Memory Access Analysis for Embedded Systems

Jesper Bjerregaard

30-1-2002 Version 12

Abstract

The memory subsystem has traditionally been a major bottleneck in the design of high performance processor based systems. As the disparity in speed between a processor and its main memory continues to widen, the need for an efficient exploitation of the memory hierarchy plays an increasingly important role in achieving good overall performance. When dealing with the design of embedded systems this is particularly important. As an embedded systems application is likely to run on the same system throughout its entire lifetime, a tailoring of the systems memory configuration to fit certain application specific requirements, can be extremely beneficial. Likewise a tailoring of a particular application to fit the parameters of the memory hierarchy, can result in a better exploitation of data locality. As the latencies in memories are orders of magnitude larger than that of modern day processors, merely small changes in an applications memory access behaviour can yield considerate performance gains. Especially in data intensive applications where large amounts of data is accessed in multi level nested loops, significant improvements can be obtained.

In this thesis a wide range of techniques for improving memory access behaviour has been covered. The primary focus has however been put on the topics of carrying out memory layout- and control- transformations on application codes. The use of the former of these is to perform a specific layout of data in main memory, thereby improving data locality. The use of control transformations can in a similar way improve data locality, by rearranging memory references in an application.

Based on the presented topics a high level compiler tool for optimizing memory access behaviour has been developed. This tool has been constructed by the aid of the SUIF compiler set, and is capable of carrying out the well known tiling transformation. The developed tool is able to perform an analysis of a given application, and to carry out the best suited tiling transformation based on this analysis. For some real-world benchmarks a 56% reduction in the number of main memory accesses has been obtained.

Preface

The work presented in this Masters thesis has been carried out by Jesper Bjerregaard at the Institute of Computer Science and Technology at the Technical University of Denmark. The course of this project has stretched over a time period of six months ending in January 2002. During this period of time I have had the pleasure of working with Jan Madsen, whom I wish to thank for his excellent guidance.

During the course of this project some tools for the practical part of the work have been used. These tools have however, turned out to be somewhat unstable. This unfortunate fact has to some extend delayed the project, and the original objectives have not been achieved. The initial problems have however after some time been overcome, and the tools have in the end turned out to be useable in the context of this project.

Jesper Bjerregaard

Contents

1 Introduction **5**

2 Survey. **7**
2.1 Objectives. 7
2.1.1 Performance. 7
2.1.2 Power. 9
2.1.3 Transformations. 9
2.2 Control Transformations. 10
2.2.1 Introduction. 10
2.2.2 Overview. 11
2.2.3 Related Work. 14
2.2.4 Summary. 17
2.3 Memory Layout Transformations. 17
2.3.1 Introduction. 17
2.3.2 Scalar variables layout. 19
2.3.3 Array Layout. 23
2.3.4 Array Padding. 24
2.3.5 Tile-based Layout. 27
2.3.6 Instruction Layout. 28
2.3.7 Selection of Dynamic Memory Layouts. 29
2.3.8 Concluding Remarks. 33
2.4 Memory hierarchy design. 33
2.4.1 Introduction. 33
2.4.2 Power-Performance tradeoffs and characteristics. 34
2.4.3 Other factors. 36
2.4.4 Estimation approaches. 38
2.4.5 Specific Estimation algorithm. 40
2.4.6 Static memory. 48
2.4.7 Conclusion. 52
2.5 Other Areas. 53
2.5.1 Introduction. 53
2.5.2 Worst Case Execution Time. 53
2.5.3 Alternative approaches. 56
2.6 Discussion. 59
2.7 Summary. 61

3 Framework. 63
3.1 Introduction 63
3.2 Overview 63
3.3 SUIF. 65
3.3.1 Introduction 65
3.3.2 Pros & cons. 66
3.3.3 Detailed description. 66
3.4 SimpleScalar 67

4 Implementation. 69
4.1 Discussion of selected implementation. 69
4.2 Implementation Specific Theory. 71
4.2.1 Data dependencies. 71
4.2.2 Dependences in loops. 72
4.2.3 Legality criteria for transformations. 75
4.2.4 Tiling aspects. 76
4.2.5 Representation of references. 78
4.2.6 Determination of reuse and equivalence classes. 80
4.2.7 Reuse and equivalence classes - a mathematical approach. 82
4.2.8 Quantifying reuse. 86
4.2.9 Limitations. 90
4.3 Implementation - Overview of code. 95
4.3.1 Tasks. 95
4.3.2 Overview. 97
4.3.3 Individual parts of the implementation. 100
4.3.4 User interface. 104
4.3.5 Shortcomings. 106
4.4 Future Work and Improvements. 108
4.4.1 Interchange 108
4.4.2 Memory Layout 109
4.4.3 Estimations 109

5 Testing. 111
5.1 Benchmarks. 111
5.2 Assumptions and metrics. 112
5.3 Results. 114
5.3.1 Matrix multiplication. 114
5.3.2 Successive Over Relaxation. 118
5.3.3 CONV. 120
5.3.4 Local summation problem. 122
5.4 Summary. 123

6 Conclusion. 125

A Code. **130**
A.1 my_tile.cc . . . 130
A.2 class for_loops . . . 134
A.3 class nested_loop . . . 136
A.4 locality.h . . . 139
A.5 locality.cc . . . 143
A.5.1 class uni_gen_set . . . 145
A.5.2 auxiliary . . . 148
A.5.3 iteration_space . . . 154
A.6 for_loop_transform.h . . . 158
A.7 for_loop_transform.cc . . . 161
A.7.1 Dependences. . . . 165
A.7.2 Locality analysis. . . . 170
A.7.3 Evaluating reuse. . . . 174
A.7.4 Auxiliary. . . . 178

B SimpleScalar installation errors. **180**

C Debugging tests. **184**
C.1 Dependence tests. . . . 184
C.2 Uniformly generated sets. . . . 193
C.3 Evaluation tests. . . . 200

Chapter 1

Introduction

In the recent years, the gap between memory and processor performance has increased significantly, thereby making the memory sub-system a large bottleneck in todays data based systems. As this gap continues to grow, the memory sub-system becomes an even more critical component, seriously degrading performance. This developement has affected the performance of both general purpose systems, as well as application specific embedded systems.

Aside from being a limiting factor for performance in terms of cycles, the memory system also contributes by consuming significant amounts of power and by occupying large fractions of the chip area. These drawbacks are especially important in the context of embedded systems, as the demand for small portable battery powered devices continues to grow.

In order to reduce the negative impacts of limited speed and energy inefficiency in todays memories, extensive research aiming at a better exploitation of the memory hierarchy has been carried out. The developed techniques can roughly be divided into two categories, namely program transformations and cache design. Program transformations are applied to an application in order to allow for a better exploitation of data locality, thereby improving its memory specific behaviour. These transformations are usually performed at a high level of abstraction, e.g. as source to source transformations, before compile-time.

Cache design techniques are used to tune the various cache parameters to fit the applications memory specific behaviour. Cache parameters such as the cache size, line size and associativity, can significantly influence the overall system performance with respect to both power and execution time.

Both of these memory optimizing techniques are well suited to be used in the developement of embedded systems, as a program constructed for such a system is likely to run there throughout its entire lifetime. This fact, as well as the developement of new technologies, allows for a costumization of the memory hierarchy to suit application specific requirements. A similar tailoring of the application code is performed by applying program transformations seeking to exploit the special characteristics of the memory hierarchy. An improvement of memory performance can thus be obtained by tailoring the memory hierarchy to fit application specific behaviour, or vice versa.

In this paper an exploration of the different memory optimizing techniques will be

performed. One or more of these techniques will be implemented in a tool, and an evaluation of their impact will be carried out. As this project is the first of its kind at the Technical University of Denmark, an emphasis on the theoretical parts of the presented topic will be made. Furthermore an attempt to cover as wide an area as possible, within the field of memory access optimization, will be made.

The paper is organized as follows :
Chapter 2 contains a survey covering previous work in the area. In chapter 3 the tools used for evaluation and parsing of programs are presented. Chapter 4 describes the actual implementation, and the results are evaluated in chapter 5. Finally a conclusion is offered in chapter 6.

Chapter 2

Survey.

2.1 Objectives.

The memory subsystem has through several years been a major bottleneck in developing high performance processor based systems. The main problem is that these systems require large memories, which in turn require longer access times. In order to cope with these conflicting requirements, modern day processor based systems are equipped with memory hierarchies consisting of multiple memory levels, that are used to reduce the number of the very time consuming accesses to bigger memories. The smaller levels in these hierarchies can be placed on-chip, and will thus allow for fast retrieval of the data present therein.

Such memory configurations exploit the fact that programs tend to reuse data and instructions that have been used recently. The fact that once the access to particular element has been established, the additional time penalty for fetching the nearby elements is very small, is also exploited. This is done by simultaneously fetching several elements from the higher levels of the memory hierarchy, whenever a data-element that is requested by the processor is not present in the innermost level. This fetching of new data will then most likely result in the eviction of another data block, that at this time was present in the cache. If the retrieved block has not been evicted from the cache when the next reference to data in the same block occurs, the data can now be accessed in the much faster innermost level of the memory hierarchy (i.e. the cache).

2.1.1 Performance.

The most common design of a memory hierarchy consists of three levels, namely the cache, main memory and disk. This hierarchy is sometimes extended by also including second level caches that reside between cache and main memory, or TLB's which are even smaller than the cache.

The innermost memory level is in most cases the cache. Access times for caches normally lie in the range of 1 to a few cycles, as it usually resides on-chip [34]. The main memory is the outermost level of RAM. This level of memory is also referred to as the off-chip memory, and access times usually vary somewhere between 5 and 50 cycles. Access times to the disk can take up to several hundred CPU cycles.

Allthough this introduction sofar only has described the handling of data storage, the same principles apply to the storage of instructions. Usually the memory banks that are positioned at lower levels than the main memory, are divided into data and instruction caches, and they are thus accessed separately. In figure 2.1 a common configuration of a memory hierarchy, is shown. The included access times illustrate approximate times for accessing the different memory components [34].

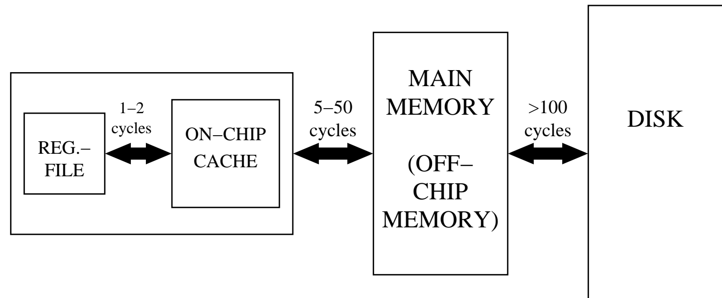


Figure 2.1: An illustration of a common memory hierarchy.

As it was former mentioned the overall strategy, for exploiting the hierarchy, is to simultaneously fetch several data elements or instructions that constitute one block of memory, from the slower memory levels into the smaller and faster levels of the hierarchy, thereby partially hiding the latency of single accesses. The number of elements that are brought into the cache at each main memory access is commonly referred to as the block size, or the line size of the cache. This number can be defined during the design phase of the memory subsystem.

Which cache line a particular data-element or instruction maps into is determined by a fixed number of the lower bits in the elements adress. That is, there exist a uniform mapping of each block of data-elements, in main memory, into the cache. As the main memory usually is much larger than the cache, several data-elements will map into the same cache line, and if this data is referenced close in time it might lead to severe performance penalties.

This feature of caches is particular important as it sometimes makes it posible to perform a layout of data and/or instructions in memory, that will ensure that data currently in use by the processor is not evicted from the cache, before the processor is done referencing it. This editing of the source code and is commonly referred to as memory layout transformations. An improvement in the reuse of data already brought into the cache can also be obtained by reordering the different memory references (control transformations), or by changing the configuration of the memory hierarchy (memory hierarchy design).

The approximate cycle estimates given above, clearly indicate that memory performance impose severe constraints on overall system capabilities. Hence the importance of improvíng the reuse of data elements and instructions, once brought into the cache, is a determinant factor for the achievement of performance gains.

2.1.2 Power.

When dealing with the reduction of power consumption, the number of transitions on bus lines is an important factor. Whenever a single bit line on one of the buses of a particular architecture changes from 1 to 0, or from 0 to 1, a power consuming charging of the existing capacitances on the line takes place. This charging and discharging of bus lines constitutes a major fraction of the overall power consumption in an embedded system, and the use of software optimizations to reduce the transition count, is therefore an obvious target for improvement. As the off-chip capacitances on a particular architecture are approximately three orders of magnitude larger than typical on-chip capacitances, the power consumption due to accesses to off-chip memory is the major source. Consequently efforts in minimizing power dissipation should concentrate on off-chip memory.

As it was the case when dealing with performance in terms of execution time, memory behaviour also constitutes a serious bottleneck when the primary goal is energy reduction.

The power dissipation introduced by memory accesses can mainly be said to consist of two different parts. One contribution from transitions on the data bus, and one stemming from transitions on the adress bus. As it is possible to predict the access sequences and thereby the exact adressing of memory locations for a particular application, the power dissipation due to transitions on the adress bus can be reduced. The power consumption introduced by transitions on the data bus is however, almost impossible to reduce using these same techniques. This is the case as there is no way of knowing what kind of data is transferred between on- and off-chip memory.

2.1.3 Transformations.

As earlier mentioned, significant performance improvements both cycle- and power- wise can be obtained by either customizing the memory hierarchy to fit application-code requirements, or by customizing the code to efficiently exploit the memory hierarchy. Both of these techniques can be used to reduce the number of data and instruction transfers between memory levels, thereby reducing execution time as well as power consumption. Furthermore the use of memory optimizing strategies alltogether, is usually targeted at loops as this is where programs generally spend most of their time.

When dealing with the costumization of application code for a particular architecture, i.e. performing program transformations, a distinction between two types of transformations are usually made. These two kinds of transformations are denoted control transformations and memory layout transformations. Control transformations are used to change the access patterns of programs in order to improve cache reuse. That is, applying control transformations to a particular application, will result in a program with reordered instructions, implying reordered memory references, that hopefully reduces memory transfers.

Memory layout transformations on the other hand, reorders the placement of data and/or instructions in the off-chip memory, in order to improve cache reuse without having to change the order of memory references.

In the next section the most commonly known control transformations will be presented. This is followed by section 2.3 in which memory layout techniques and algorithms will be described. Section 2.4 deals with memory hierarchy design, and in section 2.5

some other aspects in the area of memory optimization are presented. A summary and some concluding remarks are finally given in section 2.6.

2.2 Control Transformations.

2.2.1 Introduction.

Control transformations are used to change undesirable access patterns in applications, in order to allow for a better exploitation of the memory hierarchy. This reordering of memory references should increase cache reuse, thereby improving performance with respect to both execution time and power consumption. To illustrate the use of this kind of transformation a simple example involving interchanging of loops will be given in the following. This example will furthermore be used to illustrate some of the aspects involved when dealing with control transformations.

Loop interchange is a transformation that consists of changing the order in which the iterations are performed in a multi-level nested loop. The aim of performing such a reordering of the loop nests is to improve spatial reuse among the references. An example of interchanged loops is given in the following :
Original loop :

```
A[n][m]
for i=1 to N do
    for j=1 to N do
        A[j][i] = A[j][i] + 1
```

Interchanged loop :

```
for j=1 to N do
    for i=1 to N do
        A[j][i] = A[j][i] + 1
```

Loop interchanging is a very powerful tool in improving application performance. Provided that the arrays in the above example have been stored in row major format, the interchange will severely have increased spatial reuse in the loop, as the iterations in the innermost loop now are performed a row at a time. That is, the stride in the innermost loop is now 1, whereas it was equal to N in the original loop. Consequently the interchange allows for a large degree of reuse in the transformed loop. In the original loop however, the amount of reuse would be none, provided that the number of cache lines is less than n. That is, in that case each access to the `A` array would result in fetching new data from main memory, thereby seriously degrading performance and energy efficiency.

From a power consuming point of view the reduced off-chip memory accesses will of course be advantegeous. However even more is true. The number of transitions on the adress bus are likely to be less for the interchanged loop. If m is a power of two however, the amount of switching will remain unchanged.

When performing control transformations targeted at improving performance for the data memory hierarchy, one should keep in mind that the transformations might introduce instruction memory overhead, because of increased code size. Applying control transformations will for instance often result in an increase in the code size, thereby

leading to larger storage requirements for the particular application. Under unfortunate circumstances the instruction cache performance might also suffer from the increase in code size.

These drawbacks are however, often negligible as the data cache performance gains obtained by performing the transformation in the vast majority of cases will outweigh the negative effects. The important point to make in this context, is that the influence on code size and hence instruction cache behaviour, should not be set aside completely, as the impact can be significant in some cases. Thus a tradeoff between these two conflicting objectives might be necessary.

In the next section some more of the most common control transformations will be described. This is followed by section 2.2.3 where a review of specific earlier work in the area, is given. Finally some concluding remarks are offered in section 2.2.4.

2.2.2 Overview.

In this section some of the commonly known control transformations will be presented.

Loop Fusion.
Loop fusion is a transformation technique which combines a number of loops into a single loop, with the aim of improving cache performance. The technique is sometimes also referred to as loop jamming. The use of loop fusion is illustrated below :
Original loops :

```
for i=1 to N do
    b[i] = b[i+1] + a[i]
for i=1 to N do
    c[i] = c[i]*2 + a[i] +7
```

After loop fusion :

```
for i=1 to N do
    b[i] = b[i+1] + a[i]
    c[i] = c[i]*2 + a[i] +7
```

The main benefits of loop fusion are better exploitation of data locality and reduced loop overhead.

Loop Fission.
The transformation commonly known as loop fission consists of breaking a single loop into more than one loop, and distributing the computations and references in the original loop among these loops, thereby improving cache performance. This technique is also known as loop distribution or loop splitting. An example of applying loop fission on a suitable code fragment. is given in the following :
Original loop :

```
for i=1 to N do
    a[i] = a[i+1] * 2
    b[i] = b[i+2] + b[i]*3 + 5
```

After loop fission :

```
for i=1 to N do
    a[i] = a[i+1] * 2
for i=1 to N do
    b[i] = b[i+2] + b[i]*3 + 5
```

The potential benefits of using loop fission are :

- It reduces memory requirements as each iteration involves fewer references. This can in turn improve the cache performance significantly.
- The splitting of the involved references might reduce the number of dependencies inhabited in the loop.
- The instruction cache performance might also improve as the loop bodies become smaller.
- It might allow for a better reuse of registers, as fewer variables or constants are in use (live) at the same time.
- A drawback of loop fission is that the increased number of iterations introduces some overhead.

Loop Unswitching.
Loop unswitching consists of moving a conditional statement outside the loop it is contained within, thereby eliminating the overhead of executing the conditional in each iteration.This technique can be applied when the result of the conditional is independent of the index/iteration variable used inside the loop. When applying loop unswitching the particular loop must be replicated in each branch of the conditional in order to ensure correct execution of the program. An example of loop unswitching is illustrated in the following :
Original loop :

```
for i=1 to N
    stm1
    if cond1
        stm2
    else
        stm3
```

After loop unswitching :

```
if cond1
    for i=1 to N do
        stm1
        stm2
else
    for i=1 to N do
        stm1
        stm3
```

Loop Unrolling.
The transformation commonly known as loop unrolling consists of reducing the loop iteration count by adding instructions to the loop body. The use of loop unrolling is illustrated in the following :
Original loop :

```
for i=1 to 100 step 1 do
    B[i] = A[i] + A[i+1]
```

Unrolled loop :

```
for i=1 to 99 step 2 do
    B[i] = A[i] + A[i+1]
    B[i+1] = A[i+1] + A[i+2]
```

In this case the loop is unrolled once, and as the reference `A[i+1]` is accessed twice in each iteration it can be register allocated. Another beneficial side effect is that the loop control overhead is reduced. The code size can however, increase significantly depending on the degree of urolling. This is of course is a very undesirable effect, which could seriously degrade instruction cache performance.

Function Inlining.
Another transformation technique that is very similar to loop unrolling is function inlining. This technique consists of replacing function calls with the bodies of the respective functions.

The use of function calls in a programming language is an example of a necessary feature for improving the readability and the modularity of the code. Functions also reduce the required storage space, but they also have the side effect of degrading performance and energy efficiency compared to inlining of the code [10]. These undesirable effects arises partly because of the extra instructions that are needed to perform the actual call, as well as the return from it. Furthermore unavoidable accesses to the stack are necessary in order to save/restore registers, return adresses, base pointers and potential parameters. These memory accesses can be particularly time consuming, if they require fetching of new cache lines into the cache.

As in the case of loop unrolling performing function inlining is an attempt to increase performance at the cost of code size. Therefore a tradeoff between these two entities must be made. It can for instance be very beneficial to replace a function call with its body, when the body is very small and the call is made inside a multilevel nested loop that is executed several times.

Loop Tiling.
Loop tiling or loop blocking (as it is also referred to) is a transformation that changes the layout of a loop in order to iterate over the data accesses in tiles (or blocks), thereby reducing the number of capacity misses. The actual computations performed in the loop must of course be the same as before, but the order in which they are carried out has been changed, so that a better reuse of the data is possible. An example of a loop tiling transformation is given in the following :
Original loop :

```
for i=0 to N-1
    for j=0 to N-1
        for k=0 to N-1
            C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

Transformed loop:

```
for j'=0 to N-1 step TSIZE
    for k'=0 to N-1 step TSIZE
        /* for one tile */
        for i=0 to N-1
            for j=j' to min(j'+TSIZE-1,N-1)
                for k=k' to min(k'+TSIZE-1,N-1)
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

The use of loop tiling can be especially beneficial when the involved arrays in a loop cannot all fit into the cache at the same time. In this case the partitioning of the iteration space into TSIZE X TSIZE tiles should ensure that all the data accessed in the processing of one tile can fit into the cache, thereby reducing the number of capacity misses. Furthermore the order in which data is accessed in a loop may be more or less suited for loop tiling transformations. When the access patterns of the different arrays involved, allows for a later reuse of the fetched data, but the iteration order perecludes the exploitation of this reuse, a loop tiling transformation can enhance performance significantly. The selection of the most beneficial tile sizes is therefore very important.

The working set for a particular tile is defined as all the data elements that are accessed during the computations in the loop involving the tile. The reuse of data must be ensured by selecting the tile sizes so that the working set can fit into the cache, thereby fully eliminating capacity misses for that particular tile.

When dealing with tiled loops, cache conflicts are divided into two categories. The cache conflicts among array elements in the same tile are denoted self-interference conflicts, whereas the ones arising from competition among elements in different tiles (of the same array), or different arrays, are referred to as cross-interference conflicts.

There exists numerous articles and papers which describe the use of control transformations. An exhaustive overview of the the most commonly known transformations can be found in [2].

2.2.3 Related Work.

This section will contain a more in-depth description of some different approaches, proposed by various researcher when implementing control transformations.

Enabling transformations.
In [30] the successive use of different kinds of transformations are used to optimize application code. The important point to make in this context is, that the presence of data dependencies that prohibit the possibility of performing certain transformations, can be avoided by performing other transformations as a first step. That is, perform-

ing the transformations in the right order might enable the use of transformations that otherwise could not be applied.

The optimizations are mainly targeted at reducing power consumption for mutimedia applications. The involved transformations consists of loop- and size reduction- transformations as well as control- and data- flow transformations. Inbetween applying these different kinds of transformation to the code, yet other enabling transformations are carried out in order to be able to perform other transformations once again. The presented work constitutes a part of extensive research performed at IMEC [5], where numerous transformations are applied to applications in order to optimize performance in every possible manner.

Tiling.

In [34] a tiling algorithm is presented. The computing of the working set size is performed using techniques described in [9]. The strategy for determining the tile size consists simply of choosing the largest square tile, with dimensions equalling a multiple of the cache line size, that still makes the working set fit into the cache. In this implementation the user is also allowed to manually determine either the number of rows, the number of columns, both rows and columns or the ratio of this two entities. This approach is of course especially useful when the user have some application specific information which can be exploited.

Most of the earlier work carried out in the field of loop tiling have tried to minimize the number of conflict misses in a tiled loop by selecting appropriate tile sizes. That is, the consideration towards minimizing the number of cache conflicts have influenced the computation of tile size. This has sometimes lead to less efficient loop tiling [34].

The approach chosen by [34] consists however of selecting the tile sizes without regard to cache conflicts. These conflicts are dealt with separately, after the tile size selection has been performed. The way these conflicts are handled is by the use of a memory layout technique known as padding. This technique will be described in section 2.3, where different memory layout methods are presented. As it turns out, the approach for performing the tiling results in even better performance gains, as the use of padding allows for an optimization of the tile size independently of potential conflicts.

Matrix & vector representation.

In [8] techniques for performing both data-(memory layout) and control-(execution order) transformations is presented. The methods aim primarily at optimizing code with respect to performance, and deals also with transformations suited especially for multiprocessor systems. In this context a serious performance obstacle commonly referred to as false sharing is adressed. This phenomenom arises when different processors accesses and alters data in the same coherent block. The altering is naturally done in the respective processors local memory, but causes subsequently severe memory traffic that degrades performance.

For estimating different data- and control- transformations an algebraic notation for loops, mappings and indexations is used. This notation uses vectors and matrices for representing entities like subscript-vectors, stride-vectors, mapping-vectors, access-matrices and transformation-atrices.

To illustrate the use of this notation a small example will be described. Consider for

instance the following loop :

```
for i=1 to n
    for i=1 to n
        A[i][i+j]=....
```

For the access to the `A`-array we can define its subscript vector as `(i,i+j)` as the row indexing is done by the variable `i` and the column indexing is done by `i` added to `j`. The reasoning leading to this subscript vector could also have been done by multiplying the access matrix of the `A[i][i+j]` reference by `(i,j)` (as the iterating in the outer and inner loop is done over the variables i and j respectively). The access matrix is thus :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

A mapping vector is a vector that when multiplied by a particular subscript vector yields the offset of the referenced data element from the first element of the array. The mapping vector thus holds information of the data layout in memory. Once again considering the above example containing two nested loops and a row major memory layout, the mapping vector would be equal to `(n,1)`. A column major mapping would instead be represented by `(1,n)`. We find the offset of any element accessed by the above `A[i][i+j]` reference to be `(n,1)`·`(i,i+j)` = `(n+1)i + j`.

Another important entity in the algebraic notation is the stride vector. This vector has the property that the i'th element of the vector equals the difference between the adresses accessed when the i'th loop-variable is incremented by the particular stepsize. For the above example the stride vector is thus : `(n+1,1)`, where `n+1` is the reference stride for `A[i][i+j]` when `i` is incremented by 1 (=stepsize). Similarly 1 is the reference stride for `A[i][i+j]` when `j` is incremented by 1 (=stepsize). The stride vector can be computed by multipying the transposed access matrix by the mapping vector :

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} n \\ 1 \end{pmatrix} \quad = \quad \begin{pmatrix} n+1 \\ 1 \end{pmatrix}$$

The stride vector can be used as a measure for the possibilities of reuse to occur in a particular nested loop. As the best chances for reuse to be exploited occurs within the inner loops the last elements of the stride vector should be small. The likelihood of reuse is thus increased when the elements of the stride vector appears in decreasing order.

As the stride vector depends on both the indexing order inhabited in the loop, as well as the memory layout, these parameters should be tuned in order to achieve the optimal stride vector, thereby ensuring exploitation of data locality.

Manual optimization.

In [49] a partially manual method for optimizing applications with respect to energy and performance is used. The presented approach exploits a profiling tool to find the energy critical sections of the code, which then become targets for optimizations. A cycle accurate energy consumption simulator for mapping important code sections to the source code, is also used in the process.

Analyzing for most suited transformation.
In [54] the problems involved in determining the best optimizations for loop nests are adressed. The performed optimizations are limited to loop transformations involving interchange, reversal, skewing and tiling. The theories presented deals with both multi-dimensional arrays as well as potentially deep loop nests.

In order to perform certain transformations on a particular loop, the accesses involved are examined, and the inherent data dependence is stored in dependence vectors. The representation of the different iterations of loops are also stored in vectors, the so called index vectors. These vectors defines the iteration space of the particular loop. As the data dependence inherent in a particular loop might impose limitations on what sort of transformations are possible, the dependence vectors can be used to determine what is possible. Furthermore certain transformations can sometimes make it possible to carry out former illegal transformations (e.g. skewing can in some cases enable the use of tiling).

Apart from determining which transformations are applicable for a certain loop nest, it is also necessary to analyze the loop in order to find the most beneficial transformations. This is done by grouping the accesses into different reuse equivalence classes in the same manner as described in [37]. The four kinds of reuse classes are : self-temporal, self-spatial, group-temporal and group-spatial. An in-depth description of the meaning of this distinction between different kinds of reuse will be conducted in section 2.4 The partitioning of accesses into these types of classes is performed by mathematically aided methods. By the use of these methods the accesses are represented by one or more vectors that defines the dimensions in which the particular reference exhibits some sort of reuse. That is a dimension in the iteration space, corresponds to a particular loop.

2.2.4 Summary.

A presentation of some common and extensively used control loop transformations has been conducted in this section. The loop interchange transformation has been used to introduce the concept of control transformations. In section 2.2.2 an overview of some other control transformations was provided. The benefits as well as the drawbacks related to the different ways of optimizing performance has also been discussed. In section 2.2.3 some summaries of selected papers dealing with control transformations was given. Some of the selected papers present ways for analyzing access patterns, and finding the best suited transformations to optimize performance. When possible, multiple transformations are applied to the same application code to obtain the best performance within certain limits. Other of the reviewed papers present a widely used matrix and vector representation for describing memory layout, transformations, data dependencies and subscripts.

2.3 Memory Layout Transformations.

2.3.1 Introduction.

In the previous chapter the use of control loop transformations was described. This kind of transformations changed the iteration order in loops, thereby altering the ac-

cess patterns in order to efficiently exploit reuse, for a particular layout of the data in memory. The use of memory layout transformations however, consists of a kind of inverse approach. This kind of transformations leaves the iteration order unchanged, and attempts instead to choose a layout of data or instructions in memory, that suits the particular access patterns. The use of memory layout transformations, thus involves an analysis of an applications specific access patterns, followed by a layout strategy that based on the analysis optimizes instruction or data cache reuse.

When dealing with memory layout of arrays a distinction between row-major, column-major or tile-based memory layout is usually made. Row-major format is the most common form of layout, and it is usually the applied mapping style in compilers. In this form the elements of a row are placed in consecutive loacations in memory. This mapping is opposed to column-major format in which the elements of an array column are placed next to each other in memory. In a tile-based layout the elements of a tile of an array with specified tile-dimension lengths, are placed in consecutive locations in memory.

To illustrate the use of memory layout transformations a very simple example will be given here. The example is similar to the one presented in section 2.2.1 by which the concept of control loop transformations were introduced. Provided that the array A is stored in row-major format, the accesses in the following piece of code will exhibit very little spatial reuse :
Original loop :

```
A[n][m]
for i=1 to N do
    for j=1 to N do
        A[j][i] = A[j][i] + 1
```

In section 2.2.1 the cache reuse was significantly improved by interchanging the two for-loops. Employing a column-major mapping of the `A` array into memory will have the same effect, as this also will result in stride 1 accesses in the innermost loop.

Memory layout and control loop transformations both have the desirable property of reducing the memory bandwith requirement for a particular application. The use of memory layout transformations can sometimes however, be more powerful than loop transformations, as their use is not constrained by data dependencies. That is, no reordering of the execution order whatsoever is performed. Only the arrangement of data or instructions in memory is affected, hopefully resulting in an improvement of cache reuse.

In the area of memory layout transformations a natural distinction between instruction and data layouts exists. When dealing with the mapping of data into memory the techniques for handling scalar variables and arrays also differ. The research performed in the area of array layout is far more comprehensive than the corresponding work pertaining to scalar layout. This is because the potential performance gains are larger for arrays. The complexity of the array handling algorithms are also correspondingly greater. Furthermore the use of padding is a widely used technique when dealing with array layouts. This technique consists of inserting empty (dummy) elements between the declarations of arrays, or after each column of a particular array. This is done in order to reduce conflict misses.

In the following sections some in-depth described approaches for performing memory

layouts in different areas, will be presented. In the next two sections a thorough description of scalar and array layout techniques will be conducted. The use of padding will be described in section 2.3.4. Some references and less exhaustive descriptions of work done in the fields of tile-based layout and instruction layout will follow in the two next sections. An in-depth description of some dynamic memory layout techniques will be given in 2.3.7, followed by some concluding remarks in section 2.3.8.

2.3.2 Scalar variables layout.

In [34] a strategy for organizing data in off-chip memory in order to reduce memory traffic is presented. By applying the algorithm to a specific application, a beneficial mapping of scalar variables into memory can thus be obtained.

The employed methods assume that the branching probabilities as well as the loop bounds are known. If the loop bounds are not fixed, approximations might be obtained by profiling. Estimations of the branching probabilities will of course make it possible to achieve a higher accuracy. It is also assumed that register allocation has been performed, and that the spilled variables have been marked. Furthermore the techniques does not consider the interference of potential operatingsystem like tasks.

The overall strategy of organizing the scalar variables for a particular application involves the following steps :

1. The application is translated into an access sequence graph. The nodes represents memory references and the edges indicates a flow of control between the two nodes.
2. From the access sequence graph a closeness graph containing the same nodes (e.g. memory references) is build. The weights of the edges between nodes in this graph should be a measure/metric for the importance of keeping the two nodes in question, in the same cache-line sized area in main memory.
3. Partition the variables into clusters, so that the variables in each cluster can fit into a line in the cache. This is done by analyzing the closeness graph.
4. A cluster interference graph is build. The nodes in this graph represents clusters, and the weight of the edges indicates the desirability of mapping the two clusters into different cache lines.
5. Based on the information in the cluster interference graph the clusters are assigned profitable memory locations.

The above steps will in the following be described in greater detail. To illustrate the different steps the following fraction of code will be used:

```
....
a = 0;
b = 1;
if(c<5){
    d = e;
    f = 8;
}
else{
    for(int i=0; i<4; i++){
```

```
        g = 7;
        h = i;
    }
}
j = 5;
....
```

A node in the access sequence graph might represent either a store or a read, thus no distinction is made between those two types of memory references.

The directed edges in the graph indicates a flow of control from the one node to the other, and they also carry a weight. This weight indicates the number of times control is expected to flow to the target node from the current node, when control is at the current node. For consequtive memory references the weight is 1, whereas the two outgoing branching edges stemming from an `if`-statement (and preceeded by a memory reference) might each have a weight of 0.5. Similarly the last memory reference in a loop should have an edge pointing to the first memory reference in the same loop with a weight equal to the iteration count minus 1. How these rules are interpreted in the context of the former shown code fraction, is illustrated in figure 2.2.

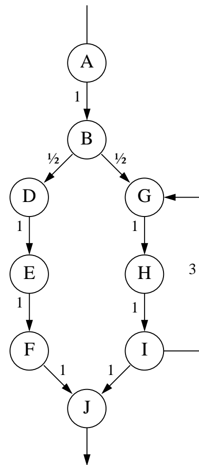


Figure 2.2: Access sequence.

The closeness graph consists of exactly the same nodes as the access sequence graph. The weights of the edges generated when building this graph should indicate the degree of desirability for keeping two variables in the same "cache-line" area in main memory. The weights of the edges are initialized with the value 0.

When building the closeness graph from the access sequence graph a simple distance function is used as a metric to determine the correlation between two references. The distance between two nodes `u`, `v` is denoted distance(`u`,`v`) and is defined as the number of variable nodes existing on a path between `u` and `v` including `u` and `v`. The path could be in either direction.

The correct weights are now assigned to the edges of the closeness graph by traversing through the access sequence graph, while considering each node in this graph. For all the nodes(`v`), that are at a distance less than the cache line size away from the node(`u`)

in question, the expected number of control flows (according to the directed edges) between these two nodes are recorded. This value is then added to the weight of the edge connecting u and v in the non-directed closeness graph. If there also exists a flow of control from `v` to `u` and this path has a distance less than the cache line size the `(u,v)` edge will receive an additional contribution to its weight when the processing of v is performed.

The preliminary version of the resulting closeness graph constructed from the former shown access sequence graph is given in figure 2.3. The graph has been build while assuming that the cache on the target processor is able to hold a maximum of three of the used scalars.

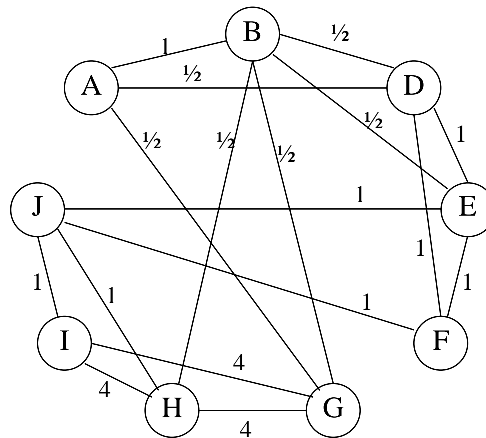


Figure 2.3: Closeness graph.

The next step is to group the variables into clusters with sizes equal to the cache line size. This is done by examining the closeness graph and selecting the clusters in a way that maximizes the total edge weights of all the clusters. Thus the clustering is basically a partitioning of the closeness graph, where the aim is to make the "cuts" separating the clusters at the low weight edges. As a high edge weight in the closeness graph indicates a good chance of exploiting spatial locality among the scalars, this partitioning criterion should result in less compulsory data cache misses. Performing an exhaustive search of the closeness graph in order to guarantee an optimal solution using this technique is however too costly. Instead an approximation algorithm for grouping the variables into clusters is employed. The steps involved are listed in the following :

1. For each node (`u`) in the closeness graph (`V`) calculate the sum S(`u`) of the edge-weights for all the edges connected to u.
2. Select the node with the highest S(`u`), and let it be the first element of a new cluster (C).
3. The variables with the highest sum of edge-weights for edges connected to nodes in C, are successively added to C. When the cache line size is reached or all variables in V has been clustered, the algorithm terminates.
4. All edges connecting C with the remaining closeness graph are deleted, and all S(`u`) values for the remaining nodes are updated. The clustering is continued at step 2.

After the clustering of variables has been performed, a so called Cluster Interference Graph (CIG) must be build. In this graph the nodes represents clusters, and the edges indicate the importance of storing clusters at memory locations that do not map into the same cache line. The CIG is also non-directed.

The algorithm for constructing the Cluster Interference Graph starts by generating one node corresponding to each cluster. The edges and their weights are then generated by performing a conversion of the former used Variable Access Sequence Graph (VASG) into a Cluster Access Sequence Graph (CASG). The conversion simply consists of replacing each node in the VASG by the cluster that contains the variable represented by the node. The conversion of a Variable Access Sequence Graph into a Cluster Access Sequence Graph is illustrated in figure 2.4. For simplicity another Access Sequence Graph than the former, is used in this example.

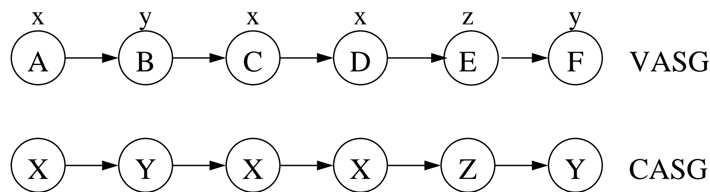


Figure 2.4: Generating a Cluster Access sequence Graph from a Variable Access Sequence Graph.

The edge weight between two nodes in the CIG is now calculated as the number of times the appearance of the same two nodes alternate in the CASG. An example of generating such a CIG from the CASG in figure 2.4 is shown in figure 2.5.

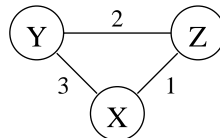


Figure 2.5: An example of a Cluster Interference Graph. (CIG)

The final step in organizing the scalar variables in main memory, consists of the assignment of the generated clusters to memory locations. In order to minimize the potential number of conflict misses the information inhabited in the Cluster Interference Graph should be exploited. A high edge-weight on the edge connecting two nodes indicates a strong desirability for letting the two corresponding clusters of variables, be mapped to different cache lines. Consequently the aim of the memory assignment phase can be formulated as follows :

Assume that all the nodes that are mapped to cache line i have a total weight-sum of W(i) on the edges connecting these nodes internally. The assignment of locations should be done in such a way that $\sum_{line} W(i)$, where the summation is performed over all the cache lines, is minimized.

The algorithm used to perform this last step, calculates the former described S values (sum of incident edge weights) for each of the clusters in the CIG, and uses this metric

for determining the order in which adresses are assigned. The clusters are then assigned to a page in memory that in size equals the number of sets in the cache times the cache line size. Each cluster is in the given order assigned to an adress that minimizes the potential number of conflicts, when mapped into the cache.

2.3.3 Array Layout.

Some of the problems involved in organizing the memory layout of arrays, in order to minimize cache conflict misses, will now be adressed. Theories for carrying out such analytical steps can be found in [18] and [34].

When dealing with arrays we do no longer take the possibility of compulsory misses into account as most arrays are much larger than the typical cache line size. Thus, there is no clustering involved when organizing arrays, as they are reluctant to fit into a cache line. The technique described here handles only the layout of one-dimensional arrays, in order to make the description simple.

The first step involved in determining a beneficial memory layout for the arrays of a specific application, consists of constructing an interference graph. This graph is simply build by generating a node for each array in the code, and connecting the nodes whose corresponding arrays are accessed in the same loop by edges. The weight of the edges is set to be the loop iteration count. If two given arrays are both accessed in different loops, the edge weight should equal the sum of those loops loop-bounds. This interference graph is used to select the the order in which memory adresses are assigned to the arrays.

For calculating the number of conflicts arising from a particular memory assignment to a specific array (`u`), a function denoted AssignmentCost is used. This function finds all the other arrays that have already been assigned a memory location and are connected to `u` by a nonzero weighted edge. For each of those arrays the additional cost (i.e. number of conflicts) is calculated if the array turns out to map to the same cache line as `u`. This can be calculated because the starting adress of both arrays, as well as the difference in their indexations, are known. Provided that they map to the same cache line the cost is calculated as the number of times references to the two arrays alternate in the loop they are accessed, times the loop iteration count.

The algorithm performing the actual assignments of memory adresses uses the sum of incident edge weights for each array, to determine the order in which to assign adresses. This is the same metric that was used earlier for clustering scalar variables (denoted S(`u`), for array `u`). In decreasing order of S(`u`) values the following procedure is performed for each array u :

For some specific starting adress and in steps of the cache line size the assignment cost of u to this adress is calculated using AssignmentCost. If this cost is less than the minimum cost found so far the cost is recorded along with the adress. When the adress examined again maps to the same cache line as the starting adress the iteration terminates, and is assigned the adress associated with the minimum cost. The location in memory where u ends is now calculated and is used as the starting adress for the memory assignment of the next array.

2.3.4 Array Padding.

Padding is a memory layout technique that consists of inserting dummy (empty) data elements in between data declarations, in order to force certain data-blocks to be mapped into different cache lines. This technique is especially useful when for instance array accesses to different arrays or different parts of the same array, continuously causes conflicts through the entire iteration of a loop. Such an unfortunate memory layout of arrays could cause many evictions of data that otherwise would have been reused. Let us for instance consider the following loop in which only two arrays are accessed :

```
for i=1 to N do
     ... = A[i] + B[i]
```

Let us assume that both `A`'s and `B`'s starting points is at a cache line boundary, and that the placement of the arrays in off-chip memory causes the first element of each array to map into the same cache line. Provided that the cache uses a direct mapping strategy, the number of off-chip accesses will in this case be equal to the number of accesses, and the performance is as a result thereof seriously degraded. Using a padding that separates the two arrays further by a distance equal to the cache line size will however enhance performance significantly. This form of layout is actually similar to the steps performed in the previous section, where memory layout of arrays was adressed.

In the following the use of padding for tiled loops will be illustrated. At first some considerations for performing a padding of single arrays will be presented. This is followed by the presentation of a technique for handling multiple arrays. A more thorough description of the use of padding can be found in [18] and [34].

Single arrays.
To illustrate the effectiveness and the use of padding in this context, the following scenario will be considered. Imagine that we are dealing with a direct mapped cache consisting of 16 sets, where each line has a size of 8 data elements. Furthermore a tile size of 8X8 elements have been selected for the tiled loop in which the only array involved is a square 1024 element array. The tile of 64 elements certainly fits in the cache (along with the rest of the working set), but the different parameters in this scenario results in an unfortunate mapping of the data into cache as severe self-interference conflicts will arise. This is illustrated in figure 2.6.

As shown in figure 2.6 the first and the fifth row of the tile will map to the same cache line. Furthermore, also the 2.,3. and 4. will conflict with the 6.,7. and 8. row respectively. However, by using padding all of the above conflicts can be avoided. A choice of padding that would ensure no self-interference conflicts during the processing of any of the 16 tiles inhabited in the array, is shown in figure 2.7.

One can be convinced that no conflicts will occur within any of the 16 tile iterations, by realizing that for any of the 16 tiles, the fetching of each of the 8 tile-rows involves adressing elements that are at a fixed distance apart. That is, the distance between the adress of the first element in tile-row no. x, and the adress of the first element in tile-row no. $x + 1$ is the same for all 16 tiles. So if no conflicts would occur during the iteration over one single tile (which has just been illustrated), then no conflicts will occur within the processing of each of the remaining 15 tiles.

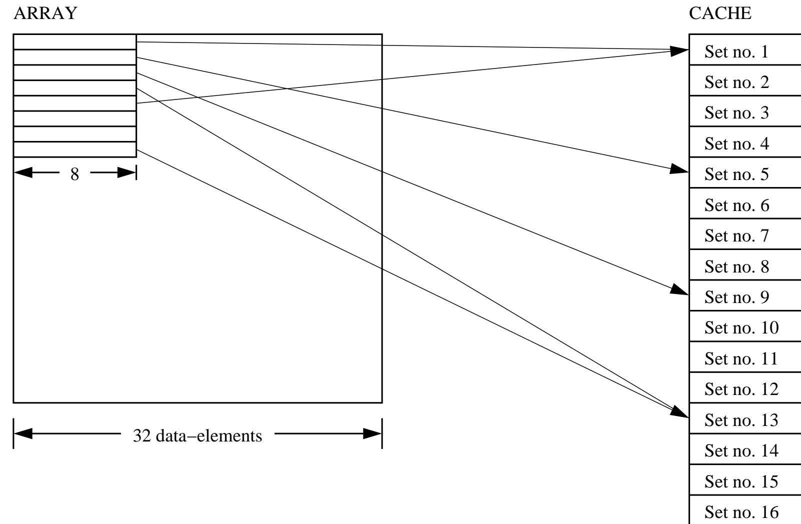


Figure 2.6: An example of self-interference in a tiled array.

One of the necessary conditions for using this kind of reasoning is, that the number of columns in the tile must be a multiple of the cache line size. This is as earlier mentioned ensured during the tile selection.

In order to find an appropriate padding of the involved array [34] uses an exhaustive algorithm that systematically explores the behaviour of the application for different padsizes. An initial step in the algorithm ensures that all the rows of the array have start adresses at a multiple of the cache line size. This adress should also be at a multiple of the cache line size, from the start adress of the first row. This is done by a padding of the array (if neccesary). The reason for performing this initial padding is that it makes the mapping into cache lines much more predictable. It is thus sufficient to check for the absence of self interference conflicts in a single tile of the array, to ensure absence of self interference conflicts for all the tiles (cf. the above discussion).

The next step in the algorithm consists of successively exploring the mapping into the cache for different padsizes. Once a padsize that ensures no self interference conflicts is found, the algorithm terminates returning this padsize. The algorithm is outlined in the following :

```
if (NoOfColumnsInArray mod LineSize == 0)
    then InitPad = 0;
else
    InitPad = LineSize - (N mod LineSize)
for PadSize = InitPad to CacheSize step LineSize
    status = OK
    Initialize LinesArray[i] to zero for all i where i<(number of lines in cache)
    for i=0 to TileRows
        for j=0 to TileColumns step LineSize
            k=(i x (NoOfColumnsInArray + PadSize) + j) mod CacheSize
        if (LinesArray[k/LineSize] == 1)
            status=CONFLICT
        else
```

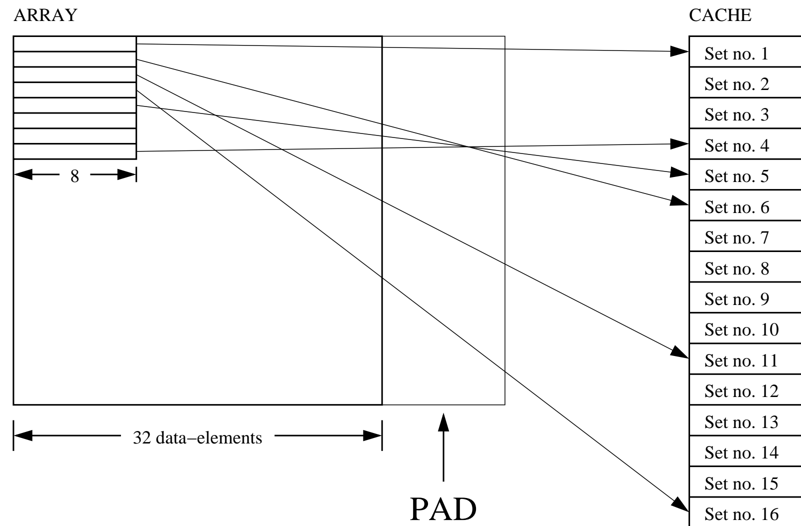


Figure 2.7: Avoiding self-interference by the use of padding.

```
            LinesArray[k/LineSize]=1
    if (status==OK)
        return PadSize
```

As shown the mapping of all the tile-elements into the cache are examined by iterating over the tile in steps of the line-size, and recording the numbers of all the occupied lines in the array LinesArray.

Multiple Arrays.

Dealing with loop tiling when multiple arrays are involved can be accomplished by incorporating minor extensions to the single-array technique. It is assumed that the involved arrays have both identical sizes and tile sizes.

Consider the two arrays represented by the two large squares, and the inhabited tiles in figure 2.8.

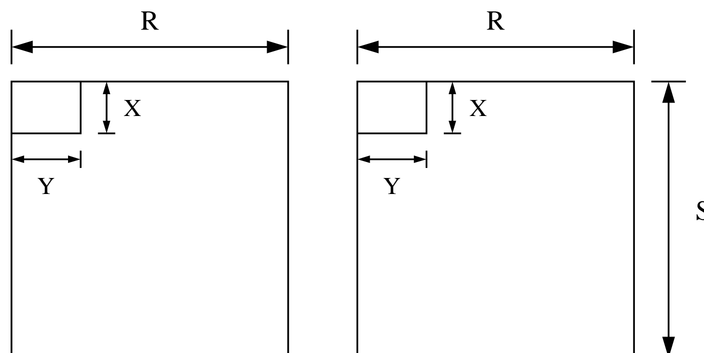


Figure 2.8: Two tiled arrays.

As shown in the figure the array dimensions are $S \times R$ and the tile sizes are denoted $X \times Y$. The steps involved in computing pad sizes and assigning array adresses for ensuring absence of self interferences (within each tile) and cross interferences (among different tiles) are :

1. Construct a new tile by placing the tiles involved next to each other, thereby forming a new rectangular shaped tile as shown in figure 2.9. The tile should be as small as possible.
2. Now consider this new tile as one coherent tile inhabited in a single array, and perform the padsize computation described in the previous section for this array. The added padding to this fiticious array should be added to both the initial arrays resulting in their new sizes becoming $S \times (R + padsize)$.
3. The last step consists of placing the two arrays in main memory at locations resulting in the distance between them being equal to $((X \times (R+padsize))``modolus'' CacheSize)$ when mapped into the cache.

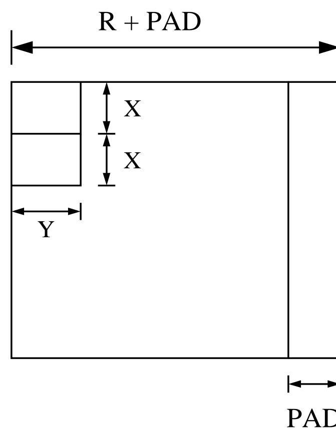


Figure 2.9: Padding of more than one array.

This placement of the arrays will together with the computed padsize in step 2 ensure, that both self-interference as well as cross-interference conflicts is avoided in each tiled loop.

2.3.5 Tile-based Layout.

In this section an approach for carrying out tile-based layout of arrays originally presented by [35], will be presented. Some of the other techniques that have been described in this paper will also be presented.

In [35] methods for enhancing performance as well as techniques for reducing power consumption in off-chip memory accesses are presented. Different hardware implemented memory access modes are in coherence with software transformations exploited to yield considerate performance gains. Furthermore some techniques for reducing power consumption in off-chip memory, by changing the data layout for arrays, are presented.

Apart from the commonly used memory access modes such as single word reads or writes, other modes can usually be employed by modern DRAMs. For instance Read-

Modify-Write (RMW) mode - where a single word is read from an adress in memory, modified, and written back to the same adress. Reads or writes of successive words in the same page can also be performed, as well as RMW operations on successive words in the same memory page.

These different ways of accessing memory data are exploited by the techniques presented by [35] in order to improve performance for specific applications. In order to make good use of the memory access modes, the data access patterns inhabited in the particular applications are analyzed, and a possible reordering of the accesses are performed. Different kinds of loop transformations might also be applied.

The techniques for reducing power consumption in off-chip memory consists basically of extensions to previous work in the area of minimization of off-chip memory accesses. It should be noted here that methods for reducing the number of cache misses and hence the number of main memory accesses in most cases also will have the beneficial side effect of reducing the power consumption [35]. The described approach is very similar to the well known performance oriented compiler optimizations, as the spatial locality and the regularity in the memory accesses is exploited.

In order to reduce the power consumption of the off-chip memory system, the number of transitions on the memory adress bus between two successive accesses must also be reduced. This task is accomplished by examining and analyzing the array reference patterns and choosing a suitable mapping of data to the main memory, based on this analysis. The different mapping styles employed are either row-major, column-major or tile-based mapping. This approach is as mentioned almost identical to the techniques aiming at maximizing data cache reuse, by dividing the iteration space of loops into tiles. In this context however, the criteria for determining the exact mapping of data into memory should primarily reduce the power consumption.

In addition to these memory layout optimizations a hardware implemented Gray Code Converter is also used to further reduce the bus transition count.

2.3.6 Instruction Layout.

In [45] a method for performing code layout transformations on applications targeted for database- and web- servers is presented. The described methods are applicable for whatever kind of application one might wish to optimze, but the proposed algorithms have been designed with this particular purpose in mind. More specifically the aim of the presented tool is intended to improve instruction cache performance for OnlineTransaction Processing Workloads (OLTP) on database- and web- servers, as this area is relatively unexplored.

The actual code optimizations are implemented with the aid of Spike, which is a program for optimizing executables for the Alpha architecture. Spike allows for different code layout transformations to be performed at different levels in the transformation phase.

The proposed approach uses profiling for performing the actual code layout optimizations. These are obtained from specific profiling tools that uses either instrumentation or sampling, and returns execution counts for the different basic blocks. A call graph that illustrates the control flow between procedures in the application is then obtained using Spike. The eges in the graph is a metric for execution frequency. A flow graph that

provides information on the internal flow of control in each procedure, is also constructed.

The next step in the transformation algorithm consists of chaining the basic blocks within each procedure together. The information inhabited in the flow graph is used to determine the internal ordering of the different basic blocks, so that successive blocks that are frequently executed is placed next to each other. Furthermore the layout of the blocks is done while attempting to reduce the number of conditional branches taken, and eliminating frequently executed unconditional branches.

After the basic block chaining has been performed each procedure is divided into different code segments, each ending with an unconditional branch or return. This partitioning step divides the procedures into smaller parts than is possible with the use of Spike. The advantages of this approach is greater flexability and ultimately better perfomance. Each code segment should consist of a few basic blocks that are likely to be executed sequentially.

The final step in the code layout optimization algorithm is to chain the code segments together, while obtaining an adjacent placement of segments with internally large execution counts. The optimization techniques has been able to reduce instruction cache misses by upto 65%.

2.3.7 Selection of Dynamic Memory Layouts.

In this section some of the work presented by [22] will be presented. This research involves an alternative approach for carrying out memory layout transformations. In [22] a dynamic approach for exploiting the data locality of a specific application is presented. The aim of the introduced method's is to change the memory layout of the different datastructures during the the execution of the program, with the purpose of improving performance compared with the earlier known static transformation algorithms, for which all the memory layouts are done at compile-time. Such an approach is useful when a program accesses the same datastructures at different program points during execution, and when at the same time, the access patterns at these points requires different memory layouts for exploiting spatial locality. The strategies presented determines when an array that is accessed in different regions of the program needs different memory layouts in some of these, and inserts the necessary code to perform the layout-transformation at run-time. In other words, the arrays representation in memory may be changed during the course of execution. The implemented algorithms and the taken approach will in the following be described in further detail.

Any kind of datastructures can be transformed to improve the data locality of the program, using the authors tool, but in order to keep things simple only multi-dimensional arrays will be treated here. Furthermore the control structures discussed are limited to sequencing, nested loops and conditional constructs (i.e. if statements), that are not allowed inside loops.

In trying to improve performance using the dynamic optimizing techniques, the tool still employs earlier known so called static optimizing strategies. Those used in this context employs a combination of loop- and data- transformations. Assuming that a single nest is to be treated, the first step is to determine the maximum inherent temporal reuse present. Hereafter a suitable loop transformation is applied to exploit this reuse.

The next step involves dividing the arrays in the (innermost) loop into two cate-

gories. One containing the arrays with temporal reuse, and one containing those arrays that do not exhibit temporal reuse. No transformations are applied to arrays in this first group, as they already exhibit temporal locality. The memory layout of the arrays in this last group however, is now transformed by applying data transformations, to exploit spatial locality for these data-sets. This must of course be done with the former used loop transformation taken into account.

Abstract program representation.
The datastructure used to represent a program is a directed graph. Each node in the graph corresponds to a loop or a possibly nested loop. Each directed edge e from a node v to a node v' indicates that there could be a control flow from the nest represented by v to the nest represented by v'. Associated with each edge (v, v') is also a weight which is the product of the estimated number of control transfers from v to v' (denoted $f_{vv'}$ for frequency) and the estimated number of exposed cache misses in v' (denoted $m_{v'}$). The name given to this sort of graph is Nest Flow Graph which is abbreviated NFG.

To illustrate both the use of the NFG datastructure as well as a computation of the frequency for a particular edge, a short code fragment and its corresponding nest flow graph will be used. The NFG is shown in figure 2.10 and the (pseudo)code from which it has been derived is presented in the following. In this code a nested loop is simply represented by the string "`Nest`", followed by a label for this nest.

```
    Nest V0
L0: Nest V1
    if(...)
        Nest V2
    else
L!:     Nest V3
        Nest V4
    if(...)
        goto L1
    Nest V5
    if(...)
        goto L0
```

In order to calculate the frequency for a specific edge we need to consider to factors. One is the probability of the branches leading to the particular edge being taken. The other is the number of iterations of all the loops enclosing the edge. Suppose we wanted to determine f_{13} for the shown program-fragment. This is done in the following way:

After the v_1 node a conditional branch might lead to the v_3 node or it might not. The probability of either of these choices is considered to be 50%. The only loop enclosing the (v_1, v_3) edge is the one going from v_5 to v_1, and if this loop iterates N times then f_{13} can simply be computed as $0.5 \cdot N$. An important observation to make at this point is, that the number of iterations in the target- nest represented by the node v_3 has not yet been taken into account. Of course this number has no bearing on the estimated frequency of control flow to v_3, but it certainly is a factor in determining the importance of the nest v_3. This iteration-count is instead incorporated into the $m_{v'}$ parameter, which is a measure for the estimated number of exposed cache misses, for a complete execu-

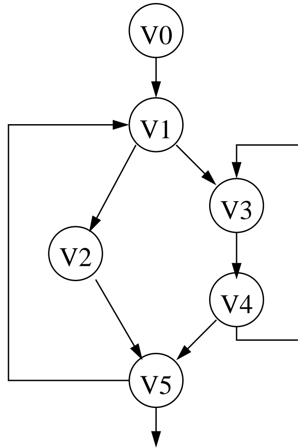


Figure 2.10: An example of a Nest Flow Graph (NFG).

tion of the innermost loop. If we assume that the loop enclosing just v_3 and v_4 iterates N' times, then f_{v3v4} can also be computed. The value of this frequency will be $0.5 \cdot N \cdot N'$.

Algorithm.
The method employed to estimate the number of exposed misses m_v for a given loop v, is taken from a technique presented by [31]. The approach is to first calculate the estimated number of exposed misses for each particular reference in the loop. This number is also referred to as the reference cost.
The way to calculate the reference cost are divided into three cases :

1. If an array reference in a loop has temporal reuse, then the cost is 1.
2. If an array reference has spatial reuse, the cost is given by : trip/(cls/stride), where trip is the number of iterations of the innermost loop. Cls is the cache line size in data items, and stride is the step size in data elements of the innermost loop.
3. If the reference in the innermost loop has neither temporal nor spatial reuse, the cost is considered to be equal to the number of iterations (trip).

In the case of spatial reuse we can consider the entity stride/cls, as a measure for the estimated number of cache misses per iteration. This number multiplied with trip must of course be equal to the reference cost.

For determining the array layouts at different program points, the static optimizing technique described by [22] is used along with the cache miss estimation technique just described. For illustative purposes we will consider a simple program which consists of a sequence of consequtive loops without any branches whatsoever. The NFG for this code is shown in figure 2.11.

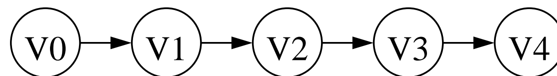


Figure 2.11: A simple example of a Nest Flow Graph.

The dynamic approach first uses the static locality optimizer described by [22] to find all the loop-nest transformations for v_0 - v_4 with the purpose of improving data locality. By applying these transformations to the code you would get what would normally be returned by the static optimizer. In the dynamic approach however, these transformations are for the time being just recorded.

The next step is to estimate the programs total number of cache misses (this number will be referred to as $cost_{04}$). Hereafter all the edges are checked, and the one with the highest value is selected as the "cut point". This is the point where it might be beneficial to change the layout of some specific array. At this point in the program a temporary partitioning of the program into two parts are made. The purpose is to check whether it is possible to obtain higher performance, if the static optimizer is run on the two partitions separately. We can do this as long as we also remember to check whether such an approach will require a change in the memory layout between these two partitions. This would of course be the case if the same array had been transformed into two different layouts, in the two program-parts.

Suppose that the selected cut point turned out to be the edge (v_2, v_3), thereby dividing the program into the two partitions consisting of the nests v_0,v_1,v_2 and v_3,v_4. This simple partitioning is illustrated in figure 2.12. The next step would then be to run the static optimizer on those two partitions, resulting in some cost values $cost_{02}$ and $cost_{34}$, which represents the estimated number of misses. Furthermore some loop- and layout-transformations are carried out, if this proves beneficial. Finally the overhead introduced by performing the layout transformations (somewhere near the cut point) should also be computed, and it could be checked whether the new total cost is lower than the former. This would be the case if $cost_{02} + overhead_{23} + cost_{34}$ was lower than $cost_{04}$, where $overhead_{23}$ represents the overhead introduced.

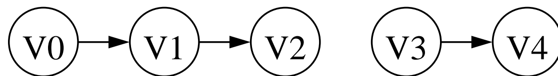


Figure 2.12: A partitioned Nest Flow Graph.

If the use of the memory layout transformation didn't improve performance relative to the static approach, then the code resulting from applying the static transformations would also be the result of using the dynamic approach. If however the layout transformations resulted in a performance gain, then the current partitioning together with the former found layout- and loop- transformations and its total cost would be recorded as the best solution found so far. A subsequent further partitioning of the NFG would then be tried in the attempt to achieve even better performance. The approach for doing this is similar to the method just described.

In [22] an algorithm for selecting the optimal points of partitioning the NFG, is also described (selection of the cut points). This is not a trivial problem as the branches in the graph allows several possible cut points which all must be examined. Once the memory layouts of the data, which might vary at different program points, have been determined, all that is left, is to carry out the appropriate dynamic layout transformations. This is of course provided that the program could benefit from different memory layouts of the same data. As these kinds of transformations are pure overhead the incentive to make

them efficient are very strong. To perform this task in the best possible way, an algorithm which tries to transform several arrays simultaneously, whenever this is possible, is used. The algorithm also makes sure that no unneccesary transformations are ever applied (eg. never performing a transformation prior to a branch if the use of the transformation depends on whether it is taken or not).

2.3.8 Concluding Remarks.

In the current chapter some different aspects of memory layout techniques have been adressed. Some analytical approaches for determining suitable memory assignments to scalar variables and arrays was presented in sections 2.3.2 and 2.3.3 respectively. In the case of scalar variables the order of accesses was examined, and those which exhibited tendencies of being accessed consequtively was mapped to the same cache line. The layout of arrays was performed in a similar way, where the number of accesses to arrays mapping into the same cache line, in the same iteration, was minimized. This was accomplished assigning appropriate starting adresses to the arrays, thereby resulting in a minimization of conflict misses.

The use of padding was described in section 2.3.4, where a technique for padding arrays in tiled loops also was presented. Less comprehensive examples of tile-based layout and instruction layout was given in sections 2.3.5 and 2.3.6 respectively. The tile-based layout of arrays was performed with the primary goal of minimizing power consumption.

Finally an algorithm for performing dynamic memory layout originally presented by Mahmut Kandemir was described. This approach involves the insertion of memory layout altering code, which is executed at run time.

2.4 Memory hierarchy design.

2.4.1 Introduction.

As the speed of processors continues to grow faster than the speed of memory in modern embedded systems, memory accesses form an increasing bottleneck in modern complex systems. Consequently the penalties associated with a cache miss becomes larger and larger relative to processor performance, and the overall system performance becomes very sensitive to cache parameters.

Modern processor based embedded systems allow the designer to costumize the on-chip memory to suit application specific requirements. This developement in integration-technology of modern day embedded systems is very useful, as especially cache- and cache-line- sizes as well as associativity are determinant factors for the overall system performance.

When parameters such as bus-widths, replacement policies and the use of additional on-chip memory levels, aside from the traditional cache, are also taken into account, the design space becomes significantly large. To be able to cope with the extremely time consuming task of examining these multiple combinations of parameter values, which is necessary to obtain an application specific designed cache, certain estimation techniques are put in to use. By applying these techniques to a particular application for

different values of the tunable cache parameters, important information pertaining to the execution time and power consumption, can be obtained. The set of cache parameters that yielded the best results, can then be chosen for the final memory configuration.

Deciding which combination of cache parameters that should determine the final architecture might also force the making of a tradeoff between power and performance. That the need for such a tradeoff actually do exist, when dealing with the tuning of certain parameters, and for certain applications, will become apparent in the following section. This section also describes earlier papers attempt do determine the characteristics of different applications for varying memory configurations (especially line-sizes). In section 2.4.3 some other factors that also have significant impact on performance and power are discussed. Some different approaches for estimating performance for different memory configurations are presented in section 2.4.4. This is followed by an in-depth description of an estimation algorithm in section 2.4.5. The use of static on-chip memory and its benefits are discussed in section 2.4.6 along with a partitioning strategy for using this memory. Finally some concluding remarks are given in section 2.4.7.

2.4.2 Power-Performance tradeoffs and characteristics.

In this section some of the tradeoffs between power and performance that are involved when configuring the on-chip memory are presented. The considered cache parameters are mainly cache size and cache line size. Furthermore some basic characteristics for common applications are examined, and an attempt to find some sort of pattern in memory behaviour will be made.

When dealing with cache design aiming at reducing power consumption, it is important not only to consider the power consumed by the fetching of a single block, but also the effect of the cache configuration on hit rates. Tailoring the cache to increase hit rates, has aside from the obvious performance gain also the desirable side effect of decreasing off-chip memory accesses, which significantly reduces power consumption [17].

In [15] an exhaustive exploration of the critical power consuming components involved in interfacing core-based designs, is presented. Also performance and area metrics are considered, and the exploration of the design space includes various kinds of core parameters, as well as the buses connecting them. More specifically, both CPU, instruction-cache, data-cache, off-chip memory and buses are considered. Moreover system parameters such as cache sizes, line sizes, associativity, off-chip memory size, bus encoding schemes, and bus widths at cross cores, are examined.

The results show that except when dealing with on-chip bus configuration, it is possible to select the other core parameters in a way that minimizes both power consumption, and execution time in most cases. When dealing with the configuration of the on-chip bus however, the results clearly indicate that a tradeoff between power and performance must be made. Large bus widths yields good performance but increases power dissipation. Not surprisingly, a reduction of the bus width decreases performace as well as power consumption. This phenomenon is explained by the fact that smaller bus widths results in less wire capacitance and hence lower power consumption. On the other hand this also affects performance, as more bus transfers now are required to fetch the same amount of data.

In [6] and [37] it has been shown that the performance for any of the involved bench-

marks improves significantly with increasing cache size, until a certain point. That is, when a certain cache size is reached, the reduction in cache misses for even larger caches becomes much smaller, and the incentive for increasing the on-chip memory is less. The tests carried out by [6] also show that the energy consumption decreases at approximately the same rate as the execution time, when the cache size is increased. At about the same point where the performance gain tends to decrease, the power dissipation increases slightly, as one might expect. That is, a point is reached where the advantages of increasing the cache further are smaller, and at this point a slight increase in power consumption is observed. This can be explained as follows. The increase in cache size contributes at this point with an energy consuming component which is larger than the power gains stemming from the reduction in cache misses, that the larger cache gives rise to.

These results are expanded to also include effects of varying the cache line size in [47] and [6]. In [47] and [6] the problems involved in simultaneously performing optimizations with respect to both loop transformations, cache configurations, and data placement strategies, are adressed. The presented methods perform the optimizations in the listed order with the aim of improving performance and minimizing area and energy consumption due to the memory system. The simulated tests shows that when dealing with the tuning of cache sizes, and especially cache line- sizes to fit performance and energy requirements, the effect on these two entities can be quite different. Depending on the application in question relatively small cache sizes will result in large miss penalties both cycle- and energy- wise. An appropriate larger cache can however, usually be found which significantly improves performance and energy dissipation, for almost any line size (the line size has however still a great impact). A further increase in the cache will at some point usually result in greater energy consumption, and the cache size should thus be chosen somewhere in the area before this happens. This is where both performance as well as energy dissipation lie within reasonable bounds.

Apart from the sections of the cache design space where the line size is unrealistically large compared with the cache size, a very clear correlation between the performance improvements, and the energy consumption seems to exist for varying line sizes. The tests simply show that the cycle count decreases and the energy dissipation increases for increasing line sizes. That the performance is improved by the selection of larger line sizes (as long as they are not to big) should come as no surprise, as this allows for better spatial reuse at little additional time costs. The reason for the increase in energy consumption stems from a number of factors. First of all the energy-wise performance does not benefit from the overall reduced time costs, of bringing larger blocks of data into the cache at the same time, as it is relatively independent of the access time to memory. Secondly as the memory access penalties has little effect on energy consumption, it benefits more from the fetching of smaller data blocks where the potential for spatial reuse is greater. Conversely the selection of larger cache lines will instead introduce an increasing amount of energy consuming transitions on the buses and in off chip circuitry.

Concluding Remarks.

When dealing with the configuration of cache sizes the results from the presented papers in this section clearly indicate that an acceptable cache size which yields good power and performance results can be obtained. Choosing the cache size that optimizes power

consumption also implies good performance behaviour in most cases. This performance can usually be improved by further increasing the cache size, but at this point the performance gains are marginal. A more explicit tradeoff exists when dealing with the selection of line sizes. In [47] the results obtained from several benchmarks roughly indicate that large line sizes favour performance on account of power, and that the inverse situation is the case for relatively small line sizes. According to the results presented by [15] reasonable compromises between power and performance can be obtained for specific line sizes. In this work the configuration of the bus width is listed as a more critical factor, when both power and performace objectives are taken into account.

2.4.3 Other factors.

In this section some selected results, from different papers, dealing with the configuration of on-chip memory are presented.

Area considerations.
Designing the memory hierarchy to fit certain application specific requirements can naturally also affect the amount of occupied chip-area. The extend of conducted research for minimizing or bounding the amount of occupied area is still very limited, compared to what has been done in order to minimize other objectives such as execution time or power consumption. The reason for this is probably that although memory is one of the most area contributing factors in embedded systems, increases can be allowed if this implies better performance and lesser power dissipation. Furthermore a reduction in power consumption has a significant impact on the amount of necessary cooling devices, which in turn impose severe size constraints. In this subsection some brief descriptions of earlier conducted research, dealing with area aspects will be given.

In [46] techniques for finding a memory configuration meeting certain area and power constraints are presented. The proposed methods allows for determining the minimum area configuration for bounded power, or minimum power configuration, for bounded area. The techniques also makes use of some of the well known loop transformations such as loop unrolling and loop fusion, to further improve the obtained results.

In [46] a discussion of area issues pertaining to memory design is also included : The use of multiport memories has some drawbacks, as they consume large amounts of power, and are also very expensive. These factors makes them impractical for embedded systems, in which single port memories are primarily used.

When dealing with the design of multiple memory banks and their individual and total sizes, a tradeoff between power dissipation and area must be made. A large memory module consume more power than the same memory capacity partitioned into several smaller parts [46]. On the other hand the smaller parts occupy a combined larger area than an equal sized coherent memory module.

In [1] Techniques for finding memory hierarchy efficient solutions customized to specific applications in embedded systems, are presented. The proposed methods aim at reducing power and area costs by tailoring the memory configuration to fit the applications requirements, thereby increasing data reuse. Several layers of memory are considered and evaluated using power- and area- cost functions.

Associativity.
The primary benefits and drawbacks of using different degrees of associativity are [28] :

Direct mapped cache : low cost, faster hit-time, high miss rate when cache is not to big

Set associative cache : higher cost, longer hit-time, lower miss rate

Selected papers describing both theories and results of the use of more or less associative caches will in the following be presented.

When accessing a set associative cache more bit line transitions are involved on average, as more than one cache line is examined. This normally results in a greater power consumption even though the capacitance for each individual bit line is smaller than in the case of a direct mapped cache [28]. The tests performed by [28] show that increasing the cache size yields higher hit rates for the instruction cache, than for the data cache. The same relative behaviour of the two caches is seen when increasing the block size. As expected the hit rate was improved in both cases. Furthermore, bit transition rate reductions of 33% and 12% were obtained by gray code conversions for the instruction- and data- cache adress buses respectively.

Other cache designs aiming at optimizing the memory hierarchy are reported by [55], that reviews designs such as MRU- and Skewed-Associative- caches. They also report of results showing that two-way associative caches increases access times by 40-51% over a direct mapped cache.

In [17] performance and power consumption is estimated for a number of different applications, while cache-size, line-size and associativity is varied. The results show that a two-way set associative cache is the best choice with respect to associativity for the vast majority of cases.

Some of the tests that have been carried out in order to investigate the impacts of set associative caches relative to direct mapped caches, have shown that set associativity usually improves hit rates as well as performance. This seems to be the case for a wide variety of benchmarks. One should though keep in mind, that these results have been obtained without prior optimizing transformations on the application code. That a direct mapped cache would be the best choice when control- and/or memory layout transformations have been applied beforehand could very well be the case. If furthermore cache parameters such as size and line size also had been tailored to suit certain application specific requirements, this conclusion could possibly be even more noticeable.

Other areas.
In [42] a technique for reducing the leakage power dissipation by turning off unused parts of the instruction cache is presented. This way of reducing the effect of leakage energy consumption is one of the relatively unexplored areas in the field of power reduction techniques [42]. This power consuming factor can for some technologies be just as large as the switching component and should therefore not be ignored.

In [32] energy dissipation is reduced in general purpose processors by using an additional very small I-cache (L0-cache) between the original I-cache and the CPU. The introduced methods dynamically analyzes the access behaviour of the running program, and uses the obtained results to guide the selection of blocks to placed in the L0-cache.

In [25] a hardware/software co-synthesis algorithm for designing suitable memory configurations along with other architecture features is presented. The work differs from most of the earlier work of this field by, apart from performing a synthesis of the hardware and software application parts, also to design an appropriate memory hierarchy. The target architecture consists of a multiprocessing environment involving both general purpose processors as well as application specific CPU's, such as DSP's and other hardware components.

In the area of hardware/software co-design the research pertaining to the partitioning of tasks between hardware and software, has so far primarily been concentrated on single CPU architectures with customized hardware components. In the recent years however, more and more work involving multiprocessor systems has been carried out [25].

In this section some approaches which also consider the effect of memory hierarchy design on the total chip area has been presented. The issue of which degree of associativity that is most beneficial for different application has also been adressed, and results from different papers dealing with this subject was presented. The most important conclusion that was made from these papers was that a set associative cache generally is a better choice than a direct mapped cache when the main objective is performance. It is however likely that a direct mapped cache will outperform an associative cache, when the application code have been a subject to optimizing transformations.

2.4.4 Estimation approaches.

Being able to estimate execution time and power consumption of a particular application on architectures with different configurations is a powerful ability, when dealing with memory hierarchy design. Fast and accurate estimations allow the designer to perform a much more comprehensive exploration of the design space, thereby improving the possibility of finding the most optimal memory configuration for the application in question. This can be accomplished by simply applying the estimation algorithms to the particular application code for a wide range of system parameter values, as it was also mentioned in the introduction to this chapter. The set of parameters that yielded the best results while eventually meeting certain other requirements can then be chosen.

Trace driven simulation is another method for estimating performance and power. This technique basically consists of running the application on a simulator while measuring execution time and possibly power. It is consequently a simple but time consuming approach. More specifically the approach consists of gathering adress traces from an adress trace generator and using the traces as input to a simulator that can provide statistics about the code. The technique is especially impractical because it is necessary to obtain several program traces in order to ensure the generation of a sufficiently descriptive profile of the application code [29].

Modern days complex processors and systems have made the developement of accurate estimation techniques more difficult. These microprocessors make extensive use of instruction level parallelizing features such as pipelining and caching in order to improve performance. These performance improving designs has however the somewhat unfortunate side effect, of seriously degrading the predictability of applications. This degradation is undesirable as it immensely complicates the estimation of execution time and power consumption on these systems.

Another area of embedded systems research commonly known as Worst Case Execution Time prediction (WCET) is also seriously affected by this developement. The computation of WCET bounds are closely related to the exploration of cache design in the sence that some sort of simulation or estimation of the application code is necessary to evaluate the effect of a particular cache design. The main difference between the two forms of used execution time estimations is that a safe worst case bound is less important when dealing with memory explorations. In this case a relatively close approximation of the execution time is sufficient as the characteristics of this entity is the important measure. These metrics can then be used to perform clever cache design choices as it is done in [6]. How this is done will be described further in the subsection denoted Related Work.

In the following a short discussion of the models used when estimating power, memory behaviour and execution time is given. This is followed by some examples of the different kinds of research that have been performed in the area. Finally some summarizing remarks are given.

Separating computations for estimation.
For estimating the influence of loop- or data- transformations, memory hierarchy design or other hardware- or software features on power consumption, different power models have been used. A common and natural way of structuring the computation of energy dissipation is to divide the calculations into off-chip data memory accesses, off-chip instruction memory accesses, and common operations. When dealing with memory accesses the parameters involved in the calculations are the voltage supply V_{dd}, the capacitances on the bus, decoding circuitry, ... and of course the number of off-chip memory accesses [56]. In some cases the power model is simplified even further. When dealing with data intensive applications (e.g. multimedia appplications) the power related to memory accesses constitutes an even bigger part, and the processor component is sometimes left out of the model [56]. For the same applications the accesses to off-chip instruction caches are sometimes also ignored (left out of the model) as the major amount of the execution time is spent in small nested loops, where evictions of instruction cache lines are unlikely.

Models intended for estimating execution time or memory access behaviour also attempts to partition the different computations needed, in order to simplify and clarify the performed computations. A natural way of dividing the computations when dealing with memory performance estimation is presented in [37]. In this work a distinction is made between the different kinds of cache misses. This also involves separating the misses imposed by arrays, as well as scalar variables. A discussion of the benefits and drawbacks that exist when performing such a modularization is given in [52]. The work presented in that paper deals with both cache- and pipeline- models to produce safe WCET bounds. Some other considerations pertaining to models for performing WCET computations are also included.

Related Work.
In [6] an exploration strategy for determining the optimal size for both the instruction- as well as the data- cache with regard to power, performance and area is presented. Instead of performing a time consuming exhaustive search among all the different combinations of data vs. instruction cache size and total cache size, the power- performance- and area-

characteristics of the application in question are exploited to yield optimal, and yet fast solutions to the cache design problem. The cache design strategy exploits the in section 2.4.2 mentioned general characteristics of program behaviour and its dependence of the cache configuration, to let specific fixpoints determine the design choices. The strategy also attempts to prioritize the the two cache types (instruction and data) in terms of performance- and power-gains so as to achieve the best possible results.

For computing the execution time and the energy consumption [6] makes use of some former developed tools and theories. The execution time is measured by the SimpleScalar tool [19], and energy models originally presented by [14] are also exploited.

In [29] methods for estimating the performance of instruction caches for DSP applications is presented. The proposed techniques exploit the fact that DSP code rarely has any conditional statements (if-then-else constructs) and often consists of one or more nested loops with fixed loop iteration counts. Especially the absense of conditional constructs makes the behaviour of the application very predictable.

In [12] methods for estimating cache performance by building analytical models for specific access patters is presented. The models are able to, apart from the cache size and line size, also to estimate execution times for different degrees of associativity.

In [15] the estimation of power consumption is performed while concentrating on the configuration of cache and its connecting buses, as these components contribute with a significant part of the total system power. The power consumption is thus estimated for all possible values of the system parameters such as cache size, associativity, bus size and encoding. The estimations are obtained by collecting sparse data information by simulation of the application in question, followed by a quick evaluation using models to predict power and performance behaviour.

Concluding Remarks.
The ability to quickly evaluate the impact of different memory hierarchy configurations when running an application on an embedded microprocessor system is very valueable. Such a fast evaluation of memory and power performance can be obtained by estimating these metrics directly from the application code using certain estimation techniques. The more accurate but also extremely time consuming trace driven simulations are impractical when the effect of several combinations of data- and/or instruction- cache sizes must be evaluated. Using much faster and only slightly less accurate estimation techniques allows in turn for a much larger design space to be explored. The best configuration that meets certain requirements or fulfills specific criteria can then be chosen by performing a possibly exhaustive search among the many possible combinations of memory parameters.

When the use of estimation techniques are targeted at providing safe upper bounds on the execution time, rather than approximate guidelines, a more sophisticated analysis is necessary. In this case the pipelining of instructions seriously complicates the task of finding safe bounds.

2.4.5 Specific Estimation algorithm.

In this section a thorough description of an algorithm that, for some specific memory parameters, and for a particular application, estimates the number of processor cycles

due to memory accesses. The developed estimation algorithm is used by [37] in an exhaustive exploration of the design space, where parameters such as scratch-pad RAM, cache size and line size are considered. The presented estimation technique attempts to group all the array accesses in a loop into different classes and uses this partitioning to provide estimates on cache-misses and -hits. Furthermore a modularization of the different computations is performed. That is, a distinction between the different sorts of cache misses (compulsory,capacity,conflict) is made and dealt with separately.

The estimation algorithm has according to [37] proven to provide good estimates on performance, as a very little variation has been observed between the estimates and actual simulation results. The theories for making estimates on the number of off-chip accesses is also presented in [37].

In the following a strategy for finding the optimal on-chip memory parameters for a specific application developed by [37] will be presented. The determined memory parameters are :

- The total size of the on-chip memory.
- The partitioning of the on-chip memory into cache- and scratch-pad- memory and the cache line size.

In choosing the optimal memory parameters different performance estimation techniques are applied to the application code. This is systematically done for all the possible sizes, within certain specified limits and in powers of 2, of the on-chip memory, the data cache and the cache line size. The scratch pad RAM size is of course equal to: on-chip RAM size - cache size.

The obvious advantage of using an analytical estimation based approach for determining the performance of a given application is that it is many times faster than a simulation based approach. Moreover the time it takes to perform this estimation is relatively independent of the size of the application code, and this allows in turn for an exhaustive exploration of the different combinations of memory parameters. A pseudo code that outlines the basic strategy for exploring the design of the memory architecture is given in the following :

```
for total on-chip memory size T, in powers of 2 :
    for cache size C in powers of 2 (as long as C<=T)
        scratch-pad memory size S := T-C
        assign variables to scratch-pad RAM for best performance
        for line size L in powers of 2 (as long as L<=C and L<=maxline)
            Estimate performance for the chosen parameters
    Record memory parameters that optimizes performance for this particular T
```

An explanation of the different steps in this code is given in the following:

Line 1 : The total on-chip memory size T is selected.

Line 2 : For each possible cache size C less than the chosen T we estimate performance.

Line 3-4 : The partitioning of program variables between scratch-pad- and cache- memory is determined using the algorithm described in [36].

Line 5-6 : All possible cache line sizes are examined and the performance for each of these values are recorded together with the chosen sizes of T and C.

Line 7 : The values of C, S and L that resulted in the best performance for this particular T are recorded.

This exploration algorithm is relatively simple and one can easily be convinced that this approach would in fact give correct results. The real challenge imposed by this analytical algorithm is the performance estimation in line 6.

This analytical algorithm estimates the total number of processor cycles used to access data in a specific application. The algorithm consists of two distinct parts, one for determining the total cycle count due to scalar variable accesses, and one for determining the same entity caused by array accesses. Both parts assume that the memory layout transformations presented in [34] have been applied to the code with the purpose of optimizing performance. Before analyzing the program it is first parsed into an appropriate datastructure in which all accesses that are not already inside any loop, is placed inside a loop with an iteration count of one. The calculations performed for handling the scalars are very simple, and these will be described briefly in the following.

Before describing the actual calculations a comment is made on the desirability for mapping the scalars into a scratch-pad RAM. Provided that a scratch-pad RAM is available, and that it is big enough to contain all the scalars, these will in fact be placed there by the data partitioning algorithm in line 4 of the exploration algorithm [36]. The reason for putting them there is of course that the accessing to the scalars is very static compared to that of arrays and they have therefore a good chance of interfering with array accesses. If for some reason some of them have to be placed in off-chip memory, it is of course necessary to be able to handle such a case. This situation could arise if for instance there is no scratch-pad RAM available, or if the scratch-pad RAM is very small.

We denote the number of cycles it takes to read the first word in a cache line by K, and assume that it takes one additional cycle to read each of the remaining words in this cache line. That is, if the cache size is denoted by L it would take $K + L$ cycles to read one complete line of words into the cache.

The estimation of the number of cycles it takes to access scalar variables assumes that all the scalars are placed in consequtive locations in memory and that the number of misses is equal to the number of lines the scalars covers. At first glance this might seem as a very optimistic estimation especially if a scratch-pad RAM is not present. The memory layout transformations that we have assumed have been performed on the application-code justifies however this assumption to some extend. At this point it should also be noted that the conflict misses incurred by the scalar variables will be dealt with later, and with this in mind the above estimation seems fairly reasonable.

We denote the number of scalar variables in the program by N and let the total number of accesses to these N variables be denoted by M. According to the above approximization there will then be $|N/L|$ misses and $M - |N/L|$ hits among the M accesses to the scalars. As the fetching of an L-word line from off-chip memory into cache costs $K + L$ cycles and access of a word present in the cache costs one cycle, the total number of cycles incurred by scalar accesses is :

$$(K + L) \cdot |N/L| + M - |N/L|$$

For estimating the total number of processor cycles introduced by array accesses another technique to determine the degree of reuse among the cache lines is used. This technique involves a partitioning of the different data references into so called reuse equivalence

classes. This approach employs the use of four different types of reuse classes, which are listed, together with their short definitions in the following :

Self- temporal : A memory reference accesses the same data location in different loop iterations

Self- spatial : A memory reference accesses the same cache line in different loop iterations

Group- temporal : More than one reference accesses the same data location in different iterations

Group- spatial : More than one reference accesses the same cache line in different iterations

From these definitions of the different types it is clear that self-temporal reuse implies self-spatial reuse and that group-temporal reuse implies group-spatial reuse.

To illustrate the use of these classes an identification of the different types will be performed on a specific example. The code used for this purpose is a two-level nested loop which is given in the following :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        A[i][j] = A[i][j] + A[i-1][j] + A[i+1][j] + A[i-1][j] +
                  A[i][j-1] + A[i][j+1] + B[i] + C[j][i]
```

The reference `C[j][i]` exhibits no reuse, (as it is defined in this context) because the loop in which it is placed iterates over the variable `j` and the distance between to successive references to the `C`-array is therefore probably large.

The reference `B[i]` exhibits self-temporal reuse as it accesses the same data location for each of the M j-iterations.

The remaining references (to the array `A`) all exhibit at least self-spatial reuse. This is because these array references are indexed by the variable `j` which is the one iterated over. The successive iterations of the innermost loop has therefore, for a particular reference to the `A`-array, a good chance of accessing data on the same cache line as the one that was accessed in the previous iteration.

However, even more is true. The references `A[i][j-1]`, `A[i][j]` and `A[i][j+1]` exhibit group-temporal reuse as the location accessed by `A[i][j+1]` in one iteration is also accessed by respectively `A[i][j]` and `A[i][j-1]` in the two following iterations. Because these three references have group-temporal reuse, they must also have group-spatial reuse. This fact could also have been obtained by observing that they occupy the same cache line in most of the iterations. This concludes the partitioning step of the references into reuse equivalence classes. To give a good overview of the obtained results, the different groups/classes are repeated here :

No reuse : `C[j][i]`
Self-temporal reuse : {`B[i]`}
Self-spatial reuse : {`A[i-1][j]`} {`A[i+1][j]`}
Group-temporal reuse : {`A[i][j-1]`,`A[i][j]`,`A[i][j+1]`}
Group-spatial reuse : none

The next step is to move all the references (i.e. groups) exhibiting self-temporal reuse

one level up in the nested loop. The reason for doing this is that the number of off-chip references to this kind of reuse class is more convieniently calculated in the immediately enclosing loop. In this loop (i.e. the i-loop) the expected number of off-chip accesses to `B[i]` is $1/L$ per iteration. Moreover, now all the remaining reuse groups in the `j`-loop have an expected number of off-chip accesses of $1/L$, as they belong to either of the three reuse types self-spatial, group-temporal or group-spatial. By moving all the self-temporal reuse references to the appropriate (possibly even higher level) enclosing loop, the calculation of the number of processor cycles due each reuse group in a loop, is simply performed by multiplying the total iteration count for the loop with $(K + L)/L$. $(K + L)$ being the penalty in processor cycles for fetching an entire cache line from the off-chip memory, and $1/L$ being the probability of a miss in this particular iteration.

When calculating the number of processor cycles introduced by the cache misses of the `B[i]` reference this is also done correctly as it now is present in the i-loop. The number of processor cycles originating from cache hits by the `B[i]` reference must of course also be taken into account. This is however easily done by making the `B[i]` reference appear j times in the `i`-loop. The $j - 1/L$ cache hits will then later be multiplied with the product of all the enclosing loops iteration counts to give the correct total number of cycles.

The only group left to consider in the given example is {`C[j][i]`} which was categorized as a "no reuse" group. The cost of such a group is considered to be 1 off-chip access per iteration. That is, all misses.

To be able to handle the general case for determining data reuse in any possible level of the loop nest and not just the innermost one (as just described) [37] refines the reuse analysis techniques of [13] by introducing some new concepts. One of the adressed problems pertains to formalizing what conditions need to be fulfilled for spatial reuse to occur at any given level in the nested loop. From this point in the code it is necessary to be able to determine if accesses from the inner loops are likely to evict data in the cache that otherwise could have been reused. If this is in fact the case, then the potential reuse group must instead be recorded as a "no reuse" group. These decisions must of course be made simultaneously with the partitioning of the references into reuse equivalence classes to make sure that the partitioning is performed correctly.

The checking whether reuse is possible or not must be done for all the self-spatial and group-spatial equivalence classes in a loop. Remember that these two types of classes are the only two we are concerned with at this point in the exploration-strategy. This is the case as the self-temporal groups have been converted to self-spatial groups and we consider the group-temporal classes as an instance of a group-spatial class. Apart from being a necessary step in the partitioning of references into classes, this estimation of potential reuses might also reveal some reuse classes that based on the previous example are less obvious, than the ones we ended up with the first time. Considering this same example once more, we discover that the data read by the `A[i][j]` reference might also be used by the `A[i-1][j]`-read in a later iteration. Assuming a perfect replacement policy this reuse requires that an entire row of the `A`-array are allowed in the cache when the iterating is done within the i-loop. Provided that we estimate all the intervening accesses not to evict the cache line needed by `A[i-1][j]`, this reference could also be a part of the {`A[i][j-1]`, `A[i][j]`, `A[i][j+1]`} group exhibiting spatial reuse.

The estimation technique used to partition the references into equivalence classes

simply sets the criterion that the total number of data accesses between to given references must be less than the cache size if reuse is to occur among them. This condition will be clarified with the following pseudo-code example :

```
for i=1 to ri
    access B[i]
    for j=1 to rj
        cj accesses
        for k=1 to rk
            ck accesses
            for l=1 to rl
                cl accesses
```

For this nest we wish to determine if the reference `B[i]` exhibits self-spatial reuse. The proposed condition for reuse to occur is that :

$$c_j \cdot r_j + c_k \cdot (r_j \cdot r_k) + c_l \cdot (r_j \cdot r_k \cdot r_l) < CacheSize - L$$

At first glance this approximation might seem a bit optimistic, but when the assumed memory layout transformations are also taken into account the condition seems more reasonable. If one imagines that each data element of each line fetched into the cache was accessed exactly one time during the iteration of the inner loops, then the estimation would match this number. This seems fairly reasonable. There will almost certainly be several elements that are accessed more than once, but then again, accesses to every element in every line is at the same time not likely to occur.

This proposed technique for estimating which references can be reused is applied to every potential self-spatial reference in a manner similar to the way it was just demonstrated in the above example. When the condition for reuse needs to be checked for a potential group-spatial reuse, we apply the same estimation technique with a few minor changes. When we examine the possibility for reuse among two different references we need, apart from checking that the line has not been evicted, also to make sure that the two references indeed will access the same row at some point. Imagine that part of an application includes the following piece of code :

```
....
for i=1 to 13 step 2
    ....
    access a[..][..]...[i+1]...[..][..]
    access a[..][..]...[i+2]...[..][..]
```

Reuse between the two shown accesses can never occur since the step size for the variable `i` percludes the two references from ever accessing the same row. Therefore it is also necessary to make sure that the difference in all the indexation levels of the two references are a multiple of that particular levels stepsize. The technique presented here assumes that no spatial reuse can ever occur between two references that do not access the same row (innermost dimension) of a particular array. This is a reasonable assumption as the cache line size normally is smaller than most of the arrays innermost dimensions.

If all the differences in the indexation levels reveal that there exists a possibility for reuse, the next step is to check whether or not the line accessed by the first reference is

likely to still be in the cache when the next reference accesses it. This is done using the same approximation technique employed in the self-spatial case. The only extension to the self-spatial approach is that we now have to calculate the total number of elements accessed before each of the indexations in the two references resolves to the same value. The appropriate way to do this is illustrated in the following example :

```
for i=0 to 10 step 2
    11 accesses
    for j=0 to 20
        5 accesses
        for k=0 to 30
            19 accesses
            A[i][j][k]
            A[i+8][j+3][k+7]
```

The total number of elements accessed before all three index expressions `[i+8][j+3][k+7]` resolves to the same values as `A[i][j][k]` did previously is calculated as :

$$(8/2) \cdot (11 + 5 \cdot 20 + 21 \cdot 20 \cdot 30) + 3 \cdot (5 + 21 \cdot 30) + 7 \cdot 21$$

If this number is less than CacheSize-L, then the two references in question exhibit group-spatial reuse.

Array conflict misses.

The techniques presented for estimating the number of cache misses so far has only dealt with compulsory- and capacity- misses. The estimation of the number of conflict misses in an application is dealt with separately. To be able to calculate a good approximation of this number a method that divides the arrays in a loop into a sort of compatability groups is used. After this partitioning step each compatability group generated will contain arrays that have compatible access patterns. In this case the definition of a set of compatible access patterns is that their index expressions only differ by a constant throughout the entire loop. That is, the differences are independent of the loop variables. This means that for two references to belong to the same compatability group, their indexes containing index variables must have the same coefficient at every level. For instance, the following two sets contain compatible references : {`A[2i][3j]`, `A[2i+5][3j+2]`} , {`A[i+7][4j]`,`B[i-3][4j+10]`} , whereas to references such as `A[i][2j]` and `B[i][j+1]` are not compatible.

The reason for partitioning the arrays into these compatability groups is that we can assume that no conflicts will occur between array references of the same group, as long as there has been performed appropriate memory layout transformations on the code. We can realize this by considering a simple example. Imagine that two arrays a and b are accessed in a loop and that they belong to the same compatability group. Assume furthermore that each element of the two arrays are accessed successively with a stepsize of one, starting with the first element of the particular array. Then these two arrays should be placed in memory such that the distance between both arrays first element will be at least one cache line apart when these are mapped into the cache. This approach would perclude any conflicts between the two arrays, and consequently if all arrays in a loop form just one compatability group then cache conflicts could be

avoided completely. This reasoning forms the background for splitting the calculation of the number of conflict misses in a loop into two separate cases. In the first case we deal with loops in which all array references form a single group. If this condition is satisfied an estimation of the total number of conflicts can simply be calculated as the ones arising from scalars conflicting with arrays and vice versa.

In the second case we deal with loops containing more than one compatability group. In these loops it is necessary apart from the just mentioned scalar/array conflicts also to consider the conflicts among arrays in the different compatability groups.
These abbreviations for different parameters are used in the following :

M_{sc} : Number of lines occupied by scalars in a loop.
M : Total number of lines occupied by the arrays in a loop.
M_{sj} : Number of lines occupied by the arrays of the compatability group S_j.
n_{sc} : Number of scalar accesses in a loop.
n_a : Number of array accesses in a loop.
C : The number of lines in the cache.

When dealing with the case of only one compatability group we realize that the probability of one of the n_a array accesses to map to the same line as one of the M_{sc} scalar lines must be M_{sc}/C. Similarly the probability of one of the n_{sc} scalar accesses to map to the same line as one of the M array lines must be M/C. Thus an estimation of the expected number of conflict misses per iteration in this type of loop could be given by $(1/C) \cdot (M \cdot n_{sc} + M_{sc} \cdot n_a)$. This estimation assumes that each access to a scalar/array that maps to the same cache line as one of the arrays/scalars actually will result in a cache miss each time the scalar/array is accessed.

When dealing with more than one compatability group in a loop we again consider the number of conflicts due to scalar accesses to be equal to $M \cdot n_{sc}/M$ as in the previous case. When estimating the number of conflicts due to array accesses from a particular group we must apart from the lines occupied by scalars also consider the lines that arrays from different compatability groups can map to. This is necessary as the arrays contained in the group in question no longer can be guaranteed to map to different lines than the arrays from other compatability groups. As in the case of just one compatability group we again calculate the number of conflict misses as the probability for other data to map to the same cache line as the group in question times the number of accesses to this group. This is done in spite of the fact that the data of this group could very well be in the cache when the access occurs. The probability of a conflict for the n_j accesses to group S_j is thus $(M - M_{sj} + M_{sc})/C$. The estimated number of conflict misses in one iteration is obtained by adding the misses due to scalar accesses and the misses due to accesses from each of the compatability groups :

$$\#conflicts = \frac{1}{C} \quad (M \cdot n_{sc} + \sum_{j=1}^{g} (M - M_{Sj} + M_{Sc}) n_j)$$

The final step in the memory estimation algorithm is to add all the results together. First the total number of cycles estimated for a full iteration of every loop is calculated. This number is obtained by adding the results from the reuse equivalence classes, the no reuse classes and the conflict analysis phase together. Finally the total number of

cycles found at each level is multiplied with the number of times this loop is executed, and adding these contributions from each level together will give the total cycle count for the entire loop nest.

The estimation of the number of processor cycles due to conflict misses assumes that the data brought into the cache in a specific loop are not replaced when the next iteration of this loop commences.

2.4.6 Static memory.

In this section the concept of static memory is introduced. This kind of memory is also commonly known as scratc-pad memory. A scratch-pad memory allows for fast access to its residing data, just as it is the case for cache memory. It is placed on-chip and is accessed through the same adress- and data- buses as the cache- and off-chip- memory. The scratch-pad memory is however mapped into an address space that is disjoint from the off-chip memory. This allows for placing data that is frequently accessed in this type of memory, guarantying a one-cycle access time as its contents are never replaced. The specific data that should be mapped to the scratch-pad memory is determined at compile time, and it will remain there throughout the applications entire execution [36].

In the remainder of this section the benefits and drawbacks of scratch-pad memory will be discussed. Some examples of its use will be provided, and the added compiler tasks which goes along with it will be described.

The use of a scratch-pad memory adds some other complex problems for optimizing code with respect to performance. The subject of transforming source code with the purpose of improving the overall data access times through a cache is well known. To exploit a scratch-pad RAM to its full potential requires however, slightly different techniques. What is needed is a strategy for identifying the critical data in an application that it would be beneficial to map into the scratch-pad RAM, thereby partitioning the data into two parts. One that is mapped into the local on-chip scratch-pad RAM and one that must reside in the off-chip RAM accessed through the cache. The use of scratch-pad memory thus adds some complexity to the tasks which must be performed by the compiler. In order to partition an applications data with the purpose of exploiting a scratch-pad RAM, it is necessary that the register allocation has been performed in advance. This goes of course for the cache related source-transformations as well.

Related work.

Some of the conducted research dealing with scratch-pad RAM has been conducted by [4] and [37]. In [4] an architecture exploiting a small scratch pad RAM localized close to the processor is presented. The objective of using such an extra mini-cache in addition to the traditional data cache is to improve performance and especially the energy behaviour. The scratch pad RAM is in this context denoted ASM, which stands for Application Specific Memory.

In [36] an algorithm for partitioning an applications data with the purpose of efficiently exploiting the use of a scratch-pad RAM is presented. The algorithm consequently maps scalars to the static RAM. The access patterns of arrays are together with access frequencies and "live"-periods used as metrics in determining the most beneficial partitioning of data. The approach taken in this work will be described more specifically in

the subsection denoted "Algorithm" later in this section. Before this partitioning strategy is presented a general discussion of the use of scratch-pad memory will be given. This discussion follows in the next section.

Partitioning discussion.
The aim of the developed program is to determine the mapping of all the applications variables into either a local on-chip scratch-pad RAM or into an off-chip RAM, so that the number of cache conflicts is minimized. Part of this aim can be obtained by mapping the variables that are estimated to cause the maximum number of conflicts to the scratch-pad RAM, thereby eliminating a lot of cache misses and optimizing the overall memory access performance.

The use of a scratch-pad RAM can be more or less beneficial depending on the particular application that is to be executed. If the application involves adressing that are data dependent a scratch-pad RAM might be very useful. This is because in this case the adressing order cannot be determined. An example of this situation is given in the following Histogram Evaluation Code. This code fraction builds a histogram of 256 brightness levels for a particular image.

```
char Brightnesslevel[512][512];
int Hist[256]; // Elements are initialized to zero.
....
for(i=0; i<512; i++){
    for(j=0; j<512; j++){
        // For each pixel (i,j) in the image
        level = Brightnesslevel[i][j];
        Hist[level] += 1;
    }
}
```

This piece of code is taken from a typical image processing application, and in this case a large number of cache conflicts might be avoided if the array hist was placed in the scratch-pad RAM.

In other cases the advantages are a lot less clear. If for instance, all the arrays are to big to fit into the scratch-pad RAM, the full potential of this new type of memory is not exploited. One might also imagine an application that exhibits little temporal reuse among the involved arrays. In such a scenario the cache conflicts do not cause any performance penalty, and a might not be beneficial at all.

There are at least two good reasons for consistently mapping scalar variables and constants to the scratch-pad memory. These kinds of data are very likely to interfere with arrays in the cache. This is because arrays reside in contiguous blocks of memory and a memory block consisting of scalar variables and constants are therefore very likely to map to the same cache line as some part of an array thereby causing conflict misses. Furthermore it has been observed that for most applications the memory space occupied by scalars is negligible compared to that of arrays [36]. This is yet another incentive to consequently map scalars and constants to the scratch-pad memory, instead of using a lot of energy developing sophisticated algorithms to identify the most frequently accessed scalars.

When it pertains to arrays, a good convention might be to always place arrays larger than the scratch-pad RAM size in off-chip memory. Otherwise the compiler would have to do a lot of extra work in order to ensure correct adressing of the array, as part of it would be mapped to the off-chip memory and part of into the on-chip scratch-pad memory. This would undoubtedly also introduce undesired overhead, and the code would be less efficient.

The live-period of a variable is also an important factor that should be taken into consideration when a partitioning is performed. This entity is defined as the time from the definition of the variable to its last use. Assuming intersecting live-periods among a specific group of variables a clever algorithm might be able to chose a partitioning that minimizes the number of intersecting live-periods. Such a strategy might reduce the potential number of cache-conflicts among variables. If all scalar variables, as earlier mentioned are mapped to the scratch-pad RAM, this discussion will of course be relevant for arrays only. For illustrative purposes let us consider the live-period distribution of the three arrays A, B and C in figure 2.13.

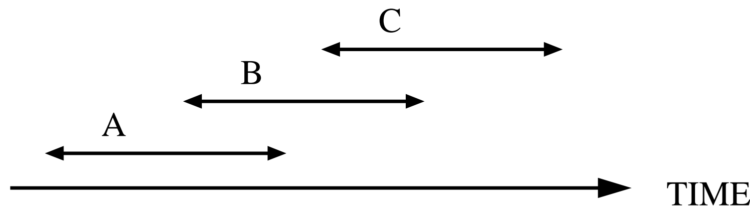


Figure 2.13: Overlapping live periods for three arrays A, B and C.

In this example array `A` intersects with `B` that again intersects with `C`. A preferable solution in this situation would be to map array b to the scratch-pad RAM, thereby making sure that the two remaining arrays would never cause any conflict misses.

Another factor that should have a great impact in determining the partitioning strategy is of course the access frequency of the array. This is because a variable that has a high number of accesses also has a high probability of introducing cache conflicts. The access frequency of an array is therefore a critical entity.

Algorithm.

The number of accesses to the elements of an array u during its entire live-period will in the following be referred to as its Variable Access Count, $VAC(u)$. Another equally important factor is the number of accesses to other arrays during the live-period of a particular array u. This entity will in the following be referred to as an arrays Interference Access Count $IAC(u)$.

As these two measures provide good indications of the number of conflicts involving a particular array, their sum must be an even better overall estimation of this number. We therefore define the Interference Factor of a variable as : $IF(u) = VAC(u) + IAC(u)$. An array with a high IF value should therefore be mapped to the scratch-pad RAM.

The theory leading to the computation of the IF value so far is well suited to be used when partitioning code involving sequences of instructions and conditional branching. When loops are also present another strategy is needed.

When accessing arrays in loops it is sometimes possible, with the aid of data alignment techniques, to make sure that there never will exist cache conflicts between certain arrays. There is however also those cases in which it might not be possible to ensure such a behaviour. An example that illustrates those two situations is given in the following :

```
for i = 0 to N-1
    access A[i]
    access B[i]
    access C[2i]
    access C[2i+1]
end for
```

As the access patterns of the arrays a and b are identical different data alignment techniques can be used to ensure that no cache conflicts between those two arrays ever occur. This is however not neccesarily the case for the array `C`. This is because the access pattern for `C` differs from that of `A` and `B`. A mapping of either the arrays `A` and `B` or the array `C` into the scratch-pad memory might then be beneficial.

To be able to make such decisions the use of another descriptive parameter that provides a measure for how many loop conflicts are likely to occur for an array would seem reasonable. This parameter is denoted LCF (Loop Conflict Factor) :

$$LCF(u) = sum_{i=1}n\Big(k(u) + sum_v k(v)\Big)$$

The outermost summation $sum_{i=1}n$ is performed over all the loops $(1, 2, ..., n)$ in which u is accessed. The innermost summation sum_v is taken over all the other arrays that are also accessed in loop i and that cannot be guaranteed to avoid cache conflicts with u using data alignment techniques. The LCF number is then a measure for the desirability to map the array in question to the scratch-pad memory. For the code shown before we have that $LCF(a) = 3$, $LCF(b) = 3$, $LCF(c) = 4$.

We now have two ways of computing an estimate for the total number of accesses of an array that might lead to cache conflicts. One that is applied on loops (LCF) and one that can be used on any other kind of program constructs (IF). When we use these two methods as just described on an application and afterwards add the numbers together we arrive at :

$$TCF(u) = IF(u) + LCF(u)$$

, where $TCF(u)$ is the Total Conflict Factor for the array u. The partitioning problem can now be formulated as follows :
For a set of arrays A_1,A_2, ... A_n in a given application, with sizes respectively equal to S_1,S_2, ... S_n and computed TCF values TCF_1,TCF_2, ... TCF_n, group the arrays into clusters so that each cluster contains arrays that do not have intersecting live-periods and so that all possible combinations of arrays are considered. Then find a subset of these clusters such that the subsets accumulated size does not exceed the scratch-pad RAM size, and such that the subsets accumulated TCF values is maximized (when choosing the final clusters each involved array should of course only appear once).

In order to perform an exhaustive analysis to find an optimal solution to the partitioning problem one would have to go through a number of extremely time-consuming tasks. First all the possible combinations of arrays that could share the same scratch-pad

RAM space (because they do not overlap in time) should be grouped into clusters (the size of the biggest array in a cluster should determine the space occupied by this group). Then all the possible combinations of these clusters that fit into the scratch-pad memory should be generated, and the one with the highest total TCF value should be picked. This approach requires $O(2^{2^n})$ time which is unacceptable.

The selected solution for the partitioning problem first groups arrays that do not overlap in time into clusters. Then a variation of the value density approximation algorithm [13] for the Knapsack Problem is used to sequentially determine the final mapping of clusters to the scratch-pad RAM. This algorithm first calculates the Access density (AD or access per size) of each cluster. This entity is a measure for the importance of mapping the arrays contained in the particular cluster to scratch-pad RAM relative to the space occupied by these arrays. It is defined as follows :

$$AD(c) = \frac{sum_{v \in c} TCF(v)}{max(size(v)|v \in c)}$$

The steps involved in the final partitioning algorithm are as follows :

1. First all the scalar variables and constants are assigned to the scratch-pad RAM, and all the arrays larger than the scratch-pad RAM's size are assigned to the off-chip RAM.
2. Generate a compatability graph for the remaining arrays. The nodes in this graph represents arrays and an edge between two nodes indicates that these two arrays have disjoint live-periods.
3. For each array u we find all the other arrays that have a size less than or equal to u and that are connected to u in the compatability graph. This generated group of arrays form a cluster for which we compute the Access Density.
4. The cluster with the highest Access Density are assigned to the scratch-pad RAM.
5. All the arrays bigger than the remaining available scratch-pad RAM are assigned to off-chip RAM.
6. We now go to step 2 and repeat the making of a new compatability graph.

This completes the presented work on partitioning application data between scratch-pad- and off-chip- memory in order to reduce cache conflicts.

2.4.7 Conclusion.

In this section some different aspects of the field of memory hierarchy design have been adressed. Several papers describing previous work in the field have been presented, and a structured overview of this research has been provided in an attempt to give some sort of insight of what has been done in the area.

In section 2.4.2 the tuning of memory hierarchy parameters to fit certain application specific requirements regarding power and performance was discussed. The results of different researchers have shown that a cache size yielding good memory behaviour both power and performance wise usually can be obtained. When dealing with the tailoring of line sizes however, a tradeoff between power and performance must be made in most cases.

In the next section some other considerations regarding area constraints was discussed, and references to some tools were given. Furthermore some results regarding cache associativity obtained by different research groups were presented. These results showed that a small degree of associativity usually improves performance over a direct mapped cache. The involved benchmarks had however not been optimized in any way previous to the running of the simulations.

In section 2.4.4 the use of performance and power estimation techniques was discussed. The ability to obtain quick estimations regarding these two objectives had proven to be very valuable, as it allows for a nearly exhaustive exploration of the design space. This in turn, provides the designer with the opportunity of finding well suited memory hierarchy parameters for the application in question.

In the next section a specific example of an algorithm able to estimate the performance of a cache was presented. The described technique used a suitable modularization of the different tasks involved, thereby separating the computations of the different kinds of cache misses.

Finally the benefits of the use of an on-chip scratch-pad RAM, which contains static data that is never evicted, was discussed in the previous section. A memory hierarchy with such an extra "cache" would be very desirable for certain applications. Scalar variables as well as arrays that are likely to interfere with other arrays could be mapped into the scratch-pad RAM, thereby yielding performance improvements. A strategy for selecting the data that would benefit the most from residing inside the scratch-pad RAM was also presented. This strategy is necessary to exploit the scratch-pad RAM to its full potential.

2.5 Other Areas.

2.5.1 Introduction.

In this chapter some work in certain research areas that didn't fit naturally into one of the three preceding chapters will be presented. Among these areas is the field of WCET-computation which was also discussed briefly in section 2.4.4 Estimation techniques, as these two fields of research are very similar. The main difference between these two areas is, as it was also prior mentioned, that safe upper bounds are needed when dealing with WCET-computations. This area is covered in section 2.5.2.

Furthermore some alternative approaches for optimizing performance or reducing power consumption in embedded systems are presented in section 2.5.3. Among the presented work are some that deals with the research areas of Multiprocessor Systems and Co-design. Also some alternative approaches for reducing power consumption, exploiting memory access instructions and exploiting instruction level paralellism are covered.

2.5.2 Worst Case Execution Time.

A vast amount of the embedded systems constructed today must in addition to cost constraints also satisfy performance constraints. As embedded systems consists of large complex components such as CPU's and ASIC's the task of computing a WCET for these building blocks as well as their compositions is not a trivial one. Furthermore the

complexity of state of the art CPU's involving the use of features such as pipelining and caches makes the computations of tight WCET bounds even more difficult, as the execution time in different parts of the program will depend on both the recent and distant history of the executed instruction trace. Furthermore the use of different levels of cache memory and especially out-of-order processors complicates the WCET computation even more.

When dealing with multiprocessor systems yet other factors need to be taken into consideration for computing the WCET. For such systems the allocation of processors (or processing elements) for different processes might result in conflicts imposing execution delays. Furthermore the scheduling of processes and the use of communication channels might also be hard to predict.

Performance analysis is also an important part of co-synthesis. Accurate performance performance estimates are an essential part of being able to meet both hard- and software requirements at minimal costs. The estimates are particular useful early in the design phase when decisions pertaining to the partitioning of tasks between hardware and software must be made.

The complexity of WCET computations can often be reduced by separating the overall computation into phases. A commonly used approach is to divide the analysis into tasks consisting of cache analysis, pipeline analysis and path analysis. This approach has been used in [52].

Performing profiling and simulation is another method for obtaining information on program behaviour and execution time. This approach has however some major drawbacks. Exhaustive simulations are impractical and less comprehensive simulation results will only cover parts of the applications behaviour and cannot provide safe bounds on the execution time.

In [53] an approach using local simulation of different basic program blocks is used to predict cache behaviour. A cache model describing which lines of data are likely to be in the cache before and after the execution of the basic blocks is used to extend the results obtained from the local cache simulation into a global prediction of the cache behaviour. In estimating the memory performance the DINERO III tool [48] is applied to the smaller code blocks to predict the altering effect of those code blocks.

In [24] an approach for estimating execution time by transforming the application assembler code into a specifically tailored simulation code is presented. After generating the assembler code it is simply transformed into a sort of assembler level C-code annotated with timing information regarding instruction scheduling, register allocation, adressing modes, memory accesses, and so on. That is, the annotations contains information about the delays imposed by the different hardware features such as caches and pipelining. This obtained model of the application code is then simulated and its behaviour is recorded.

An analogeous approach that has been used by other researchers has attempted to annotate the original source code [40] or the Control Flow Graph (CFG) [38, 44] in a similar way. These strategies is the followed by simulations. The control flow graph of the program can often be obtained by the target compiler employed.

The approaches performing annotations on the original source code is by nature not as precise as the ones that exploit the additional compiler and architecture dependent information inhabited in the control flow graph. Estimating the execution time merely

from a high level source code, the tools are forced to guess the optimizations performed by the compiler on the application code. Furthermore complex architectural features such as pipelining and caching cannot be estimated. This results of course inevitably, in less tighter execution time bounds.

In [41] an somewhat similar way of determining the Worst Case Execution Time (WCET) of an application is presented. The approach makes use of the compiler generated Control Flow Graph (CFG) to partition object-code of the application into blocks, followed by an instrumentation and execution of these. That is, the different parts of the code are altered to guarantee a worst case execution time, and the resulting code-blocks are then executed on the target architecture, while measuring the execution time.

The main reason for using this approach in determining WCET's is the increasing complexity, and hence greater modelling problems, of modern day processors. Different and complex architectures in state of the art processors makes the modelling of pipelines, caches and other hardware features difficult. As the developement of this trend isn't likely to decrease in the future an effort to construct a WCET tool that relies more on measuring than modelling seems reasonable.

[26] uses linear programming methods for estimating the WCET. The entire program is line by line assigned a cycle count representing the estimated execution time and a set of equations is used to describe the conditions that must be fulfilled for the execution of a particular program part to occur. A bound on the WCET can then be obtained by solving the system of linear equations that exhibit the control flow of the program. The major drawback of this approach is that the number of equations to be solved can be impractically large for bigger programs.

In [23] some new methods for selecting representative samples from a longer trace involving multiple samples are presented. The need for a good strategy to pick the samples that best reflects the behaviour of a specific application arises when one wishes to simulate these traces in order to obtain information about the application.

Trace driven simulation is an often used method for accurately estimating the performance of different architectures. Instruction traces could among other things be used in the design phase of a particular architecture, when important parameters pertaining to for instance branch prediction/target buffer, cache/TLB and pipelines must be made. Also memory reference traces has proved to be very useful, and has been used to great extend when memory management strategies such as paging, segmentation and memory allocation must be chosen.

There are advantages as well as drawbacks for only using samples to perform the actual simulation of the traces. The advantage of using the longer original trace is naturally that a better accuracy might be obtained. The most obvious advantage of using specific chosen samples of the original trace is that they require shorter simulation time. For some applications however, the accuracy of the estimated performance obtained by the trace driven simulation might not be better for the longer trace. This situation could arise if the behaviour of the application is very predictable, e.g. spends most of the time in a small loop.

The first step in the strategy, developed by [23], for selecting trace samples consists of grouping the samples into different clusters. The samples are grouped according to the behaviour exhibited by both the instruction- as well as the memory reference- trace. To perform this partitioning of the samples different metrics are used to characterize

them. Some of the metrics used in the grouping are :

Branch Distances : distance from branch instruction to target instruction.
Access Scatter Function : probability distribution of distances between two consecutive memory references.
Block Execution Interval : mean value of the number of instructions consecutively executed within the same main memory block (off-chip memory).

Alltogether the algorithm employs 7 different metric functions for characterizing the samples.

Finally the best samples are selected using sophisticated algorithms that attempts to calculate the distances of the samples in terms of the 7 different metrics, and picks the ones that best represents the application.

In this section some different methods for performing WCET-computations have been presented. Many of the proposed methods seem to use some sort of instrumentation of the code, in order to guarantee worst case execution times, when performing subsequent simulations of the code. The instrumentation are, for the presented examples, performed at different levels in the compilation phase, but usually at a low level.

2.5.3 Alternative approaches.

This section contains some brief descriptions of earlier work carried out in the alternative research areas mentioned in the introduction to this chapter. The covered areas are presented with the following headlines : Multiprocessor Systems, Co-Design, Alternative power reduction approaches, Exploiting memory access modes, Exploiting instruction level paralellism.

Multiprocessor Systems.
In [7] techniques for identifying and handling false sharing in multiprocessor systems is presented. These problems has also been adressed by [8], who also presented an algebraic notation useful for performing data- and control transformations. This notation is also extended by [7] in the sense that they introduce methods for representing memory layout of multidimensinal arrays using so called hyperplanes. A hyperplane is represented by a set of equations that can be used to define array elements exhibiting spatial locality. For the case of more two dimensional arrays the number of linear equations needed exeeds 1 and the hyperplanes is thus represented in a matrix structure.

In [55] an approach for exploiting data locality and parallelism on a Symmetric MultiProcessor (SMP) system at runtime is presented. Both the partitioning of different tasks in a specific application and the assignment of these to specific processors as well as data layout optimizations are performed on the application code. The system performing these optimizations is denoted Cacheminer.

A sophisticated approach is used to reorganize the tasks in an application in order to be able to partition the task into groups, where data reuse can be exploited. Such a task could for instance be a multi level nested loop where several computations involving multidimensional arrays are performed. All the processing of data in such a task is thus divided among the available processors at runtime, while different factors that all influence the overall perfomance are considered. The factor that makes a significant

performance gain possible is of course the ability to perform the processing of tasks in parallel. The factors that deminishes this gain is the decreased data cache locality, a possible load imbalance between the different processors and the scheduling overhead imposed by the partition. The impacts of all of these factors are taken into consideration when the partitioning and data layout optimizations are performed at runtime.

Co-Design.

Hardware-software co-design is an area of research in which the tasks in an embedded system is concurrently partitioned between hardware and software while considering dependencies among the two and overall performance optimization. Most of the research within this area has focused on alleviating the process of exploring the design space.

In [16] a power/performance design space exploration tool that are able to tradeoff power against performance and vice versa is presented. The tool is denoted Avalanche, and it can be used to both estimate and optimize different embedded systems with respect to both power as well as performance. The optimization and estimation steps can be applied to both power and performance simultaneously or independently.

The estimations of both power and performance (in terms of cycles) are performed based on simulation results. In the case of power estimations a measure for the total dissipated power due to memory accesses is obtained by accurately modelling the memory subsystem based on its parameters. The expected power consumption due to both cache misses as well as hits is then calculated while considering all influencing memory parameters such as cache size, associativity, tag size and so on.

Avalanche is also able to perform a hardware/software partitioning of different tasks inhabited in an application. This partitioning can be performed with respect to both power and perfomance and is the part of the Avalanche system that yields the largest improvements both power- and performance- wise.

The source to source transformations that are applied has proven to be less effective.

Alternative power reduction approaches.

The simultaneous need for modern day embedded systems to have high performance properties as well as low power consumption arises mainly because of their extensive use in battery operated portable devices [39]. Even in more power consuming static systems the need for limiting the power dissipation is great as it can reduce packaging and cooling costs and enhance reliability.

A significant fraction of the total dissipated power in embedded systems tends to come from peripheral devices. This can mainly be contributed to the fact that much larger capacitances are present in off-chip than in on-chip circuitry, and the charging and discharging of these undesirable capacitances is very power consuming.

In addition to the many conventional methods for reducing power consumption, some relatively new dynamic techniques that aim at turning off peripheral devices during their idle periods in order to reduce power consumption has been proposed. Some of the conducted work in this field has been reviewed by [39].

Yet another alternative method for reducing bus power consumption when transmitting data has been proposed by [50]. The hamming distance of two consecutive transmitted data words is computed before sending the data. If the distance is bigger than half the data word size, the inverted data is sent in order to reduce switching on the bus lines.

Exploiting memory access modes.
In [35] methods for enhancing performance as well as techniques for reducing power consumption in off-chip memory accesses are presented. Different hardware implemented memory access modes are in coherence with software transformations exploited to yield considerate performance gains. Furthermore some techniques for reducing power consumption in off-chip memory by changing the data layout for arrays are presented.

Apart from the commonly used memory access modes such as single word reads or writes, other modes can usually be employed by modern DRAMs. For instance Read-Modify-Write (RMW) mode - where a single word is read from an adress in memory, modified, and written back to the same adress. Reads or writes of successive words in the same page can also be performed as well as RMW operations on successive words in the same memory page.

These different ways of accessing memory data are exploited by the techniques presented by [35] in order to improve performance for specific applications. In order to make good use of the memory access modes the data access patterns inhabited in the particular applications are analyzed, and a possible reordering of the accesses are performed. Different kind of loop transformations might also be applied.

Exploiting instruction level paralellism.
In [33] techniques for improving the performance of microprocessors exploiting instruction level parallelism (ILP) are presented. The techniques apply transformations to the application code in order to allow the hardware to overlap the fetching of multiple cache lines from memory. The great advantage in overlapping several cache line fetches is that it makes it possible to hide a much larger part of the miss latencies than what would be possible if a cache miss should be overlapped with any other kind of instruction.

The techniques are especially concerned with the overlapping of read misses. By the use of different clustering techniques the applied transformation algorithms attempts to gather the coming read misses into sizes of the instruction windows that the target (out-of-order) processor can handle.

An example of a loop transformation that attempts to exploit miss clustering as well as spatial locality is given in the following :

```
for(...,...,i++)
    for(...,...,j++)
        .... A[i][j]
```

The above loop is as opposed to many of the regular well-known loop transformations converted into :

```
for(...,...,j++)
    for(...,...,i++)
        .... A[i][j]
```

As the entire first column of the A-array is accessed during the very first iteration of the innermost loop the number of read misses will be numerous provided that the cache line size isn't much larger than the row dimension of `A`. Furthermore the number of rows in `A` and the line size of the cache must be of magnitudes that still allow for spatial reuse

among the row elements.

2.6 Discussion.

Throughout this survey report an attempt to cover a wide range of some of the previous conducted work in the area of memory analysis has been made. In this section however, the discussion will primarily be concentrated on the considerations that should be taken into account when dealing with the optimization of memory and application-code interactions. That is, some important facts that apply for optimizing applications with respect to power and performance will be discussed.

As earlier discussed in this survey, there exists some strong incentives for optimizing embedded systems applications with respect to both power and performance. Some of the powerful tools for obtaining these goals is the use of control- and memory layout-transformations. Also the ability to design the memory hierarchy to fit certain application specific requirements can have strong impacts on power and performance.

When dealing with the task of reducing power consumption the vast majority of the conducted research have concentrated on configuring the memory hierarchy in order to meet certain power specific criteria. One of the reasons for this trend can be explained by the fact that the different parameters of a cache have significant impacts on power consumption, as they directly control the construction of the hardware. The size of a cache will for instance determine the power needed for the charging or discharging of each bit line on the bus, as well as the associativity will introduce extra hardware that also consumes power. Moreover, when configuring the memory hierarchy to fit the access patterns of an application, a reduced number of off-chip accesses can also be obtained, which again results in a reduction in power consumption. Some few alternative approaches in the area of power reduction have however, also been taken. Among these is the use of gray-code conversion to reduce the amount of switcing on adress buses.

An important point to make at this point is that performing control- or memory layout- transformation, that primarily are used for optimizing performance, also reduces power dissipation. This is again a direct result of the fewer off chip memory accesses that these transformations give rise to. This effect can also be obtained by configuring the memory hierarchy to fit the memory access patterns of a particular application, as it was just mentioned above. Thus, when designing the cache it can be constructed in a way to suit the application code, and when performing transformations the code can be tailored to fit the cache configuration.

In the area of data related optimizations, most of the research have concentrated on arrays rather than scalar variables. This fact stems from a number of reasons. Most of the data in todays applications are in some way stored in arrays and there is therefore usually much larger gains involved when dealing with arrays. Furthermore most of the execution time is normally spent in loops where the access patterns of arrays can be exploited. Especially in multi-level nested loops, where the iterations access the same array elements again and again, some large potential performance gains are present.

Some of the control transformations described earlier in this survey can for such a case be very useful. The kind of transformations that are beneficial will however, of course depend on the specific application in question. In this survey some of the

commonly known control transformations have been presented. Among these are : loop -interchange, -fusion, -fission, -unswitching, -unrolling, -tiling and function inlining. Naturally an analysis of the benefits and an investigation of the legality of performing a specific transformation must be carried out prior to the actual transformation. Especially when dealing with very data intensive applications (e.g. multimedia applications) the use of tiling transformations can yield significant gains, when the memory access patterns occur in different dimensions. That is, when for instance the access patterns consist of a square or a cube iterating over a multidimensional array.

When dealing with the configuration of the memory hierarchy, and specifically the cache size, research have shown that good results in terms of both performance and power consumption can be obtained. Trying to find a suitable cache line size will however, involve the making of a tradeoff between power and performance. These results were, obtained by performing simulations of certain benchmarks that had not been optimized for any particular cache configuration prior to the simulation. An important question in this context is whether the line size will be a much less determining factor of the final performace and power related results, if the application code has been sufficiently optimized to run on an achitecture with a certain line size. That this is in fact the case is quite possible as the cost in terms of power usually is proportional to the number of off chip memory elements fetched during the entire course of execution (ignoring the amount of bit line switching in each case).

A similar question arises when choosing among the different degrees of associativity in a particular cache configuration. Some of the conducted research in this area have shown that a small degree of associativity in most cases will yield better results than with a direct mapped cache. Applying control loop- and memory layout- transformations to the application could however, as in the former example of selecting line sizes, change this trend.

As it has been described in this survey, there currently exist three commonly used methods for reducing execution time and power consumption for applications running on embedded systems. These methods are : Control loop transformations, Memory layout transformations and Memory hierarchy design. Using all these three methods when constructing an embedded system and its application-code might ultimately yield even better results than what could have been obtained otherwise.
The obvious order in which these optimization steps should be applied is :

1. Design the memory hierarchy and select the cache parameters.

2. Apply control transformations with the selected cache parameters in mind, where it is beneficial and where data dependence constraints doesn't prohibit transformations.

3. Based on the cache parameters and the possibly altered execution order imposed by step 2, a layout of data in main memory that minimizes conflict- or compulsory misses can be performed.

This proposed order is the only one that really makes sence as control- and memory layout- transformations requires information about cache parameters, and memory layout transformations highly depend on the memory access patterns (which might be altered

by control transformations).

In this survey report there have been presented several examples of researchers that have used simulation or estimation techniques as aids in configuring the memory hierarchy to fit certain application specific requirements, during the design phase. This task might seem a bit more complicated when overall stragety of optimizing performance now also involves subsequent control- and memory layout- transformations. That is, the optimal memory configuration for the unoptimized application code need not be the same as for the optimized application code.

One way of looking at this problem might be to choose the best memory configuration for the unoptimized code, and use this design based on the point of view that even though a smaller, and thereby faster and less power consuming, cache might be a better choice for the optimized code, this selection probably won't be far from the optimal choice.

One could also use this configuration as a guidance towards a suitable configuration and then perform a simulation of the configarations that only differ slightly from this one. In this case one could also use the argument that lower degrees of associativity, and smaller cache sizes relative to the configuration obtained based on the unoptimized code might be better "guesses" than moving in the opposite direction. This strategy will definitely reduce the design space.

Furthermore some area or power constraints known from the very start in the developement of the system might provide upper bounds on the cache size.

2.7 Summary.

This survey has covered some of the previous conducted work in the area of memory exploration and optimization for embedded systems. The presentation of memory optimizing tecniques and approaches has been structured in a way that distinguishes between code transformation and memory hierarchy design. In the former approach a transformation of the specific application code is performed in order to make it fit the underlying memory hierarchy, thereby improving performance and power. In the latter case the inverse approach is taken and the memory subsystem is tuned to suit certain application specific features. When dealing with code transformations a further distinction between those that alter the control flow and those that alter the memory layout of an application has been made.

In the area of control flow transformations an overview of the most widely used transformations, and their benefits and drawbacks has been given. Some summaries of earlier papers describing particular implementations of this kind of optimization technique has also been provided.

In the area of memory layout transformations techniques for handling both scalar as well as array variables has been presented. Also the layout of instructions and dynamic memory layout approaches has been covered.

In the field of memory hierarchy design some of the tradeoffs involved when configuring the different cache parameters have been discussed. The use of estimation techniques for quick evaluations of application specific performance and power consumption have proven to be very valueable. Different approaches for implementing these techniques have been presented and a specific example of how it could be done has been given.

Furthermore the concept of scratch-pad memory has been introduced and its benefits and weaknesses has been discussed.

Finally some other areas in the field of memory exploration and optimization has been presented. These include amongst others, the fields of Worst Case Execution Time-computation, Co-design and optimization for multiprocessor systems.

Chapter 3

Framework.

In this chapter some tools which could be very useful in the context of this project will be presented. In section 3.1 an introduction to the kind of tools which could be used is given. An overview of the found tools is subsequently given in section 3.2. The specific tools are then presented and discussed individually and in greater detail in the following sections.

3.1 Introduction

When performing compiler optimization oriented research, tools for carrying out the tedious but yet complicated tasks of parsing and IR-tree generation is very useful. Likewise tools for performing the subsequent back-end generation and following evaluation of obtained performance gains is equally valueable. This is the case both when dealing with the task of performing code transformations and when memory hierarchy design is the goal.

As the primary goal for this project is to examine and analyze program code in order to obtain a better interaction between the code and the memory hierarchy, there is no need to develop additional tools if such available programs already exists. What would be really helpful when dealing with code transformations, is thus some sort of compiler which allows for analyzing and manipulating the IR-tree inbetween the front and back end generation. Furthermore some sort of evaluation tool (e.g. a simulator) which were able to calculate the effects of these manipulations would be suitable. Similar tools would also be very valueable when memory hierarchy design was the primary target, allthough only the front end of a parser would be needed for such a project. In this case estimations on cache misses could be made based on the parsed program, and the correctness of these could be evaluated by the use of the simulator.

3.2 Overview

A thorough search among available compiler- and simulator- tools have been carried out, in order to find the most promising and best suited programs for analyzing code and evaluating results in the context of this project. The use of a desirable framework for carrying out code transformations is illustrated graphically in figure 3.1.

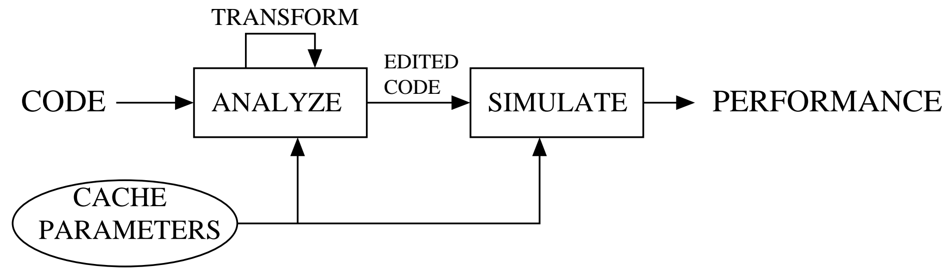


Figure 3.1: Base framework for carrying out transformations.

During the corse of searching for suitable auxiliary compilers and simulators, two very promising tools were found. The found tools are denoted The SUIF Compiler Set [11], and the SimpleScalar tool set [19].

The SUIF compiler set is a compiler that was specifically developed for performing compiler optimization research. It consists of C- and Fortran- front-ends, C- and MIPS- back-ends, and provides a suitable interface for analyzing and editing the IR-tree. The IR-tree contains all necessary information needed for performing high-level optimizations, and is thus very well suited to be used in this project.

The SimpleScalar tool set consists among other things of different simulators which are able to simulate and gather a wide range of statistics for executables. The kinds of information that can be obtained by the simulators pertains to cache hits/misses, execution time and several other profiling statistics. The simulators can simulate the behaviour of the so called SimpleScalar architecture, which is a close derivative of the MIPS architecture. Cross-compilers for generating SimpleScalar executables are also provided as a part of the tool set. Front-ends for C- and Fortran- programs exists. As the SimpleScalar tool set provides the means for quickly to obtain memory and cache behaviour statistics it is also very well suited for this project.

If one wanted to examine the impacts of code transformations on cache behaviour and performance, the SUIF compiler and the SimpleScalar tool set could thus be used for this purpose. By using the C-back-end of the SUIF-compiler, the transformed code can be compiled by the SimpleScalar cross-compiler, thereby generating SimpleScalar executables. The number of cache hits/misses can subsequently be obtained by running the appropriate simulators on these executables. The two tool sets thus provides a useful framework for carrying out code transformations and evaluating their effects. Replacing the predicates "analyze" and "simulate" on the boxes of the previous figure, with "SUIF" and "SimpleScalar" respectively, the framework illustrated in figure 3.2 is obtained.

Among the different kinds of tools which were considered, the SUIF and SimpleScalar tools appeared to be the most powerful, and almost specifically tailored for this project. Some of the other tools which were also considered can be found at :

http://www.first.gmd.de/cogent/catalog/
http://www.cs.jcu.edu.au/ alison/TONY/tony.html
http://www.softpanorama.org/Algorithms/compilers.shtml

These adresses contains extensive lists of similar programs (including links). In the

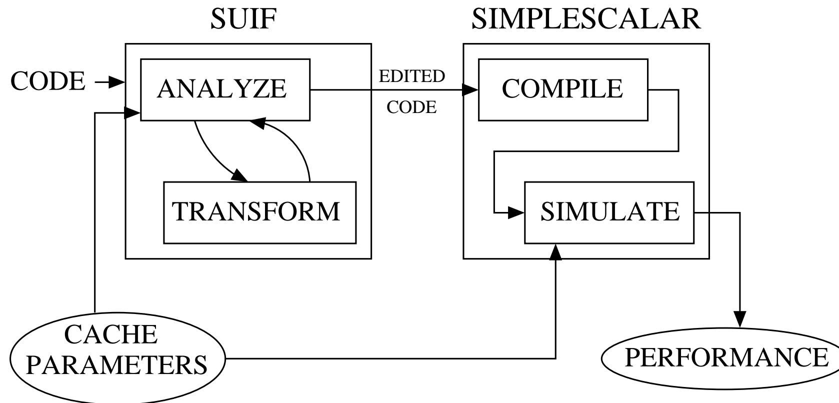


Figure 3.2: Suggested framework by using SUIF and SimpleScalar.

following two sections the SUIF and SimpleScalar tools will be described in greater detail.

3.3 SUIF.

3.3.1 Introduction

There exists two editions of the SUIF compiler system. The first edition is denoted SUIF1 and the second SUIF2. The basic functionalities of the two editions are the same, but they differ in some areas. The IR-tree representation is for instance completely different for the two systems. The reason for this is that the SUIF2 system has been expanded to contain an extensible IR-tree representation, which allows the user to make changes in the IR-tree, thereby tailoring it for specific purposes. This is the main difference between the two different implementations of SUIF.

Another feature which also has been incorporated into the SUIF2 system is the ability to integrate a userdefined compiler optimization pass together with selected front- and back-ends into a single coherent executable. This was not possible in the earlier SUIF1 implementation where the three steps of running a front-end parser, an optimization pass and lastly a back-end code generator, involved the writing of the output of each step into a file. The drawback of this approach is a that it is less efficient, as the repeated writing and reading is time rather time consuming.

The last major difference between the two systems is that the SUIF1 system is distributed along with certain very useful libraries. Some of these libraries include interfaces for performing code transformations, calculating dependencies and carrying out mathematical computations. These libraries are naturally very useful in the context of this project, and have not been ported to the SUIF2 system because the changed IR-tree structure would require the libraries to be completely rewritten.

3.3.2 Pros & cons.

In this subsection a brief discussion of the arguments which lead to the final choice of which SUIF version, will be conducted.

At first the SUIF2 system was chosen to be used as a part of the framework for performing code transformations. The strongest argument for this choice was that if a further developement of this project was to be carried out, it might not be possible to use the SUIF1 system some point in the future as the porting of SUIF1 to future days operating systems might not exist. As the structure and functionality of the SUIF2 system subsequently was explored, the system turned out to be very poorly documented. The system was distributed along with guides which explain how to use the front-ends and back-ends compilers, and with brief descriptions of the structure of a pass. Therefore the only source of information pertaining to the use and functionality of the IR-tree that is available to a user of the system is the header files for the different files which describe the IR-tree objects. Furthermore no examples for constructing a pass exists in the documentation. The SUIF1 system is on the other hand much better documented, and the distribution contains some simple examples, as well as guides for some parts of the libraries.

This unexpected drawback of the SUIF2 system thus forced the choice of choosing the SUIF1 system instead, as the task of understanding and using the SUIF2 system turned out to be to far to time-consuming. Another additional advantage of using the SUIF1 system over SUIF2 is the earlier mentioned available libraries that come with this distribution.

This switch in choosing the best suited framework, had the unfortunate effect of delaying the project to some extend. Furthermore some problems with the SUIF1 compiler system began to appear. The included transfom library turned out to generate assertion errors for some odd reason. The exact reason for this error was never exactly established, but apparently it was triggered as a result of the order in which the different SUIF libraries were included. The transform library works however quite well in the final implementation. As it was also mentioned in the preface of this thesis, these problems caused a delay of the project and as a result of this the original objectives were not fully met.

3.3.3 Detailed description.

The SUIF compiler system (both editions) has been developed by the Stanford Compiler Group at the Stanford University. SUIF is an abbrevation for Stanford University Intermediate Format, where "Intermediate Format" comes from the representation of the Intermediate Representation tree, which is used by the system. This format is the backbone of the SUIF compiler as the system is designed to provide the opportunity for conducting compiler optimization research by analyzing and manipulating this IR-tree.

The SUIF1 system contains as earlier mentioned C- and Fortran- front-ends, and C- and MIPS- back-ends. Furthermore it provides an interface for writing analyzing and/or optimizing compiler passes which operate on, and possibly modify the IR-tree representing the program under investigation. The IR-tree representation of a particular program is, after the running of a front-end on the source program, contained in a so

called suif-file. Any number of user defined passes can subsequently be applied to this suif-file. Such a pass would read the particular suif-file and after a potential transformation had been applied, the modified IR-tree would be written to another suif-file. This could then be repeated any number of times, if this is the intention. Finally the suif-file representing the possibly modified source program could be translated back to source code by using the back-end C code generator.

The front-end parser contained in the SUIF system is denoted "scc", and the generation of a suif-file from a C-program could be obtained by executing the command :

```
>scc -V -.spd input.c
```

This generates a file `input.spd`, which contains the IR-tree representation of the original C source program. After the running of one or more passes on this suif-file, by simply executing :

```
>my_pass input.spd input.out.spd
```

,if ones generated pass was denoted `my_pass`. The resulting suif-file `input.out.spd` could be converted back into C source format by the back-end `s2c` :

```
>s2c input.spd.out input.out.c
```

The interface which is provided by the SUIF1 system for generating a user defined compiler pass will also be described briefly. The code which constitutes the pass must be written in C or C++. In order to create a single executable pass one can define a procedure that is to be applied to every procedure/function contained in the IR-tree. This is accomplished by giving this function as an argument to the `suif_proc_iter(...)` function, in the `main()` program of ones pass. The `suif_proc_iter(...)` function is provided by the standard SUIF library. Furthermore some other initializing functions should be called from the `main()` program of ones pass. These functions perform all the necessary tasks of reading in the suif-file, writing it back to a file(at the end), treating command-line arguments, creating annotations in the code(used by the system), and so on.

The SUIF system also provides a standard Makefile which takes care of all the necessary tasks of creating a single standalone pass on the basis of the source code constructed by the user. The resulting pass can thus be applied to any suif-file as a single executable and will produce a possibly modified suif-file according to the actions performed in the pass. All that is needed by the user is to assign the names of ones source files to certain Makefile variables and the SUIF system takes care of the rest.

3.4 SimpleScalar

The SimpleScalar Tool Set consists of compiler, linker, assembler and simulators for the SimpleScalar architecture. The tool set allows the user to simulate programs on modern microprocessors, by the use of fast execution-driven simulation. It comes with both C- and FORTRAN- frontends which can be used to generate executables for the SimpleScalar architecture, a close derivative of the MIPS architecture. The execution of

generated binaries can be performed by one of 5 different processor simulators that in turn can provide different kinds of useful statistics about the execution process. Examples of some of the information that can be gathered during the course of execution is for instance : Execution time, profiling information and number of cache hits/misses.

The simulators provided by the tool set will also accept certain command line arguments, that specify different parameters of the architecture. Sim-outorder, which is the most powerful simulator in the tool set, is able to simulate out of order execution and will also accept arguments specifying different parameters of the processor core (no. of ALU's, no. of cache ports, ...), the memory hierarchy and branch prediction strategies.

The simulators sim-cache and sim-cheetah are especially useful for measuring the performance of the memory subsystem, as detailed memory configuration parameters can be specified as arguments at the command line. Sim-cache accepts arguments that for both instructions and data configures the TLB, level1 and possibly level2 cache. Furthermore, for each of these memory banks their block size, associativity, number of sets and replacement policy can also be given as arguments. A flushing of all caches on system calls is also possible.

Sim-cheetah, on the other hand, is capable of performing several simulations with different cache parameters simultaneously. That is, for specified intervals of line size, cache size, associativity and number of sets, sim-cheetah will perform simulations of all possible combinations of these parameters (of course these parameters are not independent of each other).

Just as it was the case with the SUIF library, some problems with the SimpleScalar tool also occurred. These problems do however not relate to the actual use of SimpleScalar. The problems occurred during the installation of the tool, where bugs in the tool were found. In order to fix these bugs an altering of the actual source code constituting the tool was necessary. Most of the fixes to these bugs were found on the web. A list of the changes that had to be made in order to obtain a working simulator are provided in appendix B.

The SimpleScalar Tool Set can be obtained at :
ftp://ftp.cs.wisc.edu/sohi/Code/simplescalar/.

Chapter 4

Implementation.

4.1 Discussion of selected implementation.

This section contains a brief introduction to the kind of memory analysis tool which has been chosen to be implemented. Furthermore a discussion of the reasons for the choice of implementation will be conducted. A brief overview of the contents of the rest of this chapter is also given.

Based on the compiler- and simulator- tools described in the previous chapter, it has been chosen to implement a memory analysis tool for carrying out tiling transformations on embedded systems applications.

Tiling transformations were previously presented in section 2.2.2, where the basics of this code transformation was introduced. The general idea of carrying out tiling transformations is to divide the iteration space of a nested loop into smaller parts (tiles), and to process each of these in turn. The example presented in section 2.2.2 involved a two dimensional tiling of the given nest, but in general any number of loops in the nest can be tiled. As the number of iterations performed in each of the loops inside a tile is relatively small, the number of elements fetched in the inner loops of the tile is reduced. This in turn allows for fetched data to be reused in the next iterations of some of the outer loops inside the tile. If such data exists it is not likely that it could have been reused in the original loop as a large number of intervening accesses in the innermost loops probably would have flushed it out of the cache. The tiles should be selected with an appropriate number of dimensions, and a size which make all the data-elements fetched during the processing of a tile fit in the cache. If this criterion is met, capacity misses during the execution of a single tile can be eliminated alltogether.

The use of tiling will be presented in greater detail in section 4.2.4. In these sections some techniques for determining the desirability of carrying out tiling transformations will also be described.

Tiling can be an extremely useful transformation when applied to certain applications. It can be especially beneficial in cases where different array references in successive iterations of a loop access elements in both the same row as well as the same column in their respective arrays. A matrix multiplication algorithm is an example of such memory access patterns. It is also a well-known case for which a tiling of the entire nest is particularly beneficial. As the references in the nested loop of a matrix multiplication

algorithm thus moves along both the rows as well as the columns of the array, the interchange of loops yields no performance gains. The correct use of tiling can however significantly reduce the number of off-chip accesses.

In cases where the references in successive iterations access elements from a number of different rows, tiling may also yield considerate performance gains. The Sucessive Over Relaxation algorithm [35] and the so called Local Summation algorithm [27] are specific real-world examples of this.

Naturally there also exist applications for which tiling transformations yield no performance gains, and just introduces loop overhead. Such applications should of course not be tiled. There definitely also exist applications which would benefit far more from other high-level cotrol-transformations, such as interchanging. Nevertheless the presence of applications that would benefit significantly from specically tailored tiling transformations, and for which no other high-level transformation is just as advantegeous, is undesputed. The ability to perform high level memory analysis and possibly subsequent tiling transformations is thus at least for some applications very valueable. This fact constitutes a part of the reasoning which has lead to the choice of implementing tiling transformations.

In order to accurately analyze a piece of specific application code for the desirability of carrying out a tiling transformation, some sort of evaluation of the potential reuse in the nested loop must be conducted. The best possible, and most accurate, measure for the potential benefits of carrying out a tiling transformation would be an estimate on the reduced number of off-chip accesses. Thus the incorporation of an algorithm which is able to make some sort of estimation on the number of off-chip accesses is preferable. Furthermore the algorithm which calculates the most beneficial dimensions and sizes of a tile, also needs to be able to gather information on the number of different cache lines that is touched during the processing of a tile. The analyzing steps which must be conducted before this task can be accomplished also involves almost all of the base work for estimating the number of off-chip accesses. In order to obtain a well working tiling transformation tool, it is therefore also necessary to construct an algorithm which can perform estimations on the number of cache misses. This algorithm is thus a beneficial side effect of performing tiling analysis. An investigation on the estimation algorithms ability to predict the values of off-chip accesses could subsequently be performed. This could involve both the relative results (compared to other tilings) and the absolute values of off-chip accesses, which could be compared with simulator results.

The SUIF1-compiler system which was presented as a part of the proposed framework in chapter 3, provides well suited tools for carying out the specific task of performing tiling transformations. This is the case as SUIF1 is distributed along with several libraries, among which some are particularly useful in the context of this project. One of these is a so called dependence library, which can be used to gather information on the existing dependences between data-elements in a nested loop. This information can subsequently be analyzed in order to ensure the legality of the kinds of transformations one wish to perform. Furthermore a transform library, which among other things is able to carry out the acual transformation is also available. The specific parameters for the transformation such as tile-dimensions and tile-sizes must be provided by the user.

In the next section of this chapter the necessary theory for implementing a working tiling transformation tool is presented. This is a rather long section as there are many

aspects which should be considered. In the following section (4.3) an overview of the code which constitutes the developed tiling transformation is provided. Finally in section 4.4 a discussion of relevant extensions and improvements for the implementation is conducted.

4.2 Implementation Specific Theory.

In this section some techniques and theories regarding the construction of a high level compiler pass will be presented. Some of the subjects adressed will apply only to the implementation of a tile pass, while others are generally applicable when dealing with high level compiler optimization. All of the described theories are necessary for constructing a well working tile pass. Among the adressed subjects are data dependencies, and mathematical approaches for evaluating and quantifying reuse. In section 4.2.1 an introduction to data dependencies will be provided. This topic is extended upon in section 4.2.2 in which data dependences in loops are considered. How to apply these theories to determine the legality of certain transformations is then adressed in the following section.

Some aspects regarding tiling are discussed in section 4.2.4 The reresentation of references which will be made use of is described in section 4.2.5. This is followed by a review of the topic of reuse and equivalence classes (section 4.2.6) and a mathematical approach for calculating these (section 4.2.7). The quantification of reuse is described in the following section, and finally some limitations to the presented theories are given in section 4.2.9.

4.2.1 Data dependencies.

When performing different kinds of control transformations on a loop it is necessary to consider the inherent data dependencies that might exist in the loop's body. This is particularly important as some transformations might be constrained by dependences imposed by the references to memory made inside the loop. Neclecting to investigate the legality of applying a particular transformation can in the worst case result in an altering of the program.

When characterizing data dependencies a distinction between three different kinds of dependences is usually made. These different types are named true dependence, anti dependence and output dependence.

A true dependence exists when an instruction writes a value (to a memory location or a register) that is later read by another instruction. Using simple variables as the place to store values, an example of true dependence is given in the following piece of code :

```
........
S1:  a = c + 5;
S2:  b = a + 12;
........
```

In this case the instruction `S1` writes a value to the variable `a`, which is later read by the instruction `S2`. Therefore `S2` has a true dependence on `S1`, and every transformation that might cause `S2` to be executed before `S1` is illegal.

An anti dependence exists when a value is read from a storage location, which is later assigned a value by another instruction. Such a dependence would for instance be present in a piece of code that had the two instructions `S1` and `S2` interchanged :

```
........
S2:  b = a + 12;
S1:  a = c + 5;
........
```

, and `S2` must in this case always be executed before `S1`.
The last type of data dependence is denoted output dependence. This kind of dependence exists when two instructions write some value to a particular memory location. An example of such a situation is given in the following :

```
S3:  d = e + 20;
S4:  d = f + e + 5;
```

An important fact that should be stressed at this point is that for a dependence to exist between two instructions (or two references) at least one of them must be a write. As it has been shown this is the case for all the three types of data dependence just mentioned.

4.2.2 Dependences in loops.

When dealing with data dependencies in multi-level nested loops, vectors is a useful way to represent a particular iteration of such a loop. More specifically an iteration of an n level nested loop can be represented by a vector $\vec{p} = (p_1, p_2, p_3, ..., p_n)$ in which p_i is the loop index for the i'th loop in the nest. The loop indexes are usually ordered from outermost to innermost, e.i. p_1 always corresponds to the outermost loop index. To illustrate the use of such an index vector, consider the following loop :

```
for i=1 to I
    for j=1 to J
        for k=1 to K
            ........
```

In this loop the iteration in which the index variables have values `i`=2, `j`=7, `k`=3, will be represented by the index vector $(2, 7, 3)$.

The lexicographic order of the index vectors in a loop will determine the order in which the iterations are performed. That is, if p_2 is lexicographically greater than p_1, then the iteration corresponding to the index vector p_2 will be executed after the iteration corresponding to the index vector p_1. For instance $(4, 1, 1)$ is lexicographically greater than $(2, 7, 3)$ and is therefore executed after iteration $(2, 7, 3)$.

To find out if there exists any dependences in a nested loop, it is sufficient to determine whether it is possible for any iteration to write a value that is read or written by any other iteration. If for instance it turns out that in two different iterations p_1 and p_2 the same data element is accessed, and at least one of the accesses is a write, then the lexicographically greatest index vector of the two will have a data dependence on the other. As an illustrative example consider the following nested loop :

```
for i = 2 to N
    for j = 1 to N-1
        A[i][j] = A[i][j] + A[i-1][j+1];
```

In iteration $(1,3)$ the array element `A[1][3]` is written. This same data element is read in a later iteration namely $(2,2)$ in which the array reference `A[i-1][j+1]` accesses the very same data element `A[1][3]`. There is thus a dependence (a true dependence) from $(1,3)$ to $(2,2)$. Furthermore the dependence can be described by the so called dependence distance which is calculated as $(2,2) - (1,3) = (1,-1)$.

In this case the dependence distance actually describes the dependences for all the iterations, as for instance the iteration $(3,3)$ is also dependent on $(2,4)$, $(4,4)$ on $(3,5)$ and so on. Therefore the dependence distance is instead denoted as a distance vector. At this point it should be noted that a distance vector by definition is lexicographically positive. This can be seen as it was calculated as the lexicographically greatest of the index vectors minus the lexicographically smaller index vector.

To stress the fact that distance vectors describe dependences among iterations, not data elements, another example, this time involving a one-dimensional array, will be given in the following :

```
for i = 1 to N
    for j = 2 to N-1
        A[j] = A[j] + A[j-1] + A[j+1];
```

At first glance the following observations are made:

- The value written by `A[j]` in one iteration of the innermost loop is read by `A[j-1]` in the next. This dependence can be describes by the distance vector $(0,1)$.
- The data element read by `A[j+1]` in one iteration of the innermost loop is written to by `A[j]` in the next. This dependence can also be describes by the distance vector $(0,1)$.
- The data element written by `A[j]` in iteration (x,y), where x and y are integers within the iteration space boundaries, is read by `A[j+1]` in iteration $(x+1,y-1)$. This dependences can be described by the distance vector $(x+1,y-1)-(x,y) = (1,-1)$.
- The data element written to by `A[j]` in one iteration is read by `A[j]` (on the right hand side) in the next iteration of the i-loop. This dependence can be describes by the distance vector $(1,0)$.

These distance vectors definitely describe the inherent dependence in the nested loop, however even more is true. For each of the found distance vectors a replacement of the first element in the vector by another positive integer (within the iteration space boundaries) will also result in a valid distance vector for the loop. For instance the reference `A[j-1]` which gave rise to the first distance vector that was found $(0,1)$ also gives rise to the distance vector $(1,1)$ and $(2,1)$ and so on. In this case the dependences can better be described by a direction vector.

A direction vector is a vector that describes dependences in the same way as it is the case for a distance vector, but where only the signs of the distance vector elements matters. For an index vector $(q_1, q_2, q_3, ...q_n)$ representing a particular iteration that

is dependent on an other iteration $(p_1, p_2, p_3, ..., p_n)$ the direction vector describing the dependence is given by $(r_1, r_2, r_3, ..., r_n)$, where the elements are :

$$
\begin{array}{lll}
r_i = " < " & if & p_i < q_i \\
r_i = " = " & if & p_i = q_i \\
r_i = " > " & if & p_i > q_i
\end{array}
$$

In this way a possibly infinite range of distance vectors can be covered, and the former mentioned set of distance vectors created by the reference `A[j-1]` : $(0,1)$, $(1,1)$, $(2,1)$ .... can be described by $(=,<)$ for $(0,1)$ and $(<,<)$ for the rest $((1,1),(2,1)....)$. A direction vector is thus a way of representing a set of distance vectors as the elements “<“ and “>“ can represent any number of values in the sets of positive and negative natural numbers respectively. That is, “<“ actually corresponds to any possible subset of Z+ and “>“ corresponds to any possible subset of Z-.

Representing a dependence that can be characterized by a single distance vector by a direction vector instead of a distance vector is an example of how information about the specific dependence is lost. For instance the former mentioned distance vector (0,1) could be described by the more general (=,<), whereby the information that the dependence exists between two successive iterations of the innermost loop is lost. Loosing this kind of information is however not very important when the dependences are used to determine the legality of certain high level control transformations. In this case only the signs of the various elements in the distance vector matters and the dependence can just as well be described by a direction vector. This point will become more clear when the legality criteria for performing high level transformations such as interchange or tiling is discussed in a later section.

Sometimes the symbols "+", "0" and "-" is used instead of "<", "=" and ">" respectively. That is, "+" corresponds to "<", "0" corresponds to "=" and "-" corresponds to ">". Furthermore the symbols <=, >= or ∗ might also be used to denote combined ranges of distances. The symbol <= thereby denotes both ranges < and =. The symbol >= denotes ranges > and =. Lastly ∗ denotes all the ranges <, = and >.

Using these symbols the direction vectors characterizing dependences in a loop can be summarized in a smaller set of directoins vectors or even a single one. This can be illustrated by once again to considering the last example, in which the distance vectors (0,1) , (1,0) , (1,-1) and (1,1) was found. These distance vectors corresponds to the direction vectors (=,<) , (<,=) , (<,>) and (<,<). Summarizing all these vectors into one yields (<=,∗) , where each direction ri in the summarizing direction vector $(r_1, r_2, ..., r_n)$ corresponds to the union of ranges found at the same nesting depth in all the other direction vectors.

This summarizing direction vectors once again leads to the loss of some information about the actual dependences. When performing this summarizing step however, one should be careful not to accidently create a direction vector that represents a possible lexicographic negative direction vector. In this case this is exactly what has happened. The direction vector (<=,∗) actually represents a possible lexicographic negative direction vector as it for instance covers the direction vector (=,>). This is a very unfortunate effect and the summarizing of direction vectors should be avoided when this may lead to possible lexicographically negative direction vectors. The main reason for this is that

it might result in a following transformation legality analysis to produce wrong results. That is, the summarized direction vector might actually cover a dependence that does not really exist, but that prohibits a certain transformation from being legal.

For the direction vectors (=,<) , (<,=) , (<,>) and (<,<) a summarizing that covers only lexicographically positive direction vectors should instead be performed. This can be done by summarizing (=,<) , (<,=) and (<,<) into (<=,<=), while leaving (<,>) unchanged.

4.2.3 Legality criteria for transformations.

As it was former mentioned the dependence vectors of a particular loop can be used to determine the legality of carrying out different kinds of transformations on this loop. The effect on the dependence vectors of performing a transformation, and the here from derived criteria that can be stated, will be described by a simple example. This example involves the interchange of loops.

When iterations in a nested loop can be represented by vectors it is possible to represent transformations such as reversal, skewing and interchange as matrix transformations. When interchanging a nested loop of depth 2 the iteration described by the index vector (x, y) is mapped to the iteration (y, x). This mapping can be formulated in a matrix notation as :

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ x \end{pmatrix}$$

It is clear that the matrix performs a linear interchange transformation on the iteration space. Moreover, when an iteration (x_2, y_2) has a dependence on (x_1, y_1) in the original iteration space (the dependence vector is (x_2, y_2) - (x_1, y_1)), then the mapping of $(x_2, y_2) = v_2$ and $(x_1, y_1) = v_1$ will be given by $M \cdot v_2 = (y_2, x_2)$ and $M \cdot v_1 = (y_1, x_1)$. This means that in the transformed iteration space the dependence vector is now given by $(y_2, x_2) - (y_1, x_1) = M \cdot v_2 - M \cdot v_1 = M \cdot (v_2 - v_1)$, which shows that the matrix also maps dependence vectors into the new iteration space. Thus if v is a dependence vector in the original iteration space, then $M \cdot v$ is a dependence vector in the transformed iteration space. More specifically $v = (v_1, v_2)$ is mapped into :

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} v1 \\ v2 \end{pmatrix} = \begin{pmatrix} v2 \\ v1 \end{pmatrix}$$

This can be summed up as follows : Interchanging loops i and j in a nested loop yields changed dependences, which can be described by interchanging elements i and j in all the dependence vectors that describe dependences in the nested loop. Furthermore, if all the dependence vectors remain lexicographically positive after the transformation, then the interchange is legal.

As an example consider a nested loop with depth 3 and with dependences (<,=,>) and (<,>,>). In this loop it is legal to interchange the two innermost loops, while the outermost loop cannot be interchanged at all. If however, a dependence described by (=,<,>) also existed in the loop then no interchange what so ever could be performed.

The same principle that was described here applies to all unimodular transformations. That is, a unimodular transformation is legal if and only if all the dependence vectors remain lexicographically positive.

The neccesary condition that must be fulfilled for a particular set of loops in a nested loop to be tileable is, that the loops targeted for tiling must be fully permutable [54]. The definition of fully permutable loops is as follows :

Loops Ii through Ij in a nested loop are fully permutable (and tileable) if and only if for each dependence vector either $(d_1, d_2, ..., d_{i-1})$ is lexicographically positive or all elements $d_i, d_{i+1}, ..., d_j$ are non-negative. To make this important criterion more clear, an example will be presented in the following:

Consider a nested loop of depth 5, with the following dependence vector : $(=,=,=,<,=)$. In this nest it is possible to tile the two innermost loops (i.e. loops 4 and 5). The $(d_1, d_2, ..., d_{i-1})$ elements are not lexicographically positive as they equal $(=,=,=)$, but on the other hand the $d_i, d_{i+1}, ..., d_j$ elements (equal to $<$ and $=$) are both non-negative and the criterion is met. In fact all possible sets of adjacent loops in this nest are tileable.

If the dependence described by the dependence vector $(<,>,>,>,=)$ was also present in the loop, then every set of adjacent loops except the ones involving the outermost loop could be tiled.

4.2.4 Tiling aspects.

The use of tiling transformations have previously been described in section 2.2.2. In this description an example of a two-dimensional tiling was also included, and the benefits and incentives for performing tiling transformations was briefly discussed. As it was mentioned in section 2.2.2 the tiling of a loop can result in a better exploitation of the inherent reuse present in the nested loop.

Performing a tiling transformation of a nested loop improves its cache reuse by dividing the nests iteration space into tiles, and transforming the loop to iterate over the tiles, one at a time. As the processing of each tile involves a lot fewer interveaning accesses, data reuse can now occur in all the loops targeted for the tiling. This is of course provided that all the data used in the processing of a tile simultaneously can be held in the cache. An illustrative example showing how reuse can occur in different loops inside the nest will be presented using the following matrix multiplication algorithm :

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C[i][k] = A[i][j] * B[j][k];
```

In this naive matrix multiplication algorithm, each line of `C`-array elements that is fetched as a result of the reference `C[i][k]` has the potential to be reused in each iteration of the `j`-loop. The reuse can however only occur, if the large amount of interveaning accesses in the iteration of the `k`-loop don't flush out the `C` elements from the cache before it can be reused. In a similar manner each line of `B` array elements that is fetched as a result of the `B[j][k]` reference has the potential to be reused during the next iteration of the `i`-loop. The array references do however still exhibit some sort of reuse, as for instance `A[i][j]` exhibits self temporal reuse in the innermost loop (see section 2.4.5 for classification of reuses). Likewise the references `C[i][k]` and `B[j][k]` both exhibit spatial reuse in the innermost loop. At this point it should be made clear that for an non-tiled loop the only reuses that are exploited are those present in the innermost loop. One can also say that

the only reuses that are exploited are those that coinside with the localized vector space of a nested loop, and for a non-tiled loop the localized vector space consists only of the innermost loop. By performing a tiling transformation however, all the loops involved in this tiling becomes a part of the localized vector space, and it is thereby possible to exploit a greater part of the inherent reuse in a nested loop.

The potential reuses in the loop that are not exploited are besides the two already mentioned reuses of `C[i][k]` (temporal) and `B[j][k]` (temporal) in the `j`- and `i`- loop respectively, also the self spatial reuse of `A[i][j]` in the middle loop. It is thus clear that unexploited reuse exists in all of the three loops, and that a tiling of all the three loops therefore would be beneficial, as long as the working set of a tile computation can fit in the cache.

In general, the effect of letting a particular loop (`j`) in a nest be a part of the tile-transformed area is that a much smaller number of iterations in the inner loops will be performed before an iteration of the `j`-loop is performed again. This is a result of the dividing of the iteration space, and this will allow a potential reuse in loop `j` to be exploited as it hereby becomes a part of the localized vector space.

In the described example involving the matrix multiplication algorithm reuses turned out to be present in all of the three loops. A tiling involving all of the three loops would therefore be very beneficial as it would allow for the reuses in all loops to be exploited. One can also say that the entire iteration space (i.e. all three loops) has been localized, and that this allows for all potential reuses to be exploited. It is actually a well known fact that matrix multiplication code is an extremely well suited target for tiling transforms, as reuses exist in all loops and as the tiling of those is perfectly legal.

In other cases however, there might not exist reuse in all of the loops, and it might not be legal to perform a tiling or another kind of transformation in order to exploit those reuses that do exist. In such cases it is beneficial to perform transformations such as interchange that orders the loops in a way that places loops with reuses innermost in the nest, before an eventual tiling of the nest is carried out. All of the performed transformations should of course still be legal, and this is not always the case.

A tiling should always include at least the two innermost loops in a nest, to really make sense. That is, a one dimensional tiling of a loop alters nothing except for introducing overhead in terms of the extra for loop. Furthermore as the effect of performing a tiling transform is to localize the iteration space, a tiling that doesn't involve one or more of the innermost loops in a nest, seems unadvantageous. For instance a tiling of some subset of the loops that does not involve the innermost loop will not result in a further localization of the iteration space, as long as the bound in the innermost loop is relatively large. A tiling should thus always on some number of the innermost loops in the nest.

The tiling of the matrix multiplication algorithm previously presented could be performed as follows :

```
for i_tile = 1 to N step T
   for j_tile = 1 to N step T
      for k_tile = 1 to N step T
         for i = i_tile to min(i_tile + T - 1, N)
            for j = j_tile to min(j_tile + T - 1, N)
```

```
          for k = k_tile to min(k_tile + T - 1, N)
            C[i][k] = A[i][j] * B[j][k];
```

Where T is the chosen tile size which ensures that the working set will fit in the cache.

The first three loops iterate over all the tiles, while three innermost loops process the computations performed within each tile. The outermost loops which control the iterations over the tiles are sometimes referred to as the controlling loops. This group of loops are fully permutable as well as the group consisting of the innermost loops are. What this means is, that within each group of loops, the loops can be interchanged without violating any depedences. That is the order of the controlling loops (named by their index variables) are currently {`i_tile`, `j_tile`, `k_tile`}, and they could just as well be laid out in the order {`j_tile`, `i_tile`, `k_tile`} or for instance {`j_tile`, `k_tile`, `i_tile`}. If this last choice is made then the layout of the code would look as follows :

```
for j_tile = 1 to N step T
   for k_tile = 1 to N step T
      for i_tile = 1 to N step T
         for i = i_tile to min(i_tile + T - 1, N)
            for j = j_tile to min(j_tile + T - 1, N)
               for k = k_tile to min(k_tile + T - 1, N)
                  C[i][k] = A[i][j] * B[j][k];
```

In this fraction of code the two middle loops could just as well be merged into one without changing the order of execution in the loop. The merging of these two loops would produce the following code :

```
for i_tile = 1 to N step T
   for j_tile = 1 to N step T
      for i = 1 to N
         for j = j_tile to min(j_tile + T - 1, N)
            for k = k_tile to min(k_tile + T - 1, N)
               C[i][k] = A[i][j] * B[j][k];
```

This form of merging loops together after tiling is commonly known as coalescing. As just mentioned applying coalescing to a tiled loop doesn't change the iteration order at all, and the coalesced loop still remain a part of the localized vector space.

4.2.5 Representation of references.

As it has been demonstrated in the previous section, unimodular transformations and tiling can be used to alter the localized iteration space in a nested loop, ultimately resulting in a better exploitation of reuse. This improvement in performance is obtained when the applied transformations succeeds in making the localized iteration space overlap with the inherent reuse in the nest. To be able to methodize the execution of such transformations, a technique for finding the potential reuse in a nested loop is needed. That is, it is necessary to be able to determine which references do exhibit some sort of reuse, and in which loop this reuse occurs (i.e. in what dimension of the iteration space). Furthermore it is also necessary to be able to quantify this reuse in order to make good

strategic decisions pertaining to the actual transformations that should be applied. That is, some sort of evaluation of which transformations would be most beneficial should be carried out. To be able to do this in an efficient and clear way a suitable representation of the array references in a nested loop must be used. This representation can then be used to establish what kinds of reuse the different references exhibit and in which dimensions this reuse occurs.

This section will provide a description of a suitable representation of array references in nested loops, and a further description of how this representation can be used in calculating a way to exploit the reuse in the best possible manner. A description of a useful representation of array references will be given in the following. The representation needs to carry information of both the subscripts of arrays as well as the the loop-positions of the index variables that are used in the subscripts. This is accomplished by letting each reference be represented by a function which maps the index vector of the nested loop to a vector containing all the subscript expressions in the reference. How this works is best illustrated by an example. Consider the following nested loop :

```
for i1 = 1 to N
    for i2 = 1 to N
        for i3 = 1 to N
            A[i1][i2+1] = B[i3 -1][i2];
```

The function representing the reference `A[i1][i2+1]` is given by $f(i) = H \cdot \vec{i} + \vec{c}$, where $\vec{i}$ is the index vector, and where H is the matrix :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

, and $\vec{c}$ is the constant vector (0,1).

It is thus clear that f(i) equals the vector (i1, i2+1) which contains exactly the index expressions of the reference `A[i1][i2+1]`. The matrix stores the positions of the index variables for each indexation of the array and the vector $\vec{c}$ stores the constant terms. One should notice that the matrix has dimensions $(loop - depth) \times (array - dimension)$.

In a similar way the H and c belonging to the `B[i3 -1][i2]` reference can be identified to be :

$$\mathbf{H} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad \mathbf{c} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

The reason for representing the references by this matrix and vector notation is that it makes it possible to calculate both reuse dimensions as well as equivalence classes by the means of simple matrix and vector operations. This will become apparent later in this section, when it is described how this is done.

For the presented example one additional comment must be made. The reference `A[i1][i2+1]` exhibits reuse in the i_3-loop and is therefore said to have reuse in the (0,0,1) direction of the iteration space. In general a reference that exhibits reuse in loop i_x in an n-level nested loop, is said to exhibit reuse in the $(i_1, i_2, i_3, ..., i_x, ..., i_{n-1}, i_n)$ dimension, where i_x equals 1 and the rest of the elements equal 0. This is a way of using vector spaces to represent the directions in which reuse is found, and these directions are the ones that are advantegeous to include in the localized iteration space.

4.2.6 Determination of reuse and equivalence classes.

When analyzing a nested loop for potential reuses a division of the different reuse-types into separate equivalence classes is prefereable. The references are divided into equivalence classes based on the same criteria that was described in section 2.4.5. The main reason for performing this partitioning of references is that the estimation of the number of off-chip accesses due to a certain set of references is determined by the number and the types of equivalence classes that the references can be divided into. An equivalence class can consist of one or more array references, and all the references belonging to the same class, must of course reference the same array. The meaning of an equivalence class which contains more than one reference is that some sort of reuse exists among these references in the current loop. If the class only contains one reference then some sort of reuse exists between the accesses to this array element in different iterations. Furthermore an equivalence class is defined to contain one or more references that exhibit a particular type of reuse. That is a specific type of reuse is associated with each equivalence class in a nested loop. The types of reuses that can occur has also been presented in section 2.4.5 and their definitions will be listed here for convienience :

Self- temporal : A memory reference accesses the same data location in different loop iterations

Self- spatial : A memory reference accesses the same cache line in different loop iterations

Group- temporal : More than one reference accesses the same data location in different iterations

Group- spatial : More than one reference accesses the same cache line in different iterations

From these definitions it should be clear that an equivalence class consisting of more that one reference only can be of one of the two types : Group-temporal or Group-spatial. Likewise an equivalence class consisting of a single reference can only be of one of the the two types : Self-temporal or Self-spatial. It should also be noted that self-temporal reuse implies self-spatial reuse and that group-temporal reuse implies group-spatial reuse (a reference to the same data element is also a reference to the same cache line).

The explanation of how the partitioning of references into equivalence classes should be done will be illustrated by the use of an example. The same code that was used in section 2.4.5 is very well suited for this purpose and will therefore also be used here. There exists however at this point some different aspects of how the partitioning of references should be carried out. This is the case as the use of tiling provides the opportunity of extending the localized vector space to also include others than just the innermost loop. This is important because the localization of the iteration space by the use of tiling may alter the original equivalence classes calculated from the original loop.

To provide the best possible description of how this affects the steps that should be taken, it will first be described how the equivalence classes can be derived from the original code, where no tiling is intended. The code looks as follows :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        A[i][j] = A[i][j-1] + A[i][j] + A[i][j+1] +
```

```
A[i-1][j] + A[i+1][j] + B[i] + C[j][i]
```

The partitioning of these references into equivalence classes resulted in the generation of the following classes, where all references belonging to the same class are placed inside the braces "{" and "}". The equivalence classes are listed according to which type of reuse they exhibit :

No reuse : `C[j][i]`
Self-temporal reuse : {`B[i]`}
Self-spatial reuse : {`A[i-1][j]`} {`A[i+1][j]`}
Group-temporal reuse : {`A[i][j-1]`, `A[i][j]`, `A[i][j+1]`}
Group-spatial reuse : none

Consider as a first example the reference `C[j][i]` which exhibits no reuse in the current loop. This is the case as the innermost loop iterates over the index variable `j`, and this in turn results in the accesses to the `C`-array (via the `C[j][i]` reference) to be in stride `M`. Therefore no spatial reuse can exist and no temporal reuse can exist either as each element of the `C`-array are only accessed once in the entire nest. This argumentation for proving that no reuse occurs for the `C[j][i]` reference is perfectly true for the way the iteration space of the loop is currently traversed. When dealing with the possibilities of extending the localized vector space however, additional care must be taken when the references are grouped into equivalence classes.

It is actually the case that the `C[j][i]` reference exhibits spatial reuse in the `i`-loop. This is a fact as each line brought in to the cache while accessing the `C`-array in the iteration of the `j`-loop can be reused in the next iteration of the `i`-loop, if the interveaning accesses has not flushed out the `C`-elements from the cache. As the loop is constructed currently the accessed `C` lines are however, very likely to have been flushed out of the cache at the time of the next `i`-loop iteration. Therefore the spatial reuse that the `C[j][i]` reference exhibits in the `i`-loop is said not to be exploited and the reference is not assigned to any equivalence class. If the i-loop however, was a part of the localized vector space, then the spatial reuse that the `C[j][i]` reference exhibits in the `i`-loop could be exploited. Alternatively this could be expressed as : The `C[j][i]` reference exhibits spatial reuse in the (1,0) dimension but as the (1,0) dimension currently is not a part of the localized vector space, the reuse is not exploited.

It is thus clear that when dealing with the investigation of potential reuses in a nested loop, all reuses in any dimension should be considered, as they might be exploited by transforming the iteration space. A tiling that involves both the `i`- and the `j`-loop in the current example would for instance allow the spatial reuse of the `C[j][i]` reference in the (1,0) dimension to be exploited. The localized vector space would then consist of the dimensions (1,0) and (0,1).

The above listed equivalence classes are thus a collection of the reuses that exist in the innermost loop, which equals the localized iteration space in the untransformed code. What is needed when one wants to analyze the nested loop for possible benefits from a tiling transformation is information about reuse in all dimensions. In a similar manner as it has been done for the `C`-array reference the `B[i]` reference can now be investigated for reuse in other dimensions. This reference also exhibits spatial reuse in the `i`-loop, which also is not exploited in the current loop.

Likewise additional reuse among the rest of the references can also be seen to exist. The rest of the references are currently divided into the three equivalence classes : {`A[i-1][j]`}, {`A[i+1][j]`} and {`A[i][j-1]`, `A[i][j]`, `A[i][j+1]`}. Apart from the reuse that have been identified to exist in the innermost loop for these equivalence classes, internal reuse actually also occurs. More specifically, temporal reuse exists among references in these three equivalence classes in the `i`-loop.

Thus, if the nested loop was to be tiled in both dimensions the three mentioned equivalence classes would instead be replaced by a single one containing all the references of the three classes.

As it was briefly mentioned at an earlier point in this chapter the generation of equivalence classes is used to estimate the number of off-chip accesses in a loop. The ability to perform such an estimation is very valueable as it allows for a precise evaluation of which kinds of transformations are most suitable. In this context the transformations that are considered actually consists of more than one transformation. I.e. what really is measured is the performance of different sets of transformations. E.g. a skewing transformation might be applied to a nested loop in order to enable a tiling of some of the loops in that nest. Several interchange transformations might also be applied to a nest in order to place the potential reuses in the innermost loops.

For such a number of different sets of transformations the estimated number of off-chip accesses per iteration of a loop can hereafter be calculated. This calculation is based on the number and types of the different equivalence classes, that are present in the nested loop. I.e. the partitioning of references into equivalence classes is absolutely essential for estimating the number of off-chip accesses in a nested loop. Hence this partitioning is also essential for the ability to evaluate the effect of transformations, and it provides a useful framework to investigate a number of different transformations. After such an investigation has been performed the transformation which results in the minimum number of off-chip accesses can then be chosen.

In order to reduce the search space, of finding a near optimal transformation of a nested loop additional analysis of the loop in question should also be performed. That is, an exhaustive search algorithm that estimates the performance of all the legal combinations of unimodular and/or tiling transformations on a nest, would be very inefficient. Therefore the information about reuses that have to be gathered in order to perform the partitioning of references into equivalence classes should also be used to reduce the search space at this point. If for instance no reuse occurs in some number of the innermost loops, then there is no point in calculating equivalence classes and estimating performance for the tiling of these loops. Instead a more analytical approach should be applied.

4.2.7 Reuse and equivalence classes - a mathematical approach.

In this section a mathematical approach for calculating reuses will be presented. This approach uses the previously described matrix notation for array accesses in nested loops.

Once again the code earlier used for the description of reuse across multiple iterations will be used to illustrate this mathematical approach for calculating reuses. The code is repeated here for convienience :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        A[i][j] = A[i][j-1] + A[i][j] + A[i][j+1] +
                  A[i-1][j] + A[i+1][j] + B[i] + C[j][i]
```

No reuse : `C[j][i]`
Self-temporal reuse : {`B[i]`}
Self-spatial reuse : {`A[i-1][j]`} {`A[i+1][j]`}
Group-temporal reuse : {`A[i][j-1]`, `A[i][j]`, `A[i][j+1]`}
Group-spatial reuse : none

The presentation for the different kinds of reuses will proceed in the same order as it has been done previously. Starting with an instance of self-temporal reuse.

Self-temporal.
When a reference exhibits self-temporal reuse, the same data element is accessed in different iterations. These different iterations can be represented by two arbitrary index vectors $\vec{i}_1$ and $\vec{i}_2$. If the reference furthermore is represented by the access marix $\vec{H}$ and the constant vector $\vec{c}$, the criterion for temporal reuse between i_1 and i_2 is that $\vec{H} \cdot \vec{i}_1 + \vec{c} = H \cdot \vec{i}_2 + \vec{c}$. This can also be written as $H \cdot (\vec{i}_1 - \vec{i}_2) = \vec{0}$. When this is the case the reuse occurs in direction $(\vec{i}_1 - \vec{i}_2)$. The reuse will however, only be exploited when this dimension is a part of the localized iteration space in the nested loop. Letting the vector $\vec{v}$ denote the distance between the two arbitrary iterations $\vec{i}_1$ and $\vec{i}_2$, the question of whether a reference represented by H and $\vec{c}$ exhibits temporal reuse can be answered by solving the equation $H \cdot \vec{v} = \vec{0}$ (i.e. independently of $\vec{c}$). Alternatively this equation can also be written as kernel(H). To illustrate this with a concrete example consider the reference `B[i]` which earlier was determined to exhibit self-temporal reuse in the `j`-loop, simply by argumenting.

The reference `B[i]` can be presented by the matrix $H = \begin{pmatrix} 1 & 0 \end{pmatrix}$, and the solution to kernel(H) is thus : span{(0,1)}.

Reuse in the (0,1) direction corresponds of course to reuse in the `j`-loop for the current nest and the result can thus be identified to concur with the previous argumentation.

It should be noted that reuse can occur in more than one direction, which can make the benefits of tiling even greater. Consider for instance the same nest with an additional loop (a `k`-loop) placed inside the `j`-loop :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        for k=0, k<M, k++
            A[i][j] = A[i][j-1] + A[i][j] + A[i][j+1] +
                      A[i-1][j] + A[i+1][j] + B[i] + C[j][i]
```

The reference `B[i]` would in this case be described by $H = [100]$, and the solution to kernel(H) is : span{(0,1,0),(0,0,1)}. In such a loop the `B[i]` reference would thus exhibit temporal reuse in both the `j`- and the `k`-loop.

Self-spatial.

Self-spatial reuse for a reference can be calculated in a very similar manner as it was done for self-temporal reuse. However, as all accesses to the same row of an array has a potential for exploiting spatial reuse, it is of lesser importance how the indexing in this dimension of the array is done. more specifically, for an arbitrary array reference like : `A[i1][i1]....[in]`, where the dimension of the array is `n`, the index expression `in` does not matter as long as the strides in the innermost loop is less than the line size. For instance in :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        A[i][j] = A[i][j*8];
```

In this loop no spatial reuse could occur for the reference `A[i][j*8]` if the line size was greater than or equal to $8 \cdot (element - size)$. If this is not the case, then spatial reuse will occur in the innermost loop. The degree of reuse will of course depend on the stride, which in this example is 8. In the following it will be assumed that all strides are of a magnitude which allows for spatial reuse.

The fact that the last index expression does not matter when examining for self-spatial reuse is reflected in the access matrix used for this purpose. As the last row of the access matrix describes the last index expression of the array reference, this row should be replaced by a row of all zero's when self-spatial reuse is calculated. Hereafter the process is exactly identical to the one described for self-temporal reuse. The task of examining a reference, represented by the access matrix H, for self-spatial reuse in a nested loop thus consists of replacing the last row in H by all zero's (thereby generating H') and solving kernel(H'). Considering once again the reference `B[i]` in the following nested loop :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        for k=0, k<M, k++
            A[i][j] = A[i][j-1] + A[i][j] + A[i][j+1] +
                      A[i-1][j] + A[i+1][j] + B[i] + C[j][i]
```

`B[i]` can be represented by $H = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$. Replacing the last row of H by a row of all zero's yields $H' = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$. The solution to kernel(H') is : span{(1,0,0),(0,1,0),(0,0,1)}, and the reference thus exhibits self-spatial reuse in all dimensions. The fact that self-spatial reuse occurs in the two innermost loops should come as no surprise, as it already has been established that `B[i]` exhibits self-temporal reuse in those two loops. Because self-temporal reuse also implies self-spatial reuse the reason for the inclusion of the directions (0,1,0) and (0,0,1) in the solution to kernel(H') is obvious. The reason that `B[i]` also exhibits self-spatial reuse in the outermost loop is also fairly obvious.

As another example consider any of the references to the `A`-array in the given loop. These all have the same access matrix H. The question of whether any of them exhibits self-spatial reuse can thus be answered by just one calculation. for these references the access matrix H is given by :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

And H' thus becomes :

$$\mathbf{H'} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The solution to kernel(H') is span(0,1,0)(0,0,1), and these directions are thus the ones in which any of the `A`-array references exhibits self-spatial reuse. As we already know, group-temporal reuse actually also exists among these references.

Group-temporal reuse.
When examining a set of references for group-temporal behaviour, only sets consisting of references with the same access matrix H need be considered. This is the same as saying that only uniformly generated sets possess a possibility for group-temporal reuse to be exploited. Thus a uniformly generated set is a set of references with identical access matrices. Two references that belong to the same uniformly generated set will exhibit group-temporal reuse within a certain localized vector space if they access the same data element inside this localized vector space. If the two references are represented by the access matrix H and the two constant vectors c_1 and c_2, and furthermore the localized vector space is denoted by L, then the condition for group-temporal reuse to occur can be formulated as :

$$\exists \vec{r} \in L : \qquad H \cdot r = \vec{c}_1 - \vec{c}_2$$

If a particular solution $\vec{r}_p$ to this equation exists then the general solution will be $\vec{r}_p$ + kernel(H). When solving this equation it is necessary to make sure that the two constant vectors $\vec{c}_1$ and $\vec{c}_2$ are not equal. That is, if the two references were identical or if the testing accidentially was performed on the same reference. This can also be formulated by the condition :

$$((span\{\vec{r}_p\} + kernel(H)) \cap L) \neq (kernel(H) \cap L)$$

Considering once again the loop :

```
for i=0, i<M, i++
    for j=0, j<M, j++
        for k=0, k<M, k++
            A[i][j] = A[i][j-1] + A[i][j] + A[i][j+1] +
                      A[i-1][j] + A[i+1][j] + B[i] + C[j][i]
```

The references `A[i][j-1]`, `A[i][j]` and `A[i][j+1]` have earlier been identified to belong to the same equivalence class, which exhibits group-temporal reuse. These references can be represented by the access matrix H :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

, and the constant vectors (0,-1), (0,0), (0,1) for references `A[i][j-1]`, `A[i][j]` and `A[i][j+1]` respectively.

Starting with references `A[i][j-1]` and `A[i][j]` a particular solution to the equation $H \cdot \vec{r} = (0, -1) - (0, 0)$ is (0,-1,0) and the general solution to this equation is therefore (0,-1,0) + ker(H). Since this solution is a part of the iteration space span{(0,1,0),(0,0,1)} the

group-temporal reuse between the references `A[i][j-1]` and `A[i][j]` can be exploited if these two directions are included in the localized vector space. In a similar manner the reference `A[i][j+1]` can likewise be shown to have group-temporal reuse with both of these references in the same two directions.

As it previously has been mentioned a localization of the entire iteration space in the given nest, would result in an equivalence class consisting of all the references : `A[i][j-1]`, `A[i][j]`, `A[i][j+1]`, `A[i-1][j]`, `A[i+1][j]`. This equivalence class has as it is shown also the two references `A[i-1][j]` and `A[i+1][j]` and the class exhibits group-temporal reuse. How this conclusion can be reached will be illustrated by showing that the references `A[i-1][j]` and `A[i][j+1]` can exhibit group-temporal reuse between them. The constant vectors representing `A[i-1][j]` and `A[i][j+1]` are respectively : (-1,0) and (0,1). A particular solution to the equation $H \cdot \vec{r} = (-1, 0) - (0, 1)$ is (-1,-1,0) and the general solution is therefore (-1,-1,0) + kernel(H). This solution is a part of the iteration space spanned by (1,0,0),(0,1,0),(0,0,1) and group-temporal reuse can thus be exploited if the localized vector space includes all of these three directions.

The calculations needed for determining if group-spatial reuse exists among a set of references is very similar to the calculations performed in the case of group-temporal reuse. To determine the directions in which group-spatial reuse occurs for two references with the access matrix H and constant vectors $\vec{c}_1$ and $\vec{c}_2$, the last row of H should be replaced by zero's and the last element in both $\vec{c}_1$ and $\vec{c}_2$ should be replaced with a zero. Hereafter the steps for determining the directions of reuse are exactly identical to the group-temporal case which has just been described.

4.2.8 Quantifying reuse.

In the preceeding sections the steps involved in the partitioning of array-references into equivalence classes have been described. This partitioning is an important task to perform when an evaluation of the the degree of reuse, and hence the number of off-chip accesses, in a nest needs to be established. This is the case as the information obtained by the partitioning steps constitutes a good basis for performing calculations pertaining to cache hits/misses. The ability to perform efficient and accurate estimations on the expected number of off-chip accesses is in turn very valueable when dealing with the problem of choosing the best transformation for a particular nest. The estimated number of off-chip accesses can thus be used as a metric for finding the most beneficial transformation.

When dealing with estimating the number of off-chip accesses for a particular localized iteration space, the references that have been put in the same equivalence class are treated together, i.e. as a single reference. This is done as the references belonging to the same equivalence class touches the same cache lines, or maybe even the same locations during the processing of a tile. One might thus say that the performance gains which are obtained because different references, in the same equivalence class, access the same cache lines are expressed in the estimation calculations in the way that these references are treated as just one. Here it is also assumed that the localization of the iteration space which lead to some particular equivalence class being formed, ensures that all the cache lines which are accessed in the processing of one tile can fit in the cache. Of course this is not always the case as conflict misses might occur internally between the elements

accessed in a tile, but in the process of estimating cache misses it is assumed that this situation does not arise.

The maximum number of off-chip accesses that can be caused by the references inside a single equivalence class can be calculated as the product of the number of iterations for each loop enclosing the reference(s). The presence of spatial and/or temporal reuse might however reduce the actual number of off-chip accesses significantly. If the spatial reuse for a particular equivalence class is exploited it can thus reduce the number of off-chip accesses by a factor equal to the line-size (l), if the accesses are stride-1. When spatial reuse exists, the degree to which it is exploited will however also depend on the size of the data-elements accessed as well as the stride in the accesses. In general, if the stride is k, and the data-element size is d, then the number of off-chip accesses can be reduced by the factor $l/(d \cdot k)$. Similarly the presence of temporal reuse in a particular dimension in an equivalence class can reduce the number of off-chip accesses by a factor equal to the tile size (S), for that particular dimension. In general, if the number of loops in which an equivalence class exhibits self-temporal reuse is denoted i, then the number of off-chip accesses due to this equivalence class can be reduced by the factor S^i, if the loops are included in the localized iteration space (and these loops all have tile-size = S).

When estimating the number of off-chip accesses for a particular nest, these rules can be applied. Care must though be taken in order to avoid making incorrect calculations by including the reduction factors of both temporal and spatial reuse, when the temporal reuse is just a special case of spatial reuse. How this situation can arise will become apparent when the formula for estimating the number of off-chip accesses is presented in the following.

The general formula for estimating the number of off-chip accesses per iteration of a loop due a single equivalence class is :

$$\frac{\#accesses}{iteration} = \frac{1}{(\frac{l}{element-size \cdot stride})^{L_{Rst} \neq L_{Rss}} \cdot S^{dim(L_{Rst})}}$$

The meaning of the symbols in this equation are :

l : The line size.
stride : The difference in the subscript expressions between two successive iterations of the loop within which the reference is placed.
$L_{Rst} \neq L_{Rss}$: An expression which equals 1 if $(L \cap R_{st}) \neq (L \cap R_{ss})$, otherwise 0.
$dim(L_{R_{st}})$: The number of dimensions spanned by the intersection of L and R_{st}.
S : The tile-size.

In order to illustrate the use and the legality of the above formula, it will in the following be applied to a series of examples. For simplicity an equivalence class containing just a single reference will be used as a first example. The approach is however exactly identical for equivalence classes containing more than one reference.
Consider the previously presented nested loop :

```
for i1=0, i1<N1, i1++
    for i2=0, i2<N2, i2++
        for i3=0, i3<N3, i3++
```

```
A[i1][i2] = A[i1][i2-1] + A[i1][i2] + A[i1][i2+1] +
            A[i1-1][i2] + A[i1+1][i2] + B[i1] +
            C[i1][2*i2] + D[i1][i2][5]
```

In this loop the reference `B[i1]` has an access matrix H, which equals : $H = [1\ 0\ 0]$. The reference exhibits self-temporal reuse in the dimensions spanned by (0,1,0) and (0,0,1), and self-spatial reuse in dimension (1,0,0). Furthermore, let the localized iteration space consist of all three loops (i.e. a tiling of all three loops is considered), and let the line-size for the cache be denoted by l. As $L \cap R_{ss}$ does not equal $L \cap R_{st}$, the expression $L_{Rst} \neq L_{Rss}$ will be set to equal 1 in the formula. This can be interpreted in the way that the self-spatial reuse present in L is not a case of self-temporal reuse, and thus a performance gain of $l/(element_size \cdot stride)$ is obtained due to this localization of the iteration space.

As the intersection of R_{st} and L spans a total of two dimensions, namely (0,1,0) and (0,0,1) the expression $dim(L_{Rst})$ will be set to equal 2 in the above formula. The self-temporal reuse present in the localized iteration space will thus yield a performance gain of S^2 in this case.

Putting these results together the estimated number of off-chip accesses per iteration of the inermost loop due to the equivalence class {`B[i1]`} can be calculated to be :

$$\frac{\#accesses}{iteration} = \frac{1}{\frac{l}{element-size \cdot stride} \cdot S^2}$$

As the instructions in the innermost loop are executed a total of $N1 \cdot N2 \cdot N3$ times, during the processing of the entire nest, the estimated total number of off-chip accesses due to this reference will be :

$$\frac{N1 \cdot N2 \cdot N3}{\frac{l}{element-size \cdot stride} \cdot S^2}.$$

Provided that one wanted to include all three loops in the tiling transformation (i.e. L consists of all three loops), the total number of off-chip accesses during the processing of the entire nest could be estimated by performing the same calculations for all the other uniformly generated sets in the nest and adding the results together. If a uniformly generated set, for some particular choice of L, consists of more than one equivalence class, then the estimate obtained by the formula can simply be multiplied by the number of equivalence classes to cover the entire uniformly generated set. That is, once a particular size of L has been chosen, the above formula need only be calculated once for the particular uniformly generated set in question. The number of generated equivalence classes that the choice of L leads to can then be multiplied by off-chip-acc./iteration, to obtain the estimated number of off-chip accesses per iteration, due to all the references belonging to the uniformly generated set. This entity is thus :

$$\frac{\#accesses}{iteration} = \frac{\#eq - classes}{(\frac{L}{element-size \cdot stride})^{L_{Rst} \neq L_{Rss}} \cdot S^{dim(L_{Rst})}}$$

Before proceeding with another example an additional comment regarding the stride (k), as it has been defined in this context, must be made. The stride equals as it appears in

the formula the distance in terms of data-elements between two data-elements accessed in two successive iterations of the loop in which the self-spatial reuse occurs. What is needed is however a metric for determing the distance between two successive accesses relative to the line-size. Therefore the stride is multiplied by the size of the data-elements of the array, in order to determine the degree of self-spatial reuse by dividing the line-size by this entity. An alternative approach would be to define the stride in terms of bytes, instead of data-elements, as it has been done here.

The stride as it has been defined here thus only depends on two factors. Namely the step-size of the particular loop in which the self-spatial reuse occurs, and the factor of the loops index variable in the last subscript expression in the reference. As an example consider the reference `C[i1][2*i2]` in the previous example. The stride for this reference is currently 2 as the factor of the index variable is 2 and the step-size of the i_2-loop in which this reference exhibits self-spatial reuse is 1.

The reason for the special definition of the parameter $L_{Rst} \neq L_{Rss}$, which determines the degree of self-spatial reuse that is exploited (if any) will in the following be briefly described.

In the case that $L \cap R_{st}$ does equal $L \cap R_{ss}$ the spatial reuse is in fact a special case of self-temporal reuse, and the potential performance gain from this reuse is expressed in formula as the factor $S^{dim(L_{Rst})}$. Therefore there exists no additional benefits from this spatial reuse and the $(L_{Rst} \neq L_{Rss})$ parameter is set to 1. As an example of this consider the reference `D[i1][i2][5]` in the previous nest. This reference exhibits self-spatial reuse in loop 3, but this reuse is actually also self-temporal and the estimated number of off-chip accesses per iteration for this references is calculated as :

$$1/(S^{dim(L_{Rst})})$$

In the case that $L \cap R_{st}$ does not equal $L \cap R_{ss}$, the reuse is self-spatial but not self-temporal and an estimated performance gain should be calculated as $l/(element - size \cdot stride)$ as previously discussed. For this "reduction factor" to be a valid measure it must however not be less than 1, as this would imply that more than one cache line needs to be fetched for a reference to a single data-element. This situation might occur when the stride and element-size are relatively large compared to the line-size. To cope with this situation the assumption that no data-element will occupy more than a single cache line is made, and the entity $l/(element - size \cdot stride)$ should be set to 1 if it originally turned out to be less than 1.

As the the approach for calculating the number of off-chip accesses per iteration of a loop now has been established, the additional steps for calculating a beneficial tile-size is fairly straight-forward. A tile-size which makes the working set fit in the cache should be chosen. I.e. the number of different memory locations touched during the processing of a tile should fit in the cache. The size of the working set can be obtained by multiplying the estimated number of off-chip accesses per iteration by $S^{dim(tile)} \cdot l$. In this expression $s^{dim(tile)}$ equals the number of iterations executed during the processing of each tile, and l equals the line-size. By solving this equation for the largest possible tile-size, a beneficial tiling can thereby be obtained.

4.2.9 Limitations.

In this section some special cases of reuse that have not been dealt with so far will be discussed. These cases of reuse identifies some shortcomings in the evaluation techniques, that have been presented in section 4.2.7. Furthermore some problems which might arise when performing a partitioning of uniformly generated sets into equivalence classes are pointed out. An introduction to the subject will be given in the following. The individual and more specific cases are handled in the subsequently.

Introduction.

In the preceeding two subsections the investigation and evaluation of reuse has been conducted while only considering the prescence of reuse in single independent dimensions. That is, in the presented examples the kernel's of the corresponding access matrices have only consisted of vector spaces spanned by vectors with only 1 non-zero element. It might however also occur that the kernel of an access matrix only has vectors (directions) which contain more than one non-zero element. As an example consider the following access matrix, which could represent an array reference `A[i1][i2+i3][i2+i3]` in a three level nested loop with index variables `i1`, `i2` and `i3` respectively :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

The kernel of this matrix equals $k \cdot (0, 1, -1)$, and this vector contains more than one non-zero element. The kernel of the same matrix with the last row replaced by zero's yield the same vector space and it can thus be concluded that this reference does not exhibit any kind of reuse in a single independent dimension of the iteration space. It does however exhibit self-temporal reuse across the dimensions (0,1,0) and (0,0,1). This can be realized by interpreting the kernel as a distance vector. For any iteration represented by the vector (x_1, x_2, x_3) where x_1, x_2 and x_3 denotes a particular iteration value for the three index variables i_1, i_2 and i_3 respectively the element accessed in this iteration will also be accessed in the iteration represented by $(x_1, x_2, x_3) + (0, 1, -1) = (x_1, x_2 + 1, x_3 - 1)$. Reuse will naturally also occur in iterations at distances of (0,2,-2), (0,3,-3), ... and so on, as long as the sum of the iteration vector and the distance vector represents an iteration which remains inside the loop bounds. One might say that reuse occurs in the direction (0,1,-1) of the iteration space.

This kind of reuse can arise when the array references contain subscript expressions in which more than one of the index variables of the nest occurs, and such cases have not been handled so far. As no litterature that covers this subject has been found, no general terms which describe these different kinds of reuse is known. Therefore the notations **"single dimensional"**- and **"multiple dimensional"** - reuse is adopted to describe these kinds of reuse. The former describe kernels with vector spaces that can be described by spans of vectors with only 1 non-zero element. The latter describe kernels with vector spaces that only can be described by spans of vectors with more than 1 non-zero element.

In the presented example the self-temporal reuse that occurs in direction (0,1,-1) can be exploited if the two innermost loops are included in the localized iteration space. The

degree of reuse that is actually present is however a more complex issue, and how this kind of reuse should be evaluated has not been answered yet. If it is assumed that all the elements accessed during the processing of one complete iteration of the i-loop can fit in the cache, then the two innermost loops in the nest can be considered a part of the localized iteration space. If furthermore the loop bounds are denoted N_1, N_2 and N_3 and the stepsize is 1, then a good estimate on the number of off-chip accesses during the processing of the entire nest would be $N_1 \cdot (N_2 + N_3)$. This estimate could not have been obtained by the evaluation techniques described in section 4.2.8, and an extended approach for evaluating this kind of reuse would therefore be preferable.

The problems and considerations involved in identifying and evaluating this kind of multiple dimensional reuse is discussed in the following section. As earlier mentioned the access patterns of array references that lead to multiple dimensional reuse (as just illustrated) can also result in a complication of the partitioning of uniformly generated sets into equivalence classes. This topic is discussed in the subsection denoted “Equivalence class generation”, which appears later in this section.

Multiple dimensional reuse.

As described in the previous subsection the evaluation and quantification of multiple dimensional reuse cannot be carried out by the previously presented methods. This section will contain a discussion and some suggestions for quantifying this kind of reuse. The discussion will be based on a few examples, and the emphasis is set on evaluating the degree of reuse based on these cases.

It is difficult to speak about the frequency of multiple dimensional reuse in common real world applications. There might exist several types of applications that that does not contain any reference patterns which for some localized iteration space exhibits multiple dimensional reuse. Clearly there is no need to be able to identify and evaluate this type of reuse if it almost never occurs. It has however been found to exist in some of the benchmarks used for this project. So even though it might not occur very often it still exists in some applications, and it might even in some cases have a significant effect on both the selection of the localized iteration space as well as the selection of tile sizes. The investigation and evaluation of multiple dimensional reuse should therefore not be completely neglected, in a data locality optimizing algorithm. At the very least, a compiler researcher should be aware of the important distinction between multiple- and single- dimensional reuse.

The examples which are used to illustrate the occurrences of multiple dimensional reuse in this section are, as it is the case for most of the other code fractions in this report, only used to illustrate the principles of the current topic, and they have thus no actual semantic meaning. An example of a fraction of code in which a reference exhibited multiple dimensional reuse has already been given in the previous section. In order to make to make the discussion more general and to point out some more factors pertaining to the investigation of multiple dimensional reuse one should consider, another example is used here. This code fraction might seem even more meaningless than the previous one presented, but it is very useful in this discussion. The code looks as follows :

```
for i1=0, i1<N1, i1++
    for i2=0, i2<N2, i2++
```

```
    for i3=0, i3<N3, i3++
        A[i1+i2+i3] [i1+i2+i3] [i1+i2+i3] =  ....
```

As the analysis of the degree of reuse that is present in this loop will be concentrated on the shown reference to the A-array no other references are included in this loop. The constant vector for the reference is the null-vector and the access matrix for the reference is :

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

The kernel for the access matrix can easily be identified to be expressed by the vector space spanned by : $k \cdot (1, -1, 0)$ and $l \cdot (0, 1, -1)$, where k and l are arbitrary constants.

As it was done during the relatively short analysis of the code presented in the previous section, it is assumed that all the elements fetched due to the reference to `A[i1+i2+i3] [i1+i2+i3] [i1+i2+i3]` during the processing of the entire nest can fit in the cache. It is furthermore assumed that no other references occur in the code. This implies once again that all the three loops can be considered a part of the localized iteration space, and that the nest during the analysis of reuse can be percepted as one big tile with space enough for all elements. The analysis could however just as well have been performed for the same loop divided into tiles, and no loss of generality occurs here.

Furthermore let the loop bounds N1, N2 and N3 equal 10, and let the size of the 3-dimensioanl array equal 30 in each dimension. That is the declaration of the array could look like : int `A[30][30][30]`. The elements accessed during the processing of the nest is thus all the elements which appear in the diagonal between `A[3][3][3]` and `A[30][30][30]`.

As the code fraction now has been displayed, and the assumptions have been stated, an analysis of the degree of reuse that occurs for the reference to the `A`-array will be conducted.

During the execution of the entire nested loop a total of $\text{N1} \cdot \text{N2} \cdot \text{N3} = 1000$ references to the A-array are made. As the actual elements accessed already have been revealed to be the ones in the diagonal between `A[3][3][3]` and `A[30][30][30]`, the total numer of off-chip accesses can easily be identified to equal 28. How an estimate on this number can be obtained be a methodogical approach that uses the information extracted by solving kernel(H) will now be discussed.

The kernel of the access matrix representing `A[i1+i2+i3][i1+i2+i3][i1+i2+i3]` was ealier identified to equal $k \cdot (1, -1, 0) + l \cdot (0, 1, -1)$, for arbitrary constants k and l. As it was mentioned in the previous section the vector (1,-1,0) can be interpreted in the way that the element accessed in a particular iteration (x, y, z), is also accessed in the iteration $(x, y, z) + (1, -1, 0) = (x + 1, y - 1, z)$, as long as the iteration represented by this iteration vector is within the bounds of the loop. Likewise the same element will also be accessed in the iterations $(x + 2, y - 2, x)$, $(x + 3, y - 3, z)$, ... and so on, as long as the constraint pertaining to the bounds of the loops still are satisfied.

For instance, in the current example the element `A[30][30][30]` is accessed only once, namely in the iteration (10,10,10), as the addition of either of the vectors (1,-1,0) or (0,1,-1) will result in an iteration vector which is not within the loop bounds. Likewise the element `A[29][29][29]` is referenced for the first time in (9,10,10) and again

in (9,10,10) + (1,-1,0) = (10,9,10), and (10,9,10) + (0,1,-1) = (10,10,9). This iteration is however the last time the element is accessed as the addition of (10,10,9) with either of the vectors (1,-1,0) or (0,1,-1) once again will result in an iteration vector which is not within the loop bounds.

It is thus clear that the number of times a particular element, which corresponding reference exhibits multiple dimensional reuse, is accessed, and hence the degree of reuse that occurs for this element, can vary to a large extend depending on the element in question. This makes it harder to make an estimate on the number of off-chip accesses caused by such a reference.

To illlustrate this enhanced complexity a simple example of single dimensional reuse can be presented for comparative purposes. Consider the following nested loop and corresponding access matrix for the `A`-array reference :

```
for i1=0, i1<N1, i1++
    for i2=0, i2<N2, i2++
        A[i1] =  ....
```

$$\mathbf{H} = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

If the assumption is made that the localized space in this nest consists only of the innermost loop, then the `A[i1]` reference exhibits self-temporal and single dimensional reuse in the innermost loop. Each element of the `A`-array that is ever accessed is thus reused `N2` times during one iteration of the `i1`-loop. There is thus no difference in the degree of reuse that exists between the accesses to the different elements of the `A`-array. Hence the estimated number of off-chip accesses due to `A[i1]` during the entire processing of the nest can easily be identified to equal `N1` (as the localized space was assumed to consist of the `i2`-loop, and the spatial reuse in the `i1`-loop is thus not exploited).

It should be noted at this point that multiple dimensional reuse possibly could occur across any number of dimensions in the iteration space. In the example given in this section, the directions of reuse were (1,-1,0) and (0,1,-1). For a reference with the following access matrix, the reuse will however occur across all four iteration dimensions :

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

This is the case as the kernel of this matrix is given by : (1,-1,-1,-1).

So far the description of the evaluation and quantifying of multiple dimensional reuse have been concentrated on the reuse that is self-temporal. The complexity of quantifying the self-spatial reuse that occurs across multiple dimensions of the iteration space seems to be of the same magnitude as it is the case for self-temporal reuse. For instance a reference with the access matrix :

$$\mathbf{H} = \begin{pmatrix} 1 & 1 \\ 2 & 1 \end{pmatrix}$$

, exhibits no self-temporal reuse but it exhibits self-spatial reuse in the (1,-2) direction.

Yet another special case which is not handled by the estimation techniques presented in section 4.2.7, is the appearance of self-spatial reuse in more than one dimension. This is not the same as multiple dimensional self-spatial reuse, as just described by the example above. When the presentation of the estimation technique was conducted presiously, it was assumed that self-spatial reuse only occurred in a single dimension. This is however not always the case. An example of this is given by the access matrix which could represent the reference `A[3][i1+i2]` in a two-level nested loop :

$$\mathbf{H} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

In this case there exists self-spatial reuse in both loop 1 and loop 2. This is the case as the kernel of this matrix with the last row replaced by all zero's yields $k \cdot (1,0) + l \cdot (0,1)$, for arbitrary constants k and l. If the number of iterations performed in each loop in this nest is denoted N_1 and N_2, and the line-size is denoted by l, then a good estimate on the number of off-chip accesses during the processing of the entire nest would be $(N_1 + N_2)/(l \cdot element - size)$. An estimation technique which is able to provide good guesses in such a case would therefore also be valueable. It should be noted that there also exists self-temporal multiple dimensional reuse in the direction (1,-1) of the iteration space.

In this section a further description of the problems involved in quantifying multiple dimensional reuse has been given. No actual solutions have been provided, but some examples that identifies some of the problems that exist have been given.

Equivalence class generation.

In section 4.2.7 a criterion for two references to be put in the same equivalence class, for some localized space L, was established. The criterion stated that if the two references constant vectors was denoted $\vec{c}_1$ and $\vec{c}_2$, and the access matrix for the uniformly generated set was denoted H, then a particular solution $\vec{r} \in L$ to $H \cdot \vec{r} = \vec{c}_1 - \vec{c}_2$, must exist for the two references to belong to the same group-temporal equivalence class. Similarly a particular solution $\vec{r} \in L$ to $H_s \cdot \vec{r} = \vec{c}_1 - \vec{c}_2$ had to exist for the two references to belong to the same group-spatial equivalence class.

The problem of determining whether such a solution exists, might however involve the examination of both a particular solution to the equation, as well as the kernel of the access matrix corresponding to the references in question. This is the case as a complete solution to the above equation can be obtained by adding the kernel of H to the particular solution $\vec{r}$. Thus, even though a particular solution $\vec{v} \notin L$" might exist to the above equation there might still exist a solution which actually belongs to L. Such a solution would satisfy the above stated criterion for the presence of group reuse and the two references should in this case be put in the same equivalence class. This will be illustrated by the use of an example.

```
for i1=0, i1<N1, i1++
    for i2=0, i2<N2, i2++
        for i3=0, i3<N3, i3++
            A[i1][i2+i3][i2+i3] =  ....  + A[i1][i2+i3+1][i2+i3];
```

The two references `A[i1][i2+i3][i2+i3]` and `A[i1][i2+i3+1][i2+i3]` in the nest have

the constant vectors (0,0,0) and (0,1,0) respectively and they belong to the same uniformly generated set as their access patterns can be described by the access matrix :

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

As no solutions whatsoever exists to the equation $H \cdot \vec{r} = \vec{c}_2 - \vec{c}_1 = (0,1,0)$, no group-temporal reuse will exist among the references in any of the possible localized iteration spaces. When examining the references for group-spatial reuse however, some solutions exist to the equation $H_s \cdot \vec{r} = (0,1,0)$. By solving for r, a particular solution such as (0,1,0), (0,0,1) or (0,2,-1) might be obtained. In the case that the generation of equivalence classes was performed for the localized space consisting only of the innermost loop (i.e. the untransformed code), it would be necessary to perform a further analysis provided that (0,1,0) or (0,2,-1) was the obtained solution. This is the case as neither of these solutions belong to the iteration space spanned by (0,0,1). Therefore it would require further computations in order to determine if there among the complete set of solutions defined by the particular solution obtained plus kernel(H_s), exists a solution belonging to L. That such a solution actually does exist have already been revealed as it was mentioned earlier that $\vec{r}$ = (0,0,1) satisfies the criterion for group-spatial reuse. As the kernel of H_s can be identified to equal $k \cdot (0,1,-1)$, for an arbitrary constant k, the solution (0,0,1) can be obtained from the complete set of solutions by : $(0,1,0) + 1 \cdot (0,1,-1)$ or $(0,2,-1) + (-2) \cdot (0,1,-1)$.

4.3 Implementation - Overview of code.

In this section the actual implementation of the tiling transformation program will be described. At first a description of the tasks that must be performed is given. During this description a distinction between the tasks that can be handled by the SUIF1 library routines and the ones that must be implemented from scratch will be made. This is followed by a by a brief overview of the entire code in section 4.3.2. This section should provide some insight into the general flow of control in the implementation. The individual parts of the implementation will subsequently be described in greater detail in section 4.3.3. This is followed by a definition of the provided user interface in section 4.3.4. Finally a listing of the capabilities as well as some shortcomings of the implementation is given in section 4.3.4.

4.3.1 Tasks.

The first step involved in carrying out a high-level optimization pass such as the one that is implemented here is of course the parsing of the target application code. As described in chapter 3, this is handled by the SUIF1 compiler system. SUIF1 also provides an interface which allows the user to process each procedure in the application code one by one.

What is needed at this point is thus some routines which can handle the task of traversing the SUIF-IR tree representing each procedure in the application, and identify

each nested loop found. The nests which meet specific criteria for what the transform library is able to handle can then be put in a suitable data structure for later processing. This task is represented by the first box in figure 4.1. The flowchart provided in this figure illustrates the different steps involved in the complete tile-pass. It should be noted that the flowchart and its boxes in no way describes the actual code which has been developed, as it merely is intended as an illustration of different steps which are involved and the order in which they occur.

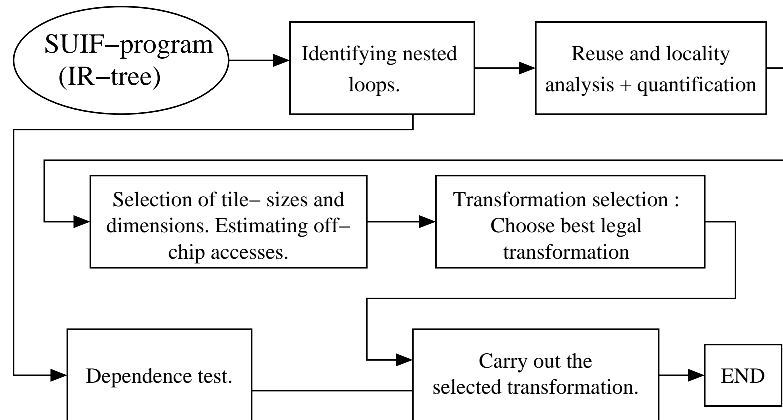


Figure 4.1: The steps involved in a complete tile-pass.

Each identified legal nest can hereafter be processed by a number of successive routines, in which the nested loop and its references are analyzed in different ways, and finally transformed. These routines are represented by the remaining boxes in the flowchart.

The first of these represents the reuse and locality analysis of the nest, as well as the subsequent quantification of the inhabited reuse. This task is by far the most complicated of the ones show here. The analysis performed in this step is conducted by using the matrix and vector representation for array references which was introduced in section 4.2.5. For this purpose the SUIF system contains some procedures which are able to extract the information in the subscript expressions of an array reference. This information about index variables and constants can then subsequently be inserted in predefined matrix and vector data-structures which are a part of the math-library provided by SUIF. This library also provides some basic operations on vectors and matrices, and some of these operations are used to carry out the in section 4.2.7 presented mathematical approaches for generating uniformly generated sets. Hereafter the references inside each uniformly generated set are partitioned into equivalence classes. This is done for each possible localization of the iteration space. Finally a quantification of the degree of reuse present is carried out for every posible localized iteration space. The results of the hereby performed calculations can then later be used to estimate the number of off-chip accesses in a nest. The quantification step is done according to the theory presented in section 4.2.8.

In the next step of the complete tile-pass the determination of tile-sizes for the different possible values of tile-dimensions, is performed. That is, the number of dimensions

of a tile equals the number of loops that is included in the localized iteration space, and tile-sizes are calculated for localized spaces consisting of loops 1 to nest-depth, 2 to nest-depth, ... , depth to depth. This step is represented by the box denoted "tile-size selection" in figure 4.1. The calculations performed at this step is done according to the theory presented in section 4.2.8. In order to carry out the computations at this step the information regarding the number of ST- and SS - reuse dimensions as well as the number of equivalence classes that was generated in the previous step, is used. As these measures were calculated for each of the possible localizations of the iteration space in the previous step, the tile-sizes can be calculated for each of these. Concurrently with the determination of tile-sizes in this step an estimate on the number of off-chip accesses for the particular localized space is also computed. These estimates are what ultimately determines which tiling is most beneficial.

The actual gathering of information pertaining to dependences among references in the nest is carried out by the routines represented by the "Dependence test" - box. The routines which accomplish this task makes use of the dependence library provided by SUIF. This library contain procedures for calculating dependences between two given references. The result of such a test will be returned in the form of one or more dependence vectors if any dependence existed. The hereby obtained dependence vectors are then stored in suitable data structures, thereby enabling other developed procedures to investigate the dependece vectors, in order to determine the legality of a particular transformation. These procedures have also been developed. The actual gathering of dependence information could in theory be done at any point in time, before the actual selection of transformation, as the information on dependences is used at this step. In the current implementation however, it must be done right after the parsing of nested for-loops, as the positions of the array-references inside the nest also is recorded at this step.

In the "transformation selection" step of the tile-pass the estimated best tiling is chosen. At first the records containing information about the different possible localizations of the iteration space is sorted according to increasing cost. The cost of a particular tiling (which are distinguished by the number of loops that they include) is considered to be the estimated number of off-chip accesses. One by one the different tilings are then considered in order of increasing cost, and the first tiling which violates no dependence constraints is selected to be carried out.

In the last step of the tile-pass the particular tiling which was estimated to yield the lowest number of off-chip accesses is carried out. The actual transformation is conducted by a call to the SUIF `tile_transform` routine which, given certain arguments can carry out the transformation of the code. The arguments which are needed establishes the loops to be tiled, as well as the tile-size.

4.3.2 Overview.

This section will provide a basic overview of the control flow in the implementation. The main parts of the code that constitutes the constructed tile-pass will be described briefly, and the contents of the involved files will also be listed.

The `main()`-function of the implementation is placed in the file `my_tile.cc`. This file thus contains function calls to the different parts of the implementation which together

forms the constructed tile-pass. An overview of the contents of this file is given in figure 4.2.

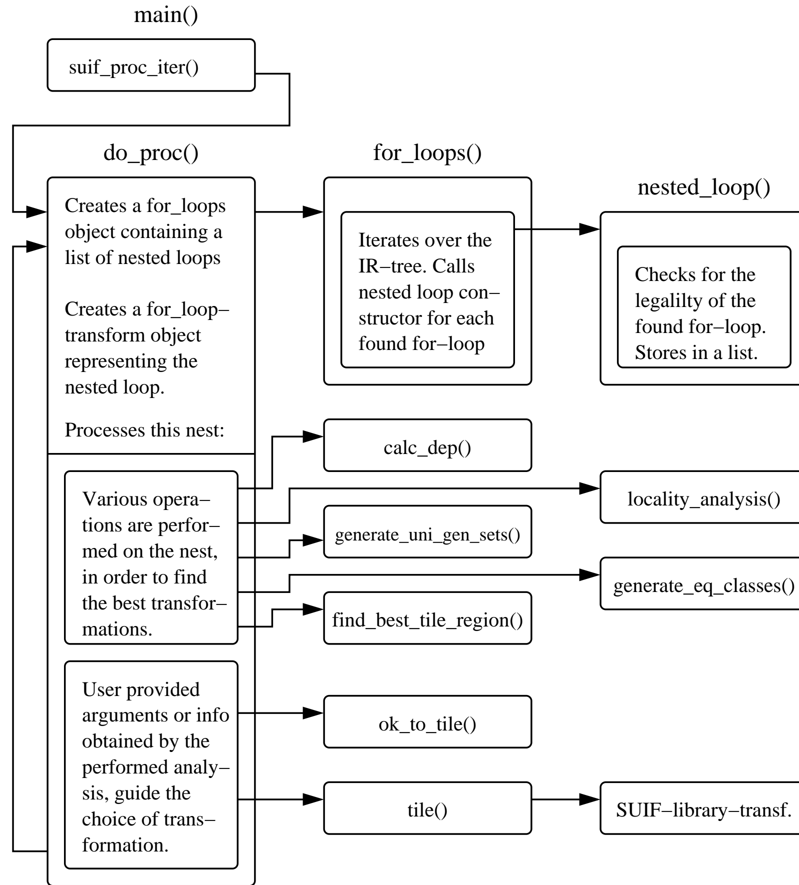


Figure 4.2: Main overview of code

In this figure the interface provided by the SUIF-system for constructing different kinds of compiler passes becomes evident. This interface, which is provided by the function `suif_proc_iter()` in the `main()` part of `my_tile.cc`, was also briefly mentioned in chapter 3. The effect of this function is to apply a function, which is given as an argument, to each of the procedures present in the SUIF input program. In the current implementation the argument given to the `suif_proc_iter()` procedure is a function denoted `do_proc()`, which is also contained in the `my_tile.cc` file. When running a pass on an input SUIF program, the `do_proc()`-function will thus be called once for each of the procedures present in the input program. It is called with a pointer to the SUIF IR-tree representing the corresponding procedure. This allows for traversing the IR-tree, identifying certain constructs, performing analytical calculations and possibly modifying the IR-tree for optimizations of different kinds.

The `do_proc()`-function consists of two distinct parts. In the first part the IR-tree is traversed and a list of pointers to nested loops that are present in the current procedure is gathered. In the second part this list of nests is traversed and each nest is analyzed

and possibly transformed.

The first part of the do_proc()-function basically consists of a call to the for_loops()-constructor. This constructor runs through the IR-tree, until it finds a for-loop. It then calls the nested_loop()-constructor, which checks for the legality of the found nest. That is, legal in these terms means that no for-loops in the nest are allowed at the same level (see section4.3.5), and that no while-loops must be present. The nested_loop()-constructor collects pointers to the for-statements in the nest. If the nest turned out to be legal, the nested loop object is appended to the list of nests in the for_loops instance. Otherwise it is deleted. After the creation of a for-loops object it thus contains a list of pointers to nested_loop objects, present in the procedure.

As it was former mentioned the second part of the do_proc()-function consists of the processing of each of the found nests. Each of these nests are handled one at a time, by creating a for_loop_transform object, which represents the particular nest, and performing operations on this object. Such an object contains function- and data-members for analyzing the particular nest with respect to reuse and locality. It also contains members for performing estimations on the number of off-chip accesses, carrying out tile-size selection algorithms and analyzing for best possible tiling. Furthermore the for_loop_transform class also contains function members for carrying out the actual tiling transformation.

As shown in figure 4.2 the first of two main steps involved in the processing of a nest (represented by a for_loop_transform object), is to perform various analyzing operations on the object. This step is represented in figure 4.2 by the calls to 5 different function members. These are denoted : calc_dep(), locality_analysis(), generate_uni_gen_sets(), generate_eq - _classes() and find_best_tile_region(). The functionality of calc_dep() is to gather information on the inhabited data-dependencies in the nest. This function also finds all array-references in the nest and stores them in a suitable data-structure for later processing. The main effect of the locality_analysis() function is to identify the access matrices and constant vectors of all found array-references.

The main functionality of the functions generate_uni_gen_sets() and generate_eq_classes(), is considered to be rather self-evident. Finally in the find_best_tile_region() function, all the obtained information regarding reuse and locality is used to estimate performance, and to find optimal tile-sizes and regions.

The second of the two main steps involved in the processing of a nest is to carry out the actual tiling. This step is represented in figure 4.2 by the last box inside do_proc(). If no specific requests for a particular tiling was made from the user, the estimated best transformation is carried out. It is however also possible for the user to specify certain command line arguments, which determines the tiling to be performed. Any tile-size relative to the calculated optimal tile-size can be chosen, for the best localized space. Furthermore it is also possible to specify the loops to be included in the tiling. If this option is set, the tiling is carried out with the optimal tile-size for that particular localized space.

As earlier mentioned the file my_tile.cc contains the main()- and do_proc()- functions of the implementation. The classes for_loops and nested_loop which were used for finding and storing nests are placed in the files for_loops.cc and nested_loop.cc respectively. The for_loop_transform class is placed in for_loop_transform.cc, and the

auxiliary data structures that this class use are placed in the file denoted `locality.cc`. These two files are rather large as all the code for analyzing nests and performing estimations is inhabited in these files.

4.3.3 Individual parts of the implementation.

In the previous section a basic overview of the implemented tile-pass was given. A follow up on this overview will be provided in this section, as each of the steps for processing a single nest will be described in greater detail. That is, all of the operations that was performed on the `for_loop_transform` object in the `do_proc()` function in figure 4.2, will be explained further. Furthermore the most important data structures will be presented.

The first step involved in the processing of a nest is, as indicated by figure 4.2 in section 4.3.2, performed by a call to the `calc_dep()` function of the `for_loop_transform` class. The tasks performed by this call will be clarified by a brief presentation in the following. An overview and description of the auxiliary functions that `calc_dep()` uses is given in figure 4.3.

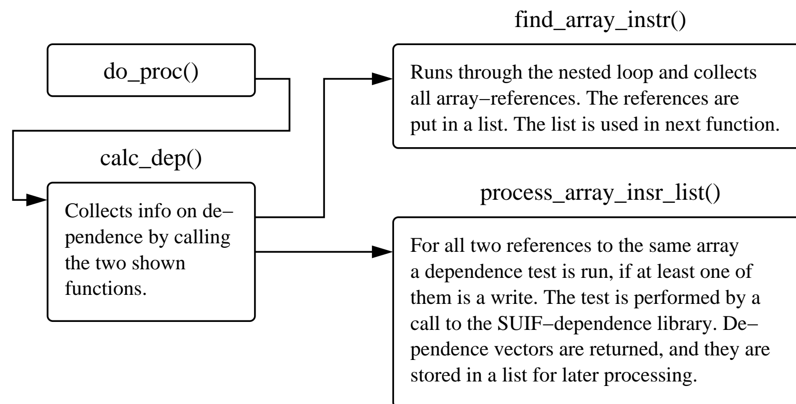


Figure 4.3: Code for calculating data dependencies

As shown the `find_array_instr()` function collects all array-references in the nest, and stores them in a list, while noting the loop in which they occurred. This important as the `calc_dep()` function therefore must be called before any of the other operations on the `for_loop_transform` object.

The subsequent `process_array_instr_list()` function performs dependence testing on all of found array references in the nest. The information gathered at this step is of the form of dependence vectors (see section 4.2.2), which are used later in `do_proc()` when the `ok_to_tile()` function is called to ensure the legality of carrying out a particular tiling transformation.

The second step involved in the processing of a nest is the gathering of access patterns for each reference. This task is performed by a call to the `for_loop_transform` function member.

`locality_analysis()`. An illustration and description of the calls to auxiliary functions that this function makes is given in figure 4.4.

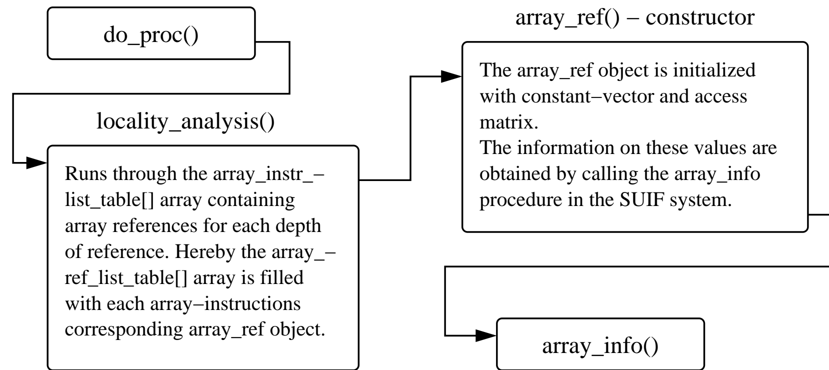


Figure 4.4: Code for performing locality analysis

The `array_instr_list_table[]` data member that the `locality_analysis()` function runs through, contains all the array references collected by `calc_dep()`. The list at index i of this array contains all instructions found in loop $i + 1$.

After the `locality_analysis()` function returns, the `array_ref_list_table[]` contains lists of all the references' access patterns.

The third step involved in the processing of a nest consists of partitioning the references into uniformly generated sets. This task can be performed based on the information found in the previous step, as two references belong to the same uniformly generated set if and only if they have the same access matrix. This step is in some sense very similar to the subsequent step, which consists of partitioning the references in each uniformly generated set into equivalence classes. The similarity, at least in the context of this implementation, is that both steps perform the task of generating and extending the same data structure. A graphical illustration of this data structure is shown in figure 4.3.3.

The structure form a tree in which all array references are divided into different "categories" at the different levels of the tree. The references are divided according to :

- The level of reference in the loop.
- The array they reference.
- The uniformly generated set to which they belong.
- The different possible localizations of the iteration space
- The equivalence class to which they belong

And this is done in the above listed order.
As it was just mentioned, the `generate_uni_gen_sets()` function carries out the task of dividing all references into uniformly generated sets, by filling in the data structure shown in figure 4.5. This data structure which is generated at this point consists only of levels down to the point of the objects denoted `uni_gen_set`. How this is accomplished is shown in figure 4.6, which illustrates how the dynamic allocation of the data structure is performed.

An additional note should be made to the actions taken inside the `uni_gen_set`-constructor. In this constructor objects of an auxiliary class denoted `iteration_space` is introduced. This class is used to represent either a certain localized space of a nested loop or one or some number of reuse dimensions for a particular reference. The reason

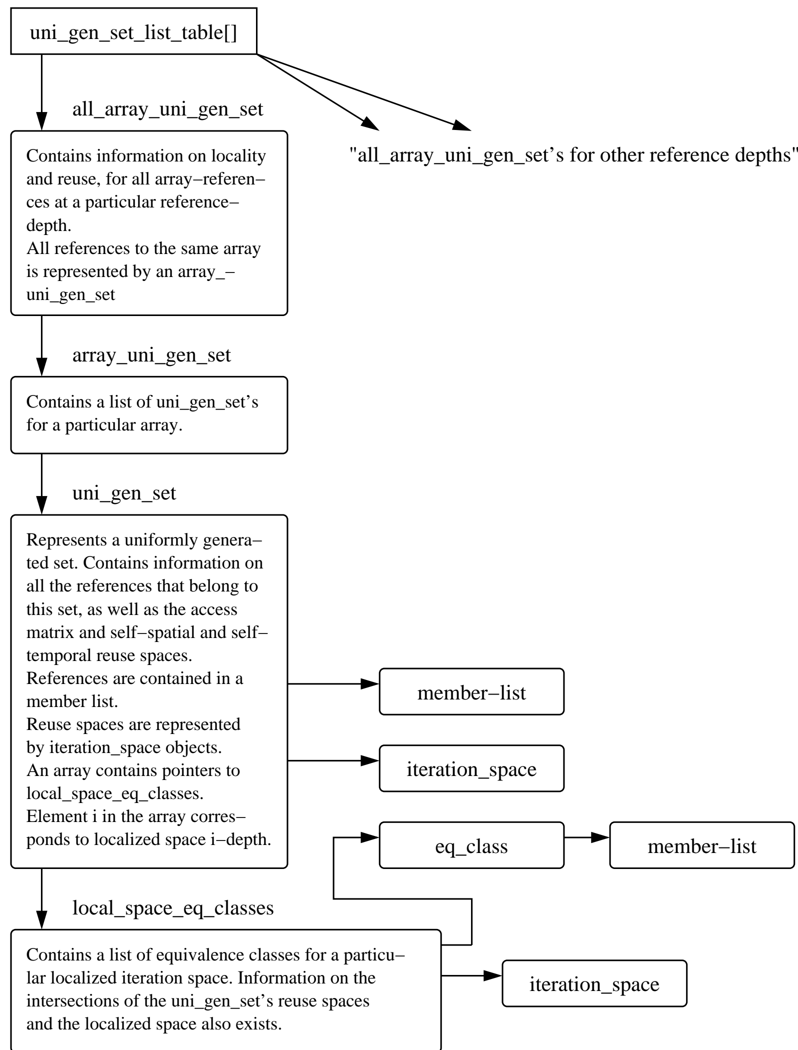


Figure 4.5: Data structure representing uniformly gen. sets and eq-classes

for developing this class is that the objects of type `vector_space` which is returned by calls to the SUIF-math library, does not contain certain needed oprations. These are for instance operations such as intersections, unions and differences between two given `vector_space` objects.

The fourth step involved in the processing of a nest consists as earlier mentioned of partitioning the references of each uniformly generated set into equivalence classes. This is done by extending the data structure shown in figure 4.5, which was constructed by `generate_uni_gen_sets()` in the previous step. A graphical illustration of how this task is carried out is given in figure 4.7.

As shown in the figure, the `generate_eq_classes()` function traverses the former mentioned data structure, and for each uniformly generated set it partitions the references into equivalence classes. This partitioning is done for each possible localization of the iteration space. That is for localized spaces 1->depth, 2->depth, ... , depth->depth.

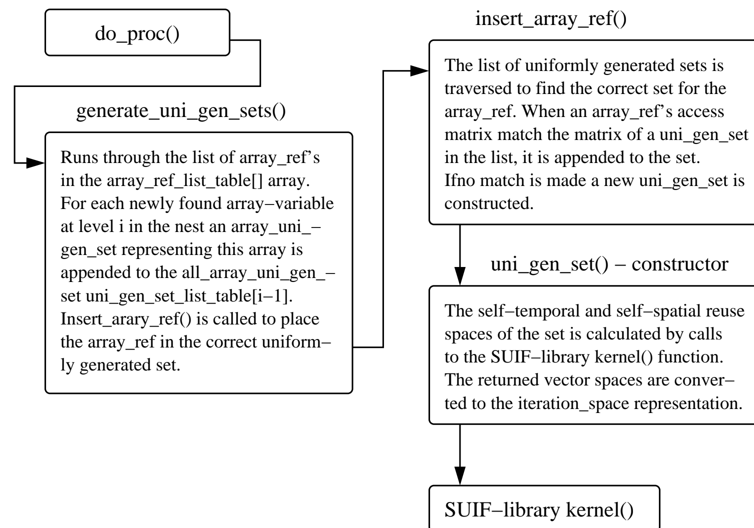


Figure 4.6: Code for generating uniformly generated sets

The actual partitioning is carried out in the `local_space_eq_classes()` constructor.

In this context it should be noted that a flaw in the SUIF system was discovered, at this point. The `local_space_eq_classes` constructor uses a list of `member`'s, which represent array references to carry out the partitioning. The list class used to contain these `member`'s is however not one of available macro-generated classes provided by the SUIF-system, as these turned out to contain a bug in at least one of their function members. This bug occurred in the `remove()` method which was needed at this point. Because it was necessary to be able to remove elements from a list and because the use of the list structures available in the C++ STL Library were unuseable with the SUIf-system (introduced redefinitions of certain functions), another solution had to be chosen. Classes containing lists of only `member`'s and corresponding iterators were therefore constructed to solve this problem. This was done by rewriting some of the source-files in the SUIF-system which defines the base list classes. For a further description of this problem see section 4.3.5.

The next step involved in the processing of a nest consists of calculating tile-sizes and estimated number of off-chip accesses, for each of the possible localizations of the iteration space. This task is handled by the function denoted `find_best_tile_region()`. The control flow of this function is illustrated in figure 4.8.

This function first collects all the information regarding reuse that has beeen obtained by the previous steps. This is done by calling the function `collect_local_space_statistics()`. The information which hereby has been gathered is then used by the function denoted `tile_size_selection()` to calculate tile-sizes and to estimate the number of off-chip accesses for the different localizations of the iteration space. After the return of this call, the calculated optimal tilings for each localization space are sorted in order of lowest cost.

By the calling of `find_best_tile_region()`, all the tasks involved in the analyzation of the nest have been completed. What remains is thus to carry out a particular tiling

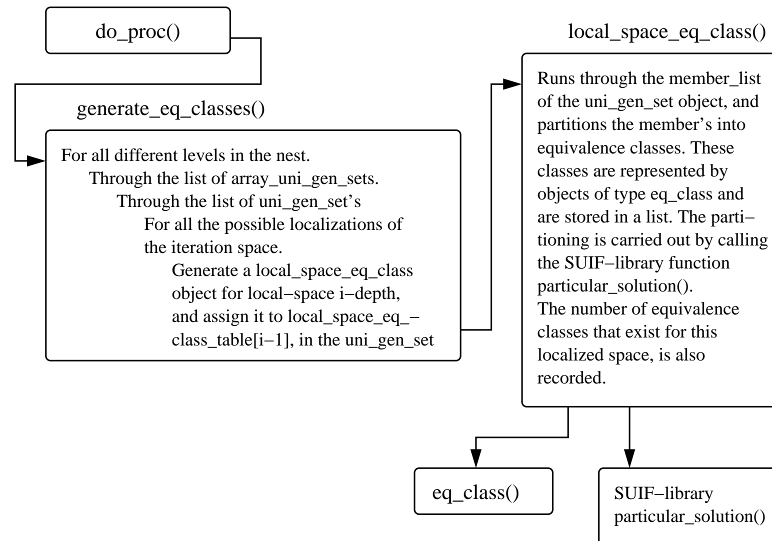


Figure 4.7: Code for generating equivalence classes

transformation, if possible.

4.3.4 User interface.

As it was mentioned in figure 4.3.2 the tile-pass will carry out the estimated most beneficial transformation, if no command-line options are provided by the user. This is done by running through the sorted array of the most beneficial tiling transformations for each possible localization of the iteration space. This sorted array was the one generated by the `find_best_tile_region()` function, in the previous step. For the optimal pair of tiling parameters (tile-size,tile-dimensions), it is then checked whether the particular tiling violates any data-dependencies, by calling the function `ok_to_tile()`. This is illustrated in the main overview of the code given in figure 4.2. This function uses the dependence vectors generated in the `calc_dep()` function to check for the legality of a particular transformation. If the tiling is illegal, the rest of the tilings are examined in order of lowest cost, until a legal tiling is found.

If the user however has specified certain tiling specific options at the command-line prompt, the tiling is instead carried out according to these options. The options available to the user are :

-cs : Cache-size

-ls : Line-size

-fl : The argument to this option specifies the first loop in the nest which should be included in the tiling. That is, a localized space consisting of loops *argument* to $nest - depth$ is requested. The requested transformation is only carried out if proves to not violate any data dependencies. If this is not the case, an error-message informing about the illegal request is printed out, and a tiling involving

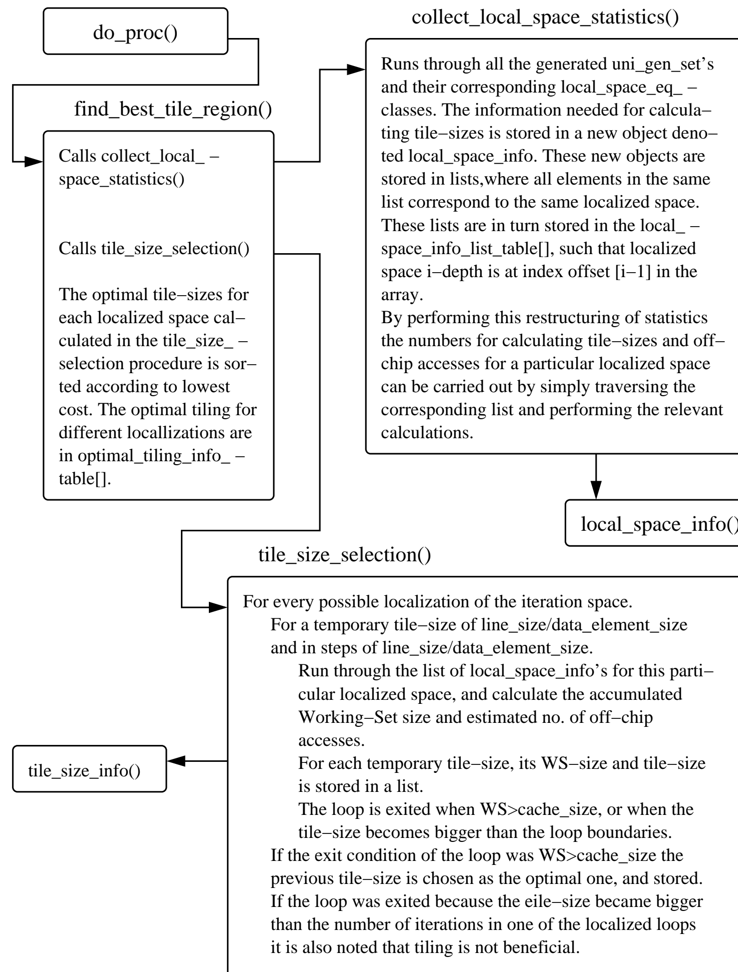


Figure 4.8: Code for finding optimal transformation

the same dimensions is tried on the next loop in the program, if any. If the argument is not within the range 1 to nest, an appropriate error message is printed out, and once again the next loop is processed. An argument which equals the nest-depth is not carried out either, as no change is made to the loacalized space by such a transformation. All tilings are performed with the optimal tile-size for the requested tile-dimension.

-ws : This option specifies that a particular tile-size is requested, for the estimated best tile-dimension. More specifically the argument to this option specifies that a particular tile-size which is $(argument - 1) \cdot (tile - size - step)$ smaller than the estimated best tile-size, should be used. In this entity the expression $tile - size - step$ is the step-size which is used for different values of the tile-size when calculating the working set size. $Tile - size - step$ is calculated as $line - size/element - size$. Furthermore if the argument value equals 0 the optimal tile-size is used. Provided that the estimated optimal tile-size for a particular nest was found to be 28, and

that $tile - size - step$ was equal to 4, the following tile-sizes for different option values would be obtained :

`-ws 1` => tile-size=28
`-ws 2` => tile-size=24
`-ws 3` => tile-size=20
`-ws 5` => tile-size=14

-bws : This option has a function very similar the -ws option, as it provides the opportunity for selecting tile-sizes that are a multiple of $tile_size_step$ bigger than the optimal tile-size. More specifically the tile-size is now chosen as $optimal - tile - size + (argument \cdot tile - size - step)$ Once again the tiling is performed for the optimal localization of the nest in question. Following the same example as given for the -ws option, the tile-sizes are now :

`-bws 1` => tile-size=32
`-bws 2` => tile-size=36
`-bws 3` => tile-size=40
`-bws 5` => tile-size=48

When using the -bws option no information pertaining to the estimated number of off-chip accesses or cache-utilization is given to the user. This is because the above tile-sizes all make the working set larger than the cache, and the applied estimation techniques are thus no longer applicable.

4.3.5 Shortcomings.

In this section a list of the capabilities as well as some of the shortcomings of the developed tool will be given. The list of capabilities which is provided is intended as a clear definition of what the tool can do. Some of the shortcomings originates from flaws or shortcomings of the SUIF1 compiler system. Others are functionalities which have not been incorporated in the implementation yet.

Capabilities.
The constructed implementation is able to perform data reuse and locality analysis of array-references in a nested loop, and to carry out a tiling of this nest, if this proves beneficial. Based on the analysis of the nest the most advantegeous tiling is performed if this transformation does not violate any data dependence constraints. All possible tilings of the original loop ordering is considered and appropriate tile-sizes are calculated in order to maximize reuse. The metric used for determining the most beneficial tiling parameters is an estimate on the number of off-chip accesses for such a tiling. This estimate is calculated simultaneously with the selection of tile-sizes, and it provides the opportunity for evaluating the correctness of the chosen transformation.

Shortcomings.

- Some kinds of nested loop constructs cannot be handled by the SUIF transform library and theses will therefore not be considered for optimization. A particular case is the appearance of several for-loops at the same level inside a nest :

  ```
  for(...){
      for(...);
      .....
      for(...);
  }
  ```

  This kind of nest will not be identified as a legal nested loop by the identification routines which constitute the introductory part of the implementation. They are therefore skipped. Furthermore while-loops are not considered as the exit conditions for such loops normally cannot be predicted at compile time. As no information on the number of executed iterations can be obtained in these cases, transformations makes no sence.
  The tool is however able to handle any other nest, which do not make the dependence testing to messy. This includes any level of for-loops, and instructions and conditional constructs in any level of the nest.

- As some flaws was discovered in the SUIF-distributed macroes for generating lists and iterators for user defined data-structures, a separate implementation of one list had to be made. The macro generated classes was not able to remove particular elements from the list, and as this ability was needed in the `local_space_eq_classes`-constructor of the implementation, a list class was created separately. This class was created by simply taking the source files used for macro generation from the SUIF distribution and performing minor editings of these. The resulting classes are contained in the files `my_list.h` and `my_list.cc`, separate from the rest of the implementation. As the `local_space_eq_classes` constructor is the only place in the implementation in which there exists a need for removing elements from a list, the macroes have been used for all other list generations in the rest of the implementation without any problems.

- When checking for equivalence class membership in the `local_space_eq_classes` constructor, only the particular solution to $H \cdot \vec{r} = \vec{c}_2 - \vec{c}_1$ returned by the Suif-math library is considered. That is, in some cases the complete solution, which can be obtained by also taking the kernel into account, can reveal another solution which actually is inside the localized iteration space. This problem was previously discussed in section 4.2.9, which can be reviewed for more details.

- The special case of multiple dimensional reuse, which occurs when reuse occurs across multiple dimensions of the iteration space, is not considered in the current implementation. An in-depth discussion of this topic was conducted in section 4.2.9, where it also was mentioned that no litterature describing such cases had been found.

- As the estimation techniques used in the current implementation does not consider

conflict misses, the tool might provide unrealistically optimistic estimations when direct-mapped caches is used. If furthermore there exists very similar access patterns of different arrays in the particular target application code, the estimations on the number of off-chip accesses might be even smaller.

- The appearence of references to arrays which contain elements of different sizes can of course occur for some applications. This difference can have an impact on the estimation of the degree of spatial reuse that occurs for certain references as the tile-size always is chosen to be a multiple of the entity line-size/element-size. The reason for choosing tile-sizes of this magnitude is, as it has earlier been discussed, that the fetching of unwanted data-elements thereby can be avoided for some cases. A good way of handling the appearance of references to arrays with elements of different sizes would therefore be to investigate the innermost loop for which element size is the most frequently occurring. This element-size could then be used to determine the steps of the tile-size, which would equal line-size/element-size. This feature has however not been implemented and a random element in the innermost loop is chosen to determine the steps of the tile-size.

4.4 Future Work and Improvements.

In this section some suggestions for extensions and improvements to the developed tool will be presented. Some of these suggestions originates from topics which have been presented in the survey report in chapter 2. The minor shortcomings of the tile-pass which were presented in section 4.3.5, will not be discussed further in this section. Instead some ideas to extensions at a higher level of abstraction will be provided.

4.4.1 Interchange

An obvious improvement to the existing pass would be to incorporate an algorithm for analyzing the possibilities and performance gains of adding interchange transformations to the pass. This, in some cases, very powerful technique for improving reuse and/or locality could be used together with the already implemented tiling transformation, in a new and improved pass. The interchange of loops would allow for placing loops with a large degree of reuse innermost, which could be followed by a tiling of these loops which might improve performance even further. This topic was also briefly discussed at the end of section 4.2.6.

As the tile-pass in its current form already is able to identify the degree of reuse in the different loops, all that is needed is to make comparisons of these different quantities. The natural approach is thus to carry out interchanging transformations as a first step, followed by a potential tiling if this proves beneficial. There might also exist dependencies which percludes certain loops from being moved in the nest. This should naturally also be considered at the first interchanging step, as it might reduce the search space, of finding an optimal transformation. After this step has been completed an algorithm for finding the best tiling of the innermost loops could be run. In more explicit terms the integrated interchange and tile pass could be constructed according to the guidelines

listed in the following. This is an almost identical approach to the one used in [54] :

1. The loops which carries no reuse and which legally can be placed outermost in the nest are moved there, as there exists no incentive to make them part of the localized iteration space (on the contrary). Furthermore the loops which do carry reuse but must remain in the outermost loops are set to placed there.

2. The remaining loops that were not set to be a part of the outermost loop-set in step1 are now considered. Among these loops every subset in which all loops contain reuse are tiled innermost in the nest, if this is legal. The performance in terms of the number of off-chip memory accesses are noted for each subset, and the one yielding the best performance is chosen.

4.4.2 Memory Layout

Another natural extension would be to include memory layout transformations such as padding to the constructed pass. The use of padding was initially presented in section 2.3.4, as a powerful technique for reducing both self-interference as well as cross- interference conflicts. In this context it should also be noted that one of the incentives for choosing square tiles, in the implementation, was that the use of padding could be subsequently applied in order to reduce conflict misses. This was also mentioned in section 4.1, where arguments for the choice of implementation were presented. By choosing to use square tiles, without any particular regard to conflicts, the use of padding thus constitutes a natural extension to the tile-pass. Choosing to integrate both interchanging, as former discussed, as well as padding is likely to yield even better results.

4.4.3 Estimations

Yet another way to improve the existing tile-pass, would be to extend the used estimation techniques to also consider conflict misses. The techniques used by [37], which were cited and discussed in section 2.4.5, might be a good choice for this purpose. Whether the integration of such extended estimation techniques would result in significant changes of the transformations selected by the pass is debatable.

What would be more interesting in this context is however, to use these estimation techniques to examine the design space of the memory hierarchy. This would allow the pass to analyze an application and try to find the best memory configuration for such a system. As an example one could consider an application for which tiling would be beneficial, and where all the data accessed occupies twice the space of the initial cache size. In such a case a doubling of the cache size would make all the data fit into the cache, and no localization of the iteration space would be necessary.

An estimation technique which considers conflict misses could also be used to explore the design space with regard to different combinations of both cache- and line- size. An exhaustive search algorithm for carrying out this task was presented in section 2.4.5. By constructing such a pass, the tradeoffs between power and performance involved in specifically the selection of the line-size could also be considered. This topic was

previously adressed in section 2.4.2. By including the effect on power consumption as a secondary goal, a guideline for estimating the bus-transition count as a function of the line-size would also be needed. That is, one could for instance suggest that the bus-transition count is linear in the line-size of the cache. Profiling statistics might provide more accurate figures to be used for this purpose.

Alternatively the SimplePower tool could be used to examine the effect of memory hierarchy design choices and/or transformations. This tool is, as the name suggests, a simulator for obtaining power consumption related statistics. Some examples of the use of this tool can be found in [51]

Chapter 5

Testing.

In this chapter the results obtained by the use of the tile-pass will be presented. Based on these results a discussion of the duability of the implemented tool will be given. The testing for the correctness of the different parts os the implementation will not be presented here. The results of this part of the testing-phase is instead provided in the appendix.

In the next section the benchmarks which have been used will be described. This is followed by section 5.2 in which some notes on the estimation techniques and assumptions are provided. This section also contains an overview of the kind of tests which will be performed. In the following sections the results of running the tile-pass on the different benchmarks will be given. The benchmarks will be handled one by one and a discussion of the results will be provided. Finally some summarizing remarks will be given in section 5.4.

5.1 Benchmarks.

The search for suitable real-world embedded systems benchmarks, to be used in this project has proven to be more than cumbersome. The reason for this stems from a number of factors. In order to provide a realistic evaluation of the useability of the implemented tool, it is first of all necessary to use embedded systems applications. It has however not been possible to locate any collection of embedded systems applications, from which source codes could be obtained. When dealing with general purpose processors the SPEC benchmarks [21] is a widely used collection of sources for evaluating and comparing performance. These source codes are, as it was just mentioned not relevant in the context of this project, as there is no way of knowing if such similar code also exist for embedded systems (see Chapter 1).

A single organization which distribute source codes for embedded systems was however found. This organization is denoted EEMBC [20] and is located in the U.S. Unfortunately the EEMBC does not allow distribution of source code outside the U.S., at present time.

Another factor which limits the amount of applications which could be used for testing, is the shortcomings of the SUIF-transform library. As it was mentioned in section 4.3.5, the library is not able to handle more than one for-loop at the same level

inside a nest. This fact has percluded some of the found benchmarks from being tested. The presence of non-constant loop bounds in an application will naturally also make the code unsuitable for testing.

Furthermore it would be beneficial to obtain some benchmarks which has the potential of being optimized. There is for instance no reason to alter a nest in which all array-references are accessed row by row, and where the only reuse is self-spatial in the innermost loops.

Some contacts at the Technical University of Denmark, have been able to provide a couple of source codes for embedded systems. These applications consisted of code used in a digital camera, and code used in a cavity detection algorithm. Unfortunately all of these turned out to be unapplicable due to one or more of the above mentioned reasons.

As a result of these somewhat unfortunate circumstances, three examples of algorithms used in real-world applications have been gathered from a number of articles. These algorithms are commonly used in image- or DSP- processing applications. Two of the algorithms are denoted CONV (convolution) [36] and SOR (Successive Over Relaxation) [35]. The third algorithm is commonly referred to as the local summation problem [27], which in some sence describes the two former algorithms. The source code of these applications exhibit in some sence very similar memory access behaviour. This is the case as some of the references in each of the algorithms traverse a particular number of rows of a two-dimensional array simultaneously. This memory access behaviour allows for an exploitation of reuse in the outermost loops, and there exists therefore potential gains by performing a tiling. Furthermore the former introduced Matrix Multiplication algorithm will also be used to demonstrate the capabilities of the tile-pass.

5.2 Assumptions and metrics.

This section will contain a description of the different tests that will be run on the benchmarks mentioned in the previous section. Furthermore some notes on what to expect from the implemented estimation technique will be given.

As it emerges from chapter 4, in which the actual implementation was presented, the tile-pass can perform estimations on the number of off-chip accesses as well as tiling-transformations. These capabilities are partly obtained by performing certain sub-tasks which enable the pass to make good choices on which tiling parameters to use. That is, an evaluation of the degree of reuse present in the different loops is performed. Furthermore the tile-sizes to be used in a potential tiling is calculated. Based on these facts, some tests for determining the following capabilities of the pass will be conducted :

- The pass will be tested for the ability of correctly identifying the most beneficial part of the iteration space to localize.
- The pass will be tested for the ability to identify the best possible tile-size, corresponding to the chosen localized space. This can be done by carrying out transformations with other tile-size parameters than the estimated optimal and performing a subsequent simulation of the transformed code.
- The incorporated estimation technique, which the tile-pass makes use of, will be tested for its accuracy.

- For some of the benchmarks, the effect of direct-mapped vs. fully associative caches, on the estimations will be evaluated.
- The performance gains that are obtainable for the different benchmarks will be calculated. The gains are measured as a reduction in the number of off-chip accesses.

The tests on each of the benchmarks will be carried out with a variety of different combinations of cache- and line- sizes. That is, for these different combinations, the tile-pass is given the particular cache- and line- size as command-line options and the simulation is subsequently carried out with the same cache parameters.

The estimation technique that is currently incorporated in the tile-pass does, as previously mentioned, only consider compulsory and capacity cache misses. That is, no estimations on the appearance of conflict misses is made. Furthermore the tiling algorithm selects the tile-sizes (for a particular number of dimensions) such that the amount of data fetched under the processing of an entire tile is as close to the cache size as possible. The cache is thereby filled up with data-elements during the execution of a tile. The degree to which the cache is used might though still depend on the number of dimensions of the tile, as well as the sizes of the data-elements and the line-size. This is a fact as the tile-sizes are examined in steps of *line-size/data-element-size* as explained in section 4.3.3. Thus, if the dimensions of the chosen tile is large, the cache is small and the line-size is large relative to the data-element-size then it is possible that a relatively small fraction of the cache will be occupied by the necessary data. This metric for the utilization of the cache can be very important for the obtained performance.

When using a fully associative cache and a Least Recently Used replacement policy, a high degree of cache utilization is definite desirable. This could ensure that all data-elements accessed during the processing of a single tile only had to be fetched once. When dealing with a direct-mapped cache however, a very high cache utilization might degrade performance, as more conflict misses are likely to occur. Whether such a reduction in performance will occur depends on the access patterns of the different array references. Situations in which numerous conflict misses appears can definitely arise if an unfortunate combination of cache-size and access patterns exist.

The cache utilization parameter is also a very important entity that should be considered when interpreting the estimations provided by the tile-pass. This is a fact as the tile selection algorithm stops examining for further tile-sizes when the tile-size exeeds the iteration count in one of the loops that are included in the localized space. This is the obvious action to take, as the loops naturally cannot be tiled by a size greater than the iteration count. The occurrence of such a situation should be interpreted in the way that the particular loops actually can be considered to be a part of the localized iteration space even without performing any tiling transformation. Thus the cache size is big enough to hold all the data accessed in the loops in question, and the reuse present in those loops can be exploited. In such a situation, it might very well be possible that elements accessed in iterations of the outermost loops are also reused, and the estimation provided can in such cases be very poor. If the cache is much larger than the working set, then a potentially very large amount of data can also be reused, and the estimations become even more misleading. It can thus be concluded that for relatively small cache utilization parameters, the smaller the cache utilization parameter is, the more unreliable the estimation will be. The cache utilization parameter is thus a very useful indication

of the correctness of the performed estimations. For obvious reasons the localized spaces consisting only of the innermost loop (i.e. the untransformed loop) are likely to have the largest number of cases where the estimations are unreliable. This is a fact as the number of elements accessed during a complete iteration of the innermost loop of course is less than the amount of data accessed in complete iterations in any of the other loops.

5.3 Results.

5.3.1 Matrix multiplication.

This section contains a presentation of the tests which were run on the matrix multiplication algorithm. A presentation of the most important results of these tests is also provided. As this section is the first of four in which the different tests on the benchmarks are presented, the comments will be more thorough in this section.

A matrix multiplication algorithm has, as previously mentioned the special property that it is both legal and beneficial to tile all three loops [54]. The code is given in the following :

```
#define N 1000
int A[N][N];
int B[N][N];
int C[N][N];
test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            for(k=0; k<N; k++){
                C[i][k] = A[i][j] * B[j][k];
            }
        }
    }
}
```

The tile-pass has been run on this exact code fraction for a number of different combinations of cache- and line- sizes. More specifically all combinations of cache-sizes 8, 16 and 32 Kb with line-sizes 8, 16 and 32 bytes have been used. These cache parameters have been given to the tile-pass by the use of the command-line options "-cs" and "-ls", as described in section 4.3.4. The estimated number of off-chip accesses for each of these completed passes have been recorded. Furthermore, different localizations of the iteration space have been requested for all of the (*cache-size*, *line-size*) combinations. As described in section 4.3.4, this can be done by specifying the "-fl" command-line option when running the tile-pass. As it was also described in that section the tile-pass automatically tries to find the best tile-sizes for whatever localized iteration space the user requests.

The transformed as well as the untransformed code have subsequently been simulated by the in chapter 3 presented SimpleScalar tool. The hereby obtained simulation results

matrix-mul.		Localized space 3-3			Localized space 2-3			Localized space 1-3		
cache	line-size	sim.	est.	ut.	sim.	est.	ut.	sim.	est.	ut.
8 Kb	8 bytes	**548**	1010	9%	**551**	523	98%	**103**	58	99%
	16 bytes	**282**	510	9%	**303**	261	98%	**83**	31	84%
	32 bytes	**167**	260	10%	**194**	131	82%	**83**	15	84%
16 Kb	8 bytes	**529**	1010	4%	**533**	516	96%	**80**	41	94%
	16 bytes	**269**	510	4%	**280**	258	90%	**54**	21	94%
	32 bytes	**150**	260	5%	**163**	129	79%	**52**	12	75%
32 Kb	8 bytes	**216**	1010	2%	**221**	511	96%	**50**	29	99%
	16 bytes	**110**	510	2%	**116**	256	96%	**31**	14	99%
	32 bytes	**60**	260	2%	**66**	128	96%	**26**	8	84%
Average		**259**	593	-	**270**	301	-	**62**	25	-

Table 5.1: Matrix Multiplication - direct mapped (off-chip acc. in thousands).

are listed together with the recorded estimation results in table 5.3.1.

As stated in table 5.1 the simulation have been performed by the use of a direct-mapped cache. The column denoted "Localized space 3-3" holds the results from the untransformed program. The columns "Localized space 2-3" and "Localized space 3-3" hold results from the tiling of the two innermost loops and the tiling of the entire nest respectively.

In the last row of the table the average values of simulation as well as estimation results are given.

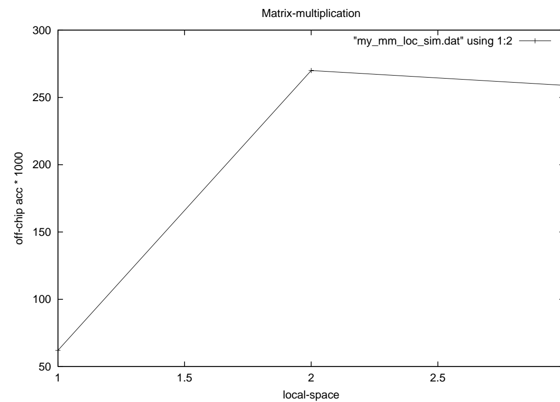
When analyzing the results in the table, it is important to keep the discussion of cache utilization which was given in the previous section in mind. The tilings of loops 2-3 and 1-3 have relatively high cache utilization rates, and the estimations given in these columns might therefore be close to the actual. When looking at the estimation results for the untransformed program (i.e. 3-3) however, the cache utilization rate appear to be very low. As discussed in the previous section this indicates that the estimation results are likely to be very pessimistic. It emerges from table 5.1 that this is indeed the case.

The estimation results stated in the second column (i.e. 2-3) is fairly accurate. The results for cache-size 32Kb deviate by an approximate factor of two, but the rest are not far from the simulation results.

The estimation results for the tiling of the entire nest are very optimistic, and they all lie below he actual values. Part of the reason for this is probably that a direct-mapped cache has been used. C.f. the discussion in the previous section, this might lead to optimistic estimates, as conflict misses are not considered in the current implementation. Furthermore the reason that these estimates are so much more optimistic than it is the case for local-space 2-3 (which on average actually are pessimistic) is probably that the bigger strides involved when processing a three dimensional tile causes a larger number of conflict misses. That is, when dealing with a cache of a specific size, a tiling which includes three dimensions are bound to have smaller tile-sizes, than a tiling of only two loops. If furthermore there exists some references in the code in question, for which the arrays are traversed row by row, then all of these elements will be placed next to each other in memory, and will therefore cause fewer conflicts in the cache. As the tile-sizes

in a tile of three dimensions on the other hand are smaller this fortunate memory layout may not be exploited to the same extend.

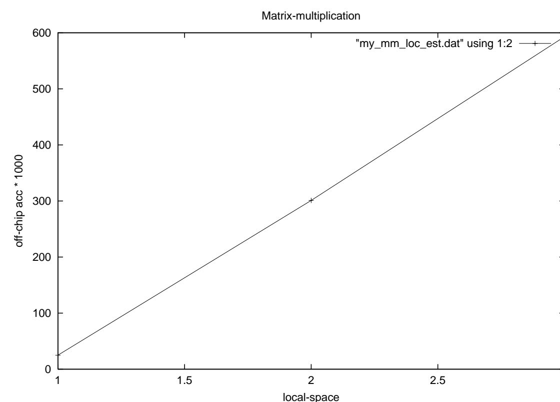
When evaluating the ability of the tile-pass to correctly identify the most beneficial iteration space to localize, there is however, no incorrect estimations. For all the possible combinations of cache- and line- sizes, the tiling of the entire nest is correctly chosen as the optimal choice. On average, the results naturally also point out that a localization of loops 1 to 3, is the most advantageous. The average results for the simulations are plotted in figure 5.1. The value on the x-axis denotes the first loop in the nest which is included in the tile. The value on the y-axis is the number of off-chip accesses in thousands.



est.

Figure 5.1: Average simulation-results for local-space x-3

For comparative purposes the estimated average number of off-chip accesses are plotted in figure 5.2.



est.

Figure 5.2: Average estimation-results for local-space x-3

As it has just been discussed, the estimations for the localized space 3-3 is not valid because of the poor cache utilization. The estimates for the localized space consisting of loops 2-3 is fairly accurate, whereas the estimations for loops 1-3 is very optimistic.

The obtained performance gains of running the code through the tile-pass are though

matrix-multiplication		-3			-2			-1		
cache-size	line-size	sim.	est.	ut.	sim.	est.	ut.	sim.	est.	ut.
8Kb	8 bytes	**105**	187	2%	**107**	68	70%	**111**	62	84%
	16 bytes	**86**	62	21%	**84**	47	37%	**80**	37	58%
	32 bytes	**168**	260	10%	**89**	47	9%	**85**	23	37%
16Kb	8 bytes	**85**	50	65%	**88**	47	75%	**80**	44	84%
	16 bytes	**59**	31	42%	**56**	27	57%	**58**	23	75%
	32 bytes	**59**	47	4%	**55**	23	18%	**53**	16	42%
32Kb	8 bytes	**56**	33	77%	**58**	31	84%	**50**	30	91%
	16 bytes	**33**	19	58%	**34**	17	70%	**35**	16	84%
	32 bytes	**26**	16	21%	**26**	12	37%	**25**	9	58%
Average		**75**	78	-	**66**	35	-	**64**	29	-

matrix-multiplication		Optimal			+1			+2		
cache-size	line-size	sim.	est.	ut.	sim.	est.	ut.	sim.	est.	ut.
8Kb	8 bytes	**103**	58	99%	**104**	-	-	**105**	-	-
	16 bytes	**83**	31	84%	**79**	-	-	**81**	-	-
	32 bytes	**83**	16	84%	**82**	-	-	**79**	-	-
16Kb	8 bytes	**80**	42	94%	**80**	-	-	**81**	-	-
	16 bytes	**54**	21	94%	**54**	-	-	**80**	-	-
	32 bytes	**52**	12	75%	**50**	-	-	**84**	-	-
32Kb	8 bytes	**50**	29	99%	**50**	-	-	**51**	-	-
	16 bytes	**31**	14	99%	**31**	-	-	**31**	-	-
	32 bytes	**26**	8	84%	**24**	-	-	**24**	-	-
Average		**62**	26	-	**61**	-	-	**68**	-	-

Table 5.2: Matrix multiplication - different tile-sizes.

significant. As indicated by both table 5.1 and figure 5.1 a reduction in the number of off-chip accesses of $100 \cdot (259 - 62)/259 = 76\%$, is obtained.

In order to evaluate the ability of the tile-pass to correctly calculate an appropriate tile-size, yet another type of tests has been performed on the matrix multiplication code. These tests have been obtained by running the tile-pass with the same cache- and line-size parameters as before, but with an additional user requested tile-size. For this purpose the "-ws" and "-bws" options described in section 4.3.4 have been put to use.

The reason for carrying out these kind of tests is that allthough significant performance gains turned out to be obtainable for the implemented tile-size selection algorithm, there might exist tiles (of the same dimensions) with different sizes which could yield even better results. The results of performing these tests are shown in table 5.2.

In this table the column denoted "Optimal", holds the results of the estimations as well as the simulations for the tile-size that was automatically selected by the tile-pass. The other columns are denoted with a number T, for which the results in the respective column are obtained by using a tile-size equal to : "Optimal-tile size" + ($T \cdot$ *tile-size-step*). As no estimations are made when a particular tile-size makes the working set larger than the cache, the columns of T equal to +1 an +2 are empty. Simulation results have though still been obtained.

The results in the table show that the tile selection algorithm selects the best tile-size in 5 out of the 9 cases. The difference in performance between the tile-sizes in the "Optimal" column and the nearby columns are though very slim, and in all but one of the cases (row 3) the actual best tile-size is in either the "-1" or "+1" column. This means that the tile selection algorithm in this case is very well suited as a tile-size very close to the optimal is selected. This trend is also indicated in figure 5.3, where the average values of the simulation results are plotted.

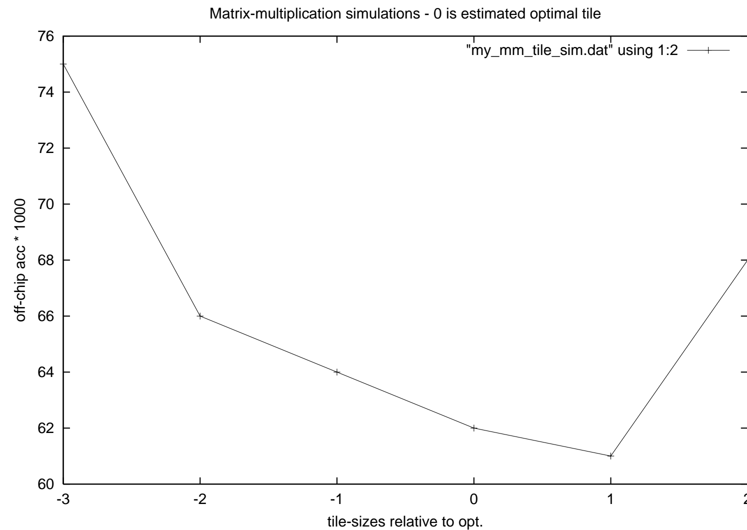


Figure 5.3: Average sim.-results for different tile-sizes

In this figure it can be seen that there is a minimum at the "+1" tile-size, but that this minimum is very close to the value value corresponding to the tile-size selected by the tile-pass ("0").

The average values obtained by the estimation technique is shown in figure 5.4

5.3.2 Successive Over Relaxation.

This section presents the tests which were run on the Successive Over Relaxation algorithm (SOR). A presentation of the most important results of these tests is also given.

The SOR algorithm is an algorithm which is frequently used in the domain of image processing applications [43]. The code kernel for this algorithm is presented in the following.

```
#define N 512
#define K 2
int A[N][N];
int B[N][N];
int C[N][N];
int D[N][N];
int E[N][N];
int U[N][N];
```

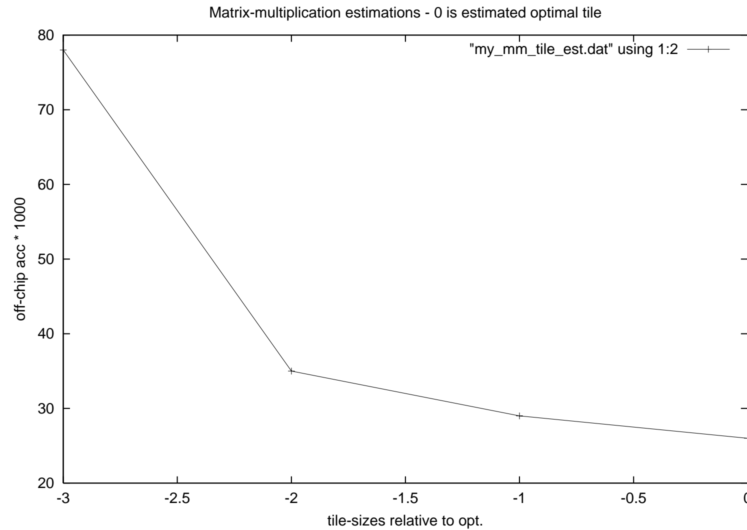


Figure 5.4: Average est.-results for different tile-sizes

```
void test1(){
    int i, j, temp;
    for(i=1; i<N-1; i++){
        for(j=1; j<N-1; j++){
            temp = A[i][j]*U[i+1][j] + B[i][j]*U[i-1][j] +
                   C[i][j]*U[i][j+1] + D[i][j]*U[i][j-1] +
                   E[i][j]*U[i][j];
            U[i][j] += K*temp*E[i][j];
        }
    }
}
```

The SOR code has been simulated on an architecture with a fully associative cache. This has been done as the use of a fully associative cache is likely to yield simulation results that correlate better with the estimations performed by the tile-pass. The reason for this is as it has earlier been mentioned that this ensures no conflicts between data elements accessed during the processing of a particular tile. That no such conflicts occur for each executed tile, is furthermore one of the assumptions that makes the used estimation technique valid.
The results of running the tile-pass on the SOR code are shown in table 5.3.

The results of estimating the number of off-chip accesses for the untransformed program are very close to the actual simulated results. All but one of the estimations lie below 1% of the simulated result. The estimation technique thus performs extremely well in this case.

The estimations for the localized space consisting of loops 1-2 are allthough they are relatively close not as precise as the estimations for the untransformed program.

The average reduction in the number of off-chip accesses can be calculated to be $100 \cdot (610 - 540)/610 = 11\%$. It should be noted that in three of the 9 cases the

SOR.		Localized space 2-2			Localized space 1-2		
cache-size	line-size	sim.	est.	ut.	sim.	est.	ut.
2Kb	8bytes	**1046**	1040	96%	**817**	780	75%
	16bytes	**523**	520	93%	**573**	390	75%
	32bytes	**261**	260	87%	**403**	195	75%
4Kb	8bytes	**1046**	1040	98%	**807**	780	84%
	16bytes	**523**	520	96%	**512**	390	84%
	32bytes	**261**	260	93%	**204**	195	37%
8Kb	8bytes	**1046**	1040	99%	**869**	780	94%
	16bytes	**523**	390	75%	**400**	390	75%
	32bytes	**261**	260	96%	**279**	195	75%
Average		**610**	592	-	**540**	455	-

Table 5.3: Successive Over Relaxation - algorithm (fully associative).

simulations actually show that the untransformed program performs better than the tiled one. The tile-pass however identifies the a tiling of both loops to be the best choice in all 9 cases.

The reason that the benefits of tiling are significantly less in this example is undoubtably that only one of a total of 6 arrays in the SOR code has an access pattern which allows for a beneficial tiling. All of the 6 arrays are furthermore of the same size, and the degree to which the reuse in the one mentioned array can be exploited is thus reduced significantly.

5.3.3 CONV.

The CONV algorithm is a frequently used convolution program in image processing applications and DSP applications [3]. It is typically used in tasks such as edge detection, regularization and morphological operations. The code kernel of the CONV algorithm is given in the following :

```
#define N 1024
#define NORM 16
#define M 4
int source[N][N];
int mask[M][M];
int dest[N][N];
test1()
{
    int x, y, temp;
    for(x=0; x<N-M; x++){
        for(y=0; y<N-M; y++){
            temp = 0;
            for(int i=0; i<M; i++){
                for(int j=0; j<M; j++){
                    temp += source[x+i][y+j] * mask[i][j];
```

CONV		Localized space 2-2			Localized space 1-2		
cache-size	line-size	sim.	est.	ut.	sim.	est.	ut.
2Kb	8bytes	**11221**	2642	99%	**11241**	1061	78%
	16bytes	**8060**	1343	96%	**8145**	549	59%
	32bytes	**6832**	697	92%	**6954**	325	31%
4Kb	8bytes	**11067**	2621	99%	**11077**	1048	95%
	16bytes	**7792**	1321	99%	**7842**	530	79%
	32bytes	**6321**	671	96%	**6385**	276	53%
8Kb	8bytes	**11015**	2611	99%	**10726**	1045	88%
	16bytes	**7684**	1310	99%	**7578**	525	77%
	32bytes	**6099**	660	99%	**6060**	267	57%
Average		**9.394**	1.542	-	**9.383**	625	-

Table 5.4: CONV Algorithm - direct-mapped.

```
                }
            }
            dest[x+M/2][y+M/2] = temp/NORM;
        }
    }
}
```

The estimations as well as the simulated results for the CONV code is shown in table 5.4. These simulated results are obtained by running the simulations on a direct mapped cache.

As it emerges from the values in the table the estimated and simulated results are extremely different. The actual simulations show that the number of off-chip accesses is far greater than the estimated, and that the differences ranges from a factor of 5 to a factor of 20. This extremely large difference might suggest that severe conflict misses will occur during the execution of the CONV code (with the shown choice of cache- and line-sizes). It should also be noted that the simulations show almost no variation between the transformed and the untransformed code. The estimations provided by the tile-pass show however, that significant performance gains are obtainable by tiling. Considering all of these facts, it is thus very likely that severe conflict misses indeed do occur.

To investigate this matter further, the exact same simulations were performed with a fully associative cache. The results that were obtained hereby are shown in table 5.5.

As indicated by the values in the table, this test proves that severe conflicts actually did occur during the simulation with the direct mapped cache. Moreover the estimates now match the simulation results to a very large extend. It has thereby been proved that the applied estimation technique in this case is very accurate, once the assumption that no conflict misses occur has been made.

The average reduction in the number of off-chip accesses can be calculated as $100 \cdot (1523 - 676)/1523 = 56\%$.

CONV		Localized space 2-2			Localized space 1-2		
cache-size	line-size	sim.	est.	ut.	sim.	est.	ut.
2Kb	8bytes	**2610**	2642	99%	**1156**	1061	78%
	16bytes	**1306**	1343	96%	**588**	549	59%
	32bytes	**653**	697	92%	**310**	325	31%
4Kb	8bytes	**2610**	2621	99%	**1211**	1048	95%
	16bytes	**1306**	1321	99%	**608**	530	79%
	32bytes	**653**	671	96%	**286**	276	53%
8Kb	8bytes	**2610**	2611	99%	**1096**	1045	88%
	16bytes	**1306**	1310	99%	**551**	525	77%
	32bytes	**653**	660	99%	**278**	267	57%
Average		**1.523**	1.542	-	**676**	625	-

Table 5.5: CONV Algorithm - fully associative.

5.3.4 Local summation problem.

In this section the tests which were run on the so called local summation algorithm are presented. This algorithm is a kind of general version of the previous presented SOR- and CONV- algorithms. That is, this kind of algorithm is very common in image processing algorithms, and the local summation algorithm presented here is a slightly different version. The access patterns does however differ to some extend. The code is taken from [27], and is given in the following:

```
#define N 1000
#define M 4
int Sum[N][N];
int Image[N][N];
test1()
{
    int i, j;
    for(i=0; i<=N-4; i++){
        for(j=0; j<=N-4; j++){
            Sum[i][j] = 0;
            for(k=0; k<M; k++)
                for(l=0; l<M; l++)
                    Sum[i][j] += Image[i+k][j+l];
        }
    }
}
```

The results of running the tile-pass on the Local summation algorithm are shown in table 5.6.

In this case it is noted that the cache utilization rates are very high for the untransformed program, thereby indicating that the estimations are valid. This is of course only the case when the interaction between code and cache parameters do not result in severe conflicts. That these estimation results indeed are valid emerges clearly from

Loc-Sum.		Localized space2-2			Localized space 1-2		
cache-size	line-size	sim.	est.	ut.	sim.	est.	ut.
2Kb	8 bytes	**2611**	2480	99%	**1370**	992	76%
	16 bytes	**1489**	1240	97%	**930**	496	56%
	32 bytes	**1114**	620	93%	**989**	248	25%
4Kb	8 bytes	**2552**	2480	99%	**1367**	992	94%
	16 bytes	**1368**	1240	99%	**732**	496	78%
	32 bytes	**868**	620	97%	**620**	248	50%
8Kb	8 bytes	**2025**	2480	99%	**1099**	992	87%
	16 bytes	**1059**	1240	99%	**618**	496	76%
	32 bytes	**622**	620	99%	**438**	248	56%
Average		**1523**	1447	-	**907**	579	-

Table 5.6: Local Summation Algorithm.

table 5.6. Once again the estimations for the localized space 1-2 turned out to be very optimistic. A reasonable explanation for this was given in section 5.3.1, and the same principle applies in this case.

In all of the cases the tile-pass have correctly identified the optimal tiling to include both loops in the nest. On average the reduction in the number of off-chip accesses can be calculated to be $100 \cdot (1523 - 907)/1523 = 40\%$.

5.4 Summary.

The implemented tile-pass has been tested on four benchmarks. These benchmarks are the SOR-algorithm, the CONV-algorithm, a matrix multiplication algorithm and an code kernel commonly referred to as the local summation problem. All but the matrix multiplication algorithm are algorithms typically used in tasks such as image processing applications and DSP applications. The benchmarks have been chosen for the evaluation of the tile-pass because they contain some references with a particular kind of memory access patterns. The reason for this is that it might be beneficial to tile applications in which there exist references with this kind of memory access behaviour.

The used estimation technique have proven to be fairly accurate in some cases, but also very unprecise in others. There seemed to exist a trend that the estimations matched the simulation results on fully associative caches very well, while the simulations on direct mapped caches were more or less unprecise. This has however, come as no surprise as the influence of conflict misses are not considered in the estimation technique. The results of estimating the number of off-chip accesses can thus be said to fit very well with what was expected.

The tile-pass has furthermore proven to be able to correctly identify the most beneficial iteration space to localize. A test performed on the matrix multiplication algorithm has shown that the pass is also able to identify the near optimal tile-size in this case.

Results obtained by running the tile-pass as well as simulations on the CONV algorithm, have stressed the need for performing padding when using direct mapped caches, in some cases. The obtained results indicated that severe conflict misses occurred. As

a result of this no performance gains were obtained for the transformed program,when simulated on a direct mapped cache. The simulation of the same program on a fully associative cache revealed however that in this case a reduction in the number of off-cip accesses of 56% was obtained.

The average reductions in the number of off-chip accesses for the other three benchmarks was recorded to be: 76%, 11% and 40%.

Chapter 6

Conclusion.

In this thesis the topic of memory performance in embedded systems has been adressed. A survey covering a wide range of techniques for analyzing and improving the memory subsystem have been presented. Among the techniques for improving performance a distinction between memory hierarchy design on the one hand and control and memory-layout transformations on the other hand has been made. The effect on both power as well as performance has furthermore been taken into account.

Based on the information obtained in the survey it has been chosen to implement a compiler pass capable of carrying out a high-level tiling transformation on a given application code. This constructed pass is apart from being able to perform the actual transformation also capable of analyzing the given code for potential unexploited data-reuse. The information obtained by this analyzing step makes it possible to carry out a tiling specifically tuned for the given application and the underlying memory subsystem.

A thorough presentation of the theories which constitute the backbone of the selected implementation has also been provided. These theories deals among other things with the topics of identifying and evaluating potential unexploited data reuse in a nested loop. In order to carry out these tasks a partitioning of the array references inside the nested loop is performed. This partitioning makes it possible together with the presented mathematical approaches for quantification of reuse, to calculate appropriate tile-sizes. These tile parameters can subsequently be used for the actual transformation. A side effect of carrying out this task is that an estimate on the number of off-chip accesses for a particular application also can be obtained.

In order to evaluate and test the dueability of the constructed tile-pass a search among available parser and simulator tools has been conducted. This search has resulted in the establishment of a framework consisting of the SUIF1 compiler system and the SimpleScalar simulator. This framework is very well suited to be used in the context of this project as the SUIF1 system allows for a parsing of the application code, whereas information pertaining to the number of off-chip accesses can be obtained by SimpleScalar. The only drawback of these tools is that especially the SUIF1 compiler system has turned out to be somewhat unstable.

In order to test the capability of the tile-pass a number of different tests have been run on four different real-world applications. The obtained estimates on the number of off-chip accesses have not surprisingly turned out to be fairly accurate when dealing with

fully associative caches. When the simulations are run on memory configurations with direct mapped caches however, the estimates are less accurate. This behaviour is also to be expected as the implemented estimation technique does not take potential conflict misses into account.

The tile-pass has furthermore been able to correctly identify the most optimal loops to tile in all of the completed tests. The also performed calculations of the optimal tile-sizes have turned out to be near the optimal choice. The most promising of the conducted tests have shown that a 76% average reduction in the number of off-chip accesses were obtainable for a matrix multiplication algorithm, on a direct mapped cache. Furthermore a 56% average reduction in the number of off-chip accesses were obtained for the CONV algorithm on a fully associative cache.

It can thus be concluded that it, at least for some applications, definitely is possible to reduce the gap between memory and processor performance by the use of control transformations. Also the use of memory layout transformations might prove to be very beneficial for some cases. In this context especially the use of padding has proven to be very useful for many purposes [34]. Extensions such as the ability to carry out interchange transformations and enhanced estimation techniques can also definitely provide even better possibilities for achieving improved memory behaviour. The partitioning of references which have been implemented in the tile pass provides excellent opportunities for such extensions.

Bibliography

[1] Tanja Van Achteren, Rudy Lauwereins, K. U. Leuven, and Francky Catthoor. Systematic data reuse exploration methodology for irregular access patterns. *IEEE*, 2000.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high performance computing. *ACM*, 1994.

[3] P. Baglietto, M. Maresca, M. Migliardi, and N. Zingirian. Image processing on high performance risc systems. -, 1995.

[4] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. Synthesis of application specific memories for power optimization in embedded systems. *ACM*, 2000.

[5] F. Catthoor and A. Vandecapelle. Dtse script illustrated on cavity detection demonstrator. -, 2000.

[6] Chaitali Chakrabarti. Cache design and exploration for low power embedded systems. *IEEE*, 2001.

[7] Alok Choudhary, Mahmut Kandemir, J. Ramanujam, and P. Banerjee. A framework for interprocedural locality optimization using both loop and data layout. -, 10.

[8] Michael Cierniak and Wei Li. Unifying data and control transformations for distributed shared-memory machines. *ACM*, 1995.

[9] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. *ACM*, 1995.

[10] Vishal Dalal and C. P. Ravikumar. Software power optimization in an embedded system. *IEEE*, 2000.

[11] SUIF Stanford University Intermidiate Format. http://www.suif.stanford.edu/suif/suif1. -, 1993.

[12] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Automatic analytical modelling for the estimation of cache misses. -, 1996.

[13] M. R. Garey and D. S. Johnson. Computers and intractibility - a guide to the theory of np-completeness. -, 1979.

[14] K. Ghoose and M. B. Kamble. Analytical energy dissipation models for low power caches. -, 1997.

[15] Tony D. Givargis, Jorg Henkel, and Frank Vahid. Interface and cache power exploration for core-based embedded systems design. *IEEE*, 1999.

[16] Joerg Henkel and Yanbing Li. Avalanche: An environment for design space exploration and optimization of low power embedded systems. -, 10.

[17] Patrick Hicks, Matthew Walnock, and Robert Michael Owens. Analysis of power consumption in memory hierarchies. *ACM*, 1997.

[18] Chung-Hsing Hsu and Ulrich Kremer. A stable and efficient loop tiling algorithm. -, 1999.

[19] Simplescalar : http://www.cs.wisc.edu/ mscalar/simplescalar.html. -. -, 1995.

[20] http://www.eembc.org/. Embedded microprocessor benchmark consortium,. *Develops and certifies real-world benchmarks and benchmark scores to help designers select the right embedded processors for their systems.*, 0.

[21] SPEC : http://www.specbench.org/. Standard performance evaluation corporation. -, 0.

[22] Mahmut Kandemir and Ismail Kadayiff. Compiler directed selection of dynamic memory layouts. -, 10.

[23] Makoto Kobayashi. Memory reference metrics and instruction trace sampling. -, 0.

[24] Mihai T. Lazarescu, Jwahar R. Bammi, Edwin Harcourt, Luciano Lavagno, and Marcello Lajolo. Compilation based software performance estimation for system level design. *IEEE*, 2000.

[25] Yanbing Li and Wayne H. Wolf and. Hardware software co-synthesis with memory hierarchies. -, 1999.

[26] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modelling. *IEEE*, 1995.

[27] Yanhong A. Liu and Scott D. Stoller. Loop optimization for aggregate array computations. -, 10.

[28] Ching long Su and Alvin M. Despain. Cache designs for energy efficiency. *IEEE*, 1995.

[29] Gabriele Luculli and Alberto Sangiovanni-Vincentelli. Analysis of dsp kernel software by implicit cache simulation. *IEEE*, 2001.

[30] K. Masselos, F. Catthoor, C. E. Goutis, and H. De Man. System-level power optimizing data-flow transformations for multimedia applications realized on programmable multimedia processors. *IEEE*, 1999.

[31] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM*, 1996.

[32] Bellas Nikolaos E, Ibrahim N. Hajj, and Constantine D. Polychronopoulos. Using dynamic cache management techniques to reduce energy in general purpose processors. *IEEE*, 2000.

[33] Vijay S. Pai and Sarita Adve. Code transformations to improve memory parallelism. -, 10.

[34] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Memory organization for improved data cache performance in embedded processors. -, 1996.

[35] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Architectural exploration and optimization of local memory in embedded systems. -, 1997.

[36] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. -, 1997.

[37] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Local memory exploration and optimization in embedded systems. -, 1999.

[38] C.Y. Park and A. P. Shaw. Experiments with a program timing tool based on source level tming schema. *IEEE*, 1991.

[39] Massoud Pedram. Power optimization and management in embedded systems. -, 10.

[40] Stefan Petters, Anette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Farber. The rear framework for emulation and analysis of embedded hard real-time systems. *IEEE*, 1997.

[41] Stefan M. Petters and Georg Farber. Bounding the execution time of real-time tasks on modern processors. *IEEE*, 2000.

[42] M. Powell, S. H. Yang, K. Roy, B. Falsafi, and T. Vijaykumar. Dri cache: a design for power efficient instruction cache. -, 2000.

[43] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical recipes in c: The art of scientific computing. -, 1992.

[44] P. Puschner and A. V. Schedl. Computing maximum task execution times - a graph based approach. -, 1995.

[45] Alex Ramirez, Luiz Andre Barosso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transactio processing workloads. -, 0.

[46] Wen-Tsong Shiue. Optimizing memory bandwith with ilp based memory exploration and assignment for low power embedded systems. *IEEE*, 2000.

[47] Wen-Tsong Shiue and Chaitali. Memory design and exploration for low power embedded systems. *IEEE*, 1999.

[48] Cache Simulator. http://www.ece.cmu.edu/ece548/tools/dinero/src. -, 1998.

[49] Tajana Simunic, Luca Benini, Giovanni De Micheli, and Mat Hans. Source code optimization and profiling of energy consumption in embedded systems. *IEEE*, 2000.

[50] M. R. Stan and W. P. Burleson. Bus invert coding for low power i/o. -, 1995.

[51] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy driven integrated hardware-software optimizations using simplepower. *ACM*, 2000.

[52] Reinhard Wilhelm and Christian Ferdinand. The abstract interpretation approach. -, 10.

[53] Fabian Wolf and Rolf Ernst. Data flow based cache prediction using local simulation. *IEEE*, 2000.

[54] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *ACM*, 1991.

[55] Yong Yan, Xiaodong Zhang, and Zhao Zhang. Cacheminer: A runtime approach to exploit cache locality on smp. *IEEE*, 1997.

[56] N. Zervas, K. Tatas, A. Argyriou, M. Dasigenis, and D. Soudris. Memory hierarchy optimization of multinedia applications on programmable embedded cores. *IEEE*, 2001.

Appendix A

Code.

A.1 my_tile.cc

```
#include <stdlib.h>
#include <suif1.h>
#include "for_loops.h"
#include "for_loop_transform.h"
#include <dependence.h>
#include <suif1/cmdparse.h>

int line_size;
int cache_size;
int user_first_tile_loop;
int user_req_ws_no;
int user_req_big_ws_no;

/**********************************************************************
 * For each procedure,                                                *
 * Need to do fill_in_access because of include_for                   *
 **********************************************************************/
void do_proc(tree_proc * tp){
/* The for_loops class contains a list of tree_for's in a procedure
   and it is now initialized with the tree_for's present in the procedure */

    proc_sym *proc_sym_p = tp->proc();
    printf("=====Processing the %s procedure=====\n",proc_sym_p->name());
    int nested_loop_number = 0;
    fill_in_access(tp);

    for_loops proc_for_loops(tp->body());
    nested_loop_list_iter nll_iter(proc_for_loops.nested_loop_list_p);
    while (!nll_iter.is_empty()){
        nested_loop *nl_p = nll_iter.step();
        nested_loop_number++;
        printf("====Processing nested loop number %d=====\n",nested_loop_number);
        for_loop_transform flt(nl_p);
        flt.calc_dependence();
        int depth = nl_p->get_depth();
        flt.locality_analysis();
        //flt.print_access_matrices();
```

```
flt.generate_uni_gen_sets();
//flt.print_uni_gen_sets();
flt.generate_eq_classes();

// Uncomment the following to get information on the partitioning.
//flt.print_uni_gen_sets_and_eq_classes();

flt.find_best_tile_region();

// Uncomment the following to get statistics, and specific estimations.
//flt.print_collected_stat();
//flt.print_est_for_all_spaces();
flt.print_optimal_tile_est();

// Code for handling command line argument user_first_loop.
if(user_first_tile_loop > flt.get_depth()){
    printf("The user defined first_loop : %d is greater than nest depth : %d\n",
           user_first_tile_loop,flt.get_depth());
    printf("No tiling performed - trying next nested loop\n");
    continue;
}
if(user_first_tile_loop == flt.get_depth()){
    printf("The user defined first_loop : %d equals the nest depth : %d\n",
           user_first_tile_loop,flt.get_depth());
    printf("No tiling performed - trying next nested loop\n");
    continue;
}
if(user_first_tile_loop){
    if(flt.ok_to_tile(user_first_tile_loop,flt.get_depth())){
        printf("Performing user requested tiling of loops %d-%d\n",
               user_first_tile_loop,flt.get_depth());
        flt.tile(flt.get_optimal_tiling_for_first_loop(user_first_tile_loop));
        flt.print_optimal_tile_est_for_loop(user_first_tile_loop);
        continue;
    }
    else{
        printf("User request denied : Not legal to tile loop %d",
             user_first_tile_loop);
        continue;
    }
}

// Code for handling the tiling with a specific tile-size, for the
// estimated best localized iteration space.
// The option -ws is handled. If its value for instance is
// 3, then the third best tile size will be used.
boolean no_legal_tiling_found = TRUE;
int best_first_loop;
int i=1;
if(user_req_ws_no){
    while(i<=depth && no_legal_tiling_found){
        // best_first_loop is set to hold the number of the first loop
        // included in the optimal tiling.
        best_first_loop = flt.get_optimal_tiling_loop(i);
        if (flt.ok_to_tile(best_first_loop,depth)){
            no_legal_tiling_found = FALSE;
```

```
            int optimal_tile_size =
                flt.get_optimal_tile_size_for_loop(best_first_loop);
            int ts_step = flt.get_tile_size_step();
            int no_of_tested_tile_sizes = optimal_tile_size/ts_step;
            if(no_of_tested_tile_sizes == 0)
                no_of_tested_tile_sizes++;
            tile_size_info_list_iter
                tsil_iter(flt.get_tile_size_local_space(best_first_loop));
            int wanted_ws_size_no = no_of_tested_tile_sizes - user_req_ws_no + 1;
            if(wanted_ws_size_no<1){
                printf("Requested ws size is too small\n");
                continue;
            }
            int current_no = 0;
            tile_size_info *tsi_p;
            while(!tsil_iter.is_empty() && current_no<wanted_ws_size_no){
                tsi_p = tsil_iter.step();
                current_no++;
            }
            optimal_tiling_info oti =
                optimal_tiling_info(tsi_p,best_first_loop);
            flt.tile(&oti);
            printf("Performed requested tiling with specific ws-size\n");
            printf("The tiling was performed for local space %d-%d\n",
                   best_first_loop,flt.get_depth());
            printf("The %d. best tile-size was requested\n",
                   user_req_ws_no);
            printf("The parameters for this tiling are :\n");
            flt.print_tile_size_info(tsi_p);
        }
        i++;
    }
    if(no_legal_tiling_found)
        printf("No legal tiling was found, trying next loop\n");
    continue;
}

// Code for handling -bws command-line option. This int is stored in
// user_req_big_ws_no and indicates that a tiling with tile-sizes of
// user_req_big_ws_no*(line_size/data_elem_size) bigger than the
// calculated optimal tiles for the optimal localization space should
// be performed.
no_legal_tiling_found = TRUE;
i=1;
if(user_req_big_ws_no){
    while(i<=depth && no_legal_tiling_found){
        // best_first_loop is set to hold the number of the first loop
        // included in the optimal tiling.
        best_first_loop = flt.get_optimal_tiling_loop(i);
        if (flt.ok_to_tile(best_first_loop,depth)){
            flt.set_selected_first_loop(best_first_loop);
            int optimal_tile_size =
                flt.get_optimal_tile_size_for_loop(best_first_loop);
            int ts_step = flt.get_tile_size_step();
            int req_tile_size =
                optimal_tile_size + user_req_big_ws_no*ts_step;
```

```
            int no_of_tested_tile_sizes = optimal_tile_size/ts_step;
            // dummy_no is not used for anything.
            int dummy_no = 100;
            optimal_tiling_info oti(best_first_loop, dummy_no,
                                    dummy_no, req_tile_size);
            flt.tile(&oti);
            no_legal_tiling_found = FALSE;
            printf("User requested tiling with tile-size bigger than the optimal was performed\n");
            printf("Tiling was performed for local-space %d-%d\n",
                   best_first_loop,flt.get_depth());
            printf("The applied tile-size is %d\n",req_tile_size);
            printf("No estimations provided - transformation only for sim. purposes\n");
        }
        i++;
    }
    if(no_legal_tiling_found)
        printf("No legal tiling was found, trying next loop\n");
    continue;
}

// Code for handling the case when no command-line arguments are given.
no_legal_tiling_found = TRUE;
i=1;
while(i<=depth && no_legal_tiling_found){
    // best_first_loop is set to hold the number of the first loop
    // included in the optimal tiling.
    best_first_loop = flt.get_optimal_tiling_loop(i);
    if (flt.ok_to_tile(best_first_loop,depth)){
        flt.set_selected_first_loop(best_first_loop);
        if(!(flt.no_tile_made_WS_fit_in_cache[best_first_loop-1])){
            if(best_first_loop == flt.get_depth()){
                printf("The optimal tiling consisted only of the innermost loop\n");
                printf("Therefore no tiling performed\n");
                }
            else{
                optimal_tiling_info * oti_p = flt.get_optimal_tiling_info_no(i);
                flt.tile(oti_p);
                printf("The tiling was performed for local-space %d-%d\n",
                       best_first_loop,flt.get_depth());
                fflush(stdout);
                printf("The parameters for the tiling are:\n");
                flt.print_tile_size_info(oti_p);
            }
        }
        else{
          printf("The WS for the optimal loops to tile fit in the cache\n");
          printf("Therefore no tiling performed\n");
        }
        no_legal_tiling_found = FALSE;
    }
    i++;
}
if(no_legal_tiling_found){
    printf("No legal tiling was found, trying next loop\n");
}
```

```
        // Prints the estimated no. of off-chip acc. and WS-sizes for all
        // localized spaces, with tile_size's = 1*,2*,...i*line_size :
        // flt.print_est_for_all_spaces();
        fflush(stdout);
    }
}


/**********************************************************************
 *  Main: set the tile size if supplied, otherwise it is 64.            *
 **********************************************************************/
main(int argc, char * argv[])
{
    static cmd_line_option my_opt_table[] = {
      {CLO_INT, "-ls", "32", &line_size},
      {CLO_INT, "-cs", "2048", &cache_size},
      {CLO_INT, "-fl", "0", &user_first_tile_loop},
      {CLO_INT, "-ws", "0", &user_req_ws_no},
      {CLO_INT, "-bws", "0", &user_req_big_ws_no}
    };

    parse_cmd_line(argc, argv, my_opt_table,
                   sizeof(my_opt_table)/sizeof(cmd_line_option));
    start_suif(argc, argv);

    printf("line_size : %d    cache_size : %d\n\n",line_size,cache_size);
    suif_proc_iter(argc, argv, do_proc, TRUE);
    printf("\n");
    fflush(stdout);
}
```

A.2 class for_loops

```
// File for_loops.h

#ifndef FOR_LOOPS_H
#define FOR_LOOPS_H

#include "nested_loop.h"

/* This declaration constructs a class named nested_loop_list_e
   which contains variables and methods for creating and manipulating
   a linked list with elements of type pointer to nested_loop.
   An iterator class named nested_loop_list_iter for traversing the
   list is also constructed */
DECLARE_LIST_CLASS(nested_loop_list, nested_loop*);

class for_loops{
    private:
        int no_of_loops;
    public:
        nested_loop_list *nested_loop_list_p;
        for_loops(tree_node_list *tnl);
        void find_for_loops(tree_node_list *tnl);
};
```

```
#endif
#include <iostream>
#include "for_loops.h"
using namespace std;

// for_loops.cc

// An instance of the for_loops object is created for each procedure
// handled in do_proc. After initialization this for loop instance will
// contain a linked list of all the perfectly nested for loops in the
// procedure
for_loops::for_loops(tree_node_list *tnl){
        no_of_loops=0;
        nested_loop_list_p = new nested_loop_list();
        find_for_loops(tnl);
    }

void for_loops::find_for_loops(tree_node_list *tnl){
    tree_node_list_iter iter(tnl);
    while(!iter.is_empty()) {

        tree_node * tn = iter.step();
        switch(tn->kind()) {
        case TREE_FOR:
            {
            tree_for * tnf = (tree_for *)tn;
            // An instance of a nested loop is created and the linked list
            // in this object is already initialized with this "tree_for*"
            nested_loop *nl_p = new nested_loop(tnf);

            // find_perf_nested_loop(tn) is called on nl_p, this function
            // member can only be called once on an nested_loop instance
            if (nl_p->is_legal_loop()){
              nested_loop_list_p->append(nl_p);
            }
            break;
            }

        case TREE_IF:
            {
            tree_if * tni = (tree_if *)tn;
            find_for_loops(tni->then_part());
            find_for_loops(tni->else_part());
            break;
            }

        case TREE_LOOP:
            {
            tree_loop * tnl = (tree_loop *)tn;
            find_for_loops(tnl->body());
            break;
            }

        case TREE_BLOCK:
            {
            tree_block * tnb = (tree_block *)tn;
```

```
            find_for_loops(tnb->body());
            break;
            }

        case TREE_INSTR:
            break;   // TREE_INSTR are the leaves of the IR-representation

        default:
            assert(0);
            break;
        }
    }
    return;
}
```

A.3 class nested_loop

```
// File nested_loop.h

#ifndef NESTED_LOOP_H
#define NESTED_LOOP_H

#include <suif1.h>

/* This declaration constructs a class named nested_loop_list_e
   which contains variables and methods for creating and manipulating
   a linked list with elements of type nested_loop*.
   An iterator class named nested_loop_list_iter for traversing the
   list is also constructed */
DECLARE_LIST_CLASS(tree_for_list, tree_for*);

class nested_loop{
    private:
        int depth;
        int perf_nested_loop;
        int legal_loop;
        tree_for_list *tree_for_list_p;
        void find_perf_nested_loop(tree_node_list *tnl,boolean found_1_instr);
        void find_perf_nested_loop(tree_node_list *tnl);
        void find_nested_loop(tree_node_list *tnl,boolean inside_if);
        void find_nested_loop(tree_node_list *tnl);

    public:
        nested_loop(tree_node *tn);
        int is_a_perf_nested_loop();
        boolean is_legal_loop();
        int get_depth();
        tree_for_list_iter *tfl_iter_p;
        tree_for_list* get_tree_for_list_p();
};
#endif

// File nested_loop.cc
#include <iostream>
#include "nested_loop.h"
```

```
using namespace std;

nested_loop::nested_loop(tree_node* tn){
    tree_for_list_p = new tree_for_list(); // _e or nothing??
    // check that tn actually is a tree_for*
    tree_for *tnf = (tree_for *)tn;
    tree_for_list_p->append(tnf);
    depth = 1;
    legal_loop = TRUE;
    tfl_iter_p = new tree_for_list_iter(tree_for_list_p);
    // The initialization of the nested_loop instance is done entirely
    // inside the constructor, if the loop turned out not to be
    // perfectly nested the instance must be deleted.
    find_nested_loop(tnf->body());
}

void nested_loop::find_perf_nested_loop(tree_node_list *tnl){
  find_perf_nested_loop(tnl,FALSE);
}

void nested_loop::find_perf_nested_loop(tree_node_list *tnl,boolean found_1_instr){
    int found_1_for_in_tree_node_list = FALSE;
    int found_1_instr_in_tree_node_list = FALSE;
    perf_nested_loop = TRUE;
    tree_node_list_iter iter(tnl);
    while(!iter.is_empty() && perf_nested_loop){
        tree_node * tn = iter.step();
        switch(tn->kind()) {
        case TREE_FOR:
            {
            if (found_1_for_in_tree_node_list ||
                found_1_instr_in_tree_node_list ||
                found_1_instr){
              perf_nested_loop = FALSE;
              break;
            }
            found_1_for_in_tree_node_list = TRUE;
            depth++;
            tree_for * tnf = (tree_for *)tn;
            tree_for_list_p->append(tnf);
            find_perf_nested_loop(tnf->body());
            break;
            }

        case TREE_IF:
            perf_nested_loop = FALSE;
            break;

        case TREE_LOOP:
            perf_nested_loop = FALSE;
            break;

        case TREE_BLOCK:
            {
            tree_block * tnb = (tree_block *)tn;
```

```
            find_perf_nested_loop(tnb->body());
            break;
            }

        case TREE_INSTR:
            found_1_instr_in_tree_node_list = TRUE;
            if (found_1_for_in_tree_node_list)
              perf_nested_loop = FALSE;
            break;

        default:
            assert(0);
            break;
        }
    }
    return;
}


// This procedure finds a nest with only simple instructions allowed
// inbetween the different for-levels. That is, instructions ... load, stores,
// and if statements.

void nested_loop::find_nested_loop(tree_node_list *tnl){
  find_nested_loop(tnl,FALSE);
}

void nested_loop::find_nested_loop(tree_node_list *tnl,boolean inside_if){
    boolean found_1_for_in_tree_node_list = FALSE;
    tree_node_list_iter iter(tnl);
    while((!iter.is_empty()) && legal_loop){
        tree_node * tn = iter.step();
        switch(tn->kind()) {
        case TREE_FOR:{
            if (found_1_for_in_tree_node_list){
              legal_loop = FALSE;
              break;
            }
            if (inside_if){
              legal_loop = FALSE;
              break;
            }
            found_1_for_in_tree_node_list = TRUE;
            depth++;
            //cout << "Entering an inner for-loop\n";
            tree_for * tnf = (tree_for *)tn;
            //          tnf->print();
            tree_for_list_p->append(tnf);
            find_nested_loop(tnf->body(),FALSE);
            break;
        }

        case TREE_IF:{
            tree_if *tf = (tree_if *)tn;
            find_nested_loop(tf->then_part(),TRUE);
            find_nested_loop(tf->else_part(),TRUE);
```

```
            break;
        }

        case TREE_LOOP:{
            legal_loop = FALSE;
            break;
        }

        case TREE_BLOCK:{
            tree_block * tnb = (tree_block *)tn;
            find_nested_loop(tnb->body(),inside_if);
            break;
        }

        case TREE_INSTR:
            break;

        default:
            assert(0);
            break;
        }
    }
    return;
}

int nested_loop::is_a_perf_nested_loop(){
    return perf_nested_loop;
}

boolean nested_loop::is_legal_loop(){
    return legal_loop;
}

int nested_loop::get_depth(){
  return depth;
}

tree_for_list* nested_loop::get_tree_for_list_p(){
  return tree_for_list_p;
}
```

A.4 locality.h

```
// locality.h

#include <suif1.h>
#include <dependence.h>
#include <suifmath.h>
#include "my_list.h"

#ifndef LOCALITY_H
#define LOCALITY_H

typedef int dimension;
```

```
#define DIMENSION_NOT_INCLUDED 0
#define DIMENSION_INCLUDED 1

class array_ref;
class uni_gen_set;
class array_uni_gen_set;
class eq_class;
class local_space_eq_classes;
class iteration_space;

extern int line_size;
extern int cache_size;

enum reuse {SELF, GROUP_SPATIAL, GROUP_TEMPORAL};

class member{
public:
    member();
    in_array *array_instr_p;
    fract_vector *constant_vector_p;
    boolean operator==(member m);
    member(member *m_p);
    member *next_e;
    member *next() const        { return next_e; }
};

DECLARE_LIST_CLASS(array_ref_list, array_ref*);
DECLARE_LIST_CLASS(member_list, member*);
DECLARE_LIST_CLASS(uni_gen_set_list, uni_gen_set*);
DECLARE_LIST_CLASS(array_uni_gen_set_list, array_uni_gen_set*);
DECLARE_LIST_CLASS(eq_class_list, eq_class*);
DECLARE_LIST_CLASS(dimension_list, dimension);

class array_ref{
private:
    in_array *array_instr_p;
    array_info *array_info_p;
    matrix *matrix_p;
    fract_vector *fract_vector_p;
    int element_size;
public:
    array_ref(in_array *instr_p, int depth, tree_for** tree_for_p_table);
    ~array_ref();
    in_array* get_array_instr_p();
    array_info *get_array_info_p();
    matrix* get_matrix_p();
    fract_vector *get_fract_vector_p();
    int get_elem_size();
};

class uni_gen_set{
private:
    int reference_depth;
    matrix *matrix_p;
    matrix *matrix_s_p;
```

```
    member_list *member_list_p;
    int no_of_members;
    int nest_depth;
    vector_space *self_spatial_vs_p;
    vector_space *self_temporal_vs_p;
    iteration_space* R_ss_p;
    iteration_space* R_st_p;
    int element_size;
    int last_row;
    int subscript_depth;

public:
    uni_gen_set(in_array *array_instr_p, matrix *matr_p,
                fract_vector *fv_p, int reference_depth, int elem_size);
    ~uni_gen_set();
    local_space_eq_classes **local_space_eq_classes_table;
    int get_no_of_members();
    int get_reference_depth();
    int get_elem_size();
    int get_index_var_factor();
    int get_nest_depth();
    int get_subscript_depth();
    member_list *get_member_list_p();
    boolean contains(matrix *matr_p);
    void add_member(in_array *array_instr_p, matrix *matr_p, fract_vector *fract_vector_p);
    iteration_space* get_R_ss_p();
    iteration_space* get_R_st_p();
    matrix *get_matrix_p();
    matrix *get_matrix_s_p();
    void print_uni_gen_set();
    void print_uni_gen_set_and_eq_classes();
};


class all_array_uni_gen_set{
private:
    array_uni_gen_set_list *array_uni_gen_set_list_p;
    int reference_depth;
public:
    all_array_uni_gen_set(int reference_depth);
    ~all_array_uni_gen_set();
    array_uni_gen_set_list *get_array_uni_gen_set_list_p();
    array_uni_gen_set_list *get_array_uni_gen_set_list(var_sym *var_sym_p);
    boolean exists(var_sym *var_sym_p);
    array_uni_gen_set *get(var_sym *var_sym_p);
    void append_array_uni_gen_set(var_sym *var_sym_p, int reference_depth);
    void print_all_array_uni_gen_set();
    void print_all_array_uni_gen_set_and_eq_classes();
};

class array_uni_gen_set{
private:
    var_sym *var_sym_p;
    uni_gen_set_list * uni_gen_set_list_p;
    int reference_depth;
public:
```

```
    array_uni_gen_set(var_sym *var_sym_p, int reference_depth);
    ~array_uni_gen_set();
    var_sym *get_var_sym_p();
    uni_gen_set_list *get_uni_gen_set_list_p();
    void print_array_uni_gen_set();
    void print_array_uni_gen_set_and_eq_classes();
};

class local_space_eq_classes{
private:
    iteration_space *L_space_p;
    iteration_space *intersection_L_Rst_p;
    iteration_space *intersection_L_Rss_p;
    eq_class_list *eq_class_list_p;
    int *acc_per_eq_class_in_loop;
    int no_of_eq_classes;
    int SS_but_not_ST_reuse_dim;
    boolean SS_reuse;
    int SS_reuse_factor;
    int reference_depth;
    int total_no_of_off_chip_acc;
    boolean intersections_equal;
    int *no_of_iter_in_loop;
    int no_of_ST_reuse_dims;
    int element_size;
    int index_var_factor;
    uni_gen_set *uniformly_gen_set_p;
    int nest_depth;
    int subscript_depth;

public:
    local_space_eq_classes(uni_gen_set *uni_gen_set_p,
                           int first_local_space_loop,
                           int *iter_in_loop);
    ~local_space_eq_classes();
    int get_total_no_of_off_chip_acc();
    int get_no_of_eq_classes();
    int get_no_of_ST_reuse_dims();
    int get_elem_size();
    int get_SS_reuse_factor();
    int get_index_var_factor();
    boolean get_SS_reuse();
    void print_eq_classes();
};

class eq_class{
private:
    member_list *member_list_p;
    uni_gen_set *uni_gen_set_p;
    reuse reuse_kind;
public:
    eq_class(uni_gen_set *uni_gen_set_p, member *member_p);
    void add(member *member_p);
    void print_members();
    void set_reuse(reuse reuse_kind);
};
```

```
class iteration_space{
private:
    dimension *dimension_space;
    int no_of_dimensions;
public:
    iteration_space(int first_loop_inside_local_space, int reference_depth);
    iteration_space(int reference_depth);
    iteration_space(fract_vector *fract_vector_p);
    ~iteration_space();
    void add_dimension(dimension dim); //crates/allocates a new dimension and inserts it
    iteration_space *intersection(iteration_space *it_space_p);
    iteration_space *union_space(iteration_space *it_space_p);

    // returns TRUE if "this" totally covers the argument.
    boolean contains(iteration_space *it_space_p);
    boolean equals(iteration_space *iteration_space_p);

    // Returns the depth of the nest.
    int get_no_of_dimensions();

    // Returns the number of dimensions that are included.
    int get_no_of_included_dims();

    // Returns TRUE if dimension i is included in the iteration space.
    boolean dimension_included(int i);

    // Returns the last dimension which is in this iteration space but not
    // in the arguments iteration space. If none => returns 0.
    int get_diff(iteration_space * it_space_p);

    void print_included_dimensions();
};

#endif
```

A.5 locality.cc

```
// locality.cc

#include "locality.h"

member::member(){
    array_instr_p = NULL;
    constant_vector_p = NULL;
    next_e = NULL;
}

boolean member::operator==(member m){
  if (array_instr_p == m.array_instr_p)
    return TRUE;
  else
    return FALSE;
}
```

```
member::member(member *m_p){
    array_instr_p = m_p->array_instr_p;
    constant_vector_p = m_p->constant_vector_p;
}

array_ref::array_ref(in_array *instr_p, int depth, tree_for **tree_for_p_table){
    array_instr_p = instr_p;
    element_size = (instr_p->elem_size())/8; // elem_size() returns size in bits
    // By calling the array_info constructor with the array instruction as
    // a parameter, access vectors representing the instruction is
    // generated. These access vectors constitute the access matrix for
    // the reference.
    array_info_p = new array_info(array_instr_p,1);
    int access_vec_no;

    //count returns the subscript depth of the array reference.
    int no_of_rows = array_info_p->count();
    matrix_p = new matrix(no_of_rows,depth);
    array_info_iter ai_iter(array_info_p);

    fract_vector_p = new fract_vector(no_of_rows);

    // Run through each subscript-expr. in the array reference. Each
    // sub.-expr. is represented by an access_vector.
    for(access_vec_no=0; !ai_iter.is_empty(); access_vec_no++){
        access_vector *access_vector_p = ai_iter.step();

        // Generate the constant vector for this subscript.
        int const_term = access_vector_p->con;
        fract fract_const_term(const_term);
        (*fract_vector_p)[access_vec_no] = fract_const_term;

        for(int index_var_no=0; index_var_no<depth; index_var_no++){
            fract temp, temp_abs;
            temp = fract(access_vector_p->val(tree_for_p_table[index_var_no]));
            temp_abs = temp.abs();

            // The elt() member returns the matrix element by reference (&)
            matrix_p->elt(access_vec_no,index_var_no) = temp_abs;
        }
    }
    assert(access_vec_no == no_of_rows);
}

array_ref::~array_ref(){
    delete array_info_p;
    delete matrix_p;
    delete fract_vector_p;
}

in_array* array_ref::get_array_instr_p(){
    return array_instr_p;
}

array_info *array_ref::get_array_info_p(){
    return array_info_p;
```

```
}

matrix* array_ref::get_matrix_p(){
    return matrix_p;
}

fract_vector* array_ref::get_fract_vector_p(){
    return fract_vector_p;
}

int array_ref::get_elem_size(){
    return element_size;
}
```

A.5.1 class uni_gen_set

```
// Uniformly generated sets are represented by uni_gen_set's :
uni_gen_set::uni_gen_set(in_array *array_instr_p, matrix *matr_p,
                         fract_vector *fv_p, int ref_depth, int elem_size){
    element_size = elem_size;
    no_of_members = 1;
    member_list_p = new member_list;
    member *member_p = new member;
    member_p->array_instr_p = array_instr_p;
    member_p->constant_vector_p = fv_p;
    member_list_p->append(member_p);
    matrix_p = matr_p;
    reference_depth = ref_depth;

    // The n() member function returns the number of columns in the matrix
    // and this equals the number of for-loops in the nest. (m x n matrix)
    nest_depth = matrix_p->n();

    // last_row == subscript_depth.
    last_row = matrix_p->m();// last_row equals the subscript depth of the
                             // array-ref. That is, A[i] => last_row=1.

    subscript_depth = last_row;

    matrix_s_p = new matrix(*matrix_p);

    R_st_p = new iteration_space(reference_depth);
    R_ss_p = new iteration_space(reference_depth);

    // This constructor variant generates a fract_vector of length
    // reference_depth and with all elements equal to zero.
    fract_vector fv_zero = fract_vector(nest_depth);

    // Replace the last row of matrix_s_p with all zero's.
    matrix_s_p->set_row(last_row-1, fv_zero);

    vector_space self_temporal_vs = matrix_p->kernel();
    vector_space self_spatial_vs = matrix_s_p->kernel();

    self_temporal_vs_p = new vector_space(self_temporal_vs);
```

```
    self_spatial_vs_p = new vector_space(self_spatial_vs);

    // The basis() function of the vector_space class converts the
    // vector space into a list of fract_vector's and returns it.
    fract_vector_list *self_temporal_fract_vector_list_p;
    self_temporal_fract_vector_list_p = self_temporal_vs_p->basis();

    fract_vector_list_iter fvl_st_iter(self_temporal_fract_vector_list_p);
    while(!fvl_st_iter.is_empty()){
        int first_non_zero;
        int i = 0;
        int indicator = 0;
        fract_vector *temp_fract_vector_p = fvl_st_iter.step();

        assert(nest_depth == temp_fract_vector_p->n());
        // Finding the dimension with temporal reuse.
        for (i=0; i<reference_depth; i++){
            if ((*temp_fract_vector_p)[i] != 0){
                first_non_zero = i;
                indicator++;
            }
        }
        // Only if the fract_vector had a single non_zero element we let
        // it count as true temporal reuse.
        if (indicator == 1){
            R_st_p->add_dimension((dimension)first_non_zero);
        }
    }

    fract_vector_list *self_spatial_fract_vector_list_p;
    self_spatial_fract_vector_list_p = self_spatial_vs_p->basis();

    fract_vector_list_iter fvl_ss_iter(self_spatial_fract_vector_list_p);
    while(!fvl_ss_iter.is_empty()){
        int first_non_zero;
        int i = 0;
        int indicator = 0;
        fract_vector *temp_fract_vector_p = fvl_ss_iter.step();
        assert(nest_depth == temp_fract_vector_p->n());
        // Finding the dimension with temporal reuse.
        for (i=0; i<reference_depth; i++){
            if ((*temp_fract_vector_p)[i] != 0){
                first_non_zero = i;
                indicator++;
            }
        }
        // Only if the fract_vector had a single non_zero element we let
        // it count as true temporal reuse.
        if (indicator == 1){
            R_ss_p->add_dimension((dimension)first_non_zero);
        }
    }
    local_space_eq_classes_table = new local_space_eq_classes*[reference_depth];
}

uni_gen_set::~uni_gen_set(){
```

```
    member_list_iter ml_iter(member_list_p);
    delete member_list_p;
    delete R_st_p;
    delete R_ss_p;
    delete [] local_space_eq_classes_table;
}

matrix *uni_gen_set::get_matrix_p(){
    return matrix_p;
}

matrix *uni_gen_set::get_matrix_s_p(){
    return matrix_s_p;
}

int uni_gen_set::get_reference_depth(){
    return reference_depth;
}

int uni_gen_set::get_elem_size(){
    return element_size;
}

int uni_gen_set::get_subscript_depth(){
    return subscript_depth;
}

void uni_gen_set::add_member(in_array *array_instr_p, matrix *matr_p, fract_vector *fract_vector_p){
    if (!(*matrix_p == *matr_p)){
        printf("The reference does not belong to this uniformly generated set\n");
        assert(0);
        return;
    }
    no_of_members++;
    member *member_p = new member;
    member_p->array_instr_p = array_instr_p;
    member_p->constant_vector_p = fract_vector_p;
    member_list_p->append(member_p);
}

void uni_gen_set::print_uni_gen_set(){
    printf("The uniformly generated set contains %d references :\n",
           no_of_members);
    printf("The members of the set are :\n");
    member_list_iter ml_iter(member_list_p);
    while(!ml_iter.is_empty()){
        member *member_p = ml_iter.step();
        print_array_access(member_p->array_instr_p);
    }
    printf("The access matrix for the set is :\n");
    matrix_p->print();
    printf("The references exhibit self-temporal reuse in loops :  ");
    R_st_p->print_included_dimensions();
    printf("The references exhibit self-spatial reuse in loops :  ");
    R_ss_p->print_included_dimensions();
    printf("\n");
```

```
}

void uni_gen_set::print_uni_gen_set_and_eq_classes(){
    printf("The uniformly generated set contains %d references :\n",
           no_of_members);
    printf("The members of the set are :\n");
    member_list_iter ml_iter(member_list_p);
    while(!ml_iter.is_empty()){
        member *member_p = ml_iter.step();
        print_array_access(member_p->array_instr_p);
    }
    printf("The access matrix for the set is :\n");
    matrix_p->print();
    printf("The references exhibit self-temporal reuse in loops :  ");
    R_st_p->print_included_dimensions();
    printf("The references exhibit self-spatial reuse in loops :  ");
    R_ss_p->print_included_dimensions();
    for (int i=0; i<reference_depth; i++){
        printf("The generated eq_class'es for local space %d-%d :\n",
               i+1,reference_depth);
        local_space_eq_classes_table[i]->print_eq_classes();
    }
    printf("\n");
}

boolean uni_gen_set::contains(matrix *matr_p){
    if (*matrix_p == *matr_p)
        return TRUE;
    else
        return FALSE;
}

int uni_gen_set::get_no_of_members(){
    return no_of_members;
}

int uni_gen_set::get_nest_depth(){
    return nest_depth;
}

member_list *uni_gen_set::get_member_list_p(){
    return member_list_p;
}

iteration_space * uni_gen_set::get_R_st_p(){
    return R_st_p;
}

iteration_space * uni_gen_set::get_R_ss_p(){
    return R_ss_p;
}
```

A.5.2 auxiliary

```
// An all_array_uni_gen_set instance is created for each loop in the nest.
```

```
// This class contains a list which represents all the different arrays
// that are present in the particular loop.

all_array_uni_gen_set::all_array_uni_gen_set(int ref_depth){
    array_uni_gen_set_list_p = new array_uni_gen_set_list;
    reference_depth = ref_depth;
}

all_array_uni_gen_set::~all_array_uni_gen_set(){
    delete array_uni_gen_set_list_p;
}

array_uni_gen_set_list *all_array_uni_gen_set::get_array_uni_gen_set_list_p(){
    return array_uni_gen_set_list_p;
}

void all_array_uni_gen_set::append_array_uni_gen_set(var_sym *var_sym_p,
                                                     int ref_depth){
    array_uni_gen_set *array_uni_gen_set_p = new array_uni_gen_set(var_sym_p,
                                                                   ref_depth);
    array_uni_gen_set_list_p->append(array_uni_gen_set_p);
}

void all_array_uni_gen_set::print_all_array_uni_gen_set(){
        array_uni_gen_set_list_iter augsl_iter(array_uni_gen_set_list_p);
        while(!augsl_iter.is_empty()){
            array_uni_gen_set *array_uni_gen_set_p = augsl_iter.step();
            array_uni_gen_set_p->print_array_uni_gen_set();
        }
}

void all_array_uni_gen_set::print_all_array_uni_gen_set_and_eq_classes(){
        array_uni_gen_set_list_iter augsl_iter(array_uni_gen_set_list_p);
        while(!augsl_iter.is_empty()){
            array_uni_gen_set *array_uni_gen_set_p = augsl_iter.step();
            array_uni_gen_set_p->print_array_uni_gen_set_and_eq_classes();
        }
}

boolean all_array_uni_gen_set::exists(var_sym *var_sym_p){
    array_uni_gen_set_list_iter augsl_iter(array_uni_gen_set_list_p);
    if (augsl_iter.is_empty())
        return FALSE;
    while(!augsl_iter.is_empty()){
        array_uni_gen_set *array_uni_gen_set_p = augsl_iter.step();
        if(var_sym_p == array_uni_gen_set_p->get_var_sym_p())
            return TRUE;
    }
    return FALSE;
}

array_uni_gen_set *all_array_uni_gen_set::get(var_sym *var_sym_p){
    array_uni_gen_set_list_iter augsl_iter(array_uni_gen_set_list_p);
    assert(!augsl_iter.is_empty());
    while(!augsl_iter.is_empty()){
        array_uni_gen_set *array_uni_gen_set_p = augsl_iter.step();
```

```
        if(var_sym_p == array_uni_gen_set_p->get_var_sym_p())
            return array_uni_gen_set_p;
    }
    assert(0);
    return NULL;
}

// The class array_uni_gen_set contains a list of all the different
// uniformly generated sets that exist for references to a particular
// array.

array_uni_gen_set::array_uni_gen_set(var_sym *vs_p, int ref_depth){
    var_sym_p = vs_p;
    uni_gen_set_list_p = new uni_gen_set_list;
    reference_depth = ref_depth;
}

array_uni_gen_set::~array_uni_gen_set(){
    delete uni_gen_set_list_p;
}

void array_uni_gen_set::print_array_uni_gen_set(){
    printf("Printing uniformly generated sets for array : ");
    var_sym_p->print();
    printf("\n\n");
    uni_gen_set_list_iter ugsl_iter(uni_gen_set_list_p);
    while(!ugsl_iter.is_empty()){
        uni_gen_set *ugs_p = ugsl_iter.step();
        ugs_p->print_uni_gen_set();
    }
}

void array_uni_gen_set::print_array_uni_gen_set_and_eq_classes(){
    printf("Printing uni_gen_set's and eq_class'es for array : ");
    var_sym_p->print();
    printf("\n\n");
    uni_gen_set_list_iter ugsl_iter(uni_gen_set_list_p);
    while(!ugsl_iter.is_empty()){
        uni_gen_set *ugs_p = ugsl_iter.step();
        ugs_p->print_uni_gen_set_and_eq_classes();
    }
}

var_sym *array_uni_gen_set::get_var_sym_p(){
    return var_sym_p;
}

uni_gen_set_list *array_uni_gen_set::get_uni_gen_set_list_p(){
    return uni_gen_set_list_p;
}

local_space_eq_classes::local_space_eq_classes(uni_gen_set *uni_gen_set_p,
                                               int first_local_space_loop,
                                               int *iter_in_loop){
    // The argument first_local_space_loop will equal 1 if for instance
    // the 1. loop in the nest is also included in the localized space.
```

```
reference_depth = uni_gen_set_p->get_reference_depth();
nest_depth = uni_gen_set_p->get_nest_depth();
element_size = uni_gen_set_p->get_elem_size();
subscript_depth = uni_gen_set_p->get_subscript_depth();
uniformly_gen_set_p = uni_gen_set_p;

// L_space_p includes the directions of the localized
// vector space, which is from "first_local_space_loop" to "reference_depth"
L_space_p = new iteration_space(first_local_space_loop,reference_depth);
eq_class_list_p = new eq_class_list();
no_of_eq_classes = 0;

// creates a copy of the member list as we will pop the elements.
my_member_list *my_ml_p = new my_member_list();
member_list_iter ml_iter(uni_gen_set_p->get_member_list_p());
while(!ml_iter.is_empty()){
    member *m_p = ml_iter.step();
    //member_list_e *member_list_e_p;
    //member_list_e_p = new member(m_p);
    member *new_m_p = new member(m_p);
    my_ml_p->append(new_m_p);
}

matrix *matr_p = uni_gen_set_p->get_matrix_p();
matrix *matr_s_p = uni_gen_set_p->get_matrix_s_p();

iteration_space* R_st_p = uni_gen_set_p->get_R_st_p();
iteration_space* R_ss_p = uni_gen_set_p->get_R_ss_p();

// The intersection function member returns a dynamically allocated
// iteration_space pointer.
intersection_L_Rst_p = L_space_p->intersection(R_st_p);
intersection_L_Rss_p = L_space_p->intersection(R_ss_p);
no_of_ST_reuse_dims = intersection_L_Rst_p->get_no_of_included_dims();

// SS_but_not_ST_reuse_dim contains the dimension in which there is SS
// reuse but not ST reuse if such a dim exists. Otherwise 0.
SS_but_not_ST_reuse_dim = 0;
SS_but_not_ST_reuse_dim =
  intersection_L_Rss_p->get_diff(intersection_L_Rst_p);

if (SS_but_not_ST_reuse_dim == 0)
  SS_reuse = FALSE;
else
  SS_reuse = TRUE;

if(SS_reuse){
    // getting the factor of the index variable for the loop where there
    // is spatial reuse, and in the last subscript-expr.
    // The indexing of the matrix starts however with zero.
    int last_row = matr_p->m();
    fract index_var_factor_fract =
        matr_p->elt(last_row-1,SS_but_not_ST_reuse_dim-1);
    if(index_var_factor_fract.denom())
        index_var_factor = index_var_factor_fract.num()/index_var_factor_fract.denom();
```

```
    if (index_var_factor == 0)
        index_var_factor = 1;
}

while(!my_ml_p->is_empty()){
    // Removes the front member element (representing an array reference)
    // from the list, and checks for group reuse among all the remaining
    // elements of the list.

    member *assigned_member_p = NULL;
    assigned_member_p = my_ml_p->pop();

    fract_vector *assigned_fract_vector_p =
            assigned_member_p->constant_vector_p;
    eq_class *eq_class_p = new eq_class(uni_gen_set_p, assigned_member_p);
    my_member_list_iter my_ml_iter(my_ml_p);

    boolean GS_reuse_membership_established = FALSE;
    boolean GT_reuse_membership_established = FALSE;
    // Checks for group-temporal reuse among all the remaining elements
    // in this while-loop.
    while(!my_ml_iter.is_empty()){
        member *current_member_p = my_ml_iter.step();

        fract_vector *current_fract_vector_p =
                        current_member_p->constant_vector_p;
        int vector_size = current_fract_vector_p->n();
        fract_vector diff_fract_vector(vector_size);
        diff_fract_vector = *assigned_fract_vector_p -
                            *current_fract_vector_p;

        boolean solution_exists = FALSE;
        boolean solution_inside_local_vector_space = FALSE;
        fract_vector fract_vector_solution;
        fract_vector_solution = matr_p->
            particular_solution(diff_fract_vector,&solution_exists);

        if (solution_exists){
            iteration_space solution_span(&fract_vector_solution);
            //printf("The fract_vector_solution spans the dim's : ");
            //solution_span.print_included_dimensions();
            solution_inside_local_vector_space =
                L_space_p->contains(&solution_span);
        }

        if (solution_inside_local_vector_space){
            // Remove the current_member from the ml_p list, as it now
            // will be assigned an eq-class. When the rest of the list
            // elements are popped at the outermost while-loop, this
            // element will no longer be present in the list.
            // The remove method returns a pointer to the member so we
            // can just insert it into the eq_class.

            member *my_removed_member_p;
            my_removed_member_p = my_ml_p->remove(current_member_p);
            // assign the current_member_copy_p to the eq-class.
```

```
            eq_class_p->add(my_removed_member_p);
            GT_reuse_membership_established = TRUE;
        }
    } // END while(!my_ml_iter.is_empty())

    if (GT_reuse_membership_established){
        eq_class_p->set_reuse(GROUP_TEMPORAL);
    }
    // No group-temporal reuse was found for assigned_member_p and
    // we now check for group-spatial reuse.
    my_member_list_iter my_ml_iter2(my_ml_p);
    while(!my_ml_iter2.is_empty()){
        member *current_member_p = my_ml_iter2.step();
        fract_vector *current_fract_vector_p =
          current_member_p->constant_vector_p;
        fract_vector diff_fract_vector = *assigned_fract_vector_p -
          *current_fract_vector_p;
        // replace_last_elem_with_zero(diff_fract_vector);
        // The []-overloaded operator returns the given element
        // by reference.
        diff_fract_vector[subscript_depth-1] = fract(0);

        boolean solution_exists;
        boolean solution_inside_local_vector_space = FALSE;
        fract_vector fract_vector_solution;
        fract_vector_solution = matr_s_p->
          particular_solution(diff_fract_vector,&solution_exists);

        if (solution_exists){
            iteration_space solution_span(&fract_vector_solution);
            solution_inside_local_vector_space =
            L_space_p->contains(&solution_span);
        }

        if (solution_inside_local_vector_space){
            // Remove the current_member from the ml_p list, as it now
            // will be assigned an eq-class. When the rest of the list
            // elements are popped at the outermost while-loop, this
            // element will no longer be present in the list.
            // The remove method returns a pointer to the member so we
            // can just insert it into the eq_class.

            member *my_removed_member_p;
            my_removed_member_p = my_ml_p->remove(current_member_p);

            // assign the current_member_copy_p to the eq-class.
            eq_class_p->add(my_removed_member_p);
            GS_reuse_membership_established = TRUE;
        }
    } // END while(!my_ml_iter2.is_empty())
    if (GS_reuse_membership_established)
      eq_class_p->set_reuse(GROUP_SPATIAL);
    eq_class_list_p->append(eq_class_p);
    no_of_eq_classes++; //
}// END while(!my_ml_p->is_empty())
acc_per_eq_class_in_loop = new int[reference_depth];
```

```
}

local_space_eq_classes::~local_space_eq_classes(){
    delete L_space_p;
    delete eq_class_list_p;
    delete acc_per_eq_class_in_loop;
    delete intersection_L_Rst_p;
    delete intersection_L_Rss_p;
}

void local_space_eq_classes::print_eq_classes(){
    eq_class_list_iter ecl_iter(eq_class_list_p);
    int i=0;
    while(!ecl_iter.is_empty()){
        i++;
        printf("Set no. %d :\n",i);
        eq_class *eq_class_p = ecl_iter.step();
        eq_class_p->print_members();
    }
}

int local_space_eq_classes::get_total_no_of_off_chip_acc(){
    return total_no_of_off_chip_acc;
}

int local_space_eq_classes::get_no_of_eq_classes(){
    return no_of_eq_classes;
}

int local_space_eq_classes::get_no_of_ST_reuse_dims(){
    return no_of_ST_reuse_dims;
}

boolean local_space_eq_classes::get_SS_reuse(){
    return SS_reuse;
}

int local_space_eq_classes::get_elem_size(){
    return element_size;
}

int local_space_eq_classes::get_SS_reuse_factor(){
    return SS_reuse_factor;
}

int local_space_eq_classes::get_index_var_factor(){
    return index_var_factor;
}
```

A.5.3 iteration_space

```
// This constructor generates an iteration space which contains all the
// dimensions included in the span of the fract_vector.
iteration_space::iteration_space(fract_vector* fv_p){
    int fract_vector_size = fv_p->n();
    dimension_space = new dimension[fract_vector_size];
```

```
    no_of_dimensions = fract_vector_size;

    // Initializing the dimension space to be empty.
    for (int i=0; i<no_of_dimensions; i++){
        dimension_space[i] = DIMENSION_NOT_INCLUDED;
    }
    // The fract_vector_size equals the the entire loop nest depth.

    // Inserts the dimension number (i.e. loop number in the nest) for
    // which the fract_vector contains an non-zero element.
    for (int i=0; i<fract_vector_size; i++){
        fract temp = (*fv_p)[i];
        // num() returns nominator, denom() : denominator.
        if (temp.num()){
            dimension_space[i] = DIMENSION_INCLUDED;
        }
    }
}

iteration_space::iteration_space(int first_loop_inside_local_space,
                                 int reference_depth){
    // Checking that we are not performing tiling-analysis for some number
    // of loops which do not include the one containing the current
    // array references.
    assert(first_loop_inside_local_space<=reference_depth);

    no_of_dimensions = reference_depth;
    dimension_space = new dimension[no_of_dimensions];

    // Initializing the dimension elements not inside the localized vector
    // space to zero.
    for (int i=0; i<(first_loop_inside_local_space-1); i++){
        dimension_space[i] = DIMENSION_NOT_INCLUDED;
    }

    // Initializing the elements which correspond to the dimensions that
    // are included in the localized vector space.
    for (int i=first_loop_inside_local_space-1; i<no_of_dimensions; i++){
        dimension_space[i] = DIMENSION_INCLUDED;
    }
}

// This constructor creates an iteration space with a number of dimensions
// equal to reference_depth, and with none of then included.
iteration_space::iteration_space(int reference_depth){
    no_of_dimensions = reference_depth;
    dimension_space = new dimension[no_of_dimensions];

    // Initializing the dimension elements not inside the localized vector
    // space to zero.
    for (int i=0; i<reference_depth; i++){
        dimension_space[i] = DIMENSION_NOT_INCLUDED;
    }
}

iteration_space::~iteration_space(){
```

```
    delete dimension_space;
}

int iteration_space::get_no_of_dimensions(){
    return no_of_dimensions;
}

boolean iteration_space::dimension_included(int i){
    if(dimension_space[i])
        return TRUE;
    else
        return FALSE;
}

boolean iteration_space::contains(iteration_space *it_space_p){
    for (int i=0; i<no_of_dimensions; i++){
        if (!dimension_space[i]){
            if (it_space_p->dimension_space[i])
                return FALSE;
        }
    }
    return TRUE;
}

void iteration_space::print_included_dimensions(){
    for (int i=0; i<no_of_dimensions; i++){
        if (dimension_space[i]){
            printf("%d ",i+1);
        }
    }
    printf("\n");
}

iteration_space *iteration_space::union_space(iteration_space *it_space_p){
    assert(no_of_dimensions == it_space_p->no_of_dimensions);

    // Allocates a new iteration space with no dimensions included yet.
    iteration_space *united_space_p = new iteration_space(no_of_dimensions);

    for (int i=0; i<no_of_dimensions; i++){
        if (dimension_space[i] || it_space_p->dimension_included(i)){
            united_space_p->dimension_space[i] = DIMENSION_INCLUDED;
        }
    }
    return united_space_p;
}

iteration_space *iteration_space::intersection(iteration_space *it_space_p){
    assert(no_of_dimensions == it_space_p->no_of_dimensions);

    // Allocates a new iteration space with no dimensions included yet.
    iteration_space *intersected_space_p = new iteration_space(no_of_dimensions);

    for (int i=0; i<no_of_dimensions; i++){
        if (dimension_space[i] && it_space_p->dimension_included(i)){
            intersected_space_p->dimension_space[i] = DIMENSION_INCLUDED;
```

```
        }
    }
    return intersected_space_p;
}

boolean iteration_space::equals(iteration_space *iteration_space_p){
    assert(no_of_dimensions == iteration_space_p->get_no_of_dimensions());
    for (int i=0; i<no_of_dimensions; i++){
        if(dimension_space[i]){
            if(!(iteration_space_p->dimension_included(i))){
                return FALSE;
            }
        }
        else{
            if(iteration_space_p->dimension_included(i)){
                return FALSE;
            }
        }
    }
    return TRUE;
}

void iteration_space::add_dimension(dimension dim){
    dimension_space[int(dim)] = DIMENSION_INCLUDED;
}

int iteration_space::get_no_of_included_dims(){
    int no_of_dimensions_included = 0;
    for (int i=0; i<no_of_dimensions; i++){
        if (dimension_space[i]){
            no_of_dimensions_included++;
        }
    }
    return no_of_dimensions_included;
}

boolean iteration_space::get_diff(iteration_space *it_space_p){
    assert(no_of_dimensions == it_space_p->no_of_dimensions);
    for (int i=no_of_dimensions-1; i>=0; i--){
        if (dimension_space[i]){
            if (!(it_space_p->dimension_space[i]))
                return i+1;
        }
    }
    return 0;
}

eq_class::eq_class(uni_gen_set *ugs_p, member *member_p){
    member_list_p = new member_list();
    member_list_p->append(member_p);
    uni_gen_set_p = ugs_p;

    // SELF reuse is the default kind.
    reuse_kind = SELF;
}
```

```
void eq_class::add(member *member_p){
    member_list_p->append(member_p);
}

void eq_class::set_reuse(reuse r_kind){
    reuse_kind = r_kind;
}

void eq_class::print_members(){
    switch(reuse_kind){
    case SELF:
      break;
    case GROUP_TEMPORAL:
      printf("The eq_class exhibits group temporal reuse among its references\n");
      break;
    case GROUP_SPATIAL:
      printf("The eq_class exhibits group spatial reuse among its references\n");
      break;
    default:
      assert(0);
      break;
    }
    member_list_iter ml_iter(member_list_p);
    while(!ml_iter.is_empty()){
        member *member_p = ml_iter.step();
        print_array_access(member_p->array_instr_p);
    }
}
```

A.6 for_loop_transform.h

```
// for_loop_transform.h

#ifndef FOR_LOOP_TRANSFORM_H
#define FOR_LOOP_TRANSFORM_H

#include <suif1.h>
#include <transform.h>
#include "nested_loop.h"
#define NO_OF_REGIONS 1
#include <useful.h>
#include <dependence.h>
#include "locality.h"

class local_space;
class local_space_info;
class tile_size_info;
class optimal_tiling_info;

/**************************************************************************
   Declare a new class which is a list of "in_array"'s
   This allows for a later instantiation like :
   <list-name> = new array_instr_list;
   , that constructs a list of pointers to in_array's.
 **************************************************************************/
```

```
DECLARE_DLIST_CLASS(array_instr_list, in_array *);
DECLARE_DLIST_CLASS(access_dep_list, dvlist *);
DECLARE_DLIST_CLASS(local_space_info_list, local_space_info *);
DECLARE_DLIST_CLASS(tile_size_info_list, tile_size_info *);

class for_loop_transform{
    private:
        nested_loop *nl_p;
        int depth;
        int tile_size;
        tree_for **tree_for_p_array;
        loop_transform *loop_transform_p;
        boolean *doalls;
        int *trip;
        // Dependences
        int data_dependence_too_messy;
        int dependence_exists;
        int no_dep_test_was_run;
        access_dep_list *access_dep_list_p;
        array_instr_list *array_instr_list_p;
        array_instr_list **array_instr_list_table;
        void process_array_instr_list(array_instr_list * ail);
        boolean dir_is_possibly_negative(direction dir);
        void find_array_instr(tree_node_list * tnl);
        void find_array_instr(tree_node_list * tnl, int current_depth);
        void find_array_instr(instruction * ins, int current_depth);

        // Locality analysis

        // array_ref_list_table is an array of pointers to array_ref_list's
        // An array_ref_list is a list of array_ref's for a particular loop
        // in the nest.
        array_ref_list **array_ref_list_table;
        all_array_uni_gen_set **uni_gen_set_list_table;
        void insert_array_ref(array_ref* array_ref_p,
                              uni_gen_set_list *uni_gen_set_list_p,
                              int reference_depth);
        boolean generated_uni_gen_sets;

        int *no_of_iter_in_loop;

        // Element at offset i, contains the product of the iteration numbers
        // for loops 1 to i+1. Is calculated from the no_of_iter_in_loop array.
        int *total_no_of_it_in_loop;

        // The element at offset i contains the est. no. of off-chip accesses
        // for a localized space of loops i+1 to loop-depth.
        int *total_no_of_iter_in_loop;
        int no_of_off_chip_acc_per_it;

        void collect_local_space_statistics();
        local_space_info_list **local_space_info_list_table;
        int *step_size_table;

        int max_no_of_tile_sizes;
        int no_of_tile_sizes_tested;
```

```
        int selected_first_loop;
        int local_space_dim;
        int est_element_size;

        tile_size_info_list **tile_size_local_space;
        optimal_tiling_info **optimal_tiling_info_table;
        int tile_size_step;

    public:
        for_loop_transform(nested_loop *nl_p_arg);
        ~for_loop_transform();
        void tile(optimal_tiling_info *oti_p);
        void calc_dependence();
        boolean ok_to_tile(int first_loop, int last_loop);
        boolean all_interchange_ok();
        boolean *no_tile_made_WS_fit_in_cache;
        int get_depth();
        int get_tile_size_step();
        int get_optimal_tile_size_for_loop(int loop_no);
        void locality_analysis();
        void print_access_matrices();
        void generate_uni_gen_sets();
        void print_uni_gen_sets();
        void generate_eq_classes();
        void print_uni_gen_sets_and_eq_classes();
        void calculate_relative_costs();
        void find_best_tile_region();
        int get_tile_region_priority_loop_no(int priority);
        void set_selected_first_loop(int first_loop);

        void tile_size_selection();
        int get_optimal_tiling_loop(int first_tile_loop);
        optimal_tiling_info *get_optimal_tiling_info_no(int first_tile_loop);

        void print_collected_stat();
        void print_est_for_local_space(int first_loop);
        void print_est_for_all_spaces();
        void print_tile_size_info(tile_size_info *tsi_p);
        void print_optimal_tile_est();
        void print_optimal_tile_est_for_loop(int loop_no);
        int get_smallest_loop_it_in_local_space(int first_loop);

        optimal_tiling_info* get_optimal_tiling_for_first_loop(int first_loop);
        tile_size_info_list *get_tile_size_local_space(int loop_no);
};

// The local_space_info class contains info on the degree of reuse for a
// particular uni_gen_set in a particular localized iteration space.
class local_space_info{
private:
    int no_of_eq_classes;
    boolean SS_reuse;
    int no_of_ST_reuse_dims;
    int no_of_iterations;
    int element_size;
    int SS_reuse_divisor;
```

```
public:
    local_space_info(int no_of_eq_classes, boolean SS_reuse,
                     int no_of_ST_reuse_dims, int no_of_iterations,
                     int elem_size, int stride);
    local_space_info(int no_of_ref, int no_of_iterations);
    int get_no_of_eq_classes();
    int get_no_of_ST_reuse_dims();
    int get_no_of_iterations();
    int get_SS_reuse_divisor();
    void print_info();
};

class tile_size_info{
private:
    int WS_size;
    int no_of_off_chip_acc;
    int tile_size;
public:
    tile_size_info(int WS_size,int no_of_off_chip_acc,int tile_size);
    int get_WS_size();
    int get_no_of_off_chip_acc();
    int get_tile_size();
};

class optimal_tiling_info : public tile_size_info{
private:
    int first_loop;
public:
    optimal_tiling_info(int first_loop,int WS_size,
                        int no_of_off_chip_acc,int tile_size);
    optimal_tiling_info(tile_size_info *tsi_p,int first_l);
    int get_first_loop();
};

#endif
```

A.7 for_loop_transform.cc

```
// for_loop_transform.cc

#include "for_loop_transform.h"
#include <transform.h>
#include <iostream>
#include "nested_loop.h"
#include <stdlib.h>
#include "locality.h"

extern int line_size;
extern int cache_size;

void find_array_instr(tree_node_list * tnl);
void find_array_instr(tree_node * tn);
void find_array_instr(instruction * instr);
```

```
for_loop_transform::for_loop_transform(nested_loop *nl_p_arg){
    nl_p = nl_p_arg;
    depth = nl_p->get_depth();
    doalls = new boolean[depth];
    tree_for_p_array = new tree_for*[depth];

    // The array_instr_list is a private data member of the
    // for_loop_transform class.

    array_instr_list_p = new array_instr_list;

    array_instr_list_table = new array_instr_list*[depth];
    for(int j=0; j<depth; j++){
        array_instr_list_table[j] = new array_instr_list;
    }

    //printf("the depth of the for-loop is : %d\n",depth);
    int i = 0;
    tree_for_list_iter tfl_iter(nl_p->get_tree_for_list_p());
        while (i<depth && !tfl_iter.is_empty()){
        tree_for_p_array[i++] = tfl_iter.step();
        //tree_for_p_array[i++]->print();
        //cout << "The tree_for arrays is assigned loop nr : " << (i) << "\n";
        //cout.flush();
    }
    assert(i==depth && tfl_iter.is_empty());

    step_size_table = new int[depth];
    no_of_iter_in_loop = new int[depth];
    for (int i=0; i<depth; i++){
        int lower_bound, upper_bound, step_size;
        if (!(tree_for_p_array[i]->lb_is_constant(&lower_bound))){
            printf("The lower bound in loop %d is not constant\n",depth+1);
            assert(0);
        }
        if (!(tree_for_p_array[i]->ub_is_constant(&upper_bound))){
            printf("The upper bound in loop %d is not constant\n",depth+1);
            assert(0);
        }
        if (!(tree_for_p_array[i]->step_is_constant(&step_size))){
            printf("The step-size in loop %d is not constant\n",depth+1);
            assert(0);
        }
        no_of_iter_in_loop[i] = (upper_bound - lower_bound)/(step_size);
        step_size_table[i] = step_size;
    }

    total_no_of_iter_in_loop = new int[depth];
    total_no_of_iter_in_loop[0] = no_of_iter_in_loop[0];
    for (int i=1; i<depth; i++){
        total_no_of_iter_in_loop[i] = total_no_of_iter_in_loop[i-1] *
                                      no_of_iter_in_loop[i];
    }

    loop_transform_p = new loop_transform(depth,tree_for_p_array,doalls);
```

```
    // DEPENDENCE TESTING RELATED VARIABLES :
    data_dependence_too_messy = FALSE;
    dependence_exists = FALSE;
    no_dep_test_was_run = FALSE;
    access_dep_list_p = NULL;

    // The access_dep_list is designed to contain lists of dependences for
    // every two accesses which are data dependent. The dependences are
    // themselves a list of dependence vectors.
    // The access_dep_list is a private data member of the
    // for_loop_transform class.
    access_dep_list_p = new access_dep_list;

    array_ref_list_table = new array_ref_list*[depth];
    for(int i=0; i<depth; i++){
        array_ref_list_table[i] = new array_ref_list();
    }

    uni_gen_set_list_table = new all_array_uni_gen_set*[depth];
    for(int i=0; i<depth; i++){
        uni_gen_set_list_table[i] = new all_array_uni_gen_set(i+1);
    }

    local_space_info_list_table = new local_space_info_list*[depth];
    for(int i=0; i<depth; i++){
        local_space_info_list_table[i] = new local_space_info_list;
    }

    max_no_of_tile_sizes = cache_size / line_size;

    tile_size_local_space = new tile_size_info_list*[depth];
    for(int i=0; i<depth; i++){
        tile_size_local_space[i] = new tile_size_info_list;
    }

    optimal_tiling_info_table = new optimal_tiling_info*[depth];
    est_element_size = 4; // Is set to 4 bytes by default, correctly set later.

    no_tile_made_WS_fit_in_cache = new boolean[depth];
    for(int i=0; i<depth; i++){
        no_tile_made_WS_fit_in_cache[i] = FALSE;
    }
}

for_loop_transform::~for_loop_transform(){
    delete [] tree_for_p_array;
    delete [] doalls;
    delete array_instr_list_p;
    delete [] array_instr_list_table;

    delete loop_transform_p;
    delete access_dep_list_p;
    delete [] array_ref_list_table;
    delete [] uni_gen_set_list_table;
}
```

```
int power(int x, int n){
    assert(n>=0);
    if(n==0)
        return 1;
    if(n>0)
        return x*power(x,n-1);
    return 1;
}

int for_loop_transform::get_optimal_tiling_loop(int priority_no){
    return (optimal_tiling_info_table[priority_no-1]->get_first_loop());
}

int for_loop_transform::get_optimal_tile_size_for_loop(int loop_no){
    return (optimal_tiling_info_table[loop_no-1]->get_tile_size());
}

tile_size_info_list *
for_loop_transform::get_tile_size_local_space(int loop_no){
    return tile_size_local_space[loop_no-1];
}

optimal_tiling_info*
for_loop_transform::get_optimal_tiling_info_no(int first_tile_loop){
    return optimal_tiling_info_table[first_tile_loop-1];
}

optimal_tiling_info*
for_loop_transform::get_optimal_tiling_for_first_loop(int first_loop){
    if(first_loop>depth){
        printf("The first loop in user requested tiling is greater than nest depth\n");
        fflush(stdout);
        assert(0);
    }
    int i = 0;
    while(i<depth){
        if(optimal_tiling_info_table[i]->get_first_loop() == first_loop)
            return optimal_tiling_info_table[i];
        i++;
    }
    assert(0);
    return optimal_tiling_info_table[0];
}

int for_loop_transform::get_smallest_loop_it_in_local_space(int first_loop){
    assert(first_loop<=depth);
    int smallest_loop_it = no_of_iter_in_loop[first_loop-1];
    if(first_loop<depth){
        for(int i=first_loop; i<depth; i++){
          if(no_of_iter_in_loop[i]<smallest_loop_it)
              smallest_loop_it = no_of_iter_in_loop[i];
        }
    }
    return smallest_loop_it;
}
```

A.7.1 Dependences.

```
void for_loop_transform::calc_dependence(){
    //printf("Calculating dep. for loop with depth : %d\n",nl_p->get_depth());
    // The body of the outermost for loop is passed to find_array_instr().
    // This procedure appends all found array references to the
    // array_instr_list

    find_array_instr(tree_for_p_array[0]->body());
    process_array_instr_list(array_instr_list_p);
}


/***************************************************************************
   This routine is called from the calc_dependence() routine when a
   complete, possibly multi-level nested, for-loop has been found. The list
   containing all the array accesses in the for-loop is given as an
   argument to this routine.
   The routine runs the DependenceTest() on all pairs of references to the
   same array and if there exists dependence it stores the returned dvlist
   in the access_dep_list.
 ***************************************************************************/
void for_loop_transform::process_array_instr_list(array_instr_list * ail){
    array_instr_list_iter iter1(ail);

    array_instr_list_iter test_iter(ail);
    //printf("This loop contains the following accesses :\n");
    while(!test_iter.is_empty()){
        in_array * ai1 = test_iter.step();
        //print_array_access(ai1);
    }
    while(!iter1.is_empty()){
        in_array * ai1 = iter1.step();
        if(is_lhs(ai1)) {
            var_sym * vs1 = get_sym_of_array(ai1);
            //printf("Calculating dependences for lhs access:\n");
            //print_array_access(ai1);

            // Traversing the list once again to find all other references
            // to the array-variable vs1.
            array_instr_list_iter iter2(ail);
            while(!iter2.is_empty()) {
                in_array * ai2 = iter2.step();
                var_sym * vs2 = get_sym_of_array(ai2);
                // if(!is_lhs(ai2)) was earlier the condition for testing
                // for dependence between vs1 and vs2.
                if(vs1 == vs2){
                    deptest_result *dep_test_result_p;
                    dvlist * access_dep = DependenceTest(ai1, ai2, 1, dep_test_result_p);
                    switch(*dep_test_result_p){
                        case dt_none: // No dep.-test was run.
                            no_dep_test_was_run = TRUE;
                            break;
                        case dt_ok:
                            // The two array accesses are data dependent.
                            // Dependence vector list returned.
                            dependence_exists = TRUE;
```

```
                            access_dep_list_p->append(access_dep);
                            break;
                        case dt_indep:
                            assert(access_dep->indep());
                            break;
                        case dt_no_common_nest:
                            // This should not be possible
                            assert(0);
                            break;
                        case dt_too_messy:
                            data_dependence_too_messy = TRUE;
                            break;
                        default:
                            assert(0);
                            break;
                    }
                }
            }
        } // end of if(is_lhs(ai1))
    }
}


boolean for_loop_transform::ok_to_tile(int first_loop, int last_loop){
    if (first_loop <= 0)
        {
        printf("Error: Argument first_loop to method ok_to_tile negative or zero\n");
        assert(0);
        }
    if (last_loop <= 0)
        {
        printf("Error: Argument last_loop to method ok_to_tile negative or zero\n");
        assert(0);
        }
    if (first_loop>last_loop)
        {
        printf("Error: Argument first_loop greater than arg. last_loop in method ok_to_tile\n");
        assert(0);
        }
    if (first_loop > depth)
        {
        printf("Error: Argument first_loop in method ok_to_tile greater than depth\n");
        assert(0);
        }
    if (last_loop > depth)
        {
        printf("Error: Argument last_loop in method ok_to_tile greater than depth\n");
        assert(0);
        }

    // access_dep_list_p is a pointer to an access_dep_list which contains
    // elements of dependence vector lists. These lists corresponds to the
    // dependences that exists between two (possibly different or the same)
    // accesses to a particular array.
    access_dep_list_iter adl_iter(access_dep_list_p);
    boolean so_far_ok_to_tile = TRUE;
```

```
        int dep_vector_no = 1;

    while(!adl_iter.is_empty() && so_far_ok_to_tile){
        dvlist *dvlist_p = adl_iter.step();
        dvlist_iter dvl_iter(dvlist_p);

        while(!dvl_iter.is_empty() && so_far_ok_to_tile){
            // Treating a single dependence vector represented by the
            // class dvlist_e. A dvlist_e contains an instance of a
            // distance_vector which again contains a list of
            // distance_vector_e's which is a direction <,>,=,<=,>=,<>,*.
            dvlist_e *dvlist_e_p = dvl_iter.step();

            distance_vector_e *distance_vector_e_p;
            boolean vector_is_pos_before_tile_loop = FALSE;
            int i = 1;
            boolean tile_loop_element_negative = FALSE;

            // Treating the first (first_loop-1) elements of the vector
            // if the first non_zero element of these first_loop-1
            // elements is positive then it is ok to tile the requested
            // loops.

            distance_vector_iter dv_iter(dvlist_e_p->dv);

            while(i<first_loop && !vector_is_pos_before_tile_loop && so_far_ok_to_tile){
              // if (!dvlist_e_p->dv->is_empty())
              assert(!dv_iter.is_empty());
                //distance_vector_e_p = dvlist_e_p->dv->pop();
                distance_vector_e_p = dv_iter.step();
                direction d = distance_vector_e_p->d.dir();
                switch(d)
                    {
                    case d_lt:
                        vector_is_pos_before_tile_loop = TRUE;
                        break;

                    case d_eq:
                        break;

                    case d_le:
                        break;

                    // d_ge d_lg d_star d_gt
                    default:
                        printf("Error: Distance vector is lex. negative\n");
                        assert(0);
                        break;
                    }
                i++;
            }

            // i will always contain the element number of the next
            // direction to be popped from the distance_vector.
            // This is ensured by the following while-loop.
            while (i < first_loop){
```

```
              //printf("Popping once, i equals : %d\n",i);
              //fflush(stdout);
              //printf("%d'th element is popped now!!!!\n",i);
              //fflush(stdout);
              distance_vector_e_p = dv_iter.step(); //dvlist_e_p->dv->pop();
                i++;
            }

            if (!vector_is_pos_before_tile_loop)
                // The legality of tiling was not determined by the
                // first first_loop-1 elements being lexicographically
                // positive, so it is necessary to check the positivity
                // of the first_loop -> last_loop elements of the
                // dependence vector.
                // All the distance elements in these loops must be pos.
                {
                for(i=first_loop; i<=last_loop; i++)
                    {
                      distance_vector_e_p = dv_iter.step();
                      direction d = distance_vector_e_p->d.dir();
                      if (dir_is_possibly_negative(d))
                        so_far_ok_to_tile = FALSE;
                    }
                }
            } // finished iterating over a dep. vector list for two accesses.
        } // finished iterating over dep. for all accesses.
    return so_far_ok_to_tile;
} // end of ok_to_tile()

boolean for_loop_transform::dir_is_possibly_negative(direction dir)
    {
    switch(dir)
        {
        case d_lt:
            return FALSE;
            break;

        case d_gt:
            return TRUE;
            break;

        case d_eq:
            return FALSE;
            break;

        case d_le:
            return FALSE;
            break;

        case d_ge:
            return TRUE;
            break;

        case d_lg:
            return TRUE;
            break;
```

```
        case d_star:
            return TRUE;
            break;
        default:
            assert(0);
            return TRUE;
            break;
        }
    }

boolean for_loop_transform::all_interchange_ok(){
    if (!dependence_exists)
        return TRUE;
    return FALSE;
}

// The main iterator over structured control-flow.
// When there is no outer for loop ail will be NULL.  ail will be assigned
// at the outermost for loops. All the array accesses for one particular
// (possibly multi-level nested) for-loop are collected in the
// array_instr_list.
// Is called with the tree_for_p_array[0]->body() as the tnl parameter in
// the for_loop_transform class.

void for_loop_transform::find_array_instr(tree_node_list * tnl){
    find_array_instr(tnl, 0);
}

void for_loop_transform::find_array_instr(tree_node_list * tnl, int current_depth)
{
  //printf("current_depth is : %d",current_depth);
  // fflush(stdout);

    tree_node_list_iter iter(tnl);
    while(!iter.is_empty()) {
        tree_node * tn = iter.step();

        switch(tn->kind()) {
        case TREE_FOR:{
            tree_for * tnf = (tree_for *)tn;
            find_array_instr(tnf->lb_list(),current_depth+1);
            find_array_instr(tnf->ub_list(),current_depth+1);
            find_array_instr(tnf->step_list(),current_depth+1);
            find_array_instr(tnf->landing_pad(),current_depth+1);
            find_array_instr(tnf->body(),current_depth+1);
            break;
        }
        case TREE_IF:{
            tree_if * tni = (tree_if *)tn;
            find_array_instr(tni->header(),current_depth);
            find_array_instr(tni->then_part(),current_depth);
            find_array_instr(tni->else_part(),current_depth);
            break;
        }
        case TREE_LOOP:{
```

```
            tree_loop * tnl = (tree_loop *)tn;
            find_array_instr(tnl->body(),current_depth);
            find_array_instr(tnl->test(),current_depth);
            break;
        }
        case TREE_BLOCK:{
            tree_block * tnb = (tree_block *)tn;
            find_array_instr(tnb->body(),current_depth);
            break;
        }
        case TREE_INSTR:{
            tree_instr * tnin = (tree_instr *)tn;
            if(array_instr_list_p)
                find_array_instr(tnin->instr(),current_depth);
            break;
        }
        default:
            assert(0);
            break;
        }
    }
}

// Iterate over all the instructions of expression trees, add array
// instructions to the list.
void for_loop_transform::find_array_instr(instruction * ins, int current_depth)
{
    fflush(stdout);
    if(ins->opcode() == io_array) {
        assert(array_instr_list_p);
        in_array * ia = (in_array *)ins;
    array_instr_list_table[current_depth]->append(ia);
    array_instr_list_p->append(ia);
    }

    for(int i=0; i<ins->num_srcs(); i++) {
        operand op(ins->src_op(i));
        if(op.is_instr())
            find_array_instr(op.instr(),current_depth);
    }
}
```

A.7.2 Locality analysis.

```
/***************************************************************************
 *  The locality analysis related procedures are listed in the following.
 ***************************************************************************/

void for_loop_transform::locality_analysis(){
    for(int i=0; i<depth; i++){
        array_instr_list_iter ail_iter(array_instr_list_table[i]);
        in_array *array_instr_p;
        while(!ail_iter.is_empty()){
            array_instr_p = ail_iter.step();
            //print_array_access(array_instr_p);
            assert(array_instr_p->opcode() == io_array);
```

```
            // The array_instr_p inserted in the array_ref is the original.
            array_ref *array_ref_p = new array_ref(array_instr_p, depth, tree_for_p_array);
            array_ref_list_table[i]->append(array_ref_p);
        }
        if(i==depth-1)
            est_element_size = (array_instr_p->elem_size())/8;
    }
}

void for_loop_transform::print_access_matrices(){
    for(int i=0; i<depth; i++){
        array_ref_list_iter arl_iter(array_ref_list_table[i]);
        if(arl_iter.is_empty())
            continue;
        printf("==== Processing loop no. : %d ====\n\n",i+1);
        while(!arl_iter.is_empty()){
            array_ref *array_ref_p = arl_iter.step();
            printf("The access matrix for array reference ");
            in_array *array_instr_p = array_ref_p->get_array_instr_p();
            matrix *matrix_p = array_ref_p->get_matrix_p();
            print_array_access(array_instr_p);
            printf("with dimensions : %d x %d is :\n",
                   matrix_p->m(),matrix_p->n());
            matrix_p->print();
            printf("\n\n");
        }
    }
}

int for_loop_transform::get_depth(){
    return depth;
}

void for_loop_transform::generate_uni_gen_sets(){
    // This variable is used by the destructor
    generated_uni_gen_sets = TRUE;

    for(int i=0; i<depth; i++){
        array_ref_list_iter arl_iter(array_ref_list_table[i]);
        while(!arl_iter.is_empty()){
            array_ref *array_ref_p;
            array_ref_p = arl_iter.step();
            var_sym *var_sym_p = get_sym_of_array(array_ref_p->get_array_instr_p());
            // The uni_gen_set_list_table[] array contains elements that are
            // instances of the all_array_uni_gen_set class.
            if (uni_gen_set_list_table[i]->exists(var_sym_p)){
                // The all_array_uni_gen_set_list already contained an
                // array_uni_gen_set instance which holds information on
                // the uniformly generated sets for this array.
                array_uni_gen_set *array_uni_gen_set_p =
                  uni_gen_set_list_table[i]->get(var_sym_p);
                uni_gen_set_list *uni_gen_set_list_p =
                  array_uni_gen_set_p->get_uni_gen_set_list_p();
                insert_array_ref(array_ref_p,uni_gen_set_list_p,i+1);
            }
            else{
```

```
                // Append an array_uni_gen_set representing all uniformly
                // generated sets for this array.
                // An array_uni_gen_set containing the var_sym and an empty
                // uni_gen_set_list is created. i+1 equals reference-depth.
                uni_gen_set_list_table[i]->append_array_uni_gen_set(var_sym_p,
                                                                     i+1);

                // Make sure that the array_uni_gen_set was appended properly
                assert(uni_gen_set_list_table[i]->exists(var_sym_p));
                array_uni_gen_set *array_uni_gen_set_p;
                array_uni_gen_set_p = uni_gen_set_list_table[i]->get(var_sym_p);
                insert_array_ref(array_ref_p,
                                 array_uni_gen_set_p->get_uni_gen_set_list_p(),
                                 i+1);
            }
        }
    } // end for ...
}

void for_loop_transform::insert_array_ref(array_ref* array_ref_p,
                                          uni_gen_set_list *uni_gen_set_list_p,
                                          int ref_depth){
    uni_gen_set_list_iter ugsl_iter(uni_gen_set_list_p);
    boolean not_found_similar_uni_gen_set = TRUE;
    while(!ugsl_iter.is_empty() && not_found_similar_uni_gen_set){
        uni_gen_set *uni_gen_set_p = ugsl_iter.step();
        matrix *uni_gen_set_matrix_p = uni_gen_set_p->get_matrix_p();
        matrix *array_ref_matrix_p = array_ref_p->get_matrix_p();
        if (*array_ref_matrix_p == *uni_gen_set_matrix_p){
            uni_gen_set_p->add_member(array_ref_p->get_array_instr_p(),
                                      array_ref_p->get_matrix_p(),
                                      array_ref_p->get_fract_vector_p());
            not_found_similar_uni_gen_set = FALSE;
        }
    }
    if (not_found_similar_uni_gen_set){
        uni_gen_set *uni_gen_set_p =
          new uni_gen_set(array_ref_p->get_array_instr_p(),
                          array_ref_p->get_matrix_p(),
                          array_ref_p->get_fract_vector_p(),
                          ref_depth,
                          array_ref_p->get_elem_size());
        uni_gen_set_list_p->append(uni_gen_set_p);
    }
}

void for_loop_transform::print_uni_gen_sets(){
    printf("\n\n==== Printing out uniformly generated sets for a loop nest ====\n\n");
    for(int i=0; i<depth; i++){
        printf("**** Printing uniformly generated sets for loop %d ****\n\n",i+1);
        uni_gen_set_list_table[i]->print_all_array_uni_gen_set();
        printf("\n");
    }
}

void for_loop_transform::generate_eq_classes(){
```

```
    for(int i=0; i<depth; i++){
      int ref_depth = i+1;
        array_uni_gen_set_list_iter augsl_iter(uni_gen_set_list_table[i]->
                                   get_array_uni_gen_set_list_p());
        while(!augsl_iter.is_empty()){
            array_uni_gen_set *array_uni_gen_set_p;
            array_uni_gen_set_p = augsl_iter.step();
            uni_gen_set_list_iter ugsl_iter(array_uni_gen_set_p
                                   ->get_uni_gen_set_list_p());
            while(!ugsl_iter.is_empty()){
                uni_gen_set *uni_gen_set_p;
                uni_gen_set_p = ugsl_iter.step();
                for(int j=0; j<ref_depth; j++){
                    local_space_eq_classes *local_space_eq_classes_p;

                    // The call to the local_space_eq_classes constructor will
                    // create eq_class'es for local-space : j+1 to ref_depth.
                    local_space_eq_classes_p =
                        new local_space_eq_classes(uni_gen_set_p,j+1,
                                                   no_of_iter_in_loop);

                    // local_space_eq_classes_table[j] contains an object
                    // local_space_eq_classes which again contains a list
                    // of eq-classes for a localized iteration space
                    // consisting of loops : j+1 to ref_depth

                    uni_gen_set_p->local_space_eq_classes_table[j] =
                            local_space_eq_classes_p;
                }
            }
        }
    }
}

void for_loop_transform::print_uni_gen_sets_and_eq_classes(){
    printf("\n\n==== Printing out uni_gen_set's and eq_class'es in a nest ====\n\n");
    for(int i=0; i<depth; i++){
        printf("**** Printing uni_gen_set's and eq_class'es for loop %d ****\n\n",i+1);
        uni_gen_set_list_table[i]->print_all_array_uni_gen_set_and_eq_classes();
        printf("\n");
    }
}

int div_round_up(int dividend, int divisor){
    int result = dividend / divisor ;
    if(dividend % divisor)
        result++;
    return result;
}

void for_loop_transform::set_selected_first_loop(int first_loop){
    selected_first_loop = first_loop;
    local_space_dim = depth - selected_first_loop + 1;
}
```

A.7.3 Evaluating reuse.

```
// This procedure obtains the costs of tiling the different possible regions
// in the nest by calling the private member function
// calc_off_chip_acc_for_local_space(). The results hereof are examined
// and are ordered in the array tile_region_priority[depth].

void for_loop_transform::find_best_tile_region(){
    // Fill in the local_space_info_list_table[depth] which elements are lists
    // for the different local spaces. These contain local_space_info's.
    collect_local_space_statistics();
    tile_size_selection();

    // Insertion sort.
    for(int j=1; j<depth; j++){
        optimal_tiling_info *key_p = optimal_tiling_info_table[j];
        int i = j-1;
        while(i>=0 && (optimal_tiling_info_table[i]->get_no_of_off_chip_acc())>
                      (key_p->get_no_of_off_chip_acc())){
          optimal_tiling_info_table[i+1] = optimal_tiling_info_table[i];
          i--;
        }
        optimal_tiling_info_table[i+1] = key_p;
    }
}

void for_loop_transform::collect_local_space_statistics(){
    for(int i=0; i<depth; i++){
      int ref_depth = i+1;
        array_uni_gen_set_list_iter augsl_iter(uni_gen_set_list_table[i]->
                                    get_array_uni_gen_set_list_p());
        while(!augsl_iter.is_empty()){
            array_uni_gen_set *array_uni_gen_set_p;
            array_uni_gen_set_p = augsl_iter.step();
            uni_gen_set_list_iter ugsl_iter(array_uni_gen_set_p
                                  ->get_uni_gen_set_list_p());
            while(!ugsl_iter.is_empty()){
                uni_gen_set *uni_gen_set_p;
                uni_gen_set_p = ugsl_iter.step();
                for(int j=0; j<ref_depth; j++){
                    // For each uni_gen_set with corresponding local_space_eq_classes
                    // that exist in loop i, the local spaces starting with
                    //loop j is dealth with here.
                    local_space_eq_classes *lsec_p =
                      uni_gen_set_p->local_space_eq_classes_table[j];
                    int stride = step_size_table[i] *
                                 lsec_p->get_index_var_factor();
                    local_space_info *local_space_info_p =
                      new local_space_info(lsec_p->get_no_of_eq_classes(),
                                           lsec_p->get_SS_reuse(),
                                           lsec_p->get_no_of_ST_reuse_dims(),
                                           total_no_of_iter_in_loop[ref_depth-1],
                                           lsec_p->get_elem_size(),
                                           stride);
                    local_space_info_list_table[j]->append(local_space_info_p);
                }
                for(int j=ref_depth; j<depth; j++){
```

```
                    int no_of_ref = uni_gen_set_p->get_no_of_members();
                    //int element_size = uni_gen_set_p->get_elem_size();

                    // total_no_of_it_in_loop[ref_depth-1] contains the product
                    // of all the enclosing loops iteration numbers.
                    local_space_info *local_space_info_p =
                      new local_space_info(no_of_ref,
                                           total_no_of_iter_in_loop[ref_depth-1]);
                    local_space_info_list_table[j]->append(local_space_info_p);
                }
            }
        }
    }
}

void for_loop_transform::print_collected_stat(){
    for(int i=0; i<depth; i++){
        printf("====== The collected info for local space %d-%d ======\n\n",
               i+1,depth);
        local_space_info_list_iter lsil_iter(local_space_info_list_table[i]);
        while(!lsil_iter.is_empty()){
            local_space_info *local_space_info_p = lsil_iter.step();
            local_space_info_p->print_info();
            printf("\n");
        }
        printf("\n");
    }
}

void for_loop_transform::tile_size_selection(){
    for(int j=0; j<depth; j++){
        // for local-space j+1 to depth.
        int smallest_loop_it = get_smallest_loop_it_in_local_space(j+1);

        int i=1;
        tile_size_step = line_size/est_element_size;
        if((line_size%est_element_size) != 0)
            tile_size_step++;
        int tile_size = tile_size_step;

        int no_of_local_space_dims = depth - j;
        int working_set_size = 0;
        boolean ts_exeeded_loop_it = FALSE;
        boolean WS_exeeded_cache_size = FALSE;
        tile_size_info *tile_size_info_p = NULL;
        while(!WS_exeeded_cache_size && !ts_exeeded_loop_it){
          // checker for tile_size = line_size * i;
            tile_size = i*tile_size_step;
            int accumulated_WS_size = 0;
            int accumulated_off_chip_acc = 0;
            local_space_info_list_iter
              lsil_iter(local_space_info_list_table[j]);
            while(!lsil_iter.is_empty()){
                local_space_info *lsi_p = lsil_iter.step();
                int no_of_it = lsi_p->get_no_of_iterations();
                int no_of_eq_cl = lsi_p->get_no_of_eq_classes();
```

```
                int no_of_ST_reuse_dims = lsi_p->get_no_of_ST_reuse_dims();
                int ST_div = power(tile_size,
                                   no_of_ST_reuse_dims);
                int ST_factor = power(tile_size,
                                      no_of_local_space_dims);
                int SS_reuse_div = lsi_p->get_SS_reuse_divisor();

                accumulated_WS_size += (ST_factor * no_of_eq_cl * line_size)/
                                   (ST_div * SS_reuse_div);
                accumulated_off_chip_acc += (no_of_it * no_of_eq_cl)/
                                        (ST_div * SS_reuse_div);;
            }
            // Element with index 0 contains corresponding total no. of off-chip
            if(accumulated_WS_size < cache_size){
                tile_size_info_p = new tile_size_info(accumulated_WS_size,
                                                      accumulated_off_chip_acc,
                                                      tile_size);
                tile_size_local_space[j]->append(tile_size_info_p);
                if(tile_size>=smallest_loop_it){
                    // Selected tile-size is greater than smallest loop bound
                    // Tiling unneccesary.
                    optimal_tiling_info *optimal_tiling_info_p =
                      new optimal_tiling_info(tile_size_info_p,j+1);
                    optimal_tiling_info_table[j] = optimal_tiling_info_p;
                    ts_exeeded_loop_it = TRUE;
                    no_tile_made_WS_fit_in_cache[j] = TRUE;
                }
            }
            else{
                // The tile_size_info* constructed in the last iteration is used
                // to generate an optimal_tiling_info*
                WS_exeeded_cache_size = TRUE;
                no_of_tile_sizes_tested = i-1;
                if(i==1){
                    tile_size_info_p = new tile_size_info(accumulated_WS_size,
                                                          accumulated_off_chip_acc,
                                                          tile_size);
                    tile_size_local_space[j]->append(tile_size_info_p);
                }
                optimal_tiling_info *optimal_tiling_info_p =
                  new optimal_tiling_info(tile_size_info_p,j+1);
                optimal_tiling_info_table[j] = optimal_tiling_info_p;
            }
            // accesses or WS-size for tile-size=line-size*1
            //total_off_chip_acc[i-1] = accumulated_off_chip_acc;
            //WS_size_for_tile_size[i-1] = accumulated_WS_size;
            i++;
        }
        if(tile_size == 0){
            printf("Minimum tile sizes equal to line-size makes ws to big for cache\n ");
            assert(0);
            return;
        }
    } // end of while(!WS_exeeded_cache_size && !ts_exeeded_loop_it)
}
```

```
void for_loop_transform::tile(optimal_tiling_info *oti_p){
    tile_size = oti_p->get_tile_size();
    int first_loop = oti_p->get_first_loop();

    assert(first_loop<=depth);
    int nregions = NO_OF_REGIONS; // NO_OF_REGIONS is set to 1
    int coalesce[NO_OF_REGIONS] = {};  // setting array elements to 0 (FALSE)
    int first[NO_OF_REGIONS+1];
    int no_of_local_space_dims = depth - first_loop + 1;

    for(int i=0; i<depth; i++){
        doalls[i] = FALSE;
    }

    int no_of_elem_acc_in_tile = 0;
    trip = new int[depth];
    for(int i=0;i<depth;i++){
        trip[i] = tile_size;
    }

    // Loop no. 1 (i.e. the first loop in the entire nest) is denoted loop no.
    // 0 in the transform library convention. The depth of the entire loop nest
    // is placed in first[1].
    first[0] = first_loop-1;
    first[1] = depth;
    loop_transform_p->tile_transform(trip, nregions, coalesce, first);
}

void for_loop_transform::print_optimal_tile_est(){
    printf("***** Optimal results for all spaces *****\n\n");
    for(int i=0; i<depth; i++){
        printf("Optimal results for tiling of loops %d - %d :\n",
               i+1,depth);
        print_tile_size_info(optimal_tiling_info_table[i]);
        printf("\n");
    }
}

void for_loop_transform::print_optimal_tile_est_for_loop(int loop_no){
    printf("***** Optimal results for tiling of loops %d-%d *****\n",
           loop_no,depth);
    tile_size_info *tsi_p = optimal_tiling_info_table[loop_no-1];
    print_tile_size_info(tsi_p);
}

void for_loop_transform::print_est_for_all_spaces(){
    printf("***** Estimations for all spaces and all tile-sizes *****\n\n");
    for(int i=0; i<depth; i++){
        print_est_for_local_space(i+1);
    }
}

void for_loop_transform::print_est_for_local_space(int first_loop){
    printf("===== Tiling of loops %d - %d estimations =====\n",
           first_loop,depth);
    tile_size_info_list_iter tsil_iter(tile_size_local_space[first_loop-1]);
```

```
    while(!tsil_iter.is_empty()){
        tile_size_info *tsi_p = tsil_iter.step();
        print_tile_size_info(tsi_p);
    }
    printf("\n");
}

void for_loop_transform::print_tile_size_info(tile_size_info *tsi_p){
    printf("Tile-size = %d, ",tsi_p->get_tile_size());
    printf("WS-size = %d, ",tsi_p->get_WS_size());
    int WS_100 = (tsi_p->get_WS_size())*100;
    char p_sign[] = "%";
    printf("Cache-ut. = %d%s, ",(WS_100/cache_size),p_sign);
    printf("No of off-chip acc. = %d\n",tsi_p->get_no_of_off_chip_acc());
    // Used for easy insertion in tex-document :
    printf(",%d,%d%s\n",tsi_p->get_no_of_off_chip_acc(),(WS_100/cache_size),p_sign);
}

int for_loop_transform::get_tile_size_step(){
    return tile_size_step;
}
```

A.7.4 Auxiliary.

```
// This constructor is called when there may exist some reuse for the
// uni_gen_set.
local_space_info::local_space_info(int no_of_eq_cl,
                                   boolean SS_r,
                                   int no_of_ST_r_d,
                                   int no_of_iter,
                                   int elem_size,
                                   int stride){
    no_of_eq_classes = no_of_eq_cl;
    SS_reuse = SS_r;
    no_of_ST_reuse_dims = no_of_ST_r_d;
    no_of_iterations = no_of_iter;
    element_size = elem_size;
    if(SS_r)
      SS_reuse_divisor = div_round_up(line_size,elem_size*stride);
    else
      SS_reuse_divisor = 1;
}

// This constructor is called when no reuse exists for the uni_gen_set.
local_space_info::local_space_info(int no_of_ref,
                                   int no_of_iter){
    no_of_eq_classes = no_of_ref;
    SS_reuse = FALSE;
    no_of_ST_reuse_dims = 0;
    no_of_iterations = no_of_iter;
    element_size = 1;
    SS_reuse_divisor = 1;
}

int local_space_info::get_no_of_eq_classes(){
    return no_of_eq_classes;
```

```
}

int local_space_info::get_no_of_ST_reuse_dims(){
    return no_of_ST_reuse_dims;
}

int local_space_info::get_no_of_iterations(){
    return no_of_iterations;
}

int local_space_info::get_SS_reuse_divisor(){
    return SS_reuse_divisor;
}

void local_space_info::print_info(){
    printf("no_of_eq_classes = %d   ",no_of_eq_classes);
    printf("no_of_ST_reuse_dims = %d\n",no_of_ST_reuse_dims);
    printf("no_of_iterations = %d\n",no_of_iterations);
    printf("SS_reuse_divisor = line_size/elem_size*stride = %d/%d*??? = %d\n",
           line_size,element_size,SS_reuse_divisor);
}

tile_size_info::tile_size_info(int WS_s,int no_of_off_ch_acc,
                               int ts){
    WS_size = WS_s;
    no_of_off_chip_acc = no_of_off_ch_acc;
    tile_size = ts;
}

int tile_size_info::get_WS_size(){
    return WS_size;
}

int tile_size_info::get_no_of_off_chip_acc(){
    return no_of_off_chip_acc;
}

int tile_size_info::get_tile_size(){
    return tile_size;
}

optimal_tiling_info::optimal_tiling_info(int first_l,int WS_size,
                                         int no_of_off_chip_acc,int tile_size)
  : tile_size_info(WS_size,no_of_off_chip_acc,tile_size){
    first_loop = first_l;
}

optimal_tiling_info::optimal_tiling_info(tile_size_info *tsi_p,int first_l)
        : tile_size_info(tsi_p->get_WS_size(),tsi_p->get_no_of_off_chip_acc(),
                         tsi_p->get_tile_size()){
    first_loop = first_l;
}

int optimal_tiling_info::get_first_loop(){
    return first_loop;
}
```

Appendix B

SimpleScalar installation errors.

This appendix contains a description of the problems encountered while installing the SimpleScalar tool set. The changes necessary to obtain a working simulator required altering parts of the source code. Most of the changes made have been gathered from different locations on the internet.
Run the "configure" script as described in the install manual :

```
>./configure --host=i586-linux --target=sslittle-na-sstrix --with-gnu-as --with-gnu-ld --prefix=/hom
```

Run make
Run make install
Build the simulators ... "make" results in :

```
[root@localhost simplesim-2.0]# make
gcc `./sysprobe -flags` -DDEBUG -O   -c sim-fast.c
gcc `./sysprobe -flags` -DDEBUG -O   -c main.c
main.c: In function `main':
main.c:200: warning: return type of `main' is not `int'
gcc `./sysprobe -flags` -DDEBUG -O   -c syscall.c
syscall.c:96: bsd/sgtty.h: No such file or directory
make: *** [syscall.o] Error 1
[1]+  Done                    emacs Makefile
[root@localhost simplesim-2.0]#
```

To work around this problem I have copied a bsd/ directory containing the sgtty.h file, to the /usr/include directory. This "bsd" package have ben obtained in a precompiled version of the libc-5.4.44 library. This action brings us a little further :

```
[root@localhost simplesim-2.0]# make
gcc `./sysprobe -flags` -DDEBUG -O   -c syscall.c
gcc `./sysprobe -flags` -DDEBUG -O   -c memory.c
gcc `./sysprobe -flags` -DDEBUG -O   -c regs.c
gcc `./sysprobe -flags` -DDEBUG -O   -c loader.c
gcc `./sysprobe -flags` -DDEBUG -O   -c ss.c
gcc `./sysprobe -flags` -DDEBUG -O   -c endian.c
gcc `./sysprobe -flags` -DDEBUG -O   -c dlite.c
gcc `./sysprobe -flags` -DDEBUG -O   -c symbol.c
gcc `./sysprobe -flags` -DDEBUG -O   -c eval.c
gcc `./sysprobe -flags` -DDEBUG -O   -c options.c
gcc `./sysprobe -flags` -DDEBUG -O   -c stats.c
```

```
gcc `./sysprobe -flags` -DDEBUG -O   -c range.c
gcc `./sysprobe -flags` -DDEBUG -O   -c misc.c
gcc -o sim-fast `./sysprobe -flags` -DDEBUG -O   sim-fast.o main.o syscall.o memory.o regs.o loader.
syscall.o: In function `ss_syscall':
syscall.o(.text+0x728): undefined reference to `bsd_ioctl'
collect2: ld returned 1 exit status
make: *** [sim-fast] Error 1
[root@localhost simplesim-2.0]#
```

After replacing the dummy.o file in /usr/lib/ with the libbsd.a from libc-5.4.44/usr/lib/ we once again get a little further.

```
[root@localhost simplesim-2.0]# make
gcc -o sim-fast `./sysprobe -flags` -DDEBUG -O   sim-fast.o main.o syscall.o memory.o regs.o loader.
/usr/bin/../lib/libbsd.a(signal.o): In function `signal':
signal.o(.text+0x22): undefined reference to `_sigintr'
collect2: ld returned 1 exit status
make: *** [sim-fast] Error 1
[root@localhost simplesim-2.0]#
```

Remove the signal.o module with "ar vd libbsd.a signal.o" executed in /usr/lib/ we once again get a little further :

```
[root@localhost simplesim-2.0]# make
gcc -o sim-fast `./sysprobe -flags` -DDEBUG -O   sim-fast.o main.o syscall.o memory.o regs.o loader.
gcc -o sim-cheetah `./sysprobe -flags` -DDEBUG -O   sim-cheetah.o main.o syscall.o memory.o regs.o l
....\\
gcc `./sysprobe -flags` -DDEBUG -O   -c sim-cache.c
gcc `./sysprobe -flags` -DDEBUG -O   -c cache.c
cache.c: In function `cache_access':
cache.c:529: conflicting types for `random'
/usr/include/stdlib.h:346: previous declaration of `random'
cache.c:529: warning: extern declaration of `random' doesn't match global one
make: *** [cache.o] Error 1
[root@localhost simplesim-2.0]#
```

The following declarations in cache.c (appproximately at line 529) have been commented out :

```
  /*
#ifndef __alpha__
 extern long random(void);
#endif
  */
```

Running make completes this task!!!!!
Changing directory to gcc-2.6.3, and running the configure script, causes no problems.
make LANGUAGES=c yields :

```
[root@localhost gcc-2.6.3]# make LANGUAGES=c
gcc -DCROSS_COMPILE -DIN_GCC -DPOSIX  -g     -I. -I. -I./config \
  -DGCC_INCLUDE_DIR=\"/home/Jesper/ss/lib/gcc-lib/sslittle-na-sstrix/2.6.3/include\" \
  -DGPLUSPLUS_INCLUDE_DIR=\"/home/Jesper/ss/lib/g++-include\" \
  -DLOCAL_INCLUDE_DIR=\"/usr/local/include\" \
  -DCROSS_INCLUDE_DIR=\"/home/Jesper/ss/lib/gcc-lib/sslittle-na-sstrix/2.6.3/sys-include\" \
```

```
  -DTOOL_INCLUDE_DIR=\"/home/Jesper/ss/sslittle-na-sstrix/include\" \
  -c `echo ./cccp.c | sed 's,^\./,,'`
cccp.c:194: conflicting types for `sys_errlist'
/usr/include/stdio.h:557: previous declaration of `sys_errlist'
make: *** [cccp.o] Error 1
[root@localhost gcc-2.6.3]#
```

Change

```
"#if defined(bsd4_4) || defined(__NetBSD__)"
```

to

```
"#if defined(bsd4_4) || defined(__NetBSD__) || defined(__linux__)"
```

and the problems with cccp.c disappears
running make LANGUAGES=c yields :

```
......
gcc -c -DCROSS_COMPILE -DIN_GCC -DPOSIX  -g     -I. -I. -I./config emit-rtl.c
gcc -c -DCROSS_COMPILE -DIN_GCC -DPOSIX  -g     -I. -I. -I./config real.c
gcc -c -DCROSS_COMPILE -DIN_GCC -DPOSIX  -g     -I. -I. -I./config dbxout.c
gcc -c -DCROSS_COMPILE -DIN_GCC -DPOSIX  -g     -I. -I. -I./config sdbout.c
sdbout.c:57: syms.h: No such file or directory
make: *** [sdbout.o] Error 1
[root@localhost gcc-2.6.3]#
```

Instead of

```
"#include <syms.h>"
```

you should use :

```
"#include <gsyms.h>"
```

running make LANGUAGES=c gives :

```
.....
-DTOOLDIR_BASE_PREFIX=\"/home/Jesper/ss/\" \
   \
  -c `echo ./gcc.c | sed 's,^\./,,'`
gcc.c:172: conflicting types for `sys_errlist'
/usr/include/stdio.h:557: previous declaration of `sys_errlist'
make: *** [gcc.o] Error 1
[root@localhost gcc-2.6.3]#
```

Change

```
"#if defined(bsd4_4) || defined(__NetBSD__)"
```

to

```
"#if defined(bsd4_4) || defined(__NetBSD__) || defined(__linux__)"
```

and the problems with gcc.c disappears
running make LANGUAGES=c gives :

```
gcc -DCROSS_COMPILE -DIN_GCC -DPOSIX  -g  -I. -I. -I./config  -o g++-cross \
   -DGCC_NAME=\"\" ./cp/g++.c version.o obstack.o ` case "gcc" in "cc") echo "" ;; esac `
```

```
./cp/g++.c:90: conflicting types for ‘sys_errlist’
/usr/include/stdio.h:557: previous declaration of ‘sys_errlist’
make: *** [g++-cross] Error 1
[root@localhost gcc-2.6.3]#
```

Change

`"#if defined(bsd4_4) || defined(__NetBSD__)"`
to
`"#if defined(bsd4_4) || defined(__NetBSD__) || defined(__linux__)"`

and the problems with cp/g++.c disappears
running make LANGUAGES=c now compiles without problems, and the following tests causes no problems.

Appendix C

Debugging tests.

C.1 Dependence tests.

In this section the testing of the developed dependence routines will be performed. For each of the listed programs, the dependence vectors which describe the dependencies in the nested loop are traversed in the order they appear in the printouts. The following printouts which start with : `Using dependence vectors :`, show how many of the dependence vectors it was necessary to examine, before a conclusion of the legality of the particular tiling could be reached. This conclusion is subsequently printet out.

```
#define N 10
int A[N][N][N][N][N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                A[i][j][k][l][m] = A[i][j+1][k][l][m];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test1.spd dep_test1.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[i,j]
A[i+1,j]
A[i,j]
B[j]
B[j]
B[j+1]
C[k]
C[k]
C[k+1]
Calculating dependences for lhs access:
```

```
A[i,j]
( 0 0 + * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( 1 0 * * * )
( 0 0 + * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )

Calculating dependences for lhs access:
B[j]
( + 0 * * * )
( 0 0 + * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + 0 * * * )
( 0 0 + * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + 1 * * * )
( 0 1 * * * )

Calculating dependences for lhs access:
C[k]
( + * 0 * * )
( 0 + 0 * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + * 0 * * )
( 0 + 0 * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + * 1 * * )
( 0 + 1 * * )
( 0 0 1 * * )


Using dependence vectors : 1
ILLEGAL to tile loops 1 to 5

Using dependence vectors : 1
ILLEGAL to tile loops 2 to 5

Using dependence vectors : 1
ILLEGAL to tile loops 3 to 5

Using dependence vectors : 1 2
ILLEGAL to tile loops 4 to 5
*/
```

```
*********************************************************************

int B[100];

test1()
{
    int i, j;
    for(i=0; i<100; i++)
        for(j=1; j<100; j++)
                B[j] = B[j-1] + B[j+1];
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test3.spd dep_test3.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 2
This loop contains the following accesses :
B[j]
B[j-1]
B[j+1]
Calculating dependences for lhs access:
B[j]
( + 0 )
( 0 0 )
( + -1 )
( + 1 )
( 0 1 )


Using dependence vectors : 1 2 3
ILLEGAL to tile loops 1 to 2
*/

*********************************************************************

int B[100];

// No tiling of this loop is legal

test1()
{
    int i, j, k, l, m;
    for(i=0; i<100; i++)
        for(j=0; j<100; j++)
          for(k=0; k<100; k++)
            for(l=0; l<100; l++)
              for(m=0; m<100; m++){
                B[j] = B[j] + B[j+1];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test6.spd dep_test6.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
```

```
B[j]
B[j]
B[j+1]
Calculating dependences for lhs access:
B[j]
( + 0 * * * )
( 0 0 + * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + 0 * * * )
( 0 0 + * * )
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + 1 * * * )
( 0 1 * * * )


Using dependence vectors : 1
ILLEGAL to tile loops 1 to 5

Using dependence vectors : 1 2
ILLEGAL to tile loops 2 to 5

Using dependence vectors : 1 2
ILLEGAL to tile loops 3 to 5

Using dependence vectors : 1 2 3
ILLEGAL to tile loops 4 to 5
*/

*********************************************************************

#define N 10
int A[N][N][N][N][N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                A[i][j+1][k][l+1][m+1] = A[i][j][k][l][m];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test7.spd dep_test7.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[i,j,k,l,m]
A[i,j+1,k,l+1,m+1]
```

```
Calculating dependences for lhs access:
A[i,j,k,l,m]
( 0 0 0 0 0 )
( 0 1 0 1 1 )


Using dependence vectors : 1 2
OK to tile loops 1 to 5

Using dependence vectors : 1 2
OK to tile loops 2 to 5

Using dependence vectors : 1 2
OK to tile loops 3 to 5

Using dependence vectors : 1 2
OK to tile loops 4 to 5
*/

*********************************************************************

#define N 10
int A[N][N][N];

// OK to tile loops : (2,3,4,5)(3,4,5)(4,5)

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                A[k][l][m] = A[k][l+1][m+1];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test8.spd dep_test8.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[k,l,m]
A[k,l+1,m+1]
Calculating dependences for lhs access:
A[k,l,m]
( + * 0 0 0 )
( 0 + 0 0 0 )
( 0 0 0 0 0 )
( + * 0 1 1 )
( 0 + 0 1 1 )
( 0 0 0 1 1 )


Using dependence vectors : 1
```

```
ILLEGAL to tile loops 1 to 5

Using dependence vectors : 1 2 3 4 5 6
OK to tile loops 2 to 5

Using dependence vectors : 1 2 3 4 5 6
OK to tile loops 3 to 5

Using dependence vectors : 1 2 3 4 5 6
OK to tile loops 4 to 5
*/

*********************************************************************

#define N 10
int A[N][N][N];

// Not ok to tile any of the loops

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                A[i][j][k] = A[i][j+1][k+1];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test9.spd dep_test9.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[i,j,k]
A[i,j+1,k+1]
Calculating dependences for lhs access:
A[i,j,k]
( 0 0 0 + * )
( 0 0 0 0 + )
( 0 0 0 0 0 )
( 0 1 1 * * )


Using dependence vectors : 1
ILLEGAL to tile loops 1 to 5

Using dependence vectors : 1
ILLEGAL to tile loops 2 to 5

Using dependence vectors : 1
ILLEGAL to tile loops 3 to 5

Using dependence vectors : 1
```

```
ILLEGAL to tile loops 4 to 5
*/

*********************************************************************

#define N 10
int A[N][N];

// OK to tile loops : (3,4,5),(4,5)

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                A[l][m] = A[l][m+1] + A[l+1][m];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test10.spd dep_test10.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[l,m]
A[l,m+1]
A[l+1,m]
Calculating dependences for lhs access:
A[l,m]
( + * * 0 0 )
( 0 + * 0 0 )
( 0 0 + 0 0 )
( 0 0 0 0 0 )
( + * * 0 1 )
( 0 + * 0 1 )
( 0 0 + 0 1 )
( 0 0 0 0 1 )
( + * * 1 0 )
( 0 + * 1 0 )
( 0 0 + 1 0 )
( 0 0 0 1 0 )


Using dependence vectors : 1
ILLEGAL to tile loops 1 to 5

Using dependence vectors : 1 2
ILLEGAL to tile loops 2 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9 10 11 12
OK to tile loops 3 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9 10 11 12
```

```
OK to tile loops 4 to 5
*/

*******************************************************************

#define N 10
int A[N][N][N][N][N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                  A[i][j][k][l][m] =A[i+j][j+1][k][l+k][m+l];
              }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test11.spd dep_test11.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[i,j,k,l,m]
A[j+i,j+1,k,k+l,l+m]
Calculating dependences for lhs access:
A[i,j,k,l,m]
( 0 0 0 0 0 )
( + 1 0 + + )
( + 1 0 + 0 )
( + 1 0 0 + )
( + 1 0 0 0 )
( 0 1 0 + + )
( 0 1 0 + 0 )
( 0 1 0 0 + )
( 0 1 0 0 0 )


Using dependence vectors : 1 2 3 4 5 6 7 8 9
OK to tile loops 1 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9
OK to tile loops 2 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9
OK to tile loops 3 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9
OK to tile loops 4 to 5
*/

*******************************************************************

#define N 10
```

```
int A[N][N][N][N][N];
int B[N][N][N][N];
int x=0,y=0,z=0,q=0;

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      x++;
      for(j=0; j<N; j++)
        for(k=0; k<N; k++){
          y++;
          for(l=0; l<N; l++){
              z++;
              for(m=0; m<N; m++){
                  A[i][j][k][l][m] = A[i+j][j+1][k][l+k][m+l];
                  B[i][j][k][l] = B[i+j][j+1][k][l+k];
              }
          }
        }
    }
}

/*
[Jesper@localhost dep_test]$ ./my_tile dep_test12.spd dep_test12.out
=====Processing the test1 procedure=====
Calculating dep. for loop with depth : 5
This loop contains the following accesses :
A[i,j,k,l,m]
A[j+i,j+1,k,k+l,l+m]
B[i,j,k,l]
B[j+i,j+1,k,k+l]
Calculating dependences for lhs access:
A[i,j,k,l,m]
( 0 0 0 0 0 )
( + 1 0 + + )
( + 1 0 + 0 )
( + 1 0 0 + )
( + 1 0 0 0 )
( 0 1 0 + + )
( 0 1 0 + 0 )
( 0 1 0 0 + )
( 0 1 0 0 0 )

Calculating dependences for lhs access:
B[i,j,k,l]
( 0 0 0 0 + )
( 0 0 0 0 0 )
( + 1 0 + * )
( + 1 0 0 * )
( 0 1 0 + * )
( 0 1 0 0 * )


Using dependence vectors : 1 2 3 4 5 6 7 8 9 10 11 12
ILLEGAL to tile loops 1 to 5
```

```
Using dependence vectors : 1 2 3 4 5 6 7 8 9 10 11 12 13 14
ILLEGAL to tile loops 2 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
OK to tile loops 3 to 5

Using dependence vectors : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
OK to tile loops 4 to 5
*/
```

C.2 Uniformly generated sets.

In this section a testing of the tile-pass' ability to partition the references into uniformly generated sets will be performed. An identification of the sets dimensions of self-temporal and self-spatial reuse is also carried out. These results are also printet out by the tile-pass.

```
#define N 10
int A[N][N][N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
          for(k=0; k<N; k++)
            for(l=0; l<N; l++)
              for(m=0; m<N; m++){
                A[k][l][m] = A[k][l+1][m+1] + A[i][l+1][m+1] + A[k][j+1][m+1] +
                  A[i][j+1][k+1] + A[i][j][k] + A[2*k][l+1][m+1] +
                  A[k][2*l+1][m+1] + A[k][2*l][m] + A[k][2*l+1][m+1];
              }
}

/*
[Jesper@localhost locality_test]$ ./my_tile uni_gen_set_test1.spd uni_gen_set_test1.out
=====Processing the test1 procedure=====


==== Printing out uniformly generated sets for a loop nest ====

**** Printing uniformly generated sets for loop 1 ****


**** Printing uniformly generated sets for loop 2 ****


**** Printing uniformly generated sets for loop 3 ****


**** Printing uniformly generated sets for loop 4 ****


**** Printing uniformly generated sets for loop 5 ****
```

```
Printing uniformly generated sets for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[k,l,m]
A[k,l+1,m+1]
The access matrix for the set is :
 0   0   1   0   0
 0   0   0   1   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  1 2
The references exhibit self-spatial reuse in loops :  1 2 5

The uniformly generated set contains 1 references :
The members of the set are :
A[i,l+1,m+1]
The access matrix for the set is :
 1   0   0   0   0
 0   0   0   1   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  2 3
The references exhibit self-spatial reuse in loops :  2 3 5

The uniformly generated set contains 1 references :
The members of the set are :
A[k,j+1,m+1]
The access matrix for the set is :
 0   0   1   0   0
 0   1   0   0   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  1 4
The references exhibit self-spatial reuse in loops :  1 4 5

The uniformly generated set contains 2 references :
The members of the set are :
A[i,j+1,k+1]
A[i,j,k]
The access matrix for the set is :
 1   0   0   0   0
 0   1   0   0   0
 0   0   1   0   0
The references exhibit self-temporal reuse in loops :  4 5
The references exhibit self-spatial reuse in loops :  3 4 5

The uniformly generated set contains 1 references :
The members of the set are :
A[2*k,l+1,m+1]
The access matrix for the set is :
 0   0   2   0   0
 0   0   0   1   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  1 2
The references exhibit self-spatial reuse in loops :  1 2 5

The uniformly generated set contains 3 references :
The members of the set are :
```

```
A[k,2*l+1,m+1]
A[k,2*l,m]
A[k,2*l+1,m+1]
The access matrix for the set is :
 0   0   1   0   0
 0   0   0   2   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  1 2
The references exhibit self-spatial reuse in loops :  1 2 5
*/

************************************************************************

#define N 10
int A[N][N][N];
int B[N][N][N];
int C[N][N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        C[j][i] = C[j][i+1] + C[2*i][j] + A[i][j][3];
        for(k=0; k<N; k++){
          A[i][j][k] = A[i][j][k+1] + A[5*i][5*j][k] + A[i*5][j*5][k+7];
          B[i][j][k] = B[i][j][k+1] + B[5*i][5*j][k] + B[i*5][j*5][k+7];
          for(l=0; l<N; l++){
            for(m=0; m<N; m++){
              A[k][l][m] = A[k][l+1][m+1] + A[i][l+1][m+1];
            }
          }
        }
      }
    }
}

/*
=====Processing the test1 procedure=====


==== Printing out uniformly generated sets for a loop nest ====

**** Printing uniformly generated sets for loop 1 ****


**** Printing uniformly generated sets for loop 2 ****

Printing uniformly generated sets for array : .C

The uniformly generated set contains 2 references :
The members of the set are :
C[j,i]
C[j,i+1]
The access matrix for the set is :
 0   1   0   0   0
```

```
 1   0   0   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  1

The uniformly generated set contains 1 references :
The members of the set are :
C[2*i,j]
The access matrix for the set is :
 2   0   0   0   0
 0   1   0   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  2

Printing uniformly generated sets for array : .A

The uniformly generated set contains 1 references :
The members of the set are :
A[i,j,3]
The access matrix for the set is :
 1   0   0   0   0
 0   1   0   0   0
 0   0   0   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :


**** Printing uniformly generated sets for loop 3 ****

Printing uniformly generated sets for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[i,j,k]
A[i,j,k+1]
The access matrix for the set is :
 1   0   0   0   0
 0   1   0   0   0
 0   0   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3

The uniformly generated set contains 2 references :
The members of the set are :
A[5*i,5*j,k]
A[5*i,5*j,k+7]
The access matrix for the set is :
 5   0   0   0   0
 0   5   0   0   0
 0   0   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3

Printing uniformly generated sets for array : .B

The uniformly generated set contains 2 references :
The members of the set are :
```

```
B[i,j,k]
B[i,j,k+1]
The access matrix for the set is :
 1   0   0   0   0
 0   1   0   0   0
 0   0   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3

The uniformly generated set contains 2 references :
The members of the set are :
B[5*i,5*j,k]
B[5*i,5*j,k+7]
The access matrix for the set is :
 5   0   0   0   0
 0   5   0   0   0
 0   0   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3


**** Printing uniformly generated sets for loop 4 ****


**** Printing uniformly generated sets for loop 5 ****

Printing uniformly generated sets for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[k,l,m]
A[k,l+1,m+1]
The access matrix for the set is :
 0   0   1   0   0
 0   0   0   1   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  1 2
The references exhibit self-spatial reuse in loops :  1 2 5

The uniformly generated set contains 1 references :
The members of the set are :
A[i,l+1,m+1]
The access matrix for the set is :
 1   0   0   0   0
 0   0   0   1   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  2 3
The references exhibit self-spatial reuse in loops :  2 3 5
*/

************************************************************************

#define N 10
int A[N][N][N];
int B[N][N][N];
int C[N][N];
```

```
test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        C[j+i][i+j] = C[j][i+1+j];
        for(k=0; k<N; k++){
          A[i][j+k][k+j] = A[i][j+k][j+k+1];
          B[i][j][k] = B[i+j+k][i+j+k][i+j+k];
          for(l=0; l<N; l++){
            for(m=0; m<N; m++){
              A[k][l][m] = A[k][l+1][m+1] + A[i][l+1][m+1+l];
            }
          }
        }
      }
    }
}

/*
[Jesper@localhost locality_test]$ ./my_tile uni_gen_set_test3.spd uni_gen_set_test3.out
=====Processing the test1 procedure=====


==== Printing out uniformly generated sets for a loop nest ====

**** Printing uniformly generated sets for loop 1 ****


**** Printing uniformly generated sets for loop 2 ****

Printing uniformly generated sets for array : .C

The uniformly generated set contains 1 references :
The members of the set are :
C[i+j,j+i]
The access matrix for the set is :
 1   1   0   0   0
 1   1   0   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :

The uniformly generated set contains 1 references :
The members of the set are :
C[j,j+i+1]
The access matrix for the set is :
 0   1   0   0   0
 1   1   0   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  1


**** Printing uniformly generated sets for loop 3 ****

Printing uniformly generated sets for array : .A
```

```
The uniformly generated set contains 2 references :
The members of the set are :
A[i,k+j,j+k]
A[i,k+j,k+j+1]
The access matrix for the set is :
 1   0   0   0   0
 0   1   1   0   0
 0   1   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :

Printing uniformly generated sets for array : .B

The uniformly generated set contains 1 references :
The members of the set are :
B[i,j,k]
The access matrix for the set is :
 1   0   0   0   0
 0   1   0   0   0
 0   0   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3

The uniformly generated set contains 1 references :
The members of the set are :
B[k+j+i,k+j+i,k+j+i]
The access matrix for the set is :
 1   1   1   0   0
 1   1   1   0   0
 1   1   1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :


**** Printing uniformly generated sets for loop 4 ****


**** Printing uniformly generated sets for loop 5 ****

Printing uniformly generated sets for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[k,l,m]
A[k,l+1,m+1]
The access matrix for the set is :
 0   0   1   0   0
 0   0   0   1   0
 0   0   0   0   1
The references exhibit self-temporal reuse in loops :  1 2
The references exhibit self-spatial reuse in loops :  1 2 5

The uniformly generated set contains 1 references :
The members of the set are :
A[i,l+1,l+m+1]
```

```
The access matrix for the set is :
 1   0   0   0   0
 0   0   0   1   0
 0   0   0   1   1
The references exhibit self-temporal reuse in loops :  2 3
The references exhibit self-spatial reuse in loops :  2 3 5
*/
```

C.3 Evaluation tests.

In this section, all of the most important abilities of the tile-pass are tested. The conducted tests determines the pass' ability to :

- Generate uniformly generated sets.
- Partition the references in each uniformly generated set into equivalence classes.
- Calculate correct tile sizes for the different localizations of a nest.
- Calculate the correct number of estimated off-chip accesses

Printouts of the performed tests are listed in the following :

```
#define N 1000

int A[N][N];
int B[N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
                B[j] = B[j] + B[j+1];
                A[i][j] = A[i-1][j] + A[i][j-1] + A[i][j] + A[i][j+1] +
                          A[i+1][j];
      }
    }
}

void main(){
  test1();
}

// SOR (Successive Over Relaxation - code)

/*
line_size : 32    cache_size : 2048

=====Processing the test1 procedure=====

==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****


**** Printing uni_gen_set's and eq_class'es for loop 2 ****

Printing uni_gen_set's and eq_class'es for array : .B
```

```
The uniformly generated set contains 3 references :
The members of the set are :
B[j]
B[j]
B[j+1]
The access matrix for the set is :
 0   1
The references exhibit self-temporal reuse in loops :  1
The references exhibit self-spatial reuse in loops :  1 2
The generated eq_class'es for local space 1-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[j]
B[j]
B[j+1]
The generated eq_class'es for local space 2-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[j]
B[j]
B[j+1]

Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 6 references :
The members of the set are :
A[i,j]
A[i-1,j]
A[i,j-1]
A[i,j]
A[i,j+1]
A[i+1,j]
The access matrix for the set is :
 1   0
 0   1
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  2
The generated eq_class'es for local space 1-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
A[i,j]
A[i-1,j]
A[i,j-1]
A[i,j]
A[i,j+1]
A[i+1,j]
The generated eq_class'es for local space 2-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
A[i,j]
A[i,j-1]
A[i,j]
A[i,j+1]
Set no. 2 :
A[i-1,j]
```

```
Set no. 3 :
A[i+1,j]


====== The collected info for local space 1-2 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 2-2 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 3   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 2 estimations =====
Tile-size 8 :  WS-size estimation = 288 ,No of off-chip acc. = 140625
Tile-size 16 :  WS-size estimation = 1088 ,No of off-chip acc. = 132812

===== Tiling of loops 2 - 2 estimations =====
Tile-size 8 :  WS-size estimation = 128 ,No of off-chip acc. = 500000
Tile-size 16 :  WS-size estimation = 256 ,No of off-chip acc. = 500000
Tile-size 24 :  WS-size estimation = 384 ,No of off-chip acc. = 500000
Tile-size 32 :  WS-size estimation = 512 ,No of off-chip acc. = 500000
Tile-size 40 :  WS-size estimation = 640 ,No of off-chip acc. = 500000
Tile-size 48 :  WS-size estimation = 768 ,No of off-chip acc. = 500000
Tile-size 56 :  WS-size estimation = 896 ,No of off-chip acc. = 500000
Tile-size 64 :  WS-size estimation = 1024 ,No of off-chip acc. = 500000
Tile-size 72 :  WS-size estimation = 1152 ,No of off-chip acc. = 500000
Tile-size 80 :  WS-size estimation = 1280 ,No of off-chip acc. = 500000
Tile-size 88 :  WS-size estimation = 1408 ,No of off-chip acc. = 500000
Tile-size 96 :  WS-size estimation = 1536 ,No of off-chip acc. = 500000
Tile-size 104 :  WS-size estimation = 1664 ,No of off-chip acc. = 500000
Tile-size 112 :  WS-size estimation = 1792 ,No of off-chip acc. = 500000
Tile-size 120 :  WS-size estimation = 1920 ,No of off-chip acc. = 500000

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 2 :
Tile-size 16 :  WS-size estimation = 1088 ,No of off-chip acc. = 132812

Optimal results for tiling of loops 2 - 2 :
Tile-size 120 :  WS-size estimation = 1920 ,No of off-chip acc. = 500000
```

```
=====Processing the main procedure=====
*/

************************************************************************

#define N 100

int A[N][N];
int B[N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        for(k=0; k<N; k++){
                B[j] = B[j] + B[j+1];
                A[i][j] = A[i-1][j];
        }
      }
    }
}

void main(){
  test1();
}

/*
line_size : 32    cache_size : 16384

=====Processing the test1 procedure=====

==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****


**** Printing uni_gen_set's and eq_class'es for loop 2 ****


**** Printing uni_gen_set's and eq_class'es for loop 3 ****

Printing uni_gen_set's and eq_class'es for array : .B

The uniformly generated set contains 3 references :
The members of the set are :
B[j]
B[j]
B[j+1]
The access matrix for the set is :
 0   1   0
The references exhibit self-temporal reuse in loops :  1 3
The references exhibit self-spatial reuse in loops :  1 2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
```

```
The eq_class exhibits group temporal reuse among its references
B[j]
B[j]
B[j+1]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[j]
B[j]
B[j+1]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
The eq_class exhibits group spatial reuse among its references
B[j]
B[j]
B[j+1]

Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[i,j]
A[i-1,j]
The access matrix for the set is :
 1   0   0
 0   1   0
The references exhibit self-temporal reuse in loops :  3
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
A[i,j]
A[i-1,j]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i,j]
Set no. 2 :
A[i-1,j]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
A[i,j]
Set no. 2 :
A[i-1,j]


====== The collected info for local space 1-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 2
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8
```

```
====== The collected info for local space 2-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 2   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 3-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 2   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 288 ,No of off-chip acc. = 17578
Tile-size 16 :  WS-size estimation = 1088 ,No of off-chip acc. = 8300
Tile-size 24 :  WS-size estimation = 2400 ,No of off-chip acc. = 5425
Tile-size 32 :  WS-size estimation = 4224 ,No of off-chip acc. = 4028
Tile-size 40 :  WS-size estimation = 6560 ,No of off-chip acc. = 3203
Tile-size 48 :  WS-size estimation = 9408 ,No of off-chip acc. = 2658
Tile-size 56 :  WS-size estimation = 12768 ,No of off-chip acc. = 2271

===== Tiling of loops 2 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 96 ,No of off-chip acc. = 46875
Tile-size 16 :  WS-size estimation = 192 ,No of off-chip acc. = 23437
Tile-size 24 :  WS-size estimation = 288 ,No of off-chip acc. = 15624
Tile-size 32 :  WS-size estimation = 384 ,No of off-chip acc. = 11718
Tile-size 40 :  WS-size estimation = 480 ,No of off-chip acc. = 9375
Tile-size 48 :  WS-size estimation = 576 ,No of off-chip acc. = 7812
Tile-size 56 :  WS-size estimation = 672 ,No of off-chip acc. = 6696
Tile-size 64 :  WS-size estimation = 768 ,No of off-chip acc. = 5859
Tile-size 72 :  WS-size estimation = 864 ,No of off-chip acc. = 5208
Tile-size 80 :  WS-size estimation = 960 ,No of off-chip acc. = 4687
Tile-size 88 :  WS-size estimation = 1056 ,No of off-chip acc. = 4260
Tile-size 96 :  WS-size estimation = 1152 ,No of off-chip acc. = 3906
Tile-size 104 :  WS-size estimation = 1248 ,No of off-chip acc. = 3604

===== Tiling of loops 3 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 96 ,No of off-chip acc. = 375000
Tile-size 16 :  WS-size estimation = 96 ,No of off-chip acc. = 187500
Tile-size 24 :  WS-size estimation = 96 ,No of off-chip acc. = 124999
Tile-size 32 :  WS-size estimation = 96 ,No of off-chip acc. = 93750
Tile-size 40 :  WS-size estimation = 96 ,No of off-chip acc. = 75000
Tile-size 48 :  WS-size estimation = 96 ,No of off-chip acc. = 62499
Tile-size 56 :  WS-size estimation = 96 ,No of off-chip acc. = 53571
```

```
Tile-size 64 :  WS-size estimation = 96 ,No of off-chip acc. = 46875
Tile-size 72 :  WS-size estimation = 96 ,No of off-chip acc. = 41665
Tile-size 80 :  WS-size estimation = 96 ,No of off-chip acc. = 37500
Tile-size 88 :  WS-size estimation = 96 ,No of off-chip acc. = 34090
Tile-size 96 :  WS-size estimation = 96 ,No of off-chip acc. = 31249
Tile-size 104 :  WS-size estimation = 96 ,No of off-chip acc. = 28845

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 3 :
Tile-size 56 :  WS-size estimation = 12768 ,No of off-chip acc. = 2271

Optimal results for tiling of loops 2 - 3 :
Tile-size 104 :  WS-size estimation = 1248 ,No of off-chip acc. = 3604

Optimal results for tiling of loops 3 - 3 :
Tile-size 104 :  WS-size estimation = 96 ,No of off-chip acc. = 28845

=====Processing the main procedure=====
*/

***********************************************************************

#define N 100

int A[N][N];
int B[N][N][N];
int C[N];

test1()
{
    int i, j, k;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        for(k=0; k<N; k++){
          A[i][k] = A[i-1][k];
          B[i][3][j] = B[i][j][k] + B[i+2][j][k];
        }
      }
    }
}

/*
line_size : 32    cache_size : 16384

=====Processing the test1 procedure=====

==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****


**** Printing uni_gen_set's and eq_class'es for loop 2 ****


**** Printing uni_gen_set's and eq_class'es for loop 3 ****
```

```
Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[i,k]
A[i-1,k]
The access matrix for the set is :
 1   0   0
 0   0   1
The references exhibit self-temporal reuse in loops :  2
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
A[i,k]
A[i-1,k]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i,k]
Set no. 2 :
A[i-1,k]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
A[i,k]
Set no. 2 :
A[i-1,k]

Printing uni_gen_set's and eq_class'es for array : .B

The uniformly generated set contains 1 references :
The members of the set are :
B[i,3,j]
The access matrix for the set is :
 1   0   0
 0   0   0
 0   1   0
The references exhibit self-temporal reuse in loops :  3
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
B[i,3,j]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
B[i,3,j]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
B[i,3,j]

The uniformly generated set contains 2 references :
The members of the set are :
B[i,j,k]
B[i+2,j,k]
The access matrix for the set is :
 1   0   0
 0   1   0
```

```
 0   0   1
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[i,j,k]
B[i+2,j,k]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
B[i,j,k]
Set no. 2 :
B[i+2,j,k]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
B[i,j,k]
Set no. 2 :
B[i+2,j,k]


====== The collected info for local space 1-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 2-3 ======

no_of_eq_classes = 2   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 3-3 ======

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8
```

```
no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 2560 ,No of off-chip acc. = 156250

===== Tiling of loops 2 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 608 ,No of off-chip acc. = 296875
Tile-size 16 :  WS-size estimation = 2240 ,No of off-chip acc. = 273437
Tile-size 24 :  WS-size estimation = 4896 ,No of off-chip acc. = 265624
Tile-size 32 :  WS-size estimation = 8576 ,No of off-chip acc. = 261718
Tile-size 40 :  WS-size estimation = 13280 ,No of off-chip acc. = 259375

===== Tiling of loops 3 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 160 ,No of off-chip acc. = 625000
Tile-size 16 :  WS-size estimation = 288 ,No of off-chip acc. = 562500
Tile-size 24 :  WS-size estimation = 416 ,No of off-chip acc. = 541666
Tile-size 32 :  WS-size estimation = 544 ,No of off-chip acc. = 531250
Tile-size 40 :  WS-size estimation = 672 ,No of off-chip acc. = 525000
Tile-size 48 :  WS-size estimation = 800 ,No of off-chip acc. = 520833
Tile-size 56 :  WS-size estimation = 928 ,No of off-chip acc. = 517857
Tile-size 64 :  WS-size estimation = 1056 ,No of off-chip acc. = 515625
Tile-size 72 :  WS-size estimation = 1184 ,No of off-chip acc. = 513888
Tile-size 80 :  WS-size estimation = 1312 ,No of off-chip acc. = 512500
Tile-size 88 :  WS-size estimation = 1440 ,No of off-chip acc. = 511363
Tile-size 96 :  WS-size estimation = 1568 ,No of off-chip acc. = 510416
Tile-size 104 :  WS-size estimation = 1696 ,No of off-chip acc. = 509615

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 3 :
Tile-size 8 :  WS-size estimation = 2560 ,No of off-chip acc. = 156250

Optimal results for tiling of loops 2 - 3 :
Tile-size 40 :  WS-size estimation = 13280 ,No of off-chip acc. = 259375

Optimal results for tiling of loops 3 - 3 :
Tile-size 104 :  WS-size estimation = 1696 ,No of off-chip acc. = 509615
*/

***********************************************************************

#define N 100

int A[N][2*N][2*N];
int B[N];
int C[N];
```

```
test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        B[j] = B[j] + B[j+1];
        for(k=0; k<N; k++){
          A[i][j][k] = A[i][j+k][j+k] + A[i][j+k][j+k+1] +
                       A[i][j+k+1][j+k] + A[i][j+k][j+k-3];
        }
      }
    }
}

void main(){
  test1();
}

// This test contains an example of a wrong partitioning of references into
// equivalence classes. This flaw has the result that for the localization
// of loops 2-3 an extra equivalence class is generated.
//

/*
line_size : 32    cache_size : 65536

=====Processing the test1 procedure=====

==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****


**** Printing uni_gen_set's and eq_class'es for loop 2 ****

Printing uni_gen_set's and eq_class'es for array : .B

The uniformly generated set contains 3 references :
The members of the set are :
B[j]
B[j]
B[j+1]
The access matrix for the set is :
 0   1   0
The references exhibit self-temporal reuse in loops :  1
The references exhibit self-spatial reuse in loops :  1 2
The generated eq_class'es for local space 1-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[j]
B[j]
B[j+1]
The generated eq_class'es for local space 2-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[j]
```

```
B[j]
B[j+1]


**** Printing uni_gen_set's and eq_class'es for loop 3 ****

Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 1 references :
The members of the set are :
A[i,j,k]
The access matrix for the set is :
 1   0   0
 0   1   0
 0   0   1
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
A[i,j,k]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i,j,k]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
A[i,j,k]

The uniformly generated set contains 4 references :
The members of the set are :
A[i,k+j,k+j]
A[i,k+j,k+j+1]
A[i,k+j+1,k+j]
A[i,k+j,k+j-3]
The access matrix for the set is :
 1   0   0
 0   1   1
 0   1   1
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :
The generated eq_class'es for local space 1-3 :
Set no. 1 :
The eq_class exhibits group spatial reuse among its references
A[i,k+j,k+j]
A[i,k+j,k+j+1]
A[i,k+j+1,k+j]
A[i,k+j,k+j-3]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
The eq_class exhibits group spatial reuse among its references
A[i,k+j,k+j]
A[i,k+j,k+j+1]
A[i,k+j+1,k+j]
A[i,k+j,k+j-3]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
The eq_class exhibits group spatial reuse among its references
```

```
A[i,k+j,k+j]
A[i,k+j,k+j+1]
A[i,k+j,k+j-3]
Set no. 2 :
A[i,k+j+1,k+j]


====== The collected info for local space 1-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 10000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1


====== The collected info for local space 2-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 10000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1


====== The collected info for local space 3-3 ======

no_of_eq_classes = 3   no_of_ST_reuse_dims = 0
no_of_iterations = 10000
SS_reuse_divisor = line_size/elem_size*stride = 32/1*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 18688 ,No of off-chip acc. = 1125156
```

```
===== Tiling of loops 2 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 2560 ,No of off-chip acc. = 1126250
Tile-size 16 :  WS-size estimation = 10240 ,No of off-chip acc. = 1126250
Tile-size 24 :  WS-size estimation = 23040 ,No of off-chip acc. = 1126250
Tile-size 32 :  WS-size estimation = 40960 ,No of off-chip acc. = 1126250
Tile-size 40 :  WS-size estimation = 64000 ,No of off-chip acc. = 1126250

===== Tiling of loops 3 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 1312 ,No of off-chip acc. = 2155000
Tile-size 16 :  WS-size estimation = 2624 ,No of off-chip acc. = 2155000
Tile-size 24 :  WS-size estimation = 3936 ,No of off-chip acc. = 2155000
Tile-size 32 :  WS-size estimation = 5248 ,No of off-chip acc. = 2155000
Tile-size 40 :  WS-size estimation = 6560 ,No of off-chip acc. = 2155000
Tile-size 48 :  WS-size estimation = 7872 ,No of off-chip acc. = 2155000
Tile-size 56 :  WS-size estimation = 9184 ,No of off-chip acc. = 2155000
Tile-size 64 :  WS-size estimation = 10496 ,No of off-chip acc. = 2155000
Tile-size 72 :  WS-size estimation = 11808 ,No of off-chip acc. = 2155000
Tile-size 80 :  WS-size estimation = 13120 ,No of off-chip acc. = 2155000
Tile-size 88 :  WS-size estimation = 14432 ,No of off-chip acc. = 2155000
Tile-size 96 :  WS-size estimation = 15744 ,No of off-chip acc. = 2155000
Tile-size 104 :  WS-size estimation = 17056 ,No of off-chip acc. = 2155000

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 3 :
Tile-size 8 :  WS-size estimation = 18688 ,No of off-chip acc. = 1125156

Optimal results for tiling of loops 2 - 3 :
Tile-size 40 :  WS-size estimation = 64000 ,No of off-chip acc. = 1126250

Optimal results for tiling of loops 3 - 3 :
Tile-size 104 :  WS-size estimation = 17056 ,No of off-chip acc. = 2155000

=====Processing the main procedure=====
*/

**********************************************************************

#define N 100

int A[N][N];
int B[N][N][N];
int C[N];

test1()
{
    int i, j, k;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        for(k=0; k<N; k++){
          A[i][3*j] = A[i-1][5*j];
          B[i][j][2*k+1] = B[i][j][2*k+2] + B[i][j][2*k+4];
        }
      }
    }
```

```
}

/*
line_size : 32    cache_size : 16384

=====Processing the test1 procedure=====

==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****


**** Printing uni_gen_set's and eq_class'es for loop 2 ****


**** Printing uni_gen_set's and eq_class'es for loop 3 ****

Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 1 references :
The members of the set are :
A[i,3*j]
The access matrix for the set is :
 1   0   0
 0   3   0
The references exhibit self-temporal reuse in loops :  3
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
A[i,3*j]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i,3*j]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
A[i,3*j]

The uniformly generated set contains 1 references :
The members of the set are :
A[i-1,5*j]
The access matrix for the set is :
 1   0   0
 0   5   0
The references exhibit self-temporal reuse in loops :  3
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
A[i-1,5*j]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i-1,5*j]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
A[i-1,5*j]

Printing uni_gen_set's and eq_class'es for array : .B
```

```
The uniformly generated set contains 3 references :
The members of the set are :
B[i,j,2*k+1]
B[i,j,2*k+2]
B[i,j,2*k+4]
The access matrix for the set is :
 1   0   0
 0   1   0
 0   0   2
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[i,j,2*k+1]
B[i,j,2*k+2]
B[i,j,2*k+4]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[i,j,2*k+1]
B[i,j,2*k+2]
B[i,j,2*k+4]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[i,j,2*k+1]
B[i,j,2*k+2]
B[i,j,2*k+4]


====== The collected info for local space 1-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 3

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 2

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 4


====== The collected info for local space 2-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 3

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 2
```

```
no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 4


====== The collected info for local space 3-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 4


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 5802 ,No of off-chip acc. = 354166

===== Tiling of loops 2 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 725 ,No of off-chip acc. = 354166
Tile-size 16 :  WS-size estimation = 2474 ,No of off-chip acc. = 302083
Tile-size 24 :  WS-size estimation = 5248 ,No of off-chip acc. = 284721
Tile-size 32 :  WS-size estimation = 9045 ,No of off-chip acc. = 276041
Tile-size 40 :  WS-size estimation = 13866 ,No of off-chip acc. = 270833

===== Tiling of loops 3 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 128 ,No of off-chip acc. = 500000
Tile-size 16 :  WS-size estimation = 192 ,No of off-chip acc. = 375000
Tile-size 24 :  WS-size estimation = 256 ,No of off-chip acc. = 333332
Tile-size 32 :  WS-size estimation = 320 ,No of off-chip acc. = 312500
Tile-size 40 :  WS-size estimation = 384 ,No of off-chip acc. = 300000
Tile-size 48 :  WS-size estimation = 448 ,No of off-chip acc. = 291666
Tile-size 56 :  WS-size estimation = 512 ,No of off-chip acc. = 285714
Tile-size 64 :  WS-size estimation = 576 ,No of off-chip acc. = 281250
Tile-size 72 :  WS-size estimation = 640 ,No of off-chip acc. = 277776
Tile-size 80 :  WS-size estimation = 704 ,No of off-chip acc. = 275000
Tile-size 88 :  WS-size estimation = 768 ,No of off-chip acc. = 272726
Tile-size 96 :  WS-size estimation = 832 ,No of off-chip acc. = 270832
Tile-size 104 :  WS-size estimation = 896 ,No of off-chip acc. = 269230

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 3 :
Tile-size 104 :  WS-size estimation = 896 ,No of off-chip acc. = 269230

Optimal results for tiling of loops 2 - 3 :
Tile-size 40 :  WS-size estimation = 13866 ,No of off-chip acc. = 270833
```

```
Optimal results for tiling of loops 3 - 3 :
Tile-size 8 :  WS-size estimation = 5802 ,No of off-chip acc. = 354166
*/

***********************************************************************

#define N 100

int A[N][N];
int B[N];
int C[N][N];

test1()
{
    int i, j, k;
    for(i=0; i<N; i++){
      B[i] = B[i+1] + B[2*i];
      for(j=0; j<N; j++){
        C[i][k] = C[i-1][k];
        for(k=0; k<N; k++){
          A[i][k] = A[i-1][k];
        }
      }
    }
}

/*
line_size : 32    cache_size : 16384

=====Processing the test1 procedure=====

==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****

Printing uni_gen_set's and eq_class'es for array : .B

The uniformly generated set contains 2 references :
The members of the set are :
B[i]
B[i+1]
The access matrix for the set is :
 1   0   0
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  1
The generated eq_class'es for local space 1-1 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
B[i]
B[i+1]

The uniformly generated set contains 1 references :
The members of the set are :
B[2*i]
The access matrix for the set is :
 2   0   0
```

```
The references exhibit self-temporal reuse in loops :
The references exhibit self-spatial reuse in loops :  1
The generated eq_class'es for local space 1-1 :
Set no. 1 :
B[2*i]


**** Printing uni_gen_set's and eq_class'es for loop 2 ****

Printing uni_gen_set's and eq_class'es for array : .C

The uniformly generated set contains 2 references :
The members of the set are :
C[i,k]
C[i-1,k]
The access matrix for the set is :
 1   0   0
 0   0   0
The references exhibit self-temporal reuse in loops :  2
The references exhibit self-spatial reuse in loops :  2
The generated eq_class'es for local space 1-2 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
C[i,k]
C[i-1,k]
The generated eq_class'es for local space 2-2 :
Set no. 1 :
C[i,k]
Set no. 2 :
C[i-1,k]


**** Printing uni_gen_set's and eq_class'es for loop 3 ****

Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 2 references :
The members of the set are :
A[i,k]
A[i-1,k]
The access matrix for the set is :
 1   0   0
 0   0   1
The references exhibit self-temporal reuse in loops :  2
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
The eq_class exhibits group temporal reuse among its references
A[i,k]
A[i-1,k]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i,k]
Set no. 2 :
A[i-1,k]
The generated eq_class'es for local space 3-3 :
```

```
Set no. 1 :
A[i,k]
Set no. 2 :
A[i-1,k]


====== The collected info for local space 1-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 100
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 100
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 4

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 10000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 2-3 ======

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 100
SS_reuse_divisor = line_size/elem_size*stride = 32/1*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 100
SS_reuse_divisor = line_size/elem_size*stride = 32/1*??? = 1

no_of_eq_classes = 2   no_of_ST_reuse_dims = 1
no_of_iterations = 10000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 2   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 3-3 ======

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 100
SS_reuse_divisor = line_size/elem_size*stride = 32/1*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 100
SS_reuse_divisor = line_size/elem_size*stride = 32/1*??? = 1

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 10000
```

```
SS_reuse_divisor = line_size/elem_size*stride = 32/1*??? = 1

no_of_eq_classes = 2   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 8448 ,No of off-chip acc. = 16912

===== Tiling of loops 2 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 6720 ,No of off-chip acc. = 34050

===== Tiling of loops 3 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 1344 ,No of off-chip acc. = 270300
Tile-size 16 :  WS-size estimation = 2688 ,No of off-chip acc. = 270300
Tile-size 24 :  WS-size estimation = 4032 ,No of off-chip acc. = 270300
Tile-size 32 :  WS-size estimation = 5376 ,No of off-chip acc. = 270300
Tile-size 40 :  WS-size estimation = 6720 ,No of off-chip acc. = 270300
Tile-size 48 :  WS-size estimation = 8064 ,No of off-chip acc. = 270300
Tile-size 56 :  WS-size estimation = 9408 ,No of off-chip acc. = 270300
Tile-size 64 :  WS-size estimation = 10752 ,No of off-chip acc. = 270300
Tile-size 72 :  WS-size estimation = 12096 ,No of off-chip acc. = 270300
Tile-size 80 :  WS-size estimation = 13440 ,No of off-chip acc. = 270300
Tile-size 88 :  WS-size estimation = 14784 ,No of off-chip acc. = 270300
Tile-size 96 :  WS-size estimation = 16128 ,No of off-chip acc. = 270300

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 3 :
Tile-size 8 :  WS-size estimation = 8448 ,No of off-chip acc. = 16912

Optimal results for tiling of loops 2 - 3 :
Tile-size 8 :  WS-size estimation = 6720 ,No of off-chip acc. = 34050

Optimal results for tiling of loops 3 - 3 :
Tile-size 96 :  WS-size estimation = 16128 ,No of off-chip acc. = 270300
*/

***********************************************************************

#define N 100
int A[N][N];
int B[N][N];
int C[N][N];

test1()
{
    int i, j, k, l, m;
    for(i=0; i<N; i++){
      for(j=0; j<N; j++){
        for(k=0; k<N; k++){
                C[i][k] = A[i][j] * B[j][k];
        }
```

```
      }
    }
}

void main(){
  test1();
}

// A printout of the results obtained by the memory analysis procedure are
// printet below. The results have been tested for correctness.

/*
line_size : 32    cache_size : 2048
=====Processing the test1 procedure=====


==== Printing out uni_gen_set's and eq_class'es in a nest ====

**** Printing uni_gen_set's and eq_class'es for loop 1 ****


**** Printing uni_gen_set's and eq_class'es for loop 2 ****


**** Printing uni_gen_set's and eq_class'es for loop 3 ****

Printing uni_gen_set's and eq_class'es for array : .C

The uniformly generated set contains 1 references :
The members of the set are :
C[i,k]
The access matrix for the set is :
 1   0   0
 0   0   1
The references exhibit self-temporal reuse in loops :  2
The references exhibit self-spatial reuse in loops :  2 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
C[i,k]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
C[i,k]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
C[i,k]

Printing uni_gen_set's and eq_class'es for array : .A

The uniformly generated set contains 1 references :
The members of the set are :
A[i,j]
The access matrix for the set is :
 1   0   0
 0   1   0
The references exhibit self-temporal reuse in loops :  3
The references exhibit self-spatial reuse in loops :  2 3
```

```
The generated eq_class'es for local space 1-3 :
Set no. 1 :
A[i,j]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
A[i,j]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
A[i,j]

Printing uni_gen_set's and eq_class'es for array : .B

The uniformly generated set contains 1 references :
The members of the set are :
B[j,k]
The access matrix for the set is :
 0   1   0
 0   0   1
The references exhibit self-temporal reuse in loops :  1
The references exhibit self-spatial reuse in loops :  1 3
The generated eq_class'es for local space 1-3 :
Set no. 1 :
B[j,k]
The generated eq_class'es for local space 2-3 :
Set no. 1 :
B[j,k]
The generated eq_class'es for local space 3-3 :
Set no. 1 :
B[j,k]


====== The collected info for local space 1-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 2-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8
```

```
no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


====== The collected info for local space 3-3 ======

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8

no_of_eq_classes = 1   no_of_ST_reuse_dims = 1
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 1

no_of_eq_classes = 1   no_of_ST_reuse_dims = 0
no_of_iterations = 1000000
SS_reuse_divisor = line_size/elem_size*stride = 32/4*??? = 8


***** Estimations for all spaces and all tile-sizes *****

===== Tiling of loops 1 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 768 ,No of off-chip acc. = 46875

===== Tiling of loops 2 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 320 ,No of off-chip acc. = 156250
Tile-size 16 :  WS-size estimation = 1152 ,No of off-chip acc. = 140624

===== Tiling of loops 3 - 3 estimations =====
Tile-size 8 :  WS-size estimation = 96 ,No of off-chip acc. = 375000
Tile-size 16 :  WS-size estimation = 160 ,No of off-chip acc. = 312500
Tile-size 24 :  WS-size estimation = 224 ,No of off-chip acc. = 291666
Tile-size 32 :  WS-size estimation = 288 ,No of off-chip acc. = 281250
Tile-size 40 :  WS-size estimation = 352 ,No of off-chip acc. = 275000
Tile-size 48 :  WS-size estimation = 416 ,No of off-chip acc. = 270833
Tile-size 56 :  WS-size estimation = 480 ,No of off-chip acc. = 267857
Tile-size 64 :  WS-size estimation = 544 ,No of off-chip acc. = 265625
Tile-size 72 :  WS-size estimation = 608 ,No of off-chip acc. = 263888
Tile-size 80 :  WS-size estimation = 672 ,No of off-chip acc. = 262500
Tile-size 88 :  WS-size estimation = 736 ,No of off-chip acc. = 261363
Tile-size 96 :  WS-size estimation = 800 ,No of off-chip acc. = 260416
Tile-size 104 :  WS-size estimation = 864 ,No of off-chip acc. = 259615

***** Optimal results for all spaces *****

Optimal results for tiling of loops 1 - 3 :
Tile-size 8 :  WS-size estimation = 768 ,No of off-chip acc. = 46875

Optimal results for tiling of loops 2 - 3 :
Tile-size 16 :  WS-size estimation = 1152 ,No of off-chip acc. = 140624

Optimal results for tiling of loops 3 - 3 :
Tile-size 104 :  WS-size estimation = 864 ,No of off-chip acc. = 259615
*/
```