System Level Platform Modeling for System-on-Chip

David Ritter

Kgs. Lyngby 2004
IMM-THESIS-2004-66

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Abstract

SoC design using IP-based design methodologies allows a large degree of design flexibility. Designers may wish to use design space exploration techniques to find optimal SoC designs for a given application. To do so, however, fast and accurate estimation of each design's performance is needed. In this project, SoC performance estimation techniques are explored, focusing on processor-based platforms. A flexible methodology, the SoC Platform Architecture Model (SPAM), is developed for specifying platform configurations, component properties, and workload properties. A set of performance models for processors, bus interconnect, and shared or private memories is presented. These models are then combined using the SPAM methodology, and applied to modeling of a variety of single processor and multiprocessor SoC platforms. The potential for extension of the SPAM modeling system is also explored.

Preface

The work presented in this Masters thesis has been carried out by David Ritter.

David Ritter 31 August 2004

Acknowledgements

I would first like to thank my parents, for always supporting me, and for their great job of proofreading. Also, thanks to my brother, Greg, for helping out despite being busy with his own work and family, and for some very useful suggestions and questions.

Thanks also to my thesis advisor, Jan Madsen, who was able to strike a perfect balance between helping out with advice and suggestions, while giving me freedom to explore my own ideas.

I should also thank all the people who helped make my stay in Denmark pleasant, productive and possible. Thanks to Nokia, for their financial support. Thanks also to my program coordinator, Flemming Stassen; to the people at the international office for all their help; and to all the friends, foreign and Danish, that make living in Denmark an enjoyable experience.

Contents

Chapter 1

Introduction

1.1 Motivation

With rapidly increasing levels of integration possible in integrated circuit (IC) technology, the design complexity of a single IC is increasing correspondingly. This design challenge has brought about new intellectual property (IP) based design methodologies, in which standard IP descriptions of diverse components are combined to create a complex system-on-chip (SoC) design. Designers typically may select from a variety of:

- third-party hard-IP cores specified at the layout level for a specific IC process
- third-party soft-IP cores, or cores described at a higher level in a hardware description language, which in many case may allow the selection of various design parameters, and which then may be compiled to produce an implementation for a particular process
- internally developed cores, which in many cases may provide high design flexibility.

By combining predesigned IP cores implementing standard functionality, an IC designer can more quickly produce a complete system design.

Often a SoC system will consist of one or more processor-based platforms, combining processors, memories, peripheral devices, and the interconnect between components. When designing a complex SoC platform, designers have great latitude in component selection, parameterization, and interconnect design. Designers can select from numerous IP cores, parameterize soft-IP cores, or utilize custom-designed cores. Furthermore, a variety of interconnection methods are available, including dedicated point-to-point interconnect, industry standard buses or custom designed buses, hierarchical buses, and various network-on-chip (NoC) designs. The wealth of design options available creates a large design space, and choosing an optimal or near optimal design for a particular workload requires a method of exploring the design space.

Many methods for exploring a design space may be possible, from complex automated methods utilizing genetic algorithms, to ad hoc evaluation of designs based on a human designer's intuition. A commonality among these methods is that they require accurate measures or estimates of the performance characteristics, such as execution speed or power consumption, when a workload is executed on particular platform designs. Clearly, measurement of the actual performance characteristics on real hardware is not feasible in most circumstances, as this would require the construction of a large

number of hardware prototypes. Alternatively, it is often possible to model different cores with existing custom, cycle-accurate, simulators. However, use of such simulators in exploration of a large design space may be problematic. Among the problems which may be associated with the use of detailed custom simulators are:

- difficulty or expense in obtaining or developing simulators for each component/core
- high simulator runtime associated with functional, cycle-accurate simulators
- difficulty in interconnecting diverse simulators to provide a full picture of the platform performance
- difficulty in integrating diverse simulators, with different input requirements and different forms of output, into a design space exploration tool.

These problems motivate the development of a single, flexible, and fast model which can produce accurate performance measures for platform designs containing diverse components and interconnection methods. Chapter 2 describes some methods presented previously in the literature for determining performance of platforms and platform components. The remainder of this report describes the design and implementation of a flexible system for performing SoC platform performance estimation.

1.2 Project Concept

A SoC platform architecture can be described in general terms as a set of interconnected, heterogeneous processing, communication and storage elements. Instruction streams executing on processing elements drive the behaviour of the system. The behavioral characteristics of the workload and architectural characteristics of processing elements determine the usage of communication and storage elements. The characteristics of the communication and storage elements in turn influence the rate of execution of the workload. The performance characteristics of the system as a whole, such as execution speed and power usage, are therefore a function of the workload and the characteristics of all architectural components working in concert. A modeling methodology for SoC platforms is required which can accurately consider the performance characteristics of various processing, communication and storage elements, as well as the interdependence of these components.

Given the wide range of potential SoC platform designs, the flexibility of the modeling methodology is extremely important. This required flexibility manifests itself in several ways:

- the methodology must be able to consider diverse components, including a wide range of processor cores, bus implementations and memory cores
- the methodology must be able to consider diverse platform designs, including variable numbers and types of components, and various interconnect topologies
- the methodology should provide simple mechanisms to allow extension with new performance models for unique or novel component designs.

Chapter 3 describes the SPAM (SoC Platform Architecture Model) methodology, which provides a flexible means of expressing various system architectures and workloads, and defines an object-oriented method of developing component performance models.

In many designs, the most important design decision affecting system performance is the choice and parameterization of a processor core. A fast, accurate, and general model of processor performance is desired, which is applicable to a wide range of processor types from simple scalar processors to superscalar processors with out-of-order execution. Considerable research has been devoted to this topic, and several fast, general models of processor performance have been proposed. In chapter 4, a model of processor performance from the literature is selected and implemented within the SPAM methodology. Additionally, some modifications to the existing model are made to extend its usefulness to additional classes of processors, and to improve its accuracy in certain conditions. The effectiveness of the processor model in accurately predicting performance for a wide range of processor architectures is verified by comparing the model's predictions to results from two cycle-accurate simulators:

- the Avrora simulator, which simulates the AVR core, a simple unpipelined scalar processor
- the SimpleScalar simulator, a superscalar processor simulator.

In a complex SoC, multiple IP cores providing various functionality operate concurrently, sharing access to various on-chip resources such as buses, shared memory, or other peripheral components. A number of possible methods exist for modeling the performance effects of sharing of a resource among concurrent activities. At a low level, a detailed cycle-accurate simulation of both the request generating components and requested resources could be made. As mentioned previously, though, detailed simulation carries high costs in terms of simulator runtimes. As an alternative to cycle-accurate simulation, models can consider exclusive allocation of a resource to a task at a high level. However, in many cases this is likely to produce extremely pessimistic and inaccurate measures of performance. A more generally useful model of shared resource usage, which combines reusability, accuracy, and high performance, is desirable. Chapter 5 describes two models of shared resources, applicable to different types of resources with different methods of resource allocation.

The performance of various shared resources, such as buses and memories, is dependent on properties of the resource, such as the width of a bus or the block size of a memory. Memory or bus performance is also dependent on the properties of the resource usage, such as the size of a bus transfer. Models of buses and memories extend the general concepts of shared resources described in chapter 5. General models of memory and bus components which consider the performance effects of various resource properties are presented in chapter 6 and 7, and implemented within the SPAM methodology.

The overall modeling concepts discussed in chapter 3 are combined with the individual component models for processors, buses and memories presented in chapters 4, 7, and 6, to arrive at a complete platform model. This model, implemented using the SPAM methodology, captures the performance effects of a variety of design choices related to each component, and considers the interdependence of the components in providing performance estimates. The model's flexibility is demonstrated by applying it to a variety of platform architectures, including single processor architectures with various memory subsystem designs, and multiprocessor platforms with shared memories accessed via a shared bus. The accuracy of the platform model is tested by comparing results with two cycle-accurate simulators:

- the SimpleScalar simulator, with the model being used to predict the effects of various processor parameters, and the results of contention between the processor front-end and back-end for unified cache access
- the MPARM simulator, a simulator for multiprocessor ARM7 platforms using an AMBA AHB bus to access shared memory.

In chapter 9, possible extensions to the model are discussed, including:

- models for additional platform components
- methods for combining models to provide more flexible and complete coverage of an individual component's design space
- methods for improving the accuracy and generality of the existing platform models
- methods for improving the modeling speed.

Additionally, limitations of the methods presented are discussed, as are features of the SPAM model which have not been utilized in the presented modeling examples.

Finally, in chapter 10, the results and contributions of the presented models and methods are summarized.

1.3 Terminology and Conventions

Throughout this report, certain terms will be used to refer to the entities which make up a SoC. In the hardware domain, a SoC may consist of one or more *platforms*, which consist of processors and all required supplementary hardware. A SoC platform is built by combining various functional cores, along with the interconnect between them. Each core and interconnection channel can have an effect on system performance. The general term *component* is used to refer to any hardware platform constituents which may have a performance effect.

In the software domain, the term *workload* is used to refer generally to an instruction stream executed on a processor-based platform. Software may be divided into *tasks*, which roughly correspond to units of software which can be scheduled by an operating system, and which execute on a single processor. In chapter 3, the concept of an *activity* is introduced, which describes a task's usage of a particular component.

Where individual names for these SoC entities are needed, platforms are given names π_i, components are named κ_i, tasks are given names τ_i, and activities are named α_i. Figure 1.1 depicts these concepts.

Specific terms used in this project which are not general terms of art are introduced in *italics*. References to source code entities used in implementation, such as classes or variables, and references to programs used, are denoted by `fixed width` type.

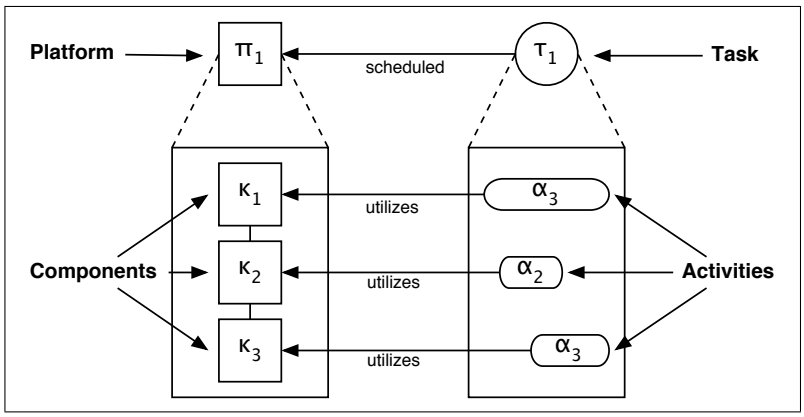


Figure 1.1: SoC Hardware and Software Entities

Chapter 2

Previous Works

Research on platform performance estimation covers a range of general areas. Significant research has been performed on component-specific performance estimation methods, particularly for processors. Other research efforts focus particularly on interconnect design, or memory hierarchy evaluation. A few performance evaluation techniques also focus on the complete platform and the interdependence of all constituent components.

While performance estimation techniques can be divided based on the considered components, a separate division can also be made with respect to the methods used to determine performance. Specifically, methods exist ranging from program instrumentation on hardware, to cycle-accurate simulation, to various methods which treat components or systems as stochastic processes, or consider performance statistically.

Design or selection of a performance estimation method involves a set of tradeoffs. Major tradeoffs must be made between the estimation method's speed, its flexibility, and its accuracy. The needs of early design phase design space exploration dictate that speed and flexibility are essential, although naturally, the accuracy of a method should not be ignored.

Most of the fast analytical methods described in this chapter are designed to achieve reasonably high accuracy without incurring a high cost for execution time. Principal methods used to achieve both high accuracy and high speed involve making key abstractions, and taking advantage of regularity in workloads. Flexibility likewise may be achieved without greatly sacrificing accuracy by making use of fundamental similarities between components to build a general model.

2.1 Processor Performance Evaluation

Processor performance evaluation has been heavily investigated in the literature. One of the most commonly used methods of evaluating performance of a processor is to construct a cycle-accurate simulator of the processor core. Such simulators can be divided into execution-driven simulators and trace-driven simulators. Execution-driven simulators duplicate the functionality of a processor core to at least the instruction level. They can thus have complex implementations, making modification for modeling architectural changes difficult.

A widely used example of an execution-driven simulator is the SimpleScalar [28, 7]

suite of simulators for a superscalar processor core and cache hierarchy. In this project the SimpleScalar toolset is used for obtaining baseline results and for gathering statistical information about workloads, as detailed in chapter 4. Numerous other examples of execution-driven simulators exist, focussing on different processor architectures, such as the AVR core [32] and the ARM processor family [10]. Detailed functional simulators necessarily have restricted flexibility: changes to the instruction set or microarchitecture require corresponding changes to the simulator internals.

Trace-driven simulators use an execution trace, obtained by running a workload on an execution-driven simulator or on actual hardware, which details the control flow and possibly some processor state during the execution of the workload. This information can be used to drive simulation of a processor with various architectural changes, such as in branch predictor or cache design, to determine the performance effect of these changes. Trace-driven simulations generally can be performed more quickly than execution-driven simulations, since the actual functionality represented by instructions in the trace does not have to be duplicated. Like execution-driven simulation, trace-driven simulation cannot deal easily with changes to the processor instruction set or some microarchitectural changes. Any changes which would affect the control flow, for example, would require reacquisition of the program trace.

A problem relevant for both execution and trace-driven simulation is the slowdown inherent in the simulation. To simulate complete workloads with detailed simulators may result in prohibitive simulator runtimes. Two classes of methods for dealing with this slowdown, trace sampling and statistical simulation, reduce simulator runtimes by reducing the size of the simulated workload or trace.

Trace sampling works by selecting portions of a workload trace, typically while ensuring through statistical methods that the sampled trace is representative of the entire workload. Conte et. al. [8] describe methods for selecting representative trace samples from execution traces, and show that trace simulation using these samples produce results with small relative errors when compared to full simulations.

Statistical simulation methods work by gathering statistical parameters from analysis of the complete workload, and then using these parameters to generate a synthetic trace. Eeckhout and De Bosschere [11] describe such a statistical simulation method. They also describe methods [12] which combine statistical simulation with analytical modeling of register usage. Their method is demonstrated as a useful technique for design space exploration, due to the lower simulation times involved and to the flexibility available in considering architecture changes or workload changes.

The methods of statistical simulation or trace sampling do not change the underlying operation of the simulator: for example, the execution-driven simulator still considers instructions in a synthetic trace on an instruction-by-instruction basis. Various analytical methods have been proposed which take properties of workloads and architectures and attempt to produce accurate performance measures without any form of simulation.

Noonburg and Shen [25] present a Markov chain model of a superscalar processor. In this model, a statespace is constructed which describes the possible microarchitectural states of the processor. This statespace is similar to the state-encoding mechanisms implicit in any functional execution-driven simulator. A traditional execution-driven simulator determines the next state based on the current processor state, the next instruction, and possibly data values in registers or memory. In contrast, the Markov

chain model defines transition probabilities between these states. From the statespace and transition probabilities, one can compute stationary state probabilities, and from these useful performance metrics can be extracted.

Zhu and Wong [34, 35] demonstrated the use of a multiple class multiple resource (MCMR) queuing system model to determine superscalar processor performance. The MCMR model is applied to processor functional units [34], with multiple classes representing instruction types and multiple resources representing the various functional units. Analytical results are derived based on queueing system theory. This method is extended by building a network of MCMR queuing systems [35], representing instruction fetch, decode, issue, execute and retire stages of a superscalar processor.

Taha and Wills [31, 30] describe an analytical instruction throughput model of superscalar processors which operates through use of an iterative method on a small sample of instructions. The operation of this model is described in more detail in chapter 4.

Analytical and statistical methods potentially provide the most useful methods for design space exploration. These methods can often produce performance estimates in a small fraction of the time required for simulation methods, and they provide flexibility in the sense that certain architectural changes can be explored by changing only a few model inputs. In contrast, modifying a cycle-accurate simulator to consider a different instruction set architecture, for example, could require extensive source code changes to the simulator. Analytical methods tend to focus on the effects of a few architectural features, and detailed examination of the performance effects of microarchitectural design choices may be outside their scope. However, for early design phase exploration, this limitation may be acceptable, as more detailed exploration can be performed later on a culled design space with more precise simulation methods.

2.2 Interconnect Performance Evaluation

A SoC architecture can make use of a variety of interconnect strategies to connect the various SoC components. While complex Network-on-Chip (NoC) designs have been proposed, these strategies are likely to be used to connect multiple processor-based platforms in a multiprocessor SoC (MPSoC). Within platforms, simpler interconnection methods including buses are likely to remain prevalent. Therefore, in this report modeling of interconnect focuses on buses.

As with processors, the performance of a bus can be determined by detailed cycle or phase-accurate simulation of the bus. However, detailed bus simulation also suffers from the same drawbacks as processor simulation, including high execution times and inflexibility. Analytical models exist which attempt to provide flexible, fast performance evaluation of a bus design. These analytical models can be divided broadly into two classes: transaction-level models, and system-level models. Transaction-level models use bus and transaction parameters to determine the delay of a single bus transaction. These models are useful for determining delays of individual operations, but they do not consider bus contention. To account for additional delays which will be incurred by bus traffic when contention occurs, system-level models which include consideration of that contention have been developed.

Knudsen and Madsen present [18] a flexible transaction-level model for estimating the

performance of communication links with different characteristics. The model considers not only the parameters of the channel itself, but also the protocol for communication, including burst modes and methods of packing, and the performance costs incurred at communication source and destination by communication drivers. However, the model does not consider contention for the communication channel.

Lahiri, Raghunathan, and Dey [19, 20] develop methods for full-system modeling of SoC communication channels. A two-phase method for modeling bus interconnect is presented [19], which operates by first extracting a trace of communication events from a system simulation, and then applying an algorithm to determine comprehensively the effects of changing bus parameters. This method is then extended to study multi-channel communication architectures [20]. These models consider the effects of contention for communication channels, and also consider dynamic effects introduced by contention delays. That is, an extra communication delay caused by bus congestion may cause future communication events to be delayed. However, the model considerably reduces simulation time compared to full-system cycle-accurate simulation since the trace of communication events is at a considerably higher granularity.

Cho et. al. [16] introduce Integer-Linear Programming (ILP) methods and heuristics for timing analysis of various SoC communication methods. These methods consider dynamic behavior introduced in the form of contention for processors or communication resources through formulation of ILP constraints. To determine the system performance, the ILP formulation must be solved. Due to the computational complexity of ILP methods, a heuristic is also developed to determine an approximate solution.

Communication system analysis, from telecommunication networks to multiprocessor interconnect, has often been analyzed with queueing systems and networks. Zhang and Chaudhary [33] use simulation of a queue system to analyze the performance of a SoC bus architecture.

2.3 Platform Performance Evaluation

Although many research efforts have focussed on evaluation of particular components of a platform, there have been some studies of full platform simulation or modeling.

Some of the models which consider interconnect design [16, 20] discussed in section 2.2 include consideration of dynamic effects introduced by the interplay between interconnect and processors. However, these methods do not include consideration of the processor design itself, but rather use execution times measured through some other means. Similarly, some processor models integrate the effects of memory and cache performance into their estimates, but also do so by using predetermined values as model inputs.

Cain et. al. [13] argue that, while researchers developing cycle-accurate simulators often focus on the precision of their model, meaning the degree to which the model considers detailed microarchitectural operations of the system, accuracy is often underemphasized. The authors argue that the accuracy of a simulator depends not only on the precision of the model, but also on the degree to which the simulator input matches what a real system will experience. Furthermore, the importance of full system simulation is demonstrated, focussing particularly on the effects of operating systems,

branch misprediction, and the effects of Direct Memory Access (DMA) transfers. A full-system, execution driven cycle-accurate simulator, PHARMsim, is developed as part of this research.

Grode presents [17] a method of modeling components at an operational level using colored Petri nets. The use of Petri nets as a generic modeling mechanism allows flexible creation and combination of diverse models. By modeling at an operational level, such as instruction level of a processor instead of microarchitectural level, the approach intends to eliminate some of the simulation complexity.

Albonesi et. al. describe [2] the STATS (System Tradeoff Analysis Toolset) framework for design space exploration of microprocessor-based systems. This framework combines cycle-accurate processor simulation with a number of analysis tools, capable of estimating the performance effects of cache, interconnect, and off-chip memory using SPICE-based analysis. A database of results is used to avoid duplication of effort and reduce overall runtime.

2.4 Summary

The methods described in this section show that the field of early design phase performance evaluation can be explored further. A number of component-specific models have been developed, each providing differing views of the component and different methods for evaluation. It may be possible to combine these diverse models into a full-system model to provide an accurate picture of SoC platform performance. Some such system-level methods have been developed, but these platform models tend to be monolithic, and thus may be difficult to modify and update. Furthermore, many system-level performance analysis suites operate at a low granularity, meaning runtimes are likely to be long.

Additionally, investigation should continue into which particular combination of models for processors, interconnect, memories and other components provide the highest accuracy, greatest modeling flexibility, and fastest performance. In the following chapters, a flexible, modular methodology for platform modeling is presented, and a particular set of component models is combined to arrive at a complete platform model.

Chapter 3

Modeling Methodology

Effective performance estimation in a design space exploration algorithm requires a systematic method for modeling diverse platform configurations, components, and workloads. The overall modeling methodology used must support the requirements for

- flexibility in platform configuration or topology
- modularity, in the sense that similar components can easily be substituted for one another in performance models, and
- extensibility such that new models for new or different types of components can be created easily and integrated with existing models.

Furthermore, the modeling methodology must consider the interdependence of components when determining performance, and the differing usage of components driven by various workloads. Finally, the modeling methods should be developed considering the need for fast performance estimation in design space exploration.

3.1 Model Concept

To allow for flexibility of configuration, one can use structural composition of separate component models into a complete platform model. For example, for the simple platform shown in figure 3.1(a), a complete platform model could be made by instantiating one component model for the processor, one model for the bus, and one for the memory. Figure 3.1(b) shows a multiprocessor platform with private and shared memories, which could similarly be modeled by instantiating component models for each constituent component. Allowing instantiation of arbitrary numbers and kinds of component models gives considerable flexibility in the nature of the platforms which can be modeled.

Other peripheral components, such as I/O devices, network interfaces, coprocessors or any other utilized component, likewise could be modeled by instantiating component-specific models. The platform models developed in this project focus, however, on the major platform components: processors, memory, and bus interconnect.

Modularity of component models can be achieved by using a common interface for all component models. The interface exposes the methods for expressing interdependencies between models, and for calculating performance estimates. Additionally, common runtime functionality is provided, such as handling dependencies, performing updates at each simulator clock, and handling events received from other components.

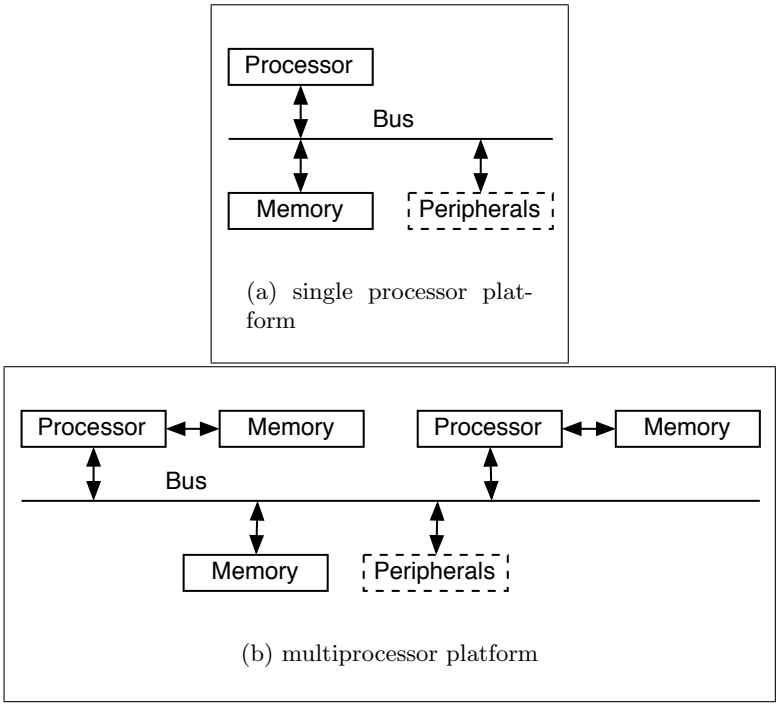


Figure 3.1: Example Platform Configurations

The use of object-oriented design techniques is a natural choice in order to present a standard component model interface. Furthermore, object-oriented inheritance provides a straightforward mechanism which allows extension of the modeling system to consider novel components. A common interface can be specified by a common base class, using abstract virtual methods where a derived component model would have to provide component-specific implementation details related to the component. By deriving child classes from the base class, new models for specific types of components can be created.

The presented requirements motivate the selection of an implementation language. The need for object orientation and for a language which can express structural composition makes SystemC a natural choice. SystemC is an extension of C++, in the form of class library, which provides hardware modeling constructs. Additionally, SystemC provides an implementation of a clocked event-based simulator, which is useful for performance modeling of complex multicomponent systems. SystemC also allows the use of existing C++ code and libraries in model implementation. For these reasons, the modeling software described in this report is implemented in SystemC. The model design widely uses the SystemC Master-Slave library because it provides guaranteed order of execution, avoiding the need for synchronization between components of the model.

One must also consider modeling of the workloads that will be be applied to the modeled platforms to determine what constructs are required to express the workload accurately. In this work, a method of describing workloads which can express both sequential and parallel usage of platform components is presented.

Section 3.2 discusses how workloads are described in the SPAM methodology as activity graphs. The concepts introduced in section 3.2 are used in section 3.3, which

describes the software architecture used to model the platform components which execute workloads. This section also presents methods by which the architecture can be extended to produce models for specific types of components. Section 3.4 describes how platforms are specified, and how structural composition of component models is used to create a model of a complete platform.

3.2 Workload Description

Workloads for processor-based platforms consist of instruction streams executing on processors. In embedded systems design, the workload is often divided into a set of distinct tasks, and the tasks and the dependencies between them form a *task graph*. Execution of a task will require utilization of the various platform components, including processor, bus interconnect, and memory. A task can be separated into a set of *activities*, with each activity describing the task's usage of a particular component.

Figure 3.2 shows the concept of activities applied to a task executing on a simple single processor platform such as that shown in figure 3.1(a). In this example, the task is executing on the processor component, illustrated by activity α_{proc}. During execution of the task, the processor is periodically executing instructions which access data memory. Execution of these instructions requires initiation of a bus transfer, with the data memory acting as a slave device on the bus and responding to the memory accesses. Hence, execution of the task also utilizes the bus and data memory platform components, denoted by activities α_{bus} and α_{mem}.

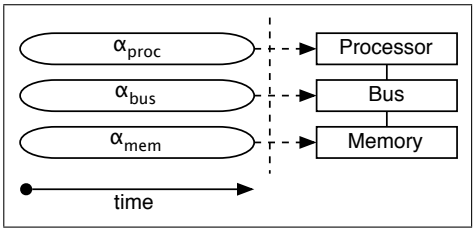


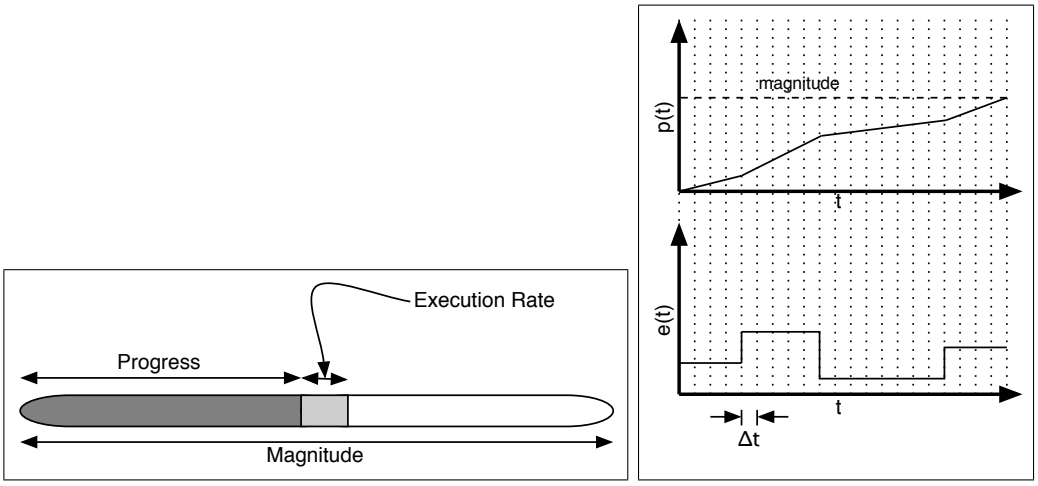
Figure 3.2: Example of Task Division into Activities

As will be discussed in chapters 4 and 5, the models used to describe processors and other platform components will perform statistical modeling of the detailed operations of those resources. Therefore, activities are described by parameters which capture statistically the many small-scale actions which make up the activity. These parameters are specified as *properties* of an activity. A model for a particular component requires a set of activity properties to estimate performance characteristics. The set of required properties will depend on the model: a processor model may require properties specifying the instruction mix, while a bus model may require a property specifying the average number of words in a bus transfer.

General Activity Properties

Four common properties of all activities are the *allocation*, *magnitude*, *progress*, and *execution rate*. An activity's allocation property indicates the component whose usage

is described by that activity. The magnitude is a generic property which specifies how much effort is required to complete an activity. The exact meaning of the magnitude is dependent on the component model. For a processor model, the magnitude of an activity may represent the total number of instructions that must be executed, while for a bus model, the total number of bytes of data transferred may be used. An activity's execution time will be proportional to its magnitude, but the exact execution time will also be dependent on the rate of execution. The progress of an activity is a simulation-time property of an activity, representing the current amount of the activity that has been completed. The simulator increments this value at each simulator clock cycle by the current execution rate, an amount determined by the component model executing the activity. Figure 3.3 illustrates the concepts of activity magnitude, progress and execution rate.



(a) activity state during one timestep

(b) progress of activity as numerical integration

Figure 3.3: Activity Execution Concept

One way of looking at the progress of an activity is as a numerical integration over time. The function being integrated is the execution rate $e(t)$ of the activity. Integration is performed using timesteps of duration Δt equal to the clock period for the simulator clock. The result of this integration is the progress $p(t)$ of the activity. When $p(t)$ equals or exceeds the activity's magnitude, the activity is complete. This is the same as saying the area under the curve of $e(t)$ versus t must equal or exceed the magnitude. Figure 3.3(b) depicts this concept.

Simulator Clock

Selection of a simulator clock rate is thus important in determining the accuracy of workload modeling. If the simulator clock period is too long, then execution time will be subjected to two potential sources of error:

- resolution errors when determining activity completion: if the simulator clock period is long, it is likely that activity progress will overshoot the required magnitude in the final simulator clock cycle
- resolution errors for execution rate changes: similarly, if the simulator clock period is long, the execution rate will not be as responsive to system changes.

Conversely, setting the simulator clock period too short will lengthen the simulation time. The correct balance between accuracy and simulation time is currently left flexible, and can be adjusted through specification of simulator and component clock rates.

Selection of the simulator clock rate does not necessarily have to correspond the clock rate of any particular platform component. The execution rates determined by components can be multiplied by a clock rate ratio to account for the difference between simulator and component clock rates.

Workload Concurrency and Sequentiality

Workloads may contain two forms of concurrency:

- concurrent activities executing on different components.
- concurrent activities sharing a single resource.

The modeling methodology provides a means of describing both forms of concurrency in workloads, and must support both forms of concurrency during performance estimation.

Additionally, workloads may require expression of sequentiality. In some cases it may be sufficient to specify the operation of an entire task with a single set of statistical parameters for each component. In complex workloads, however, one may find that certain portions of the workload have significantly different properties and utilize different platform components. Such workloads can be modeled as sequential activities with dependencies describing the temporal relationship between the activities. For example, figure 3.4 illustrates a possible example of a task executing on a processor in the multiprocessor platform shown in figure 3.1(b). In this example, the task first performs some CPU-intensive processing, utilizing data in a local memory. The task subsequently must transfer some data to or from another processor, and so it utilizes the bus and shared memory to perform a block transfer. The component usage for these two portions of the task are significantly different, and so it may be useful to model their actions sequentially.

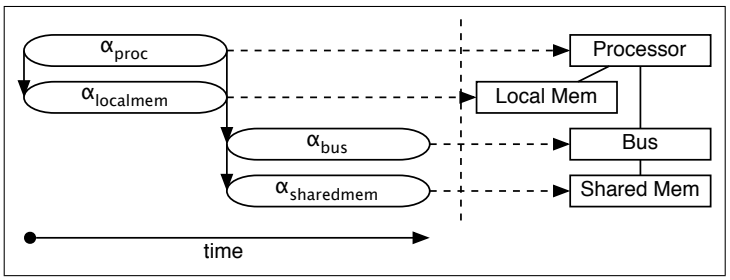


Figure 3.4: Example of Task Division into Sequential Activities

Figure 3.5(a) shows a possible example of a task described as a set of processor, memory and bus activities. In this example, activities only *trigger* dependent activities

at their completions, and dependent activities are triggered only at their initiations. This means, however, that some otherwise contiguous activities may be split in order to trigger another activity at a midpoint. An alternative method is to allow mid-activity triggering of dependent activities, as shown in figure 3.5(b). To allow midpoint triggers, the simple dependency information indicating that, for example, activity α_1 triggers activity α_2, is augmented with information regarding the point in the activity's progress at which its dependent activity begins. While the first method, with only endpoint triggering of dependencies, yields a larger graph with more activity vertices, the resulting graph may be more suitable for some forms of algorithmic analysis. However, the platform simulation method described in the report does not require complex analysis of activity graphs. Furthermore, the second method, with midpoint dependency triggering, yields a topographically simpler graph, and makes specification of workloads easier. For these reasons, the modeling methodology supports use of midpoint dependency triggering.

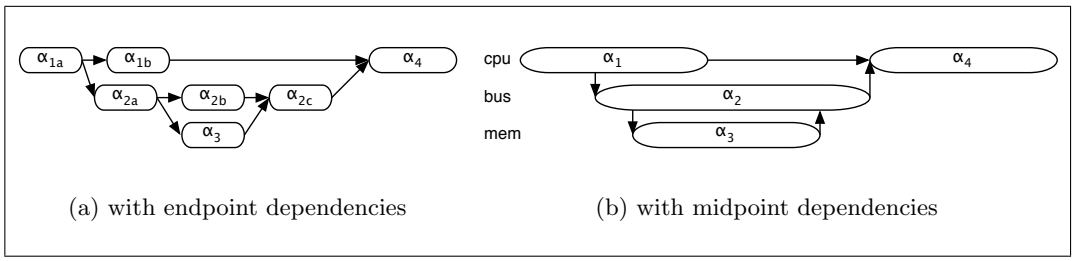


Figure 3.5: Activity Graph Examples

Dependency Types

Figure 3.5(b) also shows the need for differing types of dependencies. Activity α_2 is dependent on α_1 to begin execution; it is also dependent on α_3 to complete execution. These two types of dependencies can be termed *activation dependencies* and *completion dependencies*. Activation and completion dependencies are specified by their *source* and *sink* activities, plus the *trigger point*, which indicates the amount of source activity progress required to trigger the dependency.

Additionally, it is necessary for the model to allow results to be passed from one modeled activity to another. For example, the latency of data memory instructions in a processor may depend on the predicted performance of an activity executing on a memory component. Further examples demonstrating the usage of results sharing will be shown in chapter 8. The modeling methodology provides the functionality to describe these relationships in the form of *property dependencies*. A property dependency specifies, in addition to the source and sink activities, the name of the property of the source activity from which data will be taken, and the name of the property of the sink activity which will receive the data.

Figure 3.6 shows the information described by a set of activities and dependencies.

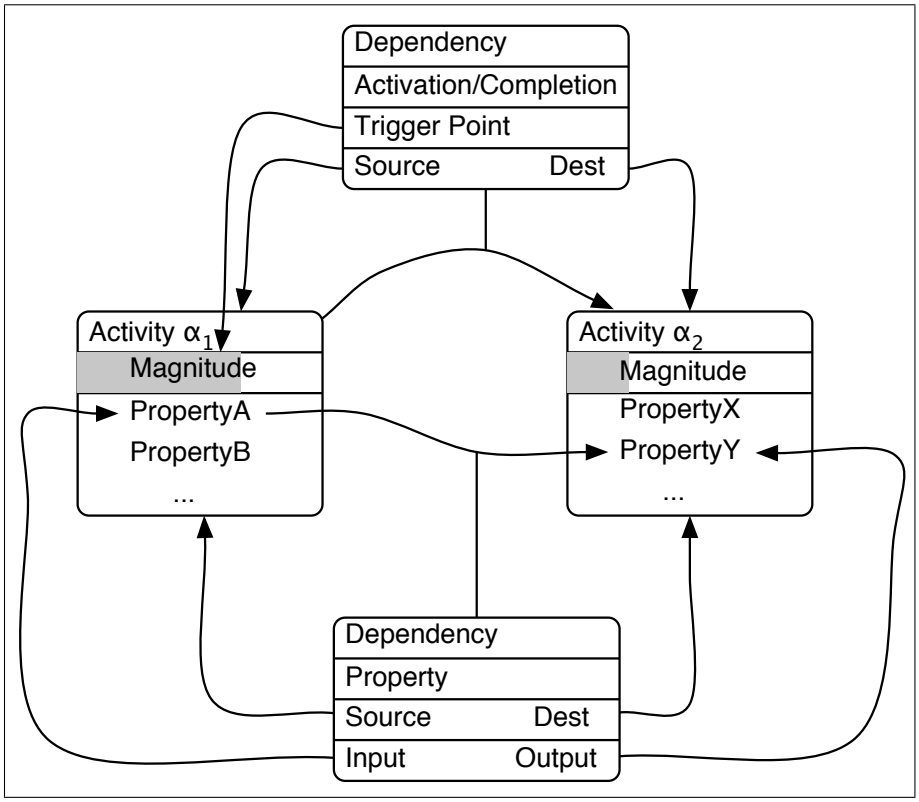


Figure 3.6: Activities and Dependencies

3.2.1 Implementation

Workload Specification

An input data file specifies the workload used in a simulation. Use of input files to specify workloads allows a single platform model executable to be used, without modification, to model any desired workload. The input file contains description of the activities, their properties, and dependencies, in the form of a directed graph, with vertices representing activities and edges representing dependencies.

Input data files are specified in XML (eXtensible Markup Language) in a format used by the FEARS (Framework for Embedded ARchitecture Synthesis) library. FEARS, which has been developed previously at DTU as part of a special project, provides an XML DTD (Data Type Definition) for directed graphs, and provides an API for accessing directed graph structures. Figure 3.7 shows a sample activity graph, with the corresponding FEARS XML description given in figure 3.8(a).

Unfortunately, some of the generality in the design of the FEARS library can lead to large, complicated XML descriptions for activity graphs. A simplified format is desirable, as it allows more human-readable and modifiable descriptions of activity graphs. A benefit of XML is that, through the use of XSL (eXtensible Stylesheet Language) stylesheets, one XML format can be translated automatically to another. This allows one human-readable format to be used to describe an activity graph, and another to be

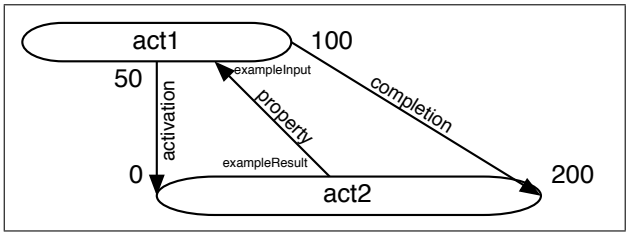


Figure 3.7: Example Activity Graph

used by the simulator runtime, with an automatic translation step to convert from one format to the other. Figure 3.8(b) shows the activity graph pictured in figure 3.7 as described in this simplified XML format.

FEARS Implementation

Once a workload has been specified in FEARS XML format, either directly or by converting from a simplified format using XSL stylesheets, an XML parser is used to read the workload description into memory. The FEARS library, as originally implemented, used a DOM (Document Object Model) parser from the Xerces-C++ distribution to convert from XML to a DOM representation. The FEARS library API then provided an abstraction layer which presented a directed graph view of the data. The abstraction layer converted calls to the FEARS API into underlying calls to the DOM objects representing the data. An overview of the original FEARS design is shown in figure 3.9(a). While functional, this method required continual accesses to the underlying data through the DOM API, in addition to the FEARS API. The result was less than satisfactory performance if access to the data represented in the DOM objects was required repeatedly.

Because of this problem, the FEARS API has been reimplemented using a SAX (Simple API for XML) parser. Parsing an XML file using SAX is generally faster than parsing the same file with a DOM parser, but more importantly, SAX performs a fundamentally different task than DOM: while a DOM parser constructs a DOM representation of an XML file, a SAX parser simply traverses the XML file and triggers events corresponding to the XML elements it finds. The reimplementation of FEARS uses the SAX events to construct a custom object model of the underlying data, instead of a DOM model. This object model is tailored to representing a directed graph, whereas the DOM is not. Because of this, and because an abstraction layer converting between FEARS API calls and DOM API calls is no longer needed, the reimplemented FEARS API is considerably faster. An overview of the reimplementation of FEARS is shown in figure 3.9(b).

To compare performance of the two FEARS implementations, a sample algorithm for enumerating paths through a directed graph was executed on a small sample input. This example implementation requires parsing of the input XML file, and performance of the pathfinding algorithm requires repeated access to the graph data. The existing, DOM-based FEARS implementation required 32.10 seconds to complete execution, while the new SAX implementation required only 0.79 seconds. Profiling shows that the majority of the execution time requirements for the DOM implementation come from DOM API

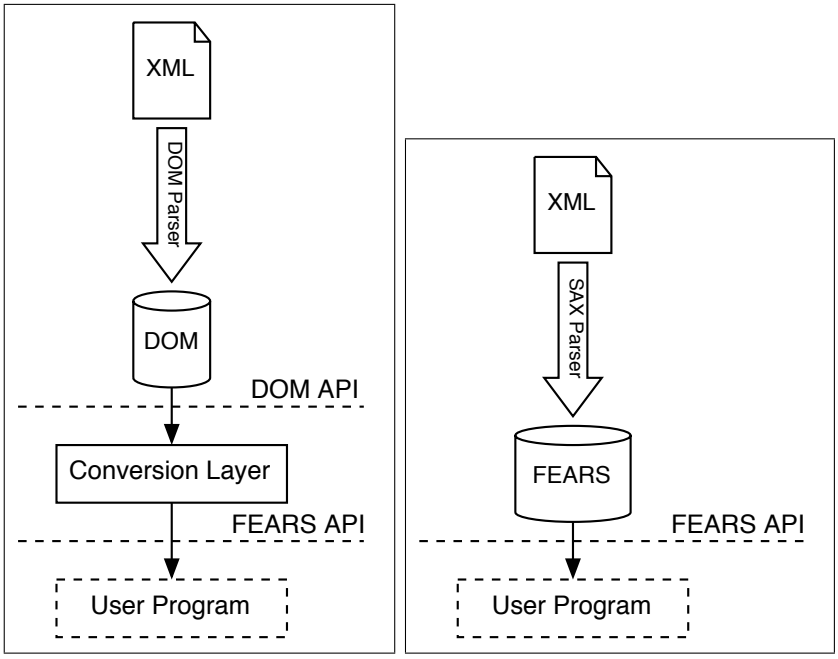
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONTENT SYSTEM "DGRAPH.dtd">

<CONTENT id="cjpeg-root">
        <STRUCTURE id="cjpeg-task">
                <NODE id="act1">
                        <INTERFACE id="act1-int">
                                <IN idint="act1-internalin" idext="act1-in"/>
                                <OUT idint="act1-internalout" idext="act1-out"/>
                        </INTERFACE>
                        <PROPERTY id="id" value="act1"/>
                        <PROPERTY id="magnitude" value="100"/>
                        <PROPERTY id="allocation" value="1"/>
                        <PROPERTY id="exampleProperty" value="1234"/>
                </NODE>

                <NODE id="act2">
                        <INTERFACE id="act2-int">
                                <IN idint="act2-internalin" idext="act2-in"/>
                                <OUT idint="act2-internalout" idext="act2-out"/>
                        </INTERFACE>
                        <PROPERTY id="id" value="act2"/>
                        <PROPERTY id="magnitude" value="200"/>
                        <PROPERTY id="allocation" value="2"/>
                        <PROPERTY id="exampleProperty" value="5678"/>
                </NODE>

                <EDGE>
                        <SOURCE id="act1-out"/>
                        <SINK id="act2-in"/>
                        <PROPERTY id="type" value="activation"/>
                        <PROPERTY id="trigger" value="50"/>
                </EDGE>

                <EDGE>
                        <SOURCE id="act1-out"/>
                        <SINK id="act2-in"/>
                        <PROPERTY id="type" value="completion"/>
                        <PROPERTY id="trigger" value="100"/>
                </EDGE>

                <EDGE>
                        <SOURCE id="act2-out"/>
                        <SINK id="act1-in"/>
                        <PROPERTY id="type" value="property"/>
                        <PROPERTY id="trigger" value="0"/>
                        <PROPERTY id="input" value="exampleResult"/>
                        <PROPERTY id="output" value="exampleInput"/>
                        <PROPERTY id="lowSlope" value="0"/>
                        <PROPERTY id="highSlope" value="1"/>
                        <PROPERTY id="funcX" value="0"/>
                        <PROPERTY id="funcY" value="0"/>
                </EDGE>
        </STRUCTURE>
</CONTENT>
```

(a) FEARS XML description

```
<?xml version="1.0" encoding="UTF-8"?>
<task id="cjpeg">
        <activity id="act1" magnitude="100" allocation="1" exampleProperty="1234"/>

        <activity id="act2" magnitude="200" allocation="2" exampleProperty="5678"/>

        <dependency type="activation" source="act1" sink="act2" trigger="50"/>
        <dependency type="completion" source="act1" sink="act2" trigger="100"/>

        <dependency type="property" source="act2" sink="act1" input="exampleResult" output="exampleInput" ratio="1"/>
</task>
```

(b) simplified XML description

Figure 3.8: Example Activity Graph XML Descriptions



(a) FEARS version 1 (b) FEARS version 2

Figure 3.9: FEARS Architecture Overview

calls and not from parsing. Thus, the relative speedup for the SAX implementation should be even larger for more complex algorithms or algorithms executed on larger datasets. Additionally, SAX parsers generally operate faster than DOM parsers, so even with small datasets and minimal post-parse processing, a small speedup in FEARS performance should still be evident.

Activity Graph Implementation

The activity graphs described previously consist of nodes, representing activities, and edges, representing dependencies. During execution of an activity graph on a SPAM model, information about the properties of the activities will be required to determine the performance characteristics of those activities, and information about the dependencies will be required to determine when the activities can begin and complete execution.

When an activity graph description is read from file, a recursive depth-first traversal of the graphs, starting from a specially labeled activity node, is performed and an Activity object is constructed to wrap each graph node. The complete set of activities is stored in a container of class `ActivityGraph`. An STL (Standard Template Library) `map` is used to track which nodes have already been processed, in case cyclic dependencies exist. The `Activity` class provides functionality in addition to that provided by the FEARS `Node` object it wraps, including tracking the progress of the activity and handling its incoming and outgoing dependencies. These operations will be described in more detail in section 3.3, where the interaction between component models and executing

activities will be discussed.

Since different activities represent a task's usage of different platform components, each activity may need to provide a different set of properties. To allow this flexibility, properties are specified as name-value pairs, with both the name and value being strings. C++ template methods were added to the FEARS API to allow for easy conversion of value strings to other data types, such as integer or floating point numbers. Additionally, some models may require arrays of properties. For example, the instruction mix for a processor is defined by the proportion of instructions belonging to each instruction class. A convenient way of representing this data is as an array of values, with one value representing the instruction proportion for each instruction class. Rather than requiring a separate name-value pair for each value in an array, methods to access comma-separated lists of values, under a single name, were added to the FEARS API.

3.3 Component Model Framework

The workload specification, in terms of activities, their properties and dependencies, is used to drive a simulation of the performance estimation platform model. This platform model consists of a set of interconnected component models. As discussed previously, the component models use a common base class to provide required functionality, and to present a common interface which can be used and extended to create custom component models.

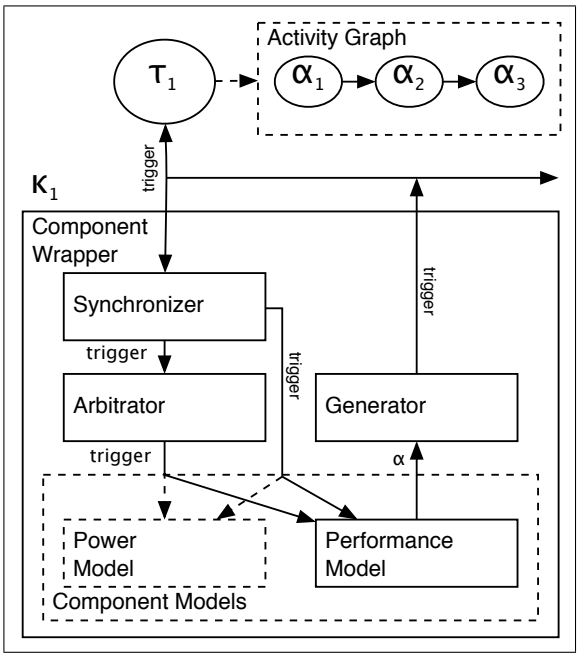


Figure 3.10: Component Model Wrapper

A major requirement of component models is to enforce the execution of the workload as specified by the task's activity graph. This includes tracking the progress of running activities, determining when activities should start or complete execution, and

determining which supplementary result values should be sent to which dependent activities.

To perform these duties, a set of helper modules is instantiated along with each component model, as shown in 3.10. This component wrapper consists of:

- a synchronizer, which determines when an activity's activation or completion dependencies are satisfied
- an arbitrator, which determines whether a component model can execute additional activities, and if so, selects an activity to run
- a generator, which, based on activity progress, determines when activation and completion dependencies are to be triggered, and sends messages to update property dependencies whenever supplementary result values change.

The component model itself consists of a performance predicting model, which is derived from a common base class. Figure 3.10 also shows the possible integration of a power predicting model. Power modeling has not yet been implemented. Some possible methods of implementing power modeling are described in chapter 9.

3.3.1 Implementation

As shown in 3.10, each component's performance estimation model is wrapped in a SystemC module, `ComponentWrapper`. The `ComponentWrapper` module is structurally composed from the helper modules and a performance estimation model, which is a component-specific model derived from the `ComponentModel` base class.

The `ComponentModel` class is itself derived from the SystemC `sc_module` class, so all component models are also SystemC modules.

During execution of a workload, each activity will progress through four operational states: idle, waiting, running, and finished. An activity will be idle until all incoming activation dependencies are satisfied. When the activation dependencies are satisfied, the activity may progress to a waiting state, or may immediately progress to a running state. The decision on which transition occurs depends on the type of component which is allocated to the activity, and on the status of the component when the transition occurs. If the component is occupied and cannot be shared, the activity transitions to waiting; otherwise, the activity transitions to running. Finally, when the activity's progress is completed and all its incoming completion dependencies are satisfied, the activity will transition to finished. This operation is shown in figure 3.11.

An activity's progression through its operational states is tracked implicitly, through the `Activity` object's progression between the different utility modules contained in the ComponentWrapper. Idle activities will be blocked in the `Synchronizer` module until their activation dependencies are satisfied; waiting activities will be blocked in the `Arbitrator` module until the component they are waiting for becomes free; and running activities are blocked in the `ComponentModel` module until they are completed. No explicit state machine is maintained for each activity. The benefit of this implicit state progression is that there is no need to traverse complete activity lists to determine which activities should transition between states. Instead, when an event occurs, only those activities which may transition as a result of that event must be processed.

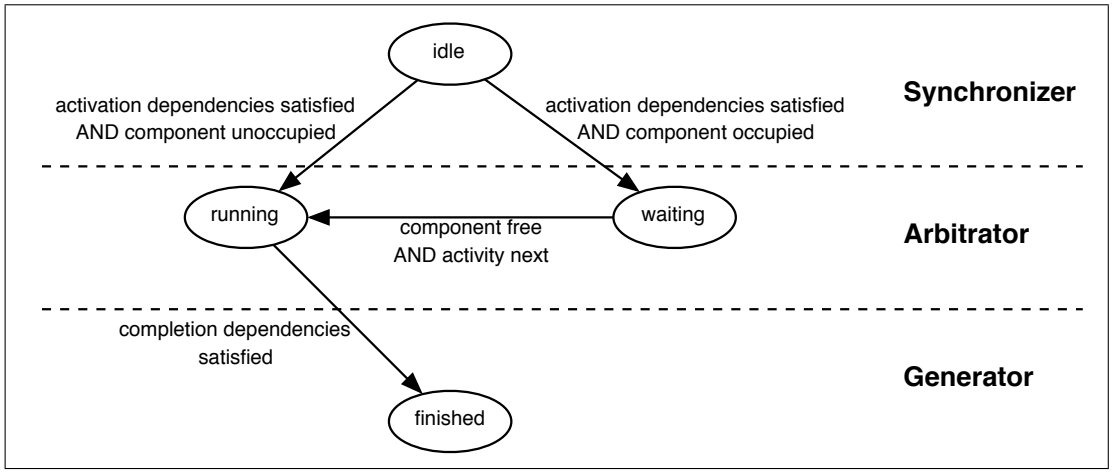


Figure 3.11: Activity Operational State Diagram

Generator

During execution of a workload, the `Generator` module generates `Trigger` objects, which are sent on a common SystemC master-slave link to all `ComponentWrapper`s in the platform model. These triggers are sent whenever the activity progress requirements of an activation or completion dependency are satisfied, or whenever new data must be sent between models due to a property dependency. The generated `Trigger` object contains the information corresponding to the associated dependency: the source and sink activities, and, for property dependencies, the source and sink properties.

When an activity's progress equals or exceeds its magnitude, the `Generator` sends a self-completion `Trigger`, which has the completed activity as both source and sink. Each activity has an implicit completion dependency on itself, which does not need to be specified in the workload description. This mechanism allows the same generalized dependency processing already implemented in the `Synchronizer` module to consider the completion of an activity's progress.

Synchronizer

The `Synchronizer` module processes all incoming activation and completion triggers, and forwards property triggers to the `ComponentModel`. The `Synchronizer` maintains separate lists of activities which are awaiting further activation or completion dependencies. Each activity in turn maintains separate lists of its expected activation and completion dependencies.

When the `Synchronizer` receives an activation `Trigger`, the sink `Activity` is informed that the specified dependency has been satisfied. The `Activity`'s activation dependency list is updated by removing the satisfied dependency. If the `Activity`'s dependency list is now empty, the `Synchronizer` can remove the activity from its list of activities awaiting activation, and forward the activation or completion `Trigger` to the `Arbitrator` module for further processing. Depending on the `Arbitrator`'s decision, the `Activity` will implicitly transition from the idle state to either the waiting or running state, by being transfered into the list of waiting or running activities. Similarly,

when a completion `Trigger` is received, the `Activity`'s completion dependency list is updated, and if all completion dependencies are satisfied, the `Synchronizer` removes the `Activity` from its list, and the `Trigger` is forwarded.

Arbitrator

The `Arbitrator` module responds to activation or completion `Triggers` forwarded to it by the `Synchronizer`. The operation of the `Arbitrator` depends on the specification of the architecture. In particular, the `Arbitrator` will respond differently depending on whether the component being modeled is a shared or exclusive component. For fully shareable components, meaning that the component theoretically can be shared by any number of activities, the `Arbitrator` becomes transparent, and simply forwards all `Triggers` to the `ComponentModel`. For partially shareable components or for exclusive components, the `Arbitrator` tracks the occupied status of the `ComponentModel` by maintaining a list of the running activities. Additionally, the `Arbitrator` maintains a FIFO queue containing waiting activities.

The list size and the potential degree of sharing indicates whether the component is occupied. If the component being modeled can be shared by N components, then the component is occupied if the size of the running list is N, and not occupied if the size of the running list is less than N. If the component being modeled is exclusive, then the component is unoccupied if the size of the running list is 0.

When the `Arbitrator` receives an activation `Trigger`, it first checks the status of the component model. If the component model is occupied, then the `Activity`, indicated by the `Trigger`'s sink, is placed in the FIFO queue. This implicitly transitions the activity to the waiting state. If the component is not occupied, the `Activity` is placed into the running list, and the trigger is forwarded to the `ComponentModel`, indicating that the activity is now in the running state. When a completion `Trigger` is received, the indicated `Activity` is immediately removed from the running list and the `Trigger` is forwarded, causing the `Activity`'s implicit transition to the finished state.

ComponentModel Base Class

The functionality for component models is divided into base class functionality provided by the `ComponentModel` class, and required child class functionality, which is declared in the base class using abstract virtual functions. The use of abstract virtual functions in C++ means that any child classes derived from the base class must provide an implementation of the virtual function; and that instances of the base class cannot be constructed.

The `ComponentModel` base class provides three main areas of functionality. First, it is responsible for receiving activation and completion `Triggers` from the `Arbitrator`, and maintaining a list of running activities. In addition, it receives property `Triggers` which are forwarded by the `Synchronizer`, and updates `Activity` properties based on these triggers. Finally, the `ComponentModel` base class updates the progress of running activities at each simulator clock tick. After updating `Activity` progress, the `Activity` objects are sent to the `Generator`, which as described above, determines whether dependencies have been satisfied.

ComponentModel Child Class

Each component model child class implementation must provide a method for estimating the rate of execution of an activity executing on the component. The platform simulator uses the execution rate result to update the progress of running activities at each simulator clock cycle. Since the method of performance estimation will be different for each type of component, this `estimate` method is declared as a C++ abstract virtual function in the base class.

The `estimate` method is called whenever membership in the set of running activities changes, or whenever properties of running activities change. Generally, the `estimate` method should then iterate through all running activities and recalculate their execution rate. Components which are being shared by multiple activities should take into account the properties of all running activities when estimating execution rate, as discussed in chapter 5.

Component models may also produce other supplementary results, in addition to the execution rate. These results may be required by property dependencies expressed in the workload. To allow flexibility in designing new models, supplementary results are also specified as name-value pairs. Like the progress property of activities, the value of a supplementary result is set at runtime, but the names of supplementary results are defined by the type of component. A method for registering supplementary results is therefore provided by component models. Since the specific supplementary results produced by each component model may differ, the `ComponentModel` base class cannot provide this functionality itself. The method for registering supplementary results is therefore declared as an abstract virtual function, and specific component model child classes must define an implementation.

During a simulator run, the supplementary results produced by a model are added as extra properties to the activities executing on the component. Generally, the values for the supplementary results are calculated at the same time as the execution rate is calculated, through the `estimate` method.

ComponentModel Properties

A design goal for the specific component models for processors, memories and bus interconnect, as described in the following chapters, is to provide flexible and general models of components such that the same model can be used to predict performance for a wide range of components. The set of component properties required to describe a particular component model will be dependent on the type of component. Therefore, as with activities described in section 3.2, a name-value pair method is used to specifiy arbitrary component properties. This functionality is provided by the FEARS API.

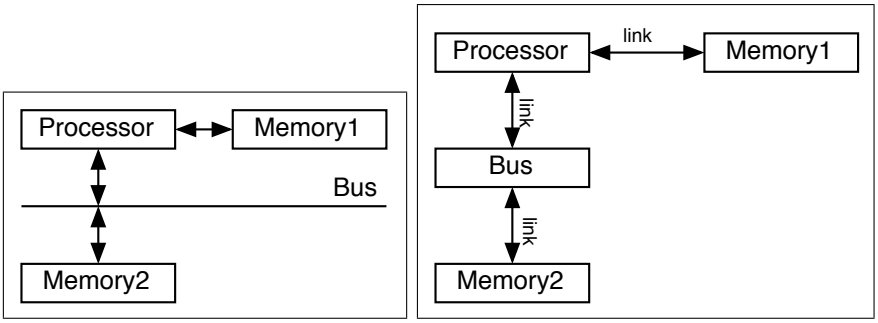
Power Models

Power estimation is a potentially important factor in design evaluation, particularly for mobile devices. Although power modeling has not been implemented in this project, the modeling framework supports the implementation of power models through the same component model mechanism used for performance modeling. This concept is discussed further in chapter 9.

3.4 Platform Model Framework

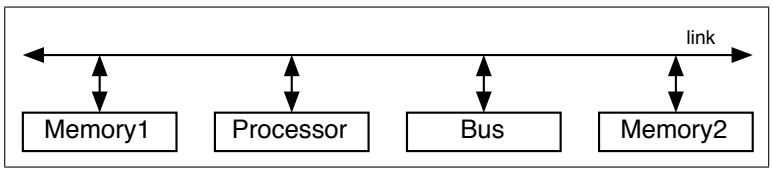
To model a complete platform, several component models with wrappers are instantiated, and their actions are simulated in parallel. Links between the `ComponentWrapper`s deliver `Trigger`s corresponding to the activation, completion and property dependencies of the workload executing on the model.

In most real platform designs, not every component will be connected to all other components. Private memories may be connected only to their associated processor, shared memories may be isolated from processors by a bus, or memories may be separated from processors by caches. It may be possible to implement explicitly the connections in the platform model by creating dedicated links only between components which have an actual connection, as shown in figure 3.12(b). Alternatively, a single shared link could be used in the platform model, as in 3.12(c), with all dependency `Trigger`s being sent to all other components, and filtering being performed to ensure that only the correct destination component acts upon the `Trigger`.



(a) example platform

(b) platform model with dedicated links



(c) platform model with single shared link

Figure 3.12: Platform Models with Different Communication Link Topologies

Using dedicated links provides two advantages:

- allows runtime checking of dependencies specified in workload descriptions
- reduces processing requirements necessary to filter all triggers at non-destination components.

However, the implementation complexity is significantly higher than using a single shared link. Since different platform models have different topologies of component connections, a flexible means of specifying, instantiating, and utilizing the dedicated links would have to be created. Furthermore, the number of components in a typical

platform is expected to be relatively small, meaning that traffic on a single link will not become prohibitively expensive. For these reasons, platform models are constructed using a single shared link.

3.4.1 Implementation

As with description of workloads, using input files to describe platform architectures allows a single executable to be used to model diverse platforms, without the need for recompiling. Platforms are therefore described as XML files. When performing a simulation, a platform model is constructed by instantiating component models as specified in the platform architecture XML file. Since component models are connected by a single shared link, it is not necessary to specify the connections between components in the XML description. A simplified XML format and XSL stylesheet can also be used for architecture description, for the reasons outlined in section 3.2. Figure 3.13 gives example XML platform descriptions corresponding to the platform shown in figure 3.12.

Once the platform architecture has been specified in a FEARS XML file, the file can be parsed by the FEARS SAX parser described previously. Then, for each node in the architecture graph, a `ComponentWrapper` can be created. Since each node in the graph may describe a different type of component, the `ComponentWrapper` instantiated for each node must contain the appropriate `ComponentModel` subclass. In SystemC, a structurally composed module like `ComponentWrapper` must create each submodule in its constructor. Therefore, a mechanism is required through which a component model child class name can be passed to the `ComponentWrapper` constructor. C++ templates provide this mechanism. The `ComponentWrapper` is therefore subclassed with a class template, `ComponentWrapperImpl`, which takes as an argument the name of the `ComponentModel` child class which should be instantiated. `ComponentWrapper`, the base class, declares the members common to all `ComponentWrapper`s. `ComponentWrapperImpl` defines the constructor implementation.

The `ComponentWrapper` base class also provides a static factory method which will create an instance of the appropriate `ComponentWrapperImpl`, based on the `model` property of a FEARS graph node. The SystemC `sc_main` function for the SPAM model constructs a platform model by iterating through the graph nodes in the FEARS architecture input file, and calling the `ComponentWrapper` factory method. The `sc_main` method then connects all the constructed `ComponentWrapperImpl`s to a single master-slave link. After also constructing the `ActivityGraph` object for the workload, as described in section 3.2, the `sc_main` function then runs the simulation, which starting with the initial `Activity`, executes the workload. The completion time of the final executing activity will determine the performance of the workload.

3.5 The ARTS Model

The ARTS model [22] presents a similar SystemC approach for modeling RTOS operation in a SoC environment. ARTS models the execution of RTOS tasks on multiple processing elements, while SPAM models sub-task activities executing on components, including processing, storage and communication elements. Aside from the level of abstraction of the intended modeling, there are several key differences between ARTS and

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CONTENT SYSTEM "DGRAPH.dtd">

<CONTENT id="pisa-root">
        <STRUCTURE id="pisa-arch">
                <NODE id="processor">
                        <INTERFACE id="processor-int"/>
                        <PROPERTY id="id" value="processor"/>
                        <PROPERTY id="modelType" value="proc"/>
                        <PROPERTY id="exampleProperty" value="1234"/>
                </NODE>
                <NODE id="memory1">
                        <INTERFACE id="memory1-int"/>
                        <PROPERTY id="id" value="memory1"/>
                        <PROPERTY id="modelType" value="mem"/>
                        <PROPERTY id="exampleProperty" value="5678"/>
                </NODE>
                <NODE id="memory2">
                        <INTERFACE id="memory2-int"/>
                        <PROPERTY id="id" value="memory2"/>
                        <PROPERTY id="modelType" value="mem"/>
                        <PROPERTY id="exampleProperty" value="1234"/>
                </NODE>
                <NODE id="bus">
                        <INTERFACE id="bus-int"/>
                        <PROPERTY id="id" value="bus"/>
                        <PROPERTY id="modelType" value="bus"/>
                        <PROPERTY id="exampleProperty" value="5678"/>
                </NODE>
        </STRUCTURE>
</CONTENT>
```

(a) FEARS XML description

```
<?xml version="1.0" encoding="UTF-8"?>
<arch id="pisa">
        <component id="processor" modelType="proc" exampleProperty="1234"/>
        <component id="memory1" modelType="mem" exampleProperty="5678"/>
        <component id="memory2" modelType="mem" exampleProperty="1234"/>
        <component id="bus" modelType="bus" exampleProperty="5678"/>
</arch>
```

(b) simplified XML description

Figure 3.13: Platform Model XML Descriptions

SPAM:

- ARTS uses a single, global synchronizer to track dependencies, whereas SPAM uses multiple, component-specific synchronizers and generators to track activity dependencies
- ARTS provides scheduler modules, which implement one of a number of RTOS scheduling algorithms, in place of SPAM's simple FCFS arbitrator
- ARTS tasks' execution times are determined randomly, between pre-specified best case and worst case limits, and tracked by a simple counter; SPAM execution times are determined by numerical integration of component-determined execution rates
- SPAM defines an object-oriented component modeling mechanism for determining execution rates
- SPAM provides mechanisms for data passing between activities, in the form of property dependencies
- ARTS provides an allocator, which models resource usage as exclusive locking of a resource for the entirety of a task's duration; SPAM considers the resources as separate components, and models potential sharing of resources as described in chapter 5
- ARTS maintains task state in a state machine for each task, while SPAM implicitly tracks activity state by passing `Activity` objects between modules
- SPAM provides an explicit wrapper class for creating components and all required modules
- SPAM provides file-based description methods for workloads and platforms; ARTS models must be constructed in code.

Of these differences, the key difference for platform modeling is the implementation of execution rate determination and progress integration. This allows changes in platform configuration to be reflected in terms of performance figures. The ability to pass data between activities through data dependencies proved necessary for platform modeling, but may not be for consideration of tasks in a multiprocessor, which typically communicate explicitly, and can thus be modeled through dependencies. Also, the consideration of resource sharing through separate component models allows a finer-grained analysis of resource usage.

On the other hand, SPAM currently loses the ability to consider different scheduling methods. This deficiency was deemed to be acceptable, as non-processor components typically do not permit complicated runtime scheduling. Processor scheduling can be either ignored, by using SPAM to predict performance of individual tasks, or considered by integrating SPAM with a higher-level model like ARTS. This concept will be discussed in chapter 9.

The remaining differences between the models are, from a conceptual perspective, relatively minor implementation and convenience details.

3.6 Summary

The concepts, classes and methods described in the previous sections provide the basis for a SoC platform modeling system. Figure 3.14 shows the hierarchy of classes introduced thus far. The class hierarchy will be extended in the following chapters as

implementations of models for particular platform components are introduced.

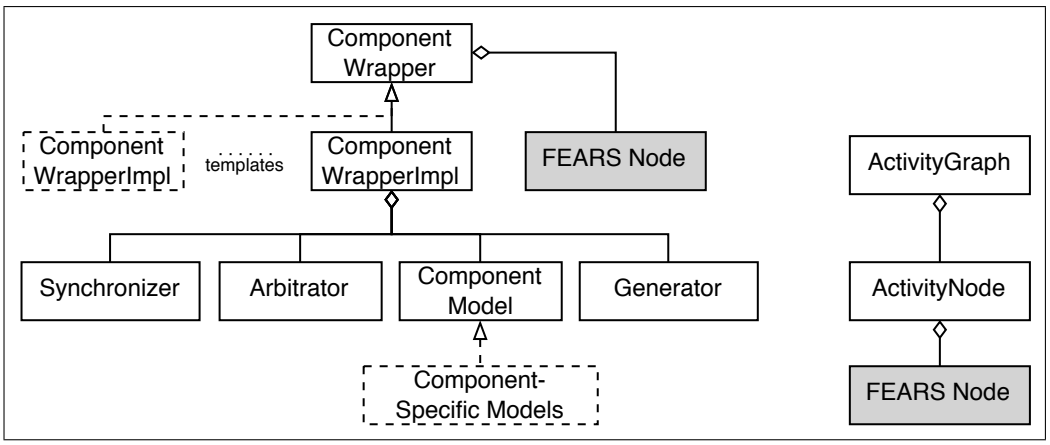


Figure 3.14: Class Diagram

Chapter 4

Processor Modeling

4.1 Introduction

4.1.1 Motivating Considerations

Determining how the architecture of a processor core affects performance under a given workload can be ascertained in several ways. One method would be to run instrumented code on an actual hardware implementation of the core and gather performance figures through instrumentation of the code or through built-in hardware support for performance measurement. This method is clearly not suitable for design space exploration, as it would necessitate investment in a large variety of possible prototype processor cores. Alternatively, one could execute the workload on a cycle-accurate simulator of the processor core. However, this method would require a similar investment in developing or obtaining cycle-accurate simulators for each instruction set architecture under consideration. Furthermore, the simulation slowdown inherent in cycle-accurate simulators means executing non-trivial workloads can result in infeasible runtimes, especially when numerous potential designs must be considered separately in a design space exploration tool.

While cycle-accurate simulation is ideal for verification of workload performance on a particular architecture, or for detailed study of the performance of a particular architecture, the need for faster performance estimation methods has motivated significant research into statistical or analytical methods for processor performance evaluation. A variety of analytical methods are discussed in chapter 2, including methods using MVA, queueing systems models, and Markov chain analyis.

Performance of a processor is influenced by numerous factors. At a very low level, performance is influenced by transistor device parameters, place and route decisions, and other low-level design and process details. At higher levels, the processor performance is affected by branch prediction methods, number of pipeline stages, number of functional units and other high-level architectural decisions. A useful fast estimation method should consider those architectural details which are modeled feasibly in an efficient manner, and those which have the greatest effect on processor performance.

Yi, Lilja and Hawkins describe [14] the use of rigorous statistical analysis to determine the relative importance, with respect to processor performance, of various design parameters of a superscalar processor. These investigations were performed with a view

to selecting appropriate simulator parameterizations when investigating novel architectural features. This analysis is also only entirely valid for the specific workloads under which it was performed. However, the results show some general trends of importance.

For design space exploration, it may be important to consider a wide range of potential processor architectures, from small scalar processors to complex superscalar processors with out-of-order execution. While it may be possible to use different models for performance evaluation of different classes of processors, care would have to be taken that the models are calibrated to one another. If, for example, a design space exploration tool compares results from a superscalar processor model which consistently overestimated performance to results from a scalar processor model which tends to underestimate performance, the conclusions of the design space exploration would be in question. Therefore it is preferable to use a single processor model which can consider as large a range of processor designs as possible, provided that such a model can produce consistently accurate results.

4.1.2 Model Selection

The requirements for a processor model for high-speed design space exploration can therefore be summarized as:

- fast execution
- consideration of key architectural features
- applicability to a wide range of processor designs
- low implementation complexity.

These requirements motivate the selection of a processor model to be implemented as part of the SoC platform model. Given the relatively comprehensive investigation of processor performance modeling which has been presented in previous literature, an existing model was selected. The selected model was extended to consider more classes of processors, modifications were made to improve the accuracy of the model predictions in some circumstances, and some additions were made to the model to allow integration into the overall platform model.

The method proposed by Taha and Wills [31] provides a computationally simple algorithm which produces reasonably accurate estimates of processor performance. The model results are produced in terms of instructions per cycle (IPC), while varying several architectural features of an out-of-order superscalar processor:

- issue window size
- reorder buffer size
- functional unit latencies, multiplicities, and throughput
- branch misprediction latency.

Other processor parameters are considered indirectly through model inputs, which must be provided through measurements on a cycle-accurate simulator. These parameters include effects of branch predictor design and memory/cache hierarchy design. The non-modeled parameters present a potential avenue of future research, as a model which could integrate the effects of these parameters would reduce the number of simulator runs that would be required to gather model inputs.

The instruction throughput model does fulfill several of the key requirements for a processor model presented above, particularly fast performance and relatively low

implementation complexity. The model also directly considers many of the key processor parameters as found by Yi, Lilja and Hawkins [14], and the most of the remainder are considered indirectly through model inputs. Moreover, while the model as presented is specific to superscalar processors, the task of generalizing a superscalar model to consider simpler architectures is likely to be more effective than trying to predict superscalar processor performance with a scalar processor model.

In section 4.2, the original instruction throughput model [31, 30] is described. In section 4.3, the modifications made to the original model in order to consider scalar processor performance are presented. The implementation of this model is described in section 4.2.1. The reader is assumed to have general knowledge of superscalar processor design and operation [27].

4.2 The Instruction Throughput Model

The instruction throughput model [31, 30] of superscalar processors considers the workload as a series of macroblocks, sequences of instructions which are separated by mispredicted branch instructions. The average length of a macroblock is determined for a particular workload by a one-time simulation run on a cycle-accurate simulator. Then, the IPC can be calculated as:

$$IPC = \frac{[NumberOfInstructionsInMacroblock]}{[ClockCyclesToExecuteMacroblock] + [PipelineRefillPenalty]} \tag{4.1}$$

The pipeline refill penalty is a known architectural property of a processor design, so what remains to be determined is the time required to execute an average macroblock. This is determined using an iterative method, where a set of instructions progress from issue to retirement, as shown in figure 4.1.

At the initiation of the iterative algorithm, a number of instructions equal to the average macroblock size are placed in the cache, i.e.:

$$C_0 = [NumberOfInstructionsInMacroblock] \tag{4.2}$$

During each iteration i, the variables C_i, W_i, R_i and $P_{n,i}$ are updated to reflect the processor's typical or average behavior during a clock cycle. In particular:

- the number of instructions waiting in cache is reduced by the fetch rate: $C_{i+1} = C_i - F_i$
- the number of instructions in the issue window is increased by the fetch rate, and decreased by the number of instructions issued to functional units for execution: $W_{i+1} = W_i + F_i - E_i$
- the number of instructions in the first pipeline stage is set to the number of instructions issued for execution: $P_{0,i+1} = E_i$
- instructions in the pipeline are advanced to the next pipeline stage: $P_{n+1,i+1} = P_{n,i}$ where $0 < n < N_{stages}$
- instructions in the last pipeline stage are considered completed; the number of completed instructions is used to help determine the number of instructions which can be removed (graduated) from the reorder buffer, and thus

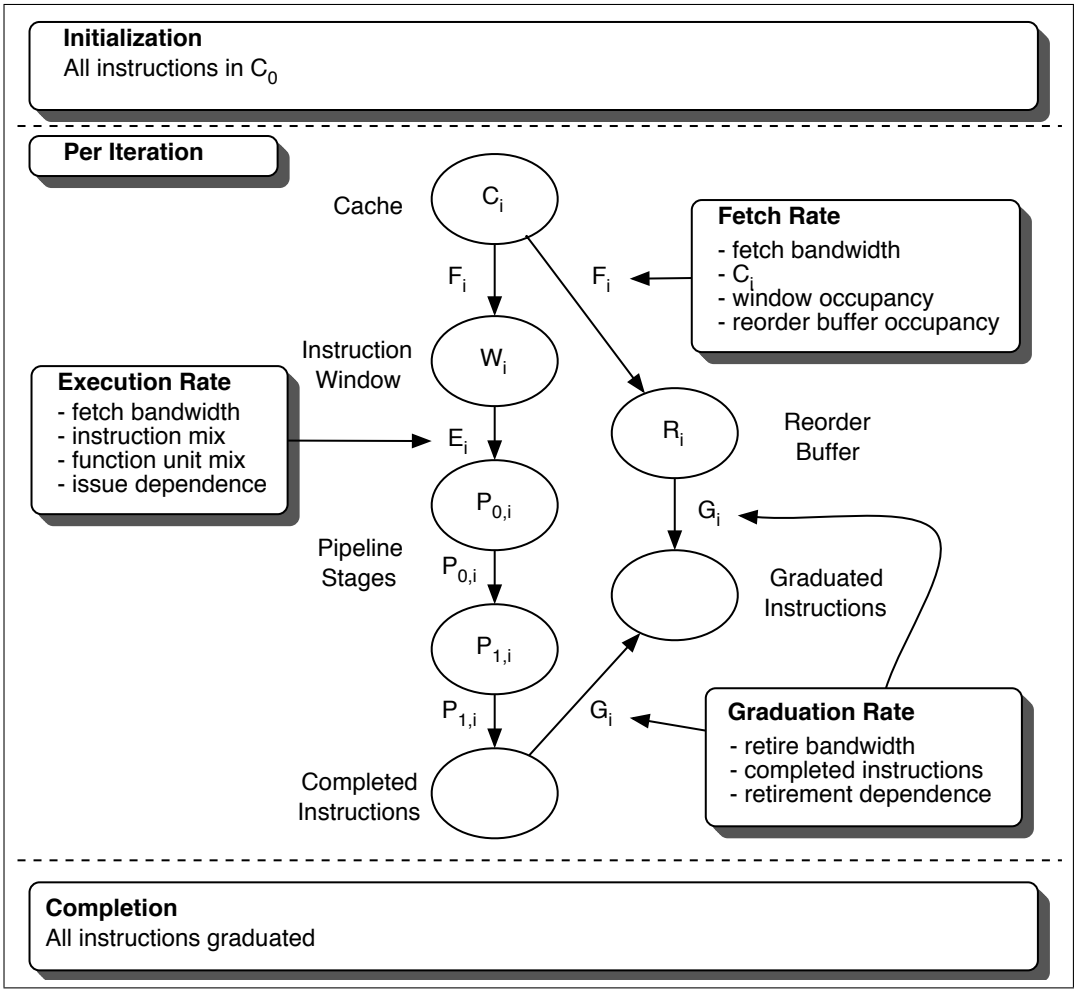


Figure 4.1: Iterative Method for Macroblock Execution

- the number of instructions in the reorder buffer is reduced by the number of graduated instructions, and increased by the number of instructions fetched: $R_{i+1} = R_i + F_i - G_i$.

Two relations are used to characterize the effects of data dependencies in a workload. An issue dependence relation is used to determine the average fraction of instructions in the issue window which can be issued. The issue dependence is measured for a workload by executing the workload on an idealized simulator which is non-ideal only in the size of its issue window. By varying the issue window size and measuring the resulting effect on issue rate, a relationship which defines the inherent limitations on issue parallelism can be found. A retire dependence relation similarly determines the number of instructions which can retire (graduate) in a cycle, depending on the number of completed instructions in the reorder buffer, and similarly can be determined by executing the workload on an idealized simulator with only reorder buffer size limited.

These dependence relations are used, along with other architecture parameters and the current state of the iterative processor model components (cache, issue window, reorder buffer and pipeline stages) to determine the rates of change described above: fetch rate F_i, execution (or issue) rate E_i, and graduation rate G_i.

The fetch rate F_i is always limited by a hard cap, determined by the processor core's hardware fetch bandwidth. The fetch rate is also limited to the workload's basic block size. This reflects an assumption that instruction fetch cannot continue correctly past a branch instruction. Additionally, the fetch rate cannot exceed the open capacity of either the instruction window or the reorder buffer. Both these capacities should be calculated while taking into account the number of departures from the respective buffers, to reflect the pipelining operation that they perform. For example, the limitation on F_i due to instruction window occupancy would be $F_i < W_{size} - W_{i-1} + E_i$, where W_{size} is the total capacity of the instruction window.

The graduation rate G_i is, like the fetch rate, limited by a hardware bandwidth, which represents constraints imposed by the instruction commit stage of the superscalar processor. Additionally, the graduation rate is limited by the retire dependence relation described above, which is evaluated for the current reorder buffer occupancy and current number of completed but unretired instructions.

The execution rate E_i is similarly limited by the issue dependence, evaluated for the current issue window occupancy. The execution rate also cannot exceed the hardware fetch rate, to reflect the fact that average, sustained throughput of the processor cannot exceed the rate at which instructions are fetched. The limited execution rate is then multiplied by a factor which accounts for the average rate at which the program instructions, defined by the instruction mix, can proceed through the set of functional units which the processor possesses. This factor is determined by calculating the ratio of constrained issue to unconstrained issue for each instruction class.

The constrained issue is calculated as:

$$I_{constrained_c} = min(I_{unconstrained_c}, T_c) \tag{4.3}$$

where $I_{constrained_c}$ and $I_{unconstrained_c}$ are the constrained and unconstrained issue rate for an instruction class c, and T_c is the total throughput for functional units servicing class c. The unconstrained issue $I_{unconstrained_c}$ is given by the unmodified execution rate E_i' multiplied by the instruction mix proportion for class c.

Having calculated the constrained and unconstrained issue for each instruction class, the unmodified execution rate $E_i^{'}$ is multiplied by the minimal factor $\frac{I_{constrained_c}}{I_{unconstrained_c}}$ to give the actual execution rate E_i.

In a processor with multiple functional units, it is quite likely that not all functional units will have the same latency. Different functional units will have different numbers of pipeline stages. The pipeline stages $P_{n,i}$ describe the per cycle state of a single, effective functional unit. The latency of this functional unit is equal to the weighted average of the functional unit latency for each instruction class. The weighting is provided by the instruction mix proportions for the workload under consideration. If the fraction of the instruction mix belonging to instruction class i is p_i, and the latency for this instruction class is L_i, then the latency of the effective functional unit is:

$$L_{eff} = \sum_{\forall i} p_i L_i \tag{4.4}$$

The number of pipeline stages N_{stages} can be set by rounding the effective latency L_{eff} to the nearest whole number.

The iterative algorithm terminates when all instructions in a macroblock have graduated. To account for the effects of instructions from previous macroblocks, the algorithm is executed twice, the second execution using the states of W_i, R_i and $P_{n,i}$ at the termination of the first execution as its initial values. This captures the effects on performance of instructions remaining in the processor reorder buffer on a macroblock's execution time. The number of iterations to reach termination for the second execution of the iterative algorithm provides the estimate for the number of cycles to execute a macroblock. This result is used with the known values for macroblock instruction count and pipeline refill penalty to calculate the estimated IPC, as shown in equation (4.1).

4.2.1 Implementation

The instruction throughput model is implemented by deriving a child class, `ProcessorModel`, from the base `ComponentModel` class described in chapter 3. The operations of the model described in the previous section are implemented in the model's estimate method, which is required for all classes derived from `ComponentModel`. The `ProcessorModel` implementation requires a certain set of properties to be defined in the processor's graph node in the architecture definition file. These properties provide the model parameters for the instruction throughput model. To execute the model on different processor configurations, all that is required is to provide a different set of properties in the architecture definition file. Table 4.1 lists the properties required by the processor model.

4.3 Scalar Processor Modeling

The instruction throughput model as described in section 4.2 is designed for estimation of superscalar processors. Many embedded cores and processor IPs intended for use in SoCs are simpler, scalar processors with in-order execution and in some cases are unpipelined designs. Some modifications to the original instruction throughput model

Property Name	Description	Units
`hwFetchRate`	hardware fetch bandwidth	inst/cycle
`hwRetireRate`	hardware instruction commit bandwidth	inst/cycle
`windowSize`	issue window size	insts
`reorderSize`	reorder buffer size	insts
`branchMissPenalty`	branch predictor misprediction penalty	cycles
`functionalUnitMults`	functional unit multiplicities (array)	
`functionalUnitLats`	functional unit latencies (array)	cycles
`functionalUnitTputs`	function unit throughputs (array)	inst/cycle/unit

Table 4.1: Processor Model Properties

are necessary to accurately predict performance of scalar and unpipelined processors. Since the operation of a scalar, in-order processor is simpler and more predictable than that of a superscalar processor, the instruction throughput model can provide a useful basis for scalar processor performance prediction.

The key difference between scalar and superscalar processors which must be considered by the model is the mechanism for issuing and retiring instructions. In a scalar processor, instruction issue is limited by definition to at most 1 instruction per cycle. This hard limit can be enforced by the issue window size used in the iterative model. Pipelined scalar processors may have their rate of instruction issue further limited by data dependencies which cannot be avoided through register forwarding. As with superscalar processors, the effect of these dependencies is modeled by an issue dependence relation, which is determined through a one-time execution of the workload on an idealized cycle-accurate simulator.

Although scalar processors do not make use of duplicate, parallel functional resources, and thus the instruction mix is not relevant for determining issue rate constraint, it is still necessary to consider the effects of the instruction mix. Certain classes of instructions may require extra cycles to execute, for instance. The use of long latency instructions may have several consequences on processor performance. The use of these instructions increases the average latency, which is accounted for by the effective pipeline length calculation described in section 4.2. If the functional unit is not fully pipelined, however, the use of long-latency instructions may also prevent the issue of subsequent instructions. A measurement of instruction mix for the workload can be combined with the number of stall cycles introduced by the instruction to calculate the effect on issue rate, and this effect can be modeled by adjusting the issue dependence relation.

For example, if a sample workload executing a particular scalar processor consists of 90% single cycle instructions, and 10% of instructions require two cycles, then the issue dependence should reflect that, in the average case, 90% of instructions should be issued each cycle. However, no instruction issue should be delayed more than 1 cycle. Since the scalar processor is modeled with an issue window size of 1, this means that if there are 0.1 instructions or less in the issue window, these represent instructions following a long-latency instruction and they must all be issued. To reflect this, the issue dependence for a scalar processor can be approximated by a curve such as that shown in figure 4.2, where p_{issue} is the average number of instructions issued per cycle

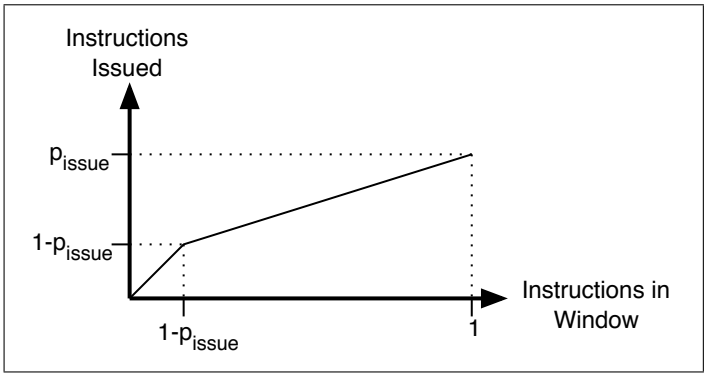


Figure 4.2: Issue Dependence Relation for Scalar Processors

in the given workload.

For unpipelined scalar processors, or processors with very short pipelines, one must consider how well the iterative processor model mimics the real operation of the processor. In the iterative model, an instruction will take at least 4 cycles to graduate: it must be fetched, then issued, then completed, and finally graduated. Thus, the iterative model assumes that the processor contains instruction fetch, instruction issue, and instruction retire stages in addition to at least one execute stage. The model can easily consider deeper pipelines by introducing more execute stages. To consider shorter pipelines, one must adjust the results to reduce what is effectively a 3 cycle latency penalty. For this reason, a correction factor has been added to the model, which is subtracted from the macroblock execution time before equation (4.1) is applied. The correction factor can be configured as needed to account for the difference between the pipeline configuration assumed by the instruction throughput model and the actual pipeline configuration of the processor under study. Thus:

$$IPC = \frac{[NumberOfInstructionsInMacroblock]}{[ClockCyclesToExecuteMacroblock]-[PipelineLatencyCorrection]+[PipelineRefillPenalty]} \quad (4.5)$$

4.4 Platform Model Integration

Some functionality must be present in the processor model to allow integration into a platform model consisting of processor(s), bus interconnect, and memory. This section describes the particular changes made to the processor model described in the previous sections which are necessary to allow integration. The construction of a complete platform model, and the results obtained from such a model, are considered further in chapter 8.

To integrate the processor model into a complete platform model, it is necessary to consider the effect of the memory subsystem, including caches, main memory and the interconnect used to access main memory. This effect must be considered in terms of both data memory access time for load/store instructions, and instruction fetch. The effects of memory access time for instruction fetch are accounted for by modifying the hardware fetch rate used in the iterative method for calculating macroblock execution time.

Because of the interdependence of processors, bus interconnect and memory, the memory access time property of the processor may not be a constant value. In a multiprocessor platform, for example, a processor may experience a longer average memory access time while another processor is accessing a shared memory, but if the other processor completes its task and becomes idle, the average access time could drop. For this reason, the memory access time property may often be the target of a property dependency from the memory or bus component which handles memory requests.

As described in chapter 3, the progress of each activity executing on each component model is incremented at each interval of the overall simulator clock by an amount equal to the execution rate of the activity. For the processor model, this execution rate is the IPC calculated by the instruction throughput model. As the IPC is dependent on the memory access time, this execution rate is recalculated each time the memory access time property is changed.

4.5 Model Modifications

The instruction throughput model produces reasonably accurate results for a range of benchmarks, as will be shown in section 4.6. However, there may be opportunities to improve the accuracy. This section proposes two possible modifications to the model. The first involves more precise handling of the effective pipeline length. The second considers possible changes to the method used to constrain issue due to functional unit limitations.

4.5.1 Pipeline Operation

The instruction throughput model operates with a single effective pipeline with a latency determined using a weighted average of the actual functional unit latencies. As such, it is likely that the calculated effective pipeline length is non-integral. Since the pipeline is modeled as a series of discrete steps, this effective pipeline length must be rounded to an integer value for use in the iterative method. This can result in a model which does not consider the effects of long-latency instructions, for example, if the bulk of the instruction mix consists of short-latency instructions.

It is relatively simple to adapt the instruction throughput model to account for fractional pipeline lengths. The pipeline can be instantiated with a length given by $\lceil L_{eff} \rceil$. The fractional remainder of L_{eff} can be saved and used to determine the proportion of instructions which proceed to the final pipeline stage. The remainder of the instructions can be removed from the pipeline one cycle early and considered completed.

As shown in section 4.6, the effect of this modification are relatively minor. Furthermore, they do not always result in improved model accuracy. However, the operation of the model with this modification is more intuitively correct than a model which assumes a worst case or best case situation through rounding of the effective pipeline length.

4.5.2 Functional Unit Contention

The instruction throughput model considers the throughput constraints imposed by functional unit availability through use of a hard cap on the number of instructions of each class issued per cycle. However, such a method does not consider the potential for temporary functional unit contention. Temporary contention can occur during certain clock cycles even if the number of functional units servicing an instruction class exceeds the average number of instructions of that class issued per cycle.

For example, consider an example processor architecture and workload in which the instruction issue window contains 2 instructions, 1 functional unit exists to serve a particular class c of instruction, and 50% of instructions belong to class c. The other instructions belong to instruction classes which have no potential for contention over functional units. The average number of instructions of class c available per cycle is 0.5×2, the proportion of instructions of class c multiplied by the issue window size. Since the number of functional units serving class c is 1, no constraint to the average rate of issue would be considered. However, this result is valid only if, in every cycle, only one instruction in the issue window belongs to class c. In reality, in some cycles, 0 or 1 instructions in the issue window would belong to class c, resulting in no contention for the functional unit; in some cycles, both instructions in the issue window would belong to class c, resulting in contention and limiting the number of instructions issued in that cycle.

This simple example illustrates a property that will be typical of many workloads and architectures, in which the potential for functional unit conflict exists even when in the average case no conflict occurs. The model can be modified to account for this possibility. To calculate the issue constraint caused by an instruction class, each instruction in the instruction window can be considered an indicator random variable with a probability p_k of belonging to instruction class c equal to the proportion of instructions in the workload's instruction mix belonging to that class. The number of instructions in the instruction window is then a binomial distributed random variable, and hence the probability of having n instructions belonging to class c in a window containing W_i instructions can be calculated as:

$$P_c = P[n_c = n] = \frac{W_i!}{n!(W_i - n)!} p_c^n (1 - p_c)^{W_i - n} \tag{4.6}$$

Given N_c functional units for class c, instruction issue is limited when the number of instructions in the issue window exceeds N_c. The probability of this occurring can be determined by summing the probabilities from (4.6) for $N_c < n \leq W_i$. Since W_i may be a non-integer value, interpolation between results found for $\lceil W_i \rceil$ and $\lfloor W_i \rfloor$ is used.

This calculation results in a non-zero probability p_c for any non-zero proportion of instructions of class c. The constrained issue for instruction class c can then be calculated as:

$$I_{constrained_c} = P_c I_{unconstrained_c} \tag{4.7}$$

As with the original model, the instruction class which possesses the smallest ratio

of constrained to unconstrained issue can be used to set the overall issue rate of the model.

The resulting constrained issue rate will be uniformly lower than the issue rate calculated by the original model. As is shown in the following section, this does not always result in an improvement in accuracy of the overall model. Possible reasons for this discrepancy are discussed in section 4.7.

4.6 Experimental Results

4.6.1 Superscalar Processors

To test the accuracy of the instruction throughput model, the model is used to predict the performance of various workloads from the MiBench testbench [15], when executed on various configurations of the SimpleScalar `sim-outorder` simulator. In addition, the results produced by the original instruction throughput method are compared to the results from the modified models presented in section 4.5. The figures of interest for all tests are the IPC predicted by the model and the IPC determined from simulation of the workload.

The instruction throughput model requires various workload information which must first be gathered by executing the workload on the SimpleScalar simulator. These simulations are performed on an idealized processor architecture, and provide workload characteristics such as instruction mix, branch predictor accuracy, average basic block size, and the issue and retire dependencies. The instruction throughput model can then be used to predict performance of the workload while varying the architectural parameters of the processor. The accuracy of the model is then checked by comparing the model results with further runs of the SimpleScalar simulator, configured to match the parameters used in the instruction throughput model.

The workloads from MiBench which are analyzed are:

- `basicmath`, from the automotive suite of testbenches
- `bitcount`, also from the automotive suite
- `dijkstra`, from the network suite
- `stringsearch`, from the office suite
- `cjpeg` and `djpeg`, from the multimedia testbench suite.

Table 4.2 lists the model properties used for modeling the tested workloads. The SimpleScalar simulator was executed with matching configuration options.

Figure 4.3 shows instructions per cycle (IPC) for these workloads as measured on the SimpleScalar `sim-outorder` simulator. Also shown are the results from three versions of the instruction throughput model:

- unmodified: the original model of Taha and Wills [31], implemented as described in section 4.2
- pipeline: the model modified as described in section 4.5.1
- contention: the model modified as described in section 4.5.2, and including the pipeline modification.

The results from each model are nearly the same, although both modifications to the original model tend to result in a small decrease in predicted IPC. Unfortunately,

Property Name		Value
hwFetchRate		4
hwRetireRate		4
windowSize		4
reorderSize		16
branchMissPenalty		3
functionalUnitMults	Integer	4
	Integer Mult/Div	1
	Floating Point	4
	Floating Point Mult/Div	1
	Memory Access	2
functionalUnitLats	Integer	1
	Integer Mult/Div	3
	Floating Point	2
	Floating Point Mult/Div	4
	Memory Access	1
functionalUnitTputs	Integer	1
	Integer Mult/Div	1
	Floating Point	1
	Floating Point Mult/Div	1
	Memory Access	1
scalarArchCorrection		0

Table 4.2: Model Properties for Modeling SimpleScalar Configuration

this serves to exacerbate a uniform underestimation already evident in the results from the original model.

For the processor configuration studied in the results shown, the major performance bottlenecks were largely due to issue window utilization and issue dependence. Furthermore, due to the small size of the default issue window (4 instructions), and the relatively low utilization of functional units, the change in prediction of functional unit contention resulted in very minor adjustments of overall IPC. Further study showed that the modification does have significant effects under conditions of higher functional unit utilization. To illustrate this, the model and simulator were applied to a processor configured with an issue window of 16 instructions, and a correspondingly larger reorder buffer. For this configuration, the predicted IPC with and without the model modifications are shown in figure 4.4, along with the IPC when simulated. This test was performed on the basicmath workload.

The results of this test show a overestimation of performance by both models, compared to the SimpleScalar simulation results. In this case, the modifications to the original instruction throughput model do in fact improve the model accuracy, albeit only slightly.

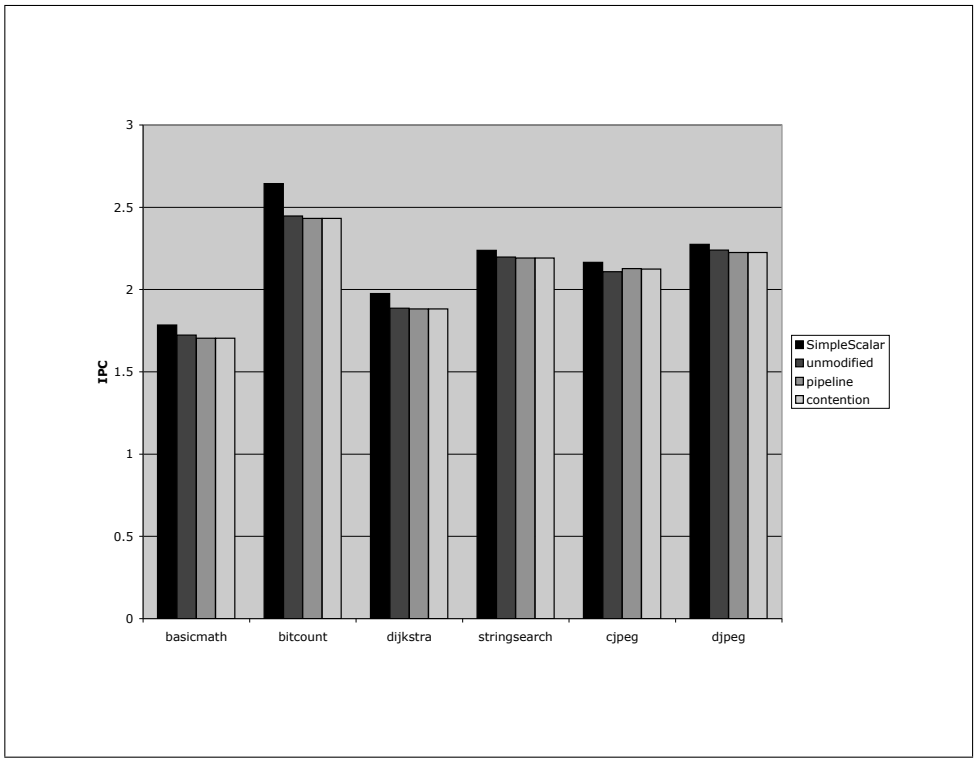


Figure 4.3: IPC Results for Superscalar Processor

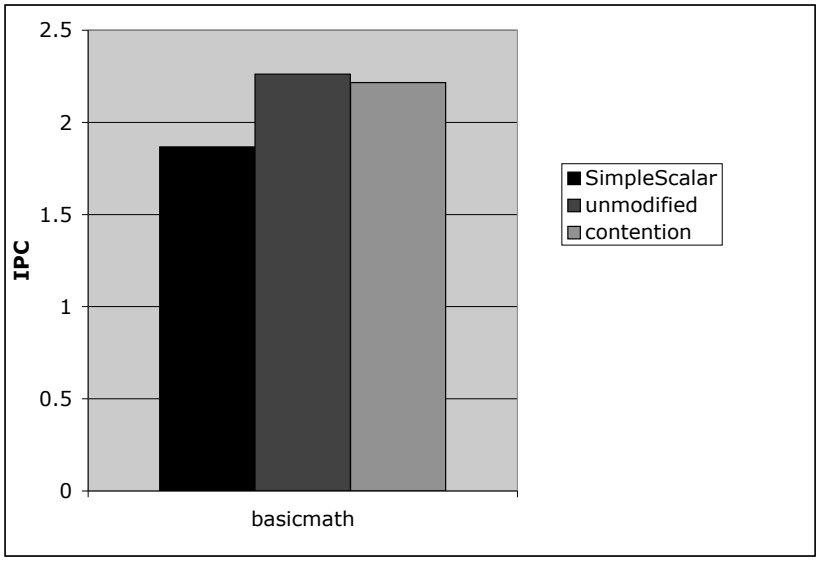


Figure 4.4: IPC Results for Superscalar Processor

4.6.2 Scalar Processors

To test the effectiveness of the model when applied to scalar processors, performance was predicted for a simple workload on an AVR core, and results were compared to the Avrora simulator. The AVR core [6] is a simple, unpipelined scalar processor core.

The Avrora [32] simulator was used to gather workload information and to serve as a baseline for comparison with model results. Due to issues involved in porting benchmark code to the AVR platform, only one benchmark program was tested. Furthermore, the simplicity of the AVR design makes its operation much more deterministic and predictable than a superscalar processor.

From initial simulation of the benchmark, which performs prime number testings, the workload parameters shown in table 4.3 were gathered. The processor model parameters for the AVR are summarized in table 4.4. As discussed in section 4.3, the instruction throughput model's issue dependence property is used to model the issue limiting effects caused by long-latency instructions. The average proportion of instructions issuing per cycle is $p_{issue} = 0.9429$. The issue dependence is then determined as described in section 4.3. Since the AVR is an unpipelined processor, the scalar architecture correction factor must be set to 3, to account for the implicit fetch, issue and retire pipeline stages assumed in the model. The AVR requires 2 cycles to complete a taken branch versus 1 cycle to complete a not taken branch. This is effectively a predict-not-taken branch prediction scheme with a 1 cycle branch misprediction penalty.

Workload Parameter	**Value**
Magnitude (Instruction Count)	8695
Basic Block Size	7.3290
Branch Predictor Miss Rate (Taken Branches)	0.8160
Issue Proportion	0.9429
Retire Dependence	1

Table 4.3: AVR Workload Parameters

Model Parameter	**Value**
Hardware Fetch Rate	1
Hardware Retire Rate	1
Issue Window Size	1
Reorder Buffer Size	100 (effectively ∞)
Branch Misprediction Penalty	1
Scalar Architecture Correction	2

Table 4.4: AVR Component Model Parameters

Given these model inputs, the predicted and actual IPC figures for the benchmark are shown in figure 4.5. As is evident, and should be expected, the model prediction is extremely close to the result of simulation.

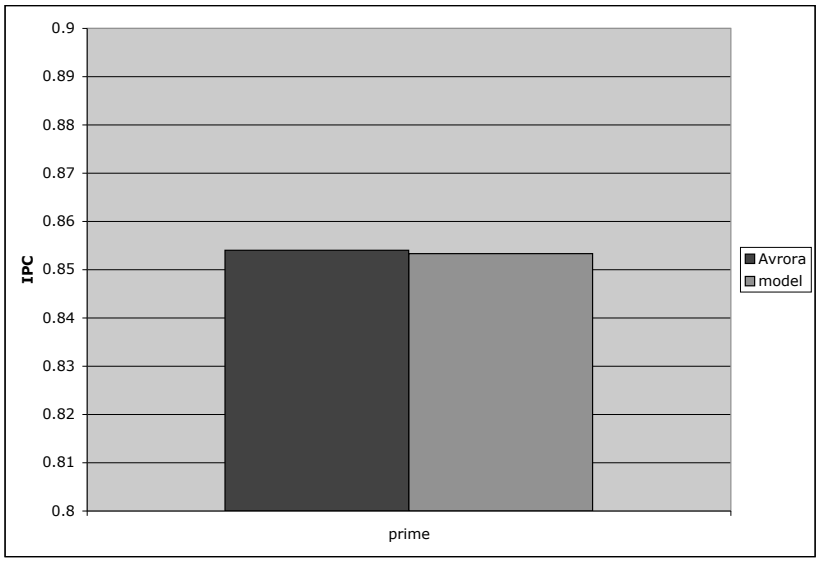


Figure 4.5: IPC Results for AVR Scalar Processor

4.7 Discussion

The instruction throughput model provides a generally accurate means of predicting performance for workloads executing on superscalar processors, and, as was shown in the previous section, can be modified to produce highly accurate estimates of performance for scalar processors. This indicates that the model can be applied usefully to a wide set of processor designs.

There may be, however, opportunities for improving the accuracy of the model. Section 4.5 presented two intuitively correct modifications which attempt to adjust some conceptual inaccuracies of the original model. However, the results in section 4.6 fail to show an overall improvement in the model's accuracy.

It is possible that inaccuracies introduced by other aspects of the model overpower the correction applied by these modifications. If, for certain processor configurations, the original model consistently underestimates performance, then the two presented modifications will in fact worsen the model's accuracy, even if the downward adjustments they apply are conceptually correct. Further study of the model and detailed analysis of superscalar processor simulations may yield insights into such possible behavior.

Alternatively, aspects of the behavior of real processors may invalidate some of the assumptions made in section 4.5. While this seems unlikely for the pipeline length adjustment, it is quite possible that the assumption of uniform probability distribution of instruction classes in the workload is incorrect for some workloads. Instructions of certain classes may be clustered together, leading to higher functional unit contention,

or may be evenly spaced due to loop constructs, possibly leading to lower functional unit contention. It seems unlikely that instructions would be distributed exactly according to the instruction mix at every clock cycle, as the original model assumes, but this assumption could lead to more accurate estimates of functional unit contention than those used in section 4.5. Further study of the behavior of actual workloads may be necessary to determine the best approach.

Cache and Branch Predictors

Another potential avenue for improvement of the instruction throughput model is to attempt to integrate analytical estimation of cache and branch predictor behavior. For an example which illustrates the potential benefit of integrating the prediction of cache and branch predictor miss rates with the processor core model, consider a small design space in which design parameters are independently modified as shown in table 4.5.

Processor Parameter	**Option 1**	**Option 2**
Fetch Bandwidth	2 inst/cycle	4 inst/cycle
Issue Window Size	4 insts	8 insts
Reorder Buffer Size	16 insts	32 insts
Integer ALU Latency	1 cycle	2 cycles
Integer ALUs	1	2
FPU Latency	2 cycles	3 cycles
Instruction Commit Bandwidth	2 inst/cycle	4 inst/cycle

Cache Parameter	**Option 1**	**Option 2**
Cache Size	32 kB	64 kB
Cache Associativity	direct-mapped	4-way set associative
Cache Block Size	16 B	32 B

Branch Predictor Parameter	**Option 1**	**Option 2**
Branch Predictor Size	256 entries	1024 entries
Branch Predictor Associativity	direct-mapped	4-way set associative
Branch Predictor History Bits	1 bit	2 bits

Table 4.5: Processor, Cache, and Branch Predictor Design Options

This design space contains 16384 distinct designs. If one ignores potential effects of branch predictor behavior on cache usage, the design space requires 8 full simulations to gather model inputs giving branch predictor miss rates for the different branch predictor designs, and 8 simulations to determine the effect of cache miss rate on the memory instruction latency. Again, ignoring possible confounding behavior between cache and branch predictor design, the same 8 simulations could be used to gather the different cache and branch predictor performance figures. Given the reported 40000× speedup for the instruction throughput model compared to cycle-accurate simulation, this means that the time required to gather model inputs from full simulation is almost 20× that required to use the model to predict performance for all 16384 designs. If the cache miss rate and branch predictor miss rates could be predicted by integrated analytical models,

only one full simulation would be required to gather model inputs. In this case, the full simulation time needed would be only approximately 2.5× that required for predicting the performance of all designs.

If the processor design is relatively more variable than the described example, the time ratio of simulation time to model time would decrease correspondingly, and likewise if the cache or branch predictor design were more variable, the time ratio would increase. Thus integration of cache and branch predictor analysis would be most beneficial for design spaces where the range of cache and branch predictor parameters is high compared to the range of variable processor parameters. In the case of many commercially available processor cores, this is in fact the case.

Another possible method to avoid the requirement of running multiple simulations to gather model inputs is to reverse the process. Instead of choosing cache parameters, determining cache performance through simulation, and then supplying these performance figures as model inputs, one could simply execute the SPAM model with a variety of cache miss rates and branch predictor misprediction rates. By doing so, one may first be able to select classes of designs which potentially provide good performance, and then use the performance results predicted over a range of cache behavior to determine the importance of the cache in determining overall design performance. If, for a class of designs, cache performance was found to be largely irrelevant, it may be sufficient to perform ad hoc design space exploration of a few cache configurations. On the other hand, if cache miss rate was found to be a crucial determinant of a platform's performance for a class of designs, then detailed analysis of cache configurations through trace simulation could be performed.

Processor Model Performance Improvement

The instruction throughput model [31] is a much faster method of performance estimation than cycle-accurate simulation. However, there may be potential for further speed improvements. The method performs an iterative analysis of instruction flow through the various processor components. This iterative analysis is bounded by mispredicted branch instructions. During execution of this macroblock, the per cycle instruction flow can be observed to proceed through a few stages:

- initially, the issue window contains no valid instructions. For a short period, fetch rate will be high and issue rate will ramp up as macroblock instructions are fetched from the cache
- similarly, instruction retire will be delayed until instructions begin propagating through the pipeline stages.
- instruction fetch may then drop due to issue window occupancy and/or reorder buffer occupancy
- if the macroblock length is sufficient, the processor model will enter a steady state period of instruction flow

The potential for performance improvement comes from the observation that the processor model enters a period of steady state instruction flow. The length of this steady state period is predictable: it ends when all instructions have graduated. The state of the processor model at the end of the steady state period is also predictable. Thus, if the beginning of steady state operation can be easily identified, all iterations

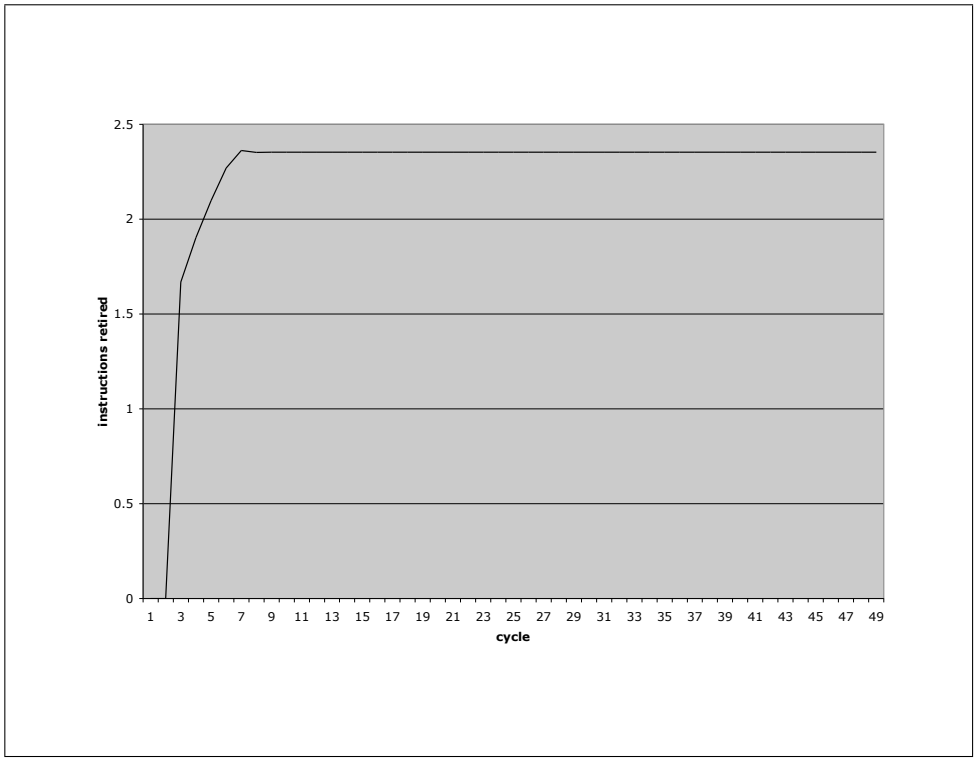


Figure 4.6: Retire Rate of Instruction Throughput Model

during this steady state period can be replaced by a single set of calculations. While these calculations would be slightly more complex than those used in a model iteration, if the steady state period is sufficiently long, the overall modeling time could be significantly reduced.

Figure 4.6 shows, as an example, the retire rate of the instruction throughput model when applied to the `cjpeg` workload. Other instruction flow rates in the model show a similar behavior. As can be seen, by cycle 11, the model enters a steady state operation, which lasts until the macroblock execution completes in cycle 49. By collapsing the execution of the steady state cycles into a single calculation, the number of iterations required by the model for this example could be reduced approximately by a factor of 4.

To perform this replacement of steady state iterations, it would be necessary to detect the beginning of steady state behavior. It may be sufficient to perform comparisons of flow rates from the current iteration with those from a few previous iterations.

Chapter 5

Shared Resource Modeling

5.1 Introduction

5.1.1 Motivation

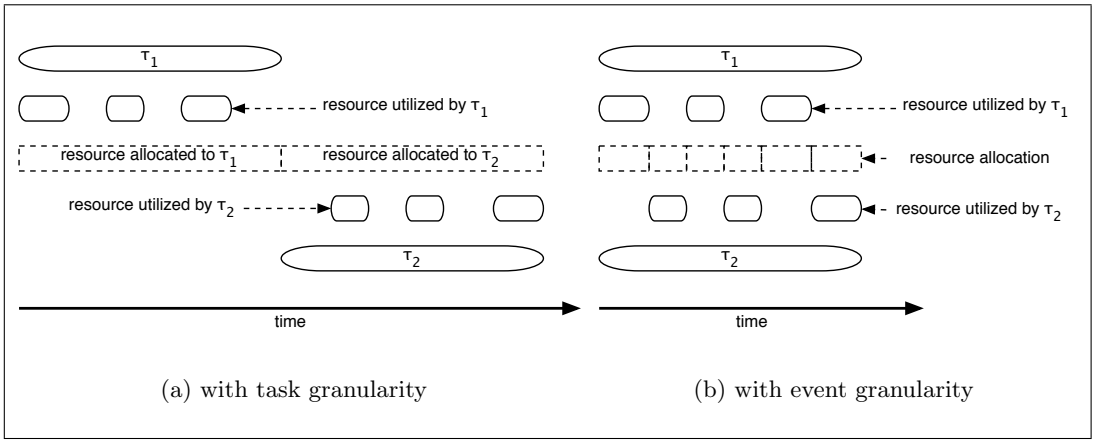
Although platform performance will often be highly correlated to the choice and design of the processor(s), other platform components can also have an important effect. In contrast to processors, whose usage is typically scheduled in embedded systems designs, usage of other components, such as buses and memories, may be either scheduled or unscheduled. With scheduled allocation of a component, exclusive usage can be assumed. However, when utilization of a component is unscheduled, it is possible for multiple accesses to be attempted simultaneously. Hence, it is important to consider the effect on performance when a component is shared by multiple activities.

The performance effects of, for instance, memory access time could be modeled in the processor model described in chapter 4 by correctly setting the latency and throughput figures of the memory instruction class' functional unit. The effects of the interconnect between processor and memory could be similarly considered by combining the communication latency with the actual delay of the memory itself. This method of modeling memory accesses would produce reasonably correct estimates if the memory accesses were serviced by a private memory and interconnect. If this condition does not hold, however, and a memory or bus is shared among multiple platform components, then some method of accounting for possible contention for the resource should be used.

One obvious example of a situation where such sharing of resources would occur would be the usage of bus interconnect and shared memory by processors in a multiprocessor platform. However, even in single processor platforms, shared resource situations may exist. A single-ported unified L2 cache, for example, may experience contention if L1 cache misses occur simultaneously for both instruction and data caches. Input devices or network interfaces may also produce unscheduled resource accesses which could be considered by a shared resource model.

5.1.2 Methods of Shared Resource Modeling

A variety of methods could be used to determine the effects of contention for shared resources on platform performance. One could perform cycle-accurate simulation of the



(a) with task granularity (b) with event granularity

Figure 5.1: Example Resource Sharing with Different Modeling Granularities

resource and all requesting components, but as previously argued, such simulation may be too slow for usage in a design space exploration method. In the ARTS model [22], resource usage is modeled by exclusively allocating resources for the duration of a task. Methods which consider communication accesses as a trace of scheduled events [19, 20] could also be used.

These methods all consider resource usage in terms of scheduled, discrete events. The methods differ in the granularity at which resource usage is considered. In cycle-accurate simulators, the granularity is at the cycle level; in event tracing models [19, 20], the granularity is at the event level; and in the ARTS model [22], the granularity is at the task level. A primary advantage of modeling with higher granularity is that it reduces the number of discrete events that must be considered, possibly by orders of magnitude. This advantage is important for early-design phase modeling or design space exploration, as large processing time requirements for executing the model could make it infeasible to model large or numerous workloads, or explore large design spaces with many potential platform designs.

The disadvantage of modeling at high granularity is that results can be highly inaccurate and misleading. Consider the example depicted in figure 5.1. In figure 5.1(a), the execution is modeled using task granularity exclusive assignment of the resource. Due to the exclusive assignment of the resource, the two tasks' executions cannot be overlapped. However, if modeled at event or cycle granularity, the tasks' executions can be correctly overlapped.

An alternative to these discrete modeling methods is to consider resource usage from a statistical perspective. Actions of a resource-using component can be modeled collectively as a statistical distribution which describes the events which are generated by the components. Statistical methods can be applied to determine effects of resource contention. This allows workload description and performance evaluation to be performed at a high granularity, while preserving much of the accuracy achieved by operating at a low granularity.

This chapter presents two methods of modeling performance effects of non-exclusive usage of resources at an activity-level granularity. The first model considers resources which maintain a FIFO ordering between customers from different sources. The con-

tention effects of these types of resources are modeled analytically using methods based on queueing systems theory. The second model considers resources which, while maintaining FIFO ordering of customers independently for each source, have arbitrary mechanisms for selecting between customers from different sources. This could include round-robin arbitration, fixed priority arbitration, or some other means of selection. Queueing networks can also be used to describe such a system, but due to the complexity of these systems, simple analytical solutions are not available. Two means for analyzing such a system, using Markov chains or using Monte Carlo simulation, are considered.

5.2 Background

5.2.1 Queueing Systems

A queueing system is a system consisting of a customer source, a single queue, and one or more customer servers. The customer source generates customers over time according to some distribution. These customers are inserted into the queue, and are removed from the queue by servers as they become available. The servers process customers, with service time determined by some process, after which the customers depart the system. Often, the queue conforms to a FIFO discipline, but this is not a necessary property of the system.

Queue systems can be categorized according to the Kendall notation, of the form A/S/n/c. In this notation the variables A, S, n and c describe:

- A: the properties of the arrival process
- S: the properties of the service process
- n: the number of servers
- c: the capacity of the system, assumed to be infinite if this variable is omitted.

A variety of arrival and service processes have been described. For the purposes of the queueing system analysis performed in this chapter, three will be of interest:

- Markovian processes, denoted in Kendall's notation by M
- deterministic processes, denoted by D
- general processes, denoted by G.

The Markovian processes are memoryless random processes. In the context of arrival processes, this corresponds to Poisson distributed arrivals. In the context of service processes, this corresponds to service with exponentially distributed service times.

A Poisson-distributed random variable takes on non-negative integer values i with a probability given by the probability density function:

$$f(i) = \frac{\lambda^i e^{-\lambda}}{i!} \tag{5.1}$$

When applied to the arrival process, this indicates that at each time step the probability of arrival is independent, and equal to λ.

An exponentially-distributed random variable takes on non-negative integer values x with a probability given by the probability density function:

$$f(x) = \mu e^{-\mu x} \tag{5.2}$$

When applied to service times for queue system customers, this indicates that the probability of service completing at each time step is independent, and equal to μ. Hence, the Poisson distribution for arrivals and exponential distribution for service times results in queue systems whose state is memoryless.

Deterministic processes are those with non-random distribution. These processes will be widely used in the following sections to consider hardware components with unchanging response times. General processes are simply arbitrary processes with unspecified parameters. In many cases, even with some unknown process properties, it is still possible to derive analytical results from queueing systems analysis.

Known analytical results and methods for queueing systems can be found in a general reference on the subject [24, 9].

5.2.2 Markov Chains

A Markov chain is a sequence of discrete random state variables in which the value of the future state is dependent only on the current state, and is independent of past states. A Markov chain forms a weighted directed graph, with vertices representing each discrete state in the statespace, and edges representing transitions between states with the edge weight indicating the probability of the transition occurring. Transitions and transition probabilities can also be specified as an $N_{states} \times N_{states}$ matrix, where N_{states} is the number of states in the statespace. The transition probabilities from each state must sum to one, including self-referencing transitions.

A Markov chain can be used to model the operation of a queuing system. State variables express properties of the queuing system, such as the current queue population. Transition probabilities are determined by properties of the arrival processes and the queuing system. Since the Markov chain is memory-less, deterministic behaviors based on past state values require introduction of additional state variables.

The statespace definition and the specification of transition probabilities can be viewed as the model inputs for a Markov chain model. From the specified transition probabilities, it is possible to calculate the stationary probabilities that the system will reside in a particular state at an arbitrary future point in time. The stationary state probabilities provide the information needed to determine performance related measures of the system behavior, including latencies, throughputs, blocking probabilities, and other figures of interest, and can also provide information useful in determining the departure process of the queuing system.

5.2.3 Notation

In the following sections, the notation $P[c]$ refers to the probability that condition c holds. For example, $P[x = y]$ is the probability that $x = y$. Shorthand notation for probability variables are introduced as needed.

Where matrices and vectors are used, matrices will be indicated using boldface type, as in $\mathbf{M}$, and vectors will be shown as $\overrightarrow{v}$.

5.3 FIFO Shared Resource Model

Some platform components may, when shared by several requesting components, service customers in a globally FIFO manner. That is, the temporal order in which requests for the resource are made is maintained when requests are serviced, regardless of which component generated which request. Such a behavior corresponds to a server, or set of servers, receiving requests in a single, unified queue, which is fed by multiple arrival processes.

Many queuing system models tend to consider service times as exponentially distributed, especially within communications research. This model of service time does not correspond to many SoC components: the time to transfer a block of memory to cache, for example, is not modeled usefully by an exponential distribution. Service times for memory or bus accesses may often be deterministic in nature. However, a simple deterministic distribution may not be adequate to fully express the properties of the resource usage. First, different types of resource access may occur: a processor may perform block reads from memory on cache miss, and also perform single word writes due to a write-through cache policy. These two types of memory accesses will require different durations for bus and memory access. Second, the resource service time requirements for a particular class of access, such as a block read, will differ depending on parameters of the architecture: the bus bandwidth, the cache block size, and so on. While the service time for a particular class of access on a particular architecture can be considered deterministic, in the more general sense the service time for a customer can be said to be deterministically calculated from properties of the customer and properties of the architecture.

The distribution of customer classes may vary depending on the request generating component. In a multiprocessor platform, for example, two processors executing different tasks may generate different amounts and types of memory or bus traffic. Similarly, the resource usage properties of customers generated by different components may vary.

Queueing system models also tend to focus on certain arrival distributions, particularly the Poisson distribution, in which the probability of an arrival occurring at any point in time is constant and independent of the state of the system. This simple distribution may be useful for modeling some platform components, such as components which respond to external events. Typical platform components such as processors, however, may show a variability in arrival rate for reasons such as unresolved data dependencies or synchronization requirements between components. For this reason it is useful to consider the relationship between arrival rate and the queueing system state. The relationship used in the FIFO shared resource model considers the effect of the population of queued customers from an arrival process on that process' arrival rate. This relationship is assumed to conform to an inverse step function, in which the arrival rate is constant below a specified cutoff point for queue population, after which the arrival rate drops to 0.

This is clearly an approximation of the reality: certain code segments may be able to issue greater numbers of memory access instructions without execution being blocked by data dependencies, for example. Further study could be made on actual workloads to determine a more realistic approximation. However, results from the memory model presented in chapter 6 indicate that, for certain workloads and configurations, this ap-

proximation may be sufficient.

For generality, it is useful to consider both single server and multiple server resources. A multiported memory, for example, could be modeled by a multiple server queueing system.

5.3.1 Model Concepts

The considerations described above lead to the following specification of a queueing system model for a FIFO shared resource:

- multiple independent arrival processes, with one arrival process corresponding to each request-generating component
- arrival rates dependent on current queue population according to an inverse step function
- multiple customer classes, with different but constant customer properties for each class
- independent distributions of customer classes generated by each arrival process
- single queue
- single or multiple servers
- customer service times deterministically based on customer and architecture properties.

Figure 5.2 illustrates this queueing system arrangement.

Despite the variability inherent in such a queueing system, the overall system complexity is still relatively low. The system consists of a single queue and one or more servers, and the major unique features are in the service time distribution, which is really a function of the class distributions of the different arrival processes; and the arrival rate, which is a modification of the general Poisson distribution.

When considering simple queuing systems, it is possible to derive analytical expressions based on the properties of the queuing system and arrival process properties. For example, for the simple M/M/1 queuing system the probability that an arriving customer will see the server occupied is $\frac{\lambda}{\mu}$, where λ is the rate of arrivals and μ is the average rate of service. However, derivation of exact analytical formulas is infeasible even for some seemingly simple queueing systems. Among these systems are those with deterministic service times. A possibility is to use Monte Carlo simulation to model the performance of the queueing system. However, it is still desirable to be able to obtain performance estimates using a quick analytical method. Furthermore, small variations from exact analytical solutions of the queueing system may be acceptable, as some error will already be introduced by model abstractions, such as the arrival distribution, customer class distribution, or service time function.

5.3.2 Analytical Results

This section presents approximate analytical formulas for key performance measures of the FIFO shared resource queueing system. These results are derived intuitively from those for more standard queuing systems. The analytical results are then compared to results obtained by Monte Carlo simulation of the system. The practicality of the model

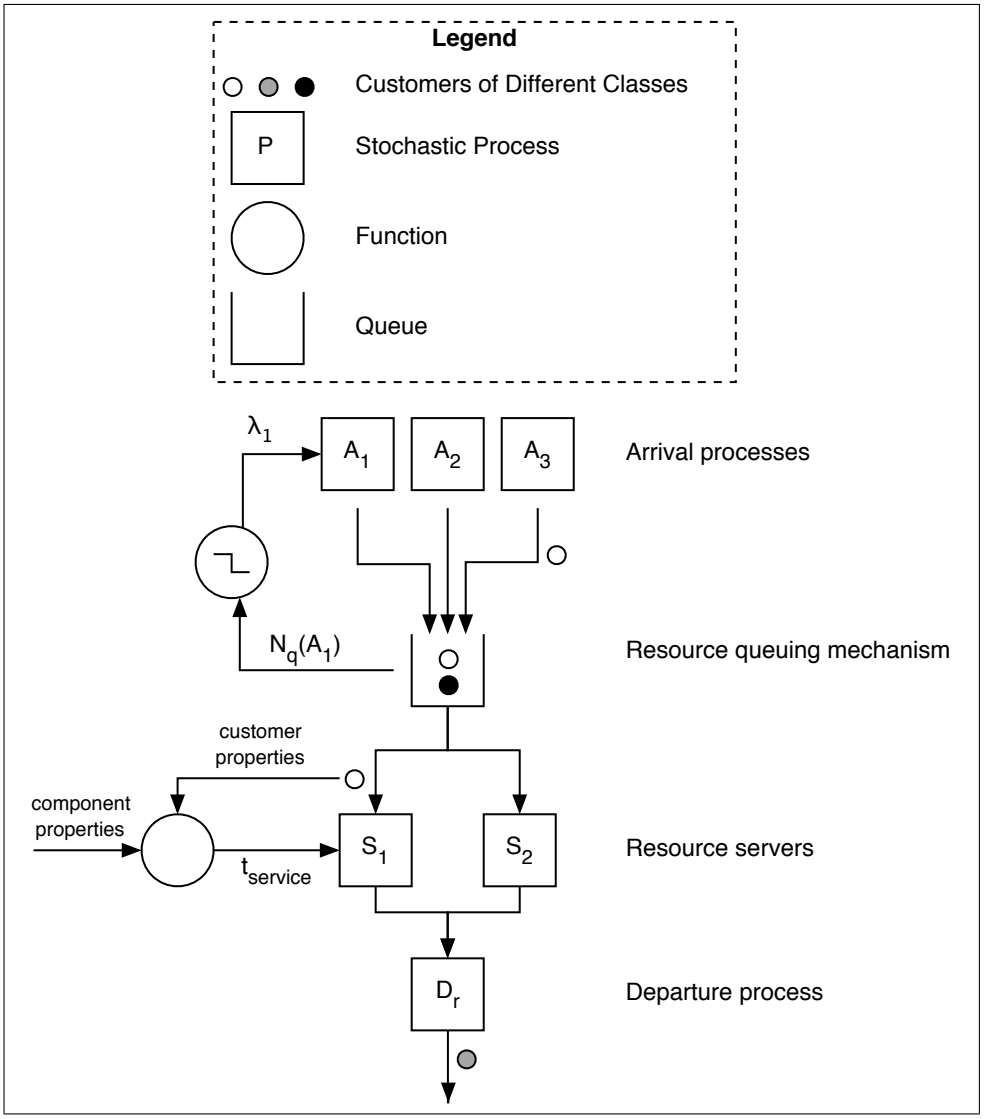


Figure 5.2: FIFO Shared Resource Model

and results are demonstrated in chapter 6, where shared memories are modeled as FIFO shared resources.

Customer Source Proportions

In the M/G/n queue system, the proportion of customers c in the system belonging to a particular arrival process j is

$$P_j = P[c = j] = \frac{\rho_j}{\rho} \tag{5.3}$$

It is possible that, since some arrival processes with different properties may experience blocking behavior, the customer proportions will be skewed. For example, if a system has long service times and thus high utilization, customers will quickly enqueue and reach the queue population cutoff points for their respective arrival processes. However, those arrival processes with higher arrival rates or lower cutoff points will more quickly reach cutoff. Subsequently, their customer proportion will begin to drop as other arrival processes continue to produce customers.

In the designs that have been considered by the model so far, these unbalancing conditions have not been generally found. Most processor-based workloads have similar constraints imposed by data dependencies, and similar proportions of memory access instructions. The expression given in (5.3) is thus used as an approximation. However, it is possible to conceive of situations where this approximation is invalid. For example, a multiprocessor with heterogeneous cache configurations could easily produce widely differing rates of memory traffic. For such cases, further consideration of the proportionality of customer sources would be needed. Alternatively, a method such as Monte Carlo simulation could be used, eliminating the need for analysis, but at the cost of possibly higher runtime.

Probability of Conflicts

The probability that a conflict between customers will occur is not a directly required performance measure, but it is useful for deriving other metrics such as average latency and throughput. Additionally, it may help provide some insight into resource usage behavior and the resulting contention.

The probability that a customer arriving at the shared resource servers will be blocked is the probability that the servers are currently occupied. This probability is commonly called the utilization, and for the general M/G/n queuing system is given by the equation, assuming symmetric servers:

$$\rho = \frac{\lambda}{n\mu} \tag{5.4}$$

where λ is the average arrival rate and μ is the average service rate. For deterministic service time t_s, this is simply $\frac{1}{t_s}$. When several arrival processes generate customers, the utilization ρ is the sum of the loads ρ_j offered by each arrival process if considered independently.

While the utilization expression in (5.4) is valid for any service time distribution, it is not generalizable over arrival distributions. Of interest is the potential effect of the modified arrival distribution on the probability of conflicts.

If an arrival process is blocked when a significant number of its customers are in the system, then its arrival rate drops towards 0 as the number of customers in the system rises. Of interest is the effect of this behavior on the conflict probability experienced by arriving customers. Arrivals which occur with a large number of customers already in the system are very likely to experience a conflict: the only way they will not experience a conflict is if there are many servers which all complete service at the time of arrival.

If the number of customers enqueued from an arrival process is sufficient to cause the arrival rate to drop to 0, then subsequent arrivals will be blocked, and will be released when the customer population drops due to completed services. The probability that these unblocked arrivals will experience a conflict is still high if they are unblocked and introduced into the system immediately after customers leave the system. In particular, an unblocked arrival will experience no conflict only if:

$$N_j - N_{d_j} < n \tag{5.5}$$

where N_j is the number of customers in the system belonging to arrival process j, N_{d_j} is the number of departing customers belonging to j, and n is as always the number of servers. For most considered resources, n is small, and so it follows that N_{d_j} is also small. Hence, it can be assumed that the probability of conflicts occurring, as expressed by ρ, is not likely to be significantly modified by blocking of arrivals.

Based on this analysis, it is reasonable to assume that the probability of conflict is not significantly modified from (5.4) due to the modified distribution of arrivals.

Average Latency

The average latency can be given by:

$$t_{lat} = t_s + t_w \tag{5.6}$$

where t_s is the average customer service time and t_w is the average wait time. The average service time is determined by the arrival processes' customer proportions given in (5.3), the customer class distributions for each arrival process, and the deterministic service time calculated for each customer class.

$$t_s = \sum_{\forall j} \sum_{\forall c} P_j \cdot P_c(j) \cdot t_s(c) \tag{5.7}$$

The average wait time is given by:

$$t_w = N_q \cdot t_s + t_r \tag{5.8}$$

where N_q is the average queue length seen by an arriving process, t_s is the average service time, and t_r is the residual service time of the customer in service.

For M/G/n queues, analytical solutions for average queue length are known. However, since the queue system for the shared resource will not have Poisson-distributed arrivals if the arrival process self-blocks, the queue length will in reality be shorter - for high utilization, considerably shorter. M/G/n queues with utilization $\rho \geq 1$, for instance, are said to be out of equilibrium and have $N_q \to \infty$.

For the FIFO shared resource queueing system, if the utilization $\rho \geq 1$, then the queue length is:

$$N_{q_{high}} \to \sum_{\forall j} N_{max_j} \tag{5.9}$$

where N_{max_j} is the cutoff point in the arrival distribution for arrival process j.

For utilization $\rho < 1$, the probability of queue saturation is low, meaning the queue system behaves approximately as a standard M/G/n queue system. Hence, the queue length can be approximated by known average queue length for the M/G/n queue:

$$N_{q_{low}} = \frac{\rho^2}{2(1-\rho)} \tag{5.10}$$

This expression approaches ∞ as $\rho \to 1$. A simple approximation can be used to give the queue length for any value of ρ:

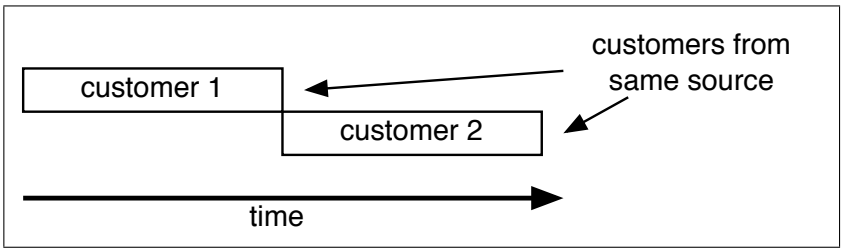
$$N_q = min(N_{q_{high}}, N_{q_{low}}) \tag{5.11}$$

Average Sustained Throughput

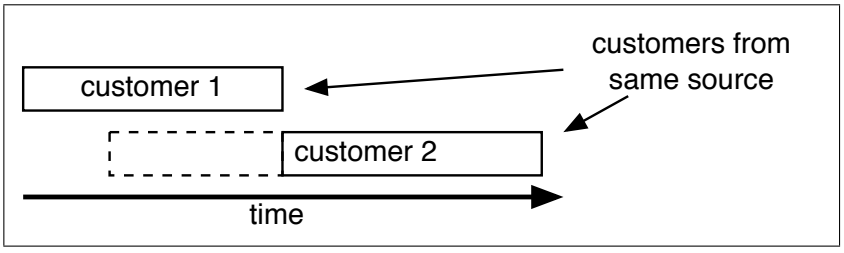
The average throughput as experienced by a particular arrival process is dependent on the service time t_s, on the probability that an arrival will be forced to wait for a customer from another arrival process, and on the average service time of the customers which must complete service to resolve the conflict. Arrivals which conflict with other customers from the same arrival process do not affect the sustained throughput as observed by that process. To confirm this property, consider the simple example shown in figure 5.3. In figure 5.3(a), two nonconflicting customers arrive in succession. In figure 5.3(b), two conflicting customers arrive from the same process. The average throughput remains unchanged. However, in figure 5.3(c), when a conflict occurs with a customer from a different arrival process, the throughput is reduced.

The probability that a conflict is with a customer not belonging to arrival process j is:

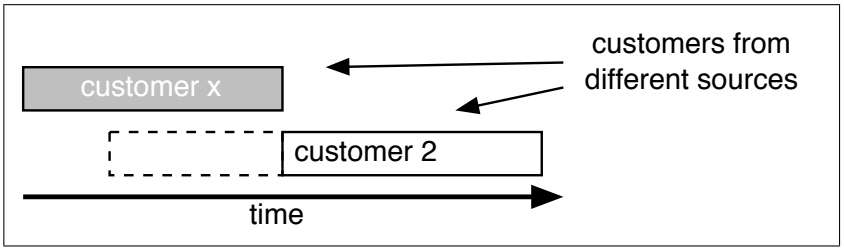
$$P_{rival} = min(\frac{\sum_{\forall i \neq j} \rho_i}{\rho}, 1) \tag{5.12}$$



(a) nonconflicting arrivals



(b) conflicting arrivals from same source



(c) conflicting arrivals from different sources

Figure 5.3: Customer Arrival Examples

Since the probability of any conflict occurring is ρ the resulting probability of conflict with a rival customer is:

$$P_{conflict} = min(\sum_{\forall i \neq j} \rho_i, \rho) \tag{5.13}$$

The residual delay is proportional to the weighted average service time of customers of processes $i \neq j$:

$$t_s = \sum_{\forall i \neq j} \sum_{\forall c} P_i \cdot P_c(i) \cdot t_s(c) \tag{5.14}$$

This reflects the fact that, of customers in the queue when an arrival enters, a proportion equal to P_i belongs to each arrival process, and each arrival process produces a distinctive mix of customer classes.

The number of rival customers in the queue is also dependent on the total utilization. If the system utilization $\rho \geq 1$, then customers will be forced to wait for a full queue, assuming blocked customers enter the system immediately when unblocked. The maximum number of complete rival customers in the queue is:

$$N_{rival_{max}} = \sum_{\forall i \neq j} N_{max_i} \tag{5.15}$$

When utilization $\rho \geq 1$, the number of customers in the queue will be approximately $N_{rival_{max}}$. There is some probability that a service departure will occur, and no corresponding arrival will occur immediately. The probability of this occurring in any cycle is:

$$p_{dep} = \left(\frac{1}{t_s}\right)^n \prod_j (1 - \lambda_j) \tag{5.16}$$

In this case, the number of rival customers in the queue will be lower. There is a decreasing probability in subsequent time steps that arrivals will continue to be absent. However, this probability can be considered small, and is thus ignored.

The maximum number of rival customers can be adjusted to take into account the possible cycles in which the queue length drops below $N_{rival_{max}}$:

$$N_{rival_{adj}} = (1 - p_{dep}) N_{rival_{max}} + p_{dep}(N_{rival_{max}} - 1) \tag{5.17}$$

When utilization $\rho < 1$, the number of customers in the queue

$$N_{rival_\rho} = (1 - P_j) \frac{\rho^2}{2(1 - \rho)} \tag{5.18}$$

However, this expression approaches ∞ as $\rho \to 1$. As a simple approximation, set:

$$N_{rival} = min(N_{rival_{adj}}, N_{rival_{\rho}}) \tag{5.19}$$

The total delay contributed by rival customers is $N_{rival} \times t_{s_{weighted}}$. The average sustained throughput for arrival process j is:

$$T_j = \frac{1}{t_{j_{avg}}} \tag{5.20}$$

where:

$$t_{j_{avg}} = t_{s_j} + P_{conflict} N_{rival} t_s \tag{5.21}$$

Departure Process

A potentially useful result of the queuing system analysis is a specification of a departure process which describes the departure rate and the distribution of classes of departures. For example, in a multiprocessor-bus-memory architecture, processors produce customers for the bus. In competing for the bus, conflicts are resolved and a stream of non-conflicting customers are passed to the memory. In such a system, the arrival process of the memory would be determined by the departure process of the bus.

Such a departure process would, for standard M/G/n queue systems, have an average departure rate given by :

$$\lambda_d = \sum_{\forall j} \lambda_j \tag{5.22}$$

If arrivals are dependent on queue population, then the effective arrival rate $\lambda_{j_{eff}} \leq \lambda_j$. In particular, during periods in which the queue population exceeds N_{max_j}, the arrival rate is limited to $1/t_{j_{avg}}$, since arrivals can only occur in tandem with departures. If $\rho \geq 1$, the probability $P[N_j = N_{max_j}] \to 1$. Otherwise, the probability $P_{j_{full}} = P[N_j = N_{max_j}]$.

The solution for $P[N_j = N_{max_j}]$ is known for M/D/1 queue systems. It is:

$$P[N_j = N_{max_j}] = (1-\rho) \sum_{k=0}^{N_{max_j}} e^{k\rho} (-1)^{N_{max_j}-k} \frac{(k\rho + N_{max_j} - k)(k\rho)^{N_{maxj}-k-1}}{(N_{max_j} - k)!} \tag{5.23}$$

The departure rate due to arrival process j is:

$$\lambda_{d_j} = (1 - P_{j_{full}})\lambda_j + P_{j_{full}} \frac{1}{t_{j_{avg}}} \tag{5.24}$$

The net departure rate λ_d is the sum of the departure rates λ_{d_j} due to each arrival process j:

$$\lambda_d = \sum_{\forall j} \lambda_{d_j} \tag{5.25}$$

The proportion of departures which result from arrival process j is given by:

$$P_{d_j} = \frac{\lambda_{d_j}}{\lambda_d} \tag{5.26}$$

Again, it should be taken into consideration that the proportionality of traffic from different sources relies on an approximation, and may not be valid when arrival processes have widely differing arrival rates or cutoff points.

5.3.3 Implementation

The FIFO shared resource model is implemented as a child class, `SharedResourceModel`, derived from the `ComponentModel` base class. The model's `estimate` method is implemented using the expression for average throughput (5.20) to determine activities' execution rates. The model also provides this value as a supplementary result. The remaining performance figures are not currently used, except where needed to calculate the average throughput.

The FIFO shared resource model is a general model which serves as a basis for modeling specific types of shared platform components. The expressions used for calculating performance require values for customer service time, which are dependent on customer properties and architecture properties. The method for calculating customer service time is unique for each potential component type, and must therefore be implemented in child classes defined for those components. The `SharedResourceModel` base class for these models uses the component-specific service times in its common `estimate` method, and therefore the method for calculating customer service time is declared as an abstract virtual function.

The resource usage patterns generated by components are modeled as one or more activities executing on the resource component, with one activity for each independent arrival process. The characteristics of the independent arrival process corresponding to each request-generating component are specified as properties of the corresponding activity. The required properties can be divided into two groups: those which are required by the base class estimate method, and those which are required by child class implementations for calculating the component-specific customer service times. The activity properties required by the FIFO shared resource base class are those that describe the basic attributes of the arrival process, such as the customer arrival rate.

The activity properties may be specified as part of the input workload XML file, or they may be calculated at runtime and passed to the shared resource activity via the property dependency mechanism described in chapter 3.

The FIFO shared resource component also requires certain base class properties describing the properties of the queueing system, plus component-specific properties

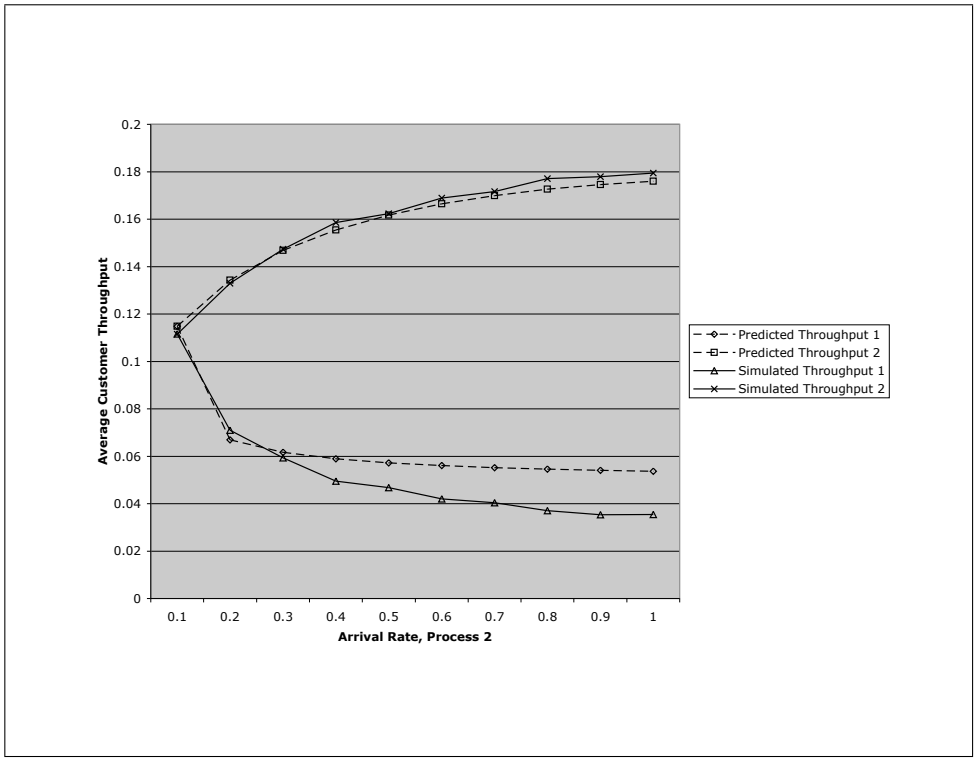


Figure 5.4: Average Throughput under Varying Arrival Rates

required for calculating architectural effects on customer service times. The values of these properties are collectively specified in the input architecture XML file.

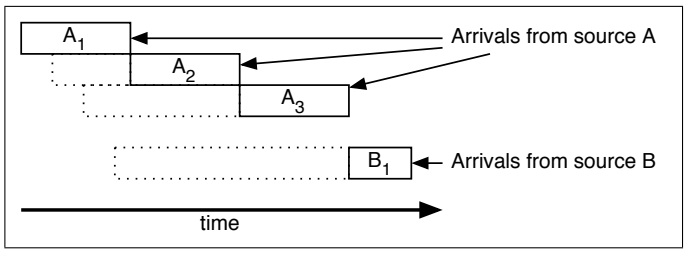
5.3.4 Experimental Results

To test the effectiveness of the analytical expressions for performance measures given in section 5.3.2, predicted results from the formulae were compared to results gathered from Monte Carlo simulation of the queueing system. For these tests, a Monte Carlo simulator was implemented, similar to that described in section 5.4.3 but considering the FIFO shared resource queueing system shown in figure 5.2.

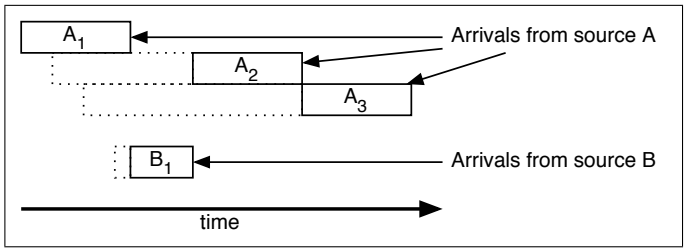
Figure 5.4 compares the predicted average throughput for customers from two different arrival processes, as one arrival process has its arrival rate varied. Each arrival process is configured to produce customers with a service time of 5 cycles, and each process has a cutoff point of 6 in-flight customers. The results show a relatively close correspondence between predicted and actual throughput, with the worst accuracy for arrival processes which have a low proportion of total utilization.

5.4 Arbitrated Shared Resource Model

While some platform components may adhere to a global FIFO queueing discipline, other resources may apply an arbitration mechanism to grant access to customers. For these components, the basic queueing system model described above may not be applicable.



(a) serviced in FIFO order



(b) serviced with priority assignments

Figure 5.5: Arbitrated Resource Example

For example, in a fixed priority arbitration scheme, a high-priority customer-generating component with a low arrival rate may experience little contention competing against a low-priority component with a high arrival rate. In the analytical model for FIFO shared resources, the effects of the priority assignment would not be considered, and as a result the model would predict a low average throughput for the high-priority component.

For example, consider the series of customer arrivals pictured in figure 5.4. In figure 5.5(a), the customers are serviced in FIFO order. In figure 5.5(b), customers from source B are given higher priority, and are serviced first. Due to the difference in arrival frequency, the FIFO ordering causes significant throughput constraint for source B. Clearly, using the FIFO shared resource model to consider this system with priority-based arbitration will result in highly inaccurate estimation of the performance of B.

A queueing system model which does consider various arbitration mechanisms and source-dependent queueing can be developed. The basic concepts of such a queueing system model are discussed in section 5.4.1. Unfortunately, due to the complexity of such a system, deriving analytical expressions for calculating performance results of the model becomes difficult. Two avenues for producing performance results from the arbitrated shared resource queueing system are explored. In section 5.4.2, analysis of the system using Markov chains is attempted. Section 5.4.3 describes the use of a Monte Carlo simulator to obtain approximate performance results for the queueing system model.

5.4.1 Model Concepts

The queuing system model of an arbitrated shared resource combines several aspects of resource usage. The first is explicit or implicit queuing of resource requests within individual customer-generating components. Additionally, the queuing model considers contention for the resource server(s). It is possible, as well, for the resource itself to perform explicit queuing of customers. Alternatively, the resource may be exposed directly to requests from components, i.e. with a queue length of 0.

The set of components accessing a shared resource are characterized by their arrival process, their local queue length and queuing discipline, and their customer classes and customer property distributions. The shared resource itself is characterized by its queue length, queuing discipline, number of servers and server properties. The service time of a customer is determined based on a resource-specific calculation involving customer properties and server properties. An example of the queuing model of a shared resource system is shown in figure 5.6.

5.4.2 Markov Chain Analysis

A Markov chain model can be created which describes the operation of the arbitrated shared resource queuing system. For simplicity of analysis, this model considers systems with no resource queueing. The state space of the Markov chain has dimensions:

- number of customers $0 \leq N_i \leq N_{i_{max}}$ in local queue i.
- class $c_j \in C$ of customer in resource server j.
- accrued service time $0 \leq t_j \leq t_s(c_j)$ of customer with class c_j in resource server j, where $t_s(c_j)$ is the deterministic service time of a customer of class c_j.

Since local queues are assumed to block arrivals when full, additional states are required to indicate that the queue is full and an arrival is blocked. Blocked customers always enter the queue immediately when space becomes available. These states are denoted as having queue population $N^*_{i_{max}}$. A special customer class c_{empty} can be used to denote states in which a server is empty.

Having defined the Markov chain state space, the transition probabilities can then be determined. A large number of states are transition incompatible and thus transitions between them have a probability of 0. Informally, this includes

- transitions that change the customer class c_j, other than those transitions which change the customer class to or from c_{empty}.
- transitions that change the customer class c_j to c_{empty} where the source state of the transition does not have $t_j = t_s(c_j)$.
- transitions that do not increase the accrued service time t_j by 1.
- transitions that increase the number of customers in a local queue by more than 1.

Additionally, certain states are unreachable, namely those in which any local queue population $N_i > 0$ and any server has customer class c_{empty}.

The probabilities for the remaining transitions are defined by the arrival rates of the different arrival processes, and their distributions of customer classes. Note that the distribution of customer classes departing a local queue is identical to the distribution of customer classes generated by the associated arrival process.

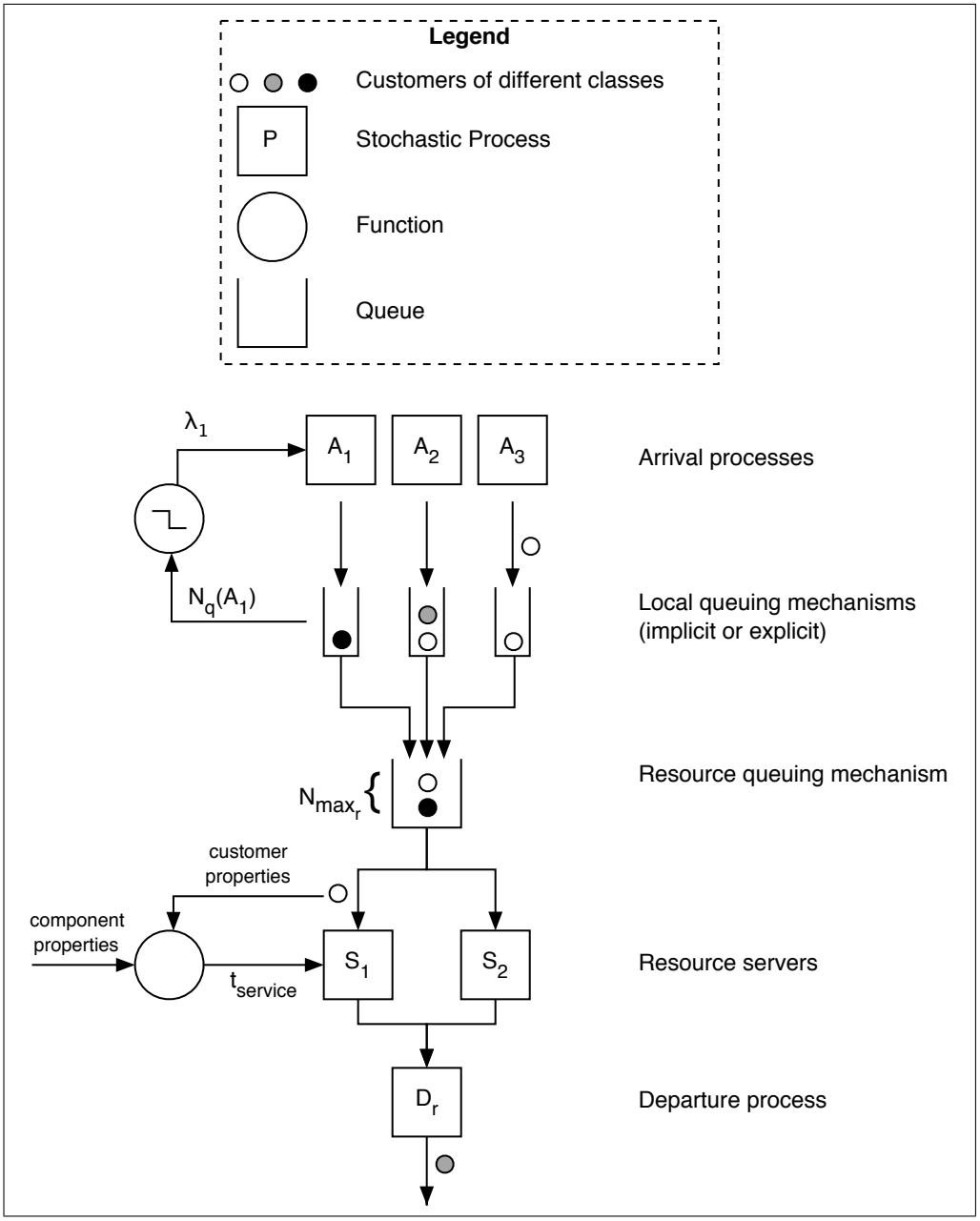


Figure 5.6: Shared Resource Model

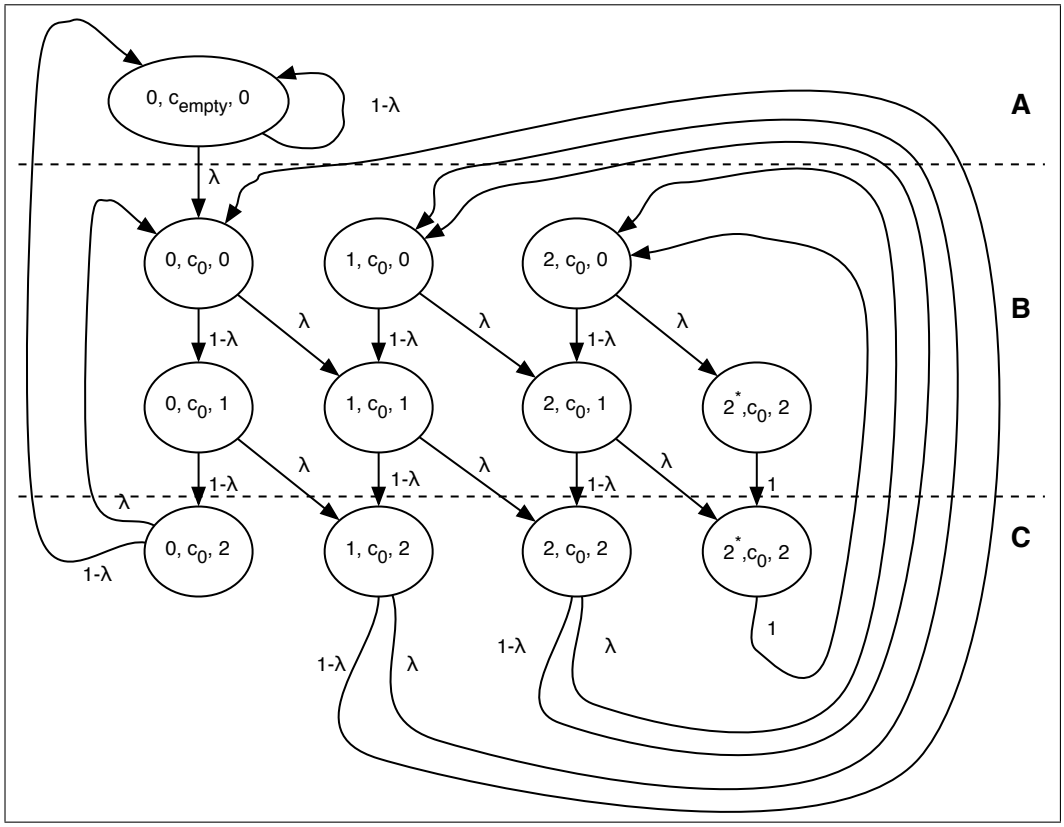


Figure 5.7: Markov Chain Model for Simple Shared Resource

For an example illustrating a Markov chain state space and transition probabilities, consider the Markov chain for a system with a single arrival process with arrival rate λ, an associated local queue with length $N_{0_{max}} = 2$, a single customer class c_0 with service time $t_{s_0} = 2$, and a single server. The Markov chain for this system is depicted in figure 5.7.

The simple single server Markov chain can be divided into three regions defined by the status of the customer in service. If there is no customer in service, the system is in the initial state labeled A. While a customer is in service, the system is in the set of states labeled B. Finally, when the customer reaches completion, the system is in the set of states labeled C. Within regions B and C, the Markov chain has been graphically depicted with the accrued service time of the customer increasing in the vertical direction, and the queue population increasing in the horizontal direction.

Transition from region A to region B occurs with probability λ. Transitions within region B always progress in the vertical direction, and progress one step in the horizontal direction (increase the queue population) with probability λ, or have no effect on queue population with probability $1 - \lambda$. In the special case where a customer arrives when the queue is full, a transition occurs to the special states with queue population 2^*, in which no further arrivals can occur. Finally, in region C, the probabilities for arrival determine whether the queue population decreases (probability $1 - \lambda$) or remains constant (probability λ). In the special case where the queue population is zero, the

system returns to the empty state with probability $1 - \lambda$. In the special case where an arrival is blocked, the arrival is unblocked and immediately enters the system, meaning the queue population always stays constant.

This simple Markov chain model must be extended in several dimensions to fully describe the general arbitrated shared resource model. It is obvious that increasing the deterministic service time or the queue length would simply extend the depicted model in the horizontal and vertical directions, respectively. Introducing additional customer classes introduces a third dimension to the depicted model, with parallel independent realizations of regions B and C having identical horizontal dimensions (local queue length) but possibly differing vertical dimensions (service times). No direct transitions would exist between these parallel sub-chains within region B. The empty state (region A) and the customer completion states (region C) would transition to the initial states of region B of the sub-chain associated with customer class c_j with probability $\lambda p_c(j,$ where $p_c(j)$ is the probability of arrival process j generating a customer of class c.

The presence of multiple arrival processes introduces additional dimensions to the Markov chain model. The transition probabilities for modifying the state variables related to the queue populations due to independent arrival processes feeding independent queues can be derived mechanically.

The probability that a certain subset of arrivals occurs simultaneously is:

$$P_{I_a} = \prod_{\forall i \in I_a} \lambda_i \prod_{\forall i \notin I_a} (1 - \lambda_i) \tag{5.27}$$

where I_a is the subset of arrival processes which generate simultaneous arrivals.

The final extension to the depicted Markov chain model is the introduction of multiple servers. This introduces additional dimensions to the model indicating the class and accrued service time of the customer in each server. The Markov chain is then divided into orthogonal intersecting regions A_j, B_j, and C_j indicating the empty state, customer in progress states, and customer completion states for server j. The system empty state $A^{'}$ indicates all servers empty. The transition rules within the state space defined by the state variable for one server remain the same as defined above. The transition probabilities within the overall state space can be calculated by forming the product of the independent probabilities defined within the subspaces defined for each server, in a manner similar to that shown for the independent arrival processes modifying the independent state variables encoding queue population.

From the Markov chain model it is possible to determine exact analytical expressions for the stationary probabilities for each state in the system. Based on the stationary probabilties, one can determine the average queue length, blocking probability, and other important properties of the shared resource model. The derived performance results are presented after the methods for determining stationary probabilities are described.

Stationary Probabilities

It is possible to derive expressions for the stationary probabilities for the states defined in the Markov chain model using the balance and normalizing equations:

$$\overrightarrow{\pi} = \overrightarrow{\pi}\mathbf{P} \tag{5.28}$$

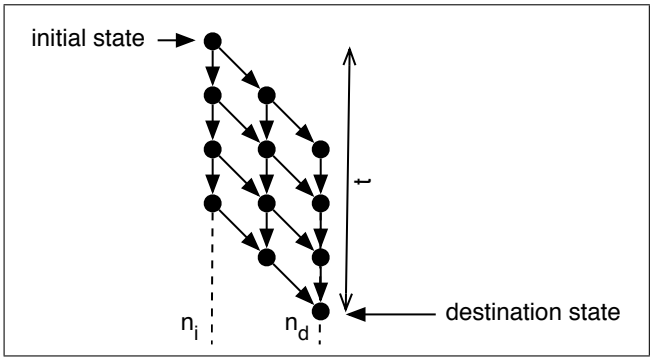


Figure 5.8: State Progress

$$|\overrightarrow{\pi}| = 1 \tag{5.29}$$

where $\overrightarrow{\pi}$ is the vector of stationary probabilities and $\mathbf{P}$ is the transition probability matrix.

Starting with a simplified model, similar to that depicted in figure 5.7 but with arbitrary local queue size N_{max} and arbitrary service time t_s, some generalized expressions can be calculated. Define $\pi_{n,t}$ as the stationary probability for the state with queue population n, and accrued service time t. Define $\pi_{\emptyset}$ as the stationary probability of the state in which the combined population of the server and queue is 0.

From the transition probability matrix $\mathbf{P}$ and equation 5.28 , the stationary probability for the empty state satisfies:

$$\pi_{\emptyset} = \frac{1-\lambda}{\lambda}\pi_{0,t_s} \tag{5.30}$$

The stationary probability for states representing the initiation of a new customer service satisfy:

$$\pi_{0,1} = \pi_{0,t_s} + (1-\lambda)\pi_{1,t_s} \tag{5.31}$$

$$\pi_{N_{max},1} = \lambda\pi_{N_{max},t_s} + \pi_{N^*_{max},t_s} \tag{5.32}$$

$$\pi_{n,1} = \lambda\pi_{n,t_s} + (1-\lambda)\pi_{n+1,t_s} \tag{5.33}$$

The stationary probability for the states representing a customer in progress, including the final state before completion, satisfy the general expression:

$$\pi_{n,t} = \lambda\pi_{n-1,t-1} + (1-\lambda)\pi_{n,t-1} \tag{5.34}$$

One can derive expressions for $\pi_{n,t}$ in terms of the initiation state stationary probabilities $\pi_{n,1}$. Consider the graph shown in figure 5.8. This graph, representing the temporal and queue length progression of the queue system, is a 2-dimensional grid, and transitions occur in only one direction in each dimension. This means that there are a certain subset of valid paths from any initial state $\pi_{n_i,1}$ to any destination state

$\pi_{n_d,t}$. Each of these paths consists of transitions with probability λ or $1-\lambda$, corresponding to transitions in the diagonal or horizontal respectively. Due to the limitation on transition direction, it follows that any path from an initial state $\pi_{n_i,1}$ to a destination state $\pi_{n_d,t}$ must consist of $n_d - n_i$ diagonal steps and $t - n_d + n_i - 1$ horizontal steps. Thus, each path from $\pi_{n_i,1}$ to $\pi_{n_d,t}$ is equally likely, and occurs with a probability:

$$\lambda^{n_d-n_i}(1-\lambda)^{t-n_d+n_i-1} \tag{5.35}$$

The stationary probability for a given state $\pi_{n,t}$ can be determined by considering all paths from all initial states $\pi_{n_i,1}$ which have valid transition paths to $\pi_{n,t}$. The set of valid initial states is limited to include those with $n_i \leq n$, due to the limitation on transition direction. Additionally, the set of valid initial states is limited by $n_i + t \geq n$, as each time increment can correspond to a single increment of queue length.

All paths from each initial state to the considered destination state must be enumerated. The enumeration of always-increasing paths in a 2-dimensional grid is a known problem. The number of paths between two grid points separated by x and y steps in the horizontal and vertical dimension, respectively, is given by:

$$p = \frac{(x+y)!}{x!y!} \tag{5.36}$$

Combining equations (5.35) and (5.36), and summing over the initial states $n - t < n_i < n$, yields the general expression for stationary probability $\pi_{n,t}$:

$$\pi_{n,t} = \sum_{n_i=max(0,n-N_{max})}^{n} \pi_{n,1}\lambda^{n-n_i}(1-\lambda)^{t-n+n_i-1}\frac{t!}{(n-n_i)!(t-n+n_i-1)!} \tag{5.37}$$

For the special states indicating blocked arrival:

$$\pi_{n^*,t} = \sum_{t_i=1}^{t} \lambda\pi_{n,t} \tag{5.38}$$

Substituting into equation 5.30:

$$\pi_{\emptyset} = \frac{(1-\lambda)^{t+1}}{\lambda}\pi_{0,1} \tag{5.39}$$

From equations 5.31 through 5.33 one can write:

$$\pi_{0,1} = a_{0,0}\pi_{0,1} + a_{0,1}\pi_{1,1} \tag{5.40}$$

$$\pi_{n,1} = a_{n,0}\pi_{0,1} + ... + a_{n,n+1}\pi_{n,1} \tag{5.41}$$

$$\pi_{N_{max},1} = a_{N_{max},0}\pi_{0,1} + ... + a_{N_{max},N_{max}}\pi_{N_{max},1} \tag{5.42}$$

where coefficients $a_{n,m}$ are the sum of all coefficients on $\pi_{m,1}$ when determining $\pi_{n,1}$. These coefficients can be determined by multiplying the appropriate terms from equation 5.37 by 1, λ or $(1-\lambda)$ as determined by equations 5.31 through 5.33 . The resulting system of linear equations for $\pi_{n,1}$ is in upper Hessenberg form, and can thus be solved mechanically. A matrix, representing the coefficients of a system of linear equations, is in upper Hessenberg form if all coefficients lying further than 1 location below the main diagonal have value 0.

From the solutions for $\pi_{n,1}$, the stationary probabilities for all states in the Markov chain can be determined using (5.37). Unfortunately, solution of the system of linear equations does not yield a simple formulaic expression for stationary probabilities, but automatic determination of the stationary probabilities for any set of parameters t_s, N_{max} and λ can be implemented.

Extending the Markov Chain Model

As discussed previously, various potential shared resource systems may require variations to the simple model discussed above. The variations considered include multiple customer classes, multiple arrival processes and multiple servers.

The existence of multiple customer classes creates parallel execution planes (regions B and C) which can interact only through the empty state $c_\emptyset$ or through customer completions (region C). The result is that multiple sets of linear equations must be solved, with the coefficient factors (1, λ and $(1-\lambda)$) relating π_{m,t_s} to $\pi_{n,1}$ each multiplied by the customer class probability p_{c_j} from the distribution of customer classes. Additionally, the independent planes may consider differing service times.

The introduction of multiple arrival processes and associated local queues introduces additional orthogonal dimensions. During each time step, the system may increment any combination of local queue population state variables. The probability of incrementing any queue population state variable is independent. Since the temporal progression of the Markov chain is independent of the queue population progression, and the queue population progressions are orthogonal, it is possible to consider the queue population progressions for each local queue as an independent parallel execution plane, in a manner similar to that described for multiple customer classes. In this case, stationary probabilities for an independent plane should be calculated for each combination of arrival process and differing customer class service time. The coefficient factors for completions again can be modified: the probability that the next served customer is of class c_j, and thus the coefficient factor for arrival in the associated execution plane, is the average probability that a customer is of class c_j, weighted by the probability that the associated local queue length is n.

Similarly, the introduction of multiple servers means that several orthogonal state variables and associated execution planes exist, one for each server/customer class combination. Completion transitions may occur between execution planes representing that the new customer is of a different class.

Performance Metrics

The goal of the shared resource model is to provide performance metrics such as average latency, average sustainable throughput, blocking probability, probability of immediate service, and to provide statistical characteristics for departure processes. These figures can be determined using stationary probabilities for selected sets of states.

For example:

- Average throughput for an arrival process can be determined by summing the stationary probabilities of all region C states corresponding to that arrival process, as each of these states indicates a customer completion.
- Average queue length can be found by taking the weighted average of the stationary probabilities of customer initiation states at the beginning of region B, with the queue length n of the state applied as weighting.
- Average latency can be determined based on average queue length and service times.
- Departure process specification can be determined based on stationary probabilities of states in region C, and customer class distributions.

Discussion

Increasing the number of dimensions in a Markov chain can dramatically increase the size of the state space, and the size of the resulting set of linear equations that must be solved to determine stationary probabilities. Ideally, one would want to minimize the state space which must be considered to arrive at the desired performance measures. For example, the generalized expressions for $\pi_{n,1}$ and π_{n,t_s} make it possible to compute the throughput without considering the stationary probabilities of the intermediate states $\pi_{n,t}$.

Another method of reducing the state space which must be considered is to defer the selection of the arrival process and customer class of the customer in service for each server. Consider a system with two arrival processes which each generate different distributions of two customer classes. The two customer classes have service times t_{s_1} and t_{s_2}. One can construct a Markov chain for this system in which the customer class in service is not defined in the state space. During the temporal progression for a customer in this Markov chain, a completion probability can be calculated at each step which is based on the probabilities that the arrival was from a particular generating component, and the distribution of customer classes and their associated service times generated by that component.

However, state space expansion remains a problem even when collapsing some dimensions and replacing them with modified transition probabilities. For example, solution of a system which tracked all customer classes within a single plane such as that shown in figure 5.7 would still require additional dimensions to independently track the state of each arrival process and each server. The addition of dimensions to consider extra arrival processes is especially problematic, as it means that the Markov chain state space grows exponentially as arrival-generating components are added to a platform. Thus, considering highly parallel multiprocessors, systems with coprocessors, or systems with a variety of other traffic generating components could become costly.

5.4.3 Monte Carlo Simulation

An alternative method of obtaining results for a complex queueing system is to run a simulation of the system using probabilistically generated input events. Such a simulator is known as a Monte Carlo simulator. Markov chain analysis has, in theory and with suitable heuristics, the possibility of obtaining analytical expressions for performance figures based on model parameters. However, usage of Monte Carlo simulation has several key advantages. A Monte Carlo simulation is relatively simple to implement compared to the numerical analysis algorithms which may be necessary to analyze a Markov chain. Additionally, use of a Monte Carlo simulator allows for introduction of different arrival distributions, service time distributions, arbitration mechanisms and queueing disciplines, without the need for reanalysis of a complex system. Finally, without simplification, the Markov chain's statespace can easily grow too large for efficient analysis.

5.4.4 Implementation

Due to the current advantages it provides, the modeling of arbitrated shared resources is performed using Monte Carlo simulation. A component model subclass, `MCResourceModel`, is derived from the `ComponentModel` base class, as for other component models described previously. The `estimate` method of this model must run a Monte Carlo simulation to obtain the execution rates for activities executing on the shared component.

The requirements of this method of performance estimation raise certain implementation issues. Since activity execution already occurs within the context of the overall SPAM simulation, the `MCResourceModel` must run its own sub-simulation whenever its estimate method is called. Although this Monte Carlo simulation considers discrete events that occur over several component clock cycles, its goal is to produce a long term average execution rate for an activity. Therefore, it is desired that the entire multicycle Monte Carlo sub-simulation runs within a single SPAM simulator clock cycle.

To accomplish this requires certain adaptations of the SystemC simulation environment. SystemC does not directly support nesting of simulation contexts. That is, after a simulation is started by calling the `sc_start` method, it is impossible to setup and start another simulation. Furthermore, it is impossible to construct new SystemC modules in the current simulation context while the simulation is running.

The SystemC simulation facilities, however, can provide useful functionality. Therefore, the `MCResourceModel` is structurally composed from a pair of static SystemC modules:

- `MCArrivalGenerator`, which provides the ability to model multiple independent arrival processes
- `MCServer`, which provides the ability to model one or more resource servers, and which uses the abstract virtual method `getServiceTime` to determine the component-specific service times for customers.

The `MCArrivalGenerator`, which is instantiated when the global SPAM simulation begins, must be capable of being reinitialized flexibly whenever the `MCResourceModel`'s `estimate` method is called, to reflect changing properties of resource usage. The initialization is handled by the estimate method, which creates `MCArrivalProcess` objects,

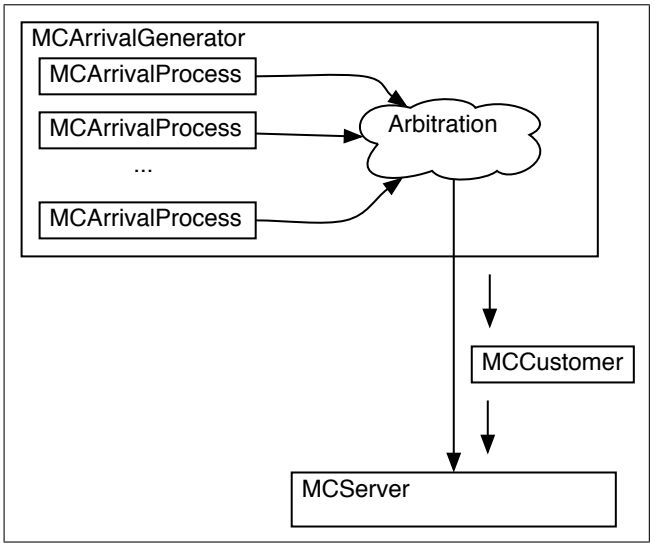


Figure 5.9: Monte Carlo Simulator Implementation

each with the arrival process properties for one activity. These objects are passed to the `MCArrivalGenerator` instance through a SystemC master-slave link, before Monte Carlo simulation begins.

The Monte Carlo simulation is executed, within a single clock cycle of the global SPAM simulation, by applying a for loop iteration to create an effective clock signal. Since no delay in the SystemC simulation is associated with each loop iteration, all iterations can be completed within a simulator cycle. This effective clock causes execution of slave processes in the `MCArrivalGenerator` and `MCServer`, which together perform cycle-level Monte Carlo simulation of the shared resource system.

During execution of the Monte Carlo simulation, the `MCArrivalGenerator` uses random number generation, and the probabilities defined by the `MCArrivalProcess` objects, to determine if customers are generated in each clock cycle, and what customer class they belong to. The customer's properties are used to construct an `MCCustomer` object, which keeps track of the customer's class, its execution status, and its history in the system.

The `MCServer` module tracks the state of the resource queue, and the occupancy of each resource server. Whenever a resource server is not occupied, the `MCServer` requests the next customer by accessing a master input link connected to the `MCArrivalGenerator`. This causes the `MCArrivalGenerator` to execute an arbitration function, which selects one `MCCustomer` available in one of the local customer queues and passes is back to the `MCServer` on the master-slave link.

The organization of the Monte Carlo simulator is summarized in figure 5.9.

Customers travel through four phases during their lifetime in the system:

- waiting in local queue
- waiting in resource queue
- in service
- completed

When an `MCCustomer` is generated, its arrival time is noted, and it is immediately

inserted into the local queue associated with its `MCArrivalProcess`, where it will wait until it is selected to proceed to the resource. When the `MCServer` requests a customer, the `MCCustomer` may be selected by the arbitration process of the `MCArrivalGenerator`, if it is at the head of its local queue. If the `MCServer` has a non-zero length resource queue, the `MCCustomer` may now wait in the resource queue until a server becomes available. In many cases, however, a resource will have a zero-length queue, with all queueing behavior occurring in the generating components. In these cases, `MCCustomer`s sent to the `MCServer` will enter service immediately. When an `MCCustomer` enters service, through the resource queue or otherwise, the start of service time is noted, and the customer and resource properties are used to determine service time. The service time is also stored in the `MCCustomer`, and decremented at each clock tick until it reaches 0. At this point, the `MCCustomer` completes service. Its completion time is noted.

The `MCCustomer` class provides static methods which maintain statistics for customers. These statistics are listed in table 5.1. The statistics are measured separately as seen by each arrival process. At customer arrival, the `arrivedCount` is incremented. All other statistics counters are updated when a customer completes, based on the customer's stored arrival, service and completion times.

Property Name	**Description**
`arrivedCount`	number of customers arriving during simulation
`completedCount`	number of customers completing service during simulation
`averageLatency`	average latency from arrival to completion for completed customers
`averageService`	average latency from service start to completion for completed customers
`averageQueueOccupancy`	average number of customers in queue as seen by arriving customers
`averageServerOccupancy`	average number of customers in servers as seen by arriving customers
`utilization`	proportion of arriving customers who see no available servers on arrival

Table 5.1: Monte Carlo Simulator Customer Statistics

The average throughput figure for arrival processes are used by the `MCResourceModel`'s `estimate` method to determine each activity's execution rate. The other statistical measures may be useful as supplementary results. Currently, the latency figure for each activity is provided as a supplementary result. Additionally, the gathered statistical information can be useful for gaining insight into the system's performance.

5.5 Summary

This chapter introduced two new component models for estimating performance of shared resources. The models serve as a basis for modeling of specific shared resource components. The first model, presented in section 5.3, models FIFO-conforming resources using analytical formulae derived from standard queueing system models. The second model, presented in section 5.4, models general arbitrated shared resources using Monte Carlo simulation. Each model has been implemented as a class derived from `ComponentModel`, as shown in figure 5.10. These classes are abstract, and provide a base class which can be extended to derive models of specific shared resource components. Two such examples of shared components, shared memories and buses, are described in the following chapters.

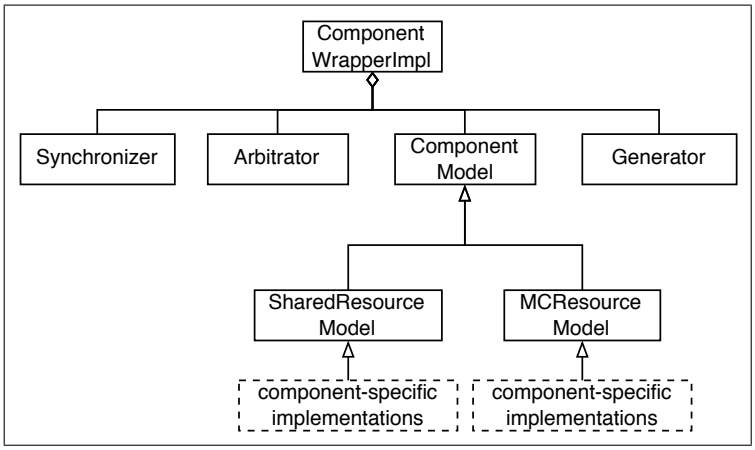


Figure 5.10: Class Diagram with Shared Resource Models

Chapter 6

Memory Modeling

With the increasing performance gap between processors and memory, an important factor in determining the rate of execution for processor-based platform workloads is the performance of its memory hierarchy. The design of a memory hierarchy is in general quite flexible. Multiple levels of caches can be used, each with potential variability in size, associativity, block size, and other parameters. Caches can be divided into separate instruction and data caches, or combined into a unified cache. Private instruction or data memories can be directly connected to processors. Custom memory hierarchies can be created, with portions of an address space mapped to small, high-speed memories and other address regions mapped to slower memories. Shared memories may be used for flexibility and communication in multiprocessor platforms.

Some research has been performed on quantifiably predicting performance benefits of these methods. Certain areas of research, such as analytical prediction of cache miss rate, have proven difficult. Sophisticated prediction methods such as locality surfaces [29] are promising, but combine both high computational expense and often very low accuracy. Often, trace simulation is resorted to as a means of determining performance of different cache designs.

Promising research has been performed in design space exploration of full cache and memory hierarchies [1]. It could be useful for future work to be performed integrating these methods into a SPAM component model. In this project, however, the focus is on the performance estimation of single standalone or shared memories.

6.1 Model Concepts

When modeling a memory, it is important to consider the sharing behavior to which it is being subjected. A memory could be subjected to sharing by being used by multiple processors simultaneously. However, if the memory is accessed only through a shared bus, then any contention for the memory will occur when processors attempt to initiate a bus transfer. The bus effectively resolves contention and orders memory accesses before they are serviced by the memory. In this case, the requirement of the memory model is simply to provide to the bus an estimate of the memory's response time.

Alternatively, a unified instruction and data memory can be seen as a memory being shared between the front-end and back-end of a single processor. If this memory is

directly connected to the processor, by interconnect that is not shared by other platform components, then it is possible to model the memory delay and contention together. Instruction and data memory accesses from a processor's instruction stream can be said to conform approximately to a FIFO queueing discipline. The FIFO queueing behavior occurs due to the inherent instruction order of the workload. Some minor changes in ordering may occur in an out-of-order superscalar processor, but these can be considered to be relatively minor deviations from a FIFO ordering. Therefore, a unified memory may be modeled as a FIFO shared resource, as described in section 5.3.

Arrival rates and other properties of the concurrent arrival processes can be determined by one-time simulation of the workload to gather information on the memory access patterns. The arrival rates used for the memory model may correspond to the frequency of memory instructions for activities modeling data memory access, or they may be based on cache miss rates for unified memories separated from the processor by a cache. In either case, the frequency of access also may vary with processor performance, and therefore may be provided to the memory model through a property dependency.

Customer service times for the memory shared resource model are determined by a simple analytical expression, which captures a few key memory parameters. This method may be seen as a placeholder for a more complete method of analyzing memory designs. However, it gives some degree of flexibility in modeling memories, and more importantly, provides a basis for studying the effects of contention for memories.

$$[ServiceTime] = [TransferSetup] + \frac{[TransferSize]}{[PortSize][TransferRate]} \tag{6.1}$$

6.2 Implementation

A component model class MemoryModel is derived from the SharedResourceModel base class for FIFO shared resources. The base class provides the implementation for the estimate method, which determines contention effects on performance when several activities share the memory. Arrival process parameters, as discussed previously, are provided either as fixed activity properties based on one-time workload simulation, or through property dependencies from the request-generating components. The architectural properties of the memory are provided as properties of the component in the input architecture XML file. Customer service times are provided by the MemoryModel child class, and are calculated according to (6.1).

6.3 Experimental Results

The effects of memory performance and contention is studied using the SimpleScalar simulator toolset. The sim-cache simulator provides a number of customization options for the SimpleScalar processor's cache hierarchy. To provide a simple shared memory scenario for which arrival process and resource properties can be modified easily, a large fully associative unified L2 cache can be instantiated; if sufficiently large, the cache will behave effectively as main memory. SimpleScalar then allows modification of the L2 cache latency and the cache block size. In this scenario, the L2 cache memory is

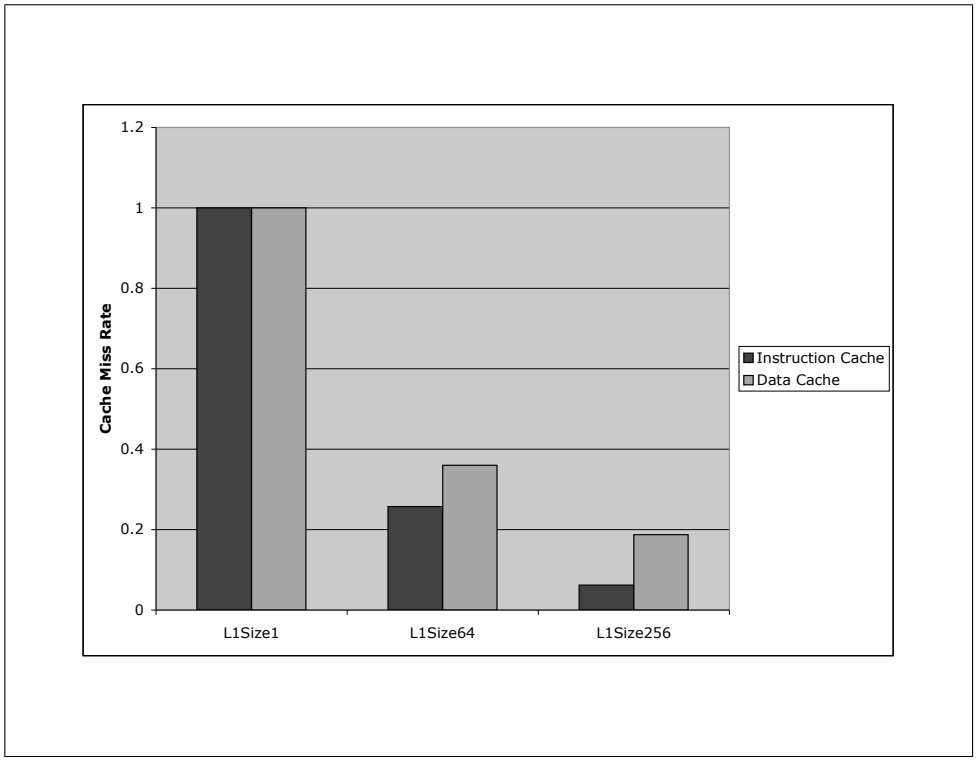


Figure 6.1: L1 Cache Miss Rates

shared between two activities, one representing instruction L1 cache misses and the other representing data L1 cache misses. By modifying the L1 cache sizes, it is possible to vary independently the arrival rates of these two activities. This allows predictions of the memory model to be tested in situations with a range of contention.

The `cjpeg` workload was executed with several cache configurations. This workload has a memory instruction proportion of 0.2574. Figure 6.1 shows the L1 cache miss rates for the workload with three cache sizes: with 1 block (essentially no cache), 64 blocks and 256 blocks in each L1 cache. The cache block size is 1 processor word in each case. The cache miss rates shown give a range of behaviors. In the extreme case where L1 cache size is minimal, contention should occur whenever a memory access instruction is executed. For the 256 block cache, the miss rates for both caches is small, and contention should be a relatively infrequent occurrence.

Figure 6.2 shows the resulting IPC predicted by the platform model with processor and memory. In each case, the model result is within 10% of the actual simulated performance. Results for additional workloads and cache configurations are similar. The largest difference occurs when the contention rate should be low. It is possible that the shared resource model is overestimating the rate of contention. It is also possible, however, that the inaccuracy stems from problems in applying instruction memory throughput to the processor model. Further study of the model operation in these conditions is necessary to determine what, if any, modifications should be made to improve accuracy.

Further validation of the memory model occurs in chapter 8, when the model is com-

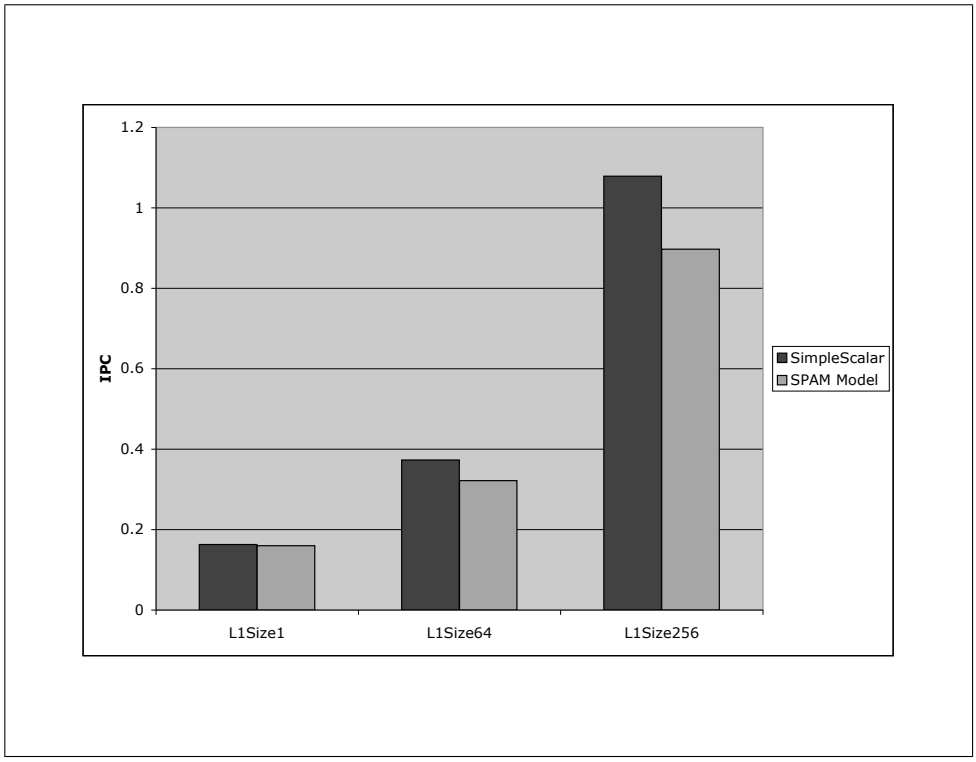


Figure 6.2: Processor with Memory IPC

bined with models for processors and buses to predict performance of a multiprocessor platform with shared memory.

Chapter 7

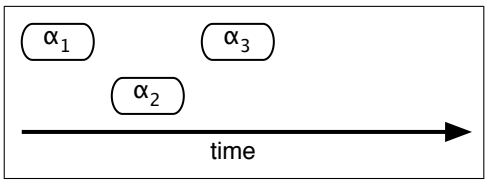
Bus Modeling

The advent of complex SoCs with large numbers of processor cores is leading to the development of novel on-chip communication architectures, including such methods as packet-switched Network-on-Chip (NoC) implementations. However, this in not likely to lead to the complete elimination of simple shared buses from the SoC design space. First, many applications will continue to require less processing power than that provided by large multiprocessor SoC. In these smaller designs, bus interconnect may be adequate, and implementing a NoC may cost more in terms of chip area and design time than any possible performance benefit warrants. Second, even in complex NoC-based systems, individual processors will likely need to be connected to local memories and other devices, and these individual platforms will then be connected to each other by the NoC. Within each platform, bus interconnect and dedicated point-to-point connections will remain.

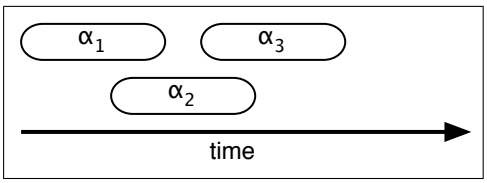
The performance of a bus-connected platform will be to some extent dependent on the properties of the bus and of the bus traffic. To quantify the performance effect, it is necessary to consider the interaction between bus traffic generated by different sources. For example, slowing the speed of the bus between a processor and memory obviously will increase the delay incurred by memory access instructions. The effect of this extra delay will be more pronounced if the cache miss rate is high, or if the processor is directly connected to memory; and the effect of slower bus transfers will be even more pronounced if other devices are putting traffic on the bus, since the increased duration of each transfer will increase the probability of contention. Bus contention will cause the observed performance of the bus as seen by each traffic source to worsen more substantially than the effect caused by increased transfer latency alone.

Figure 7.1 illustrates this effect. In figure 7.1(a), a set of activities utilizing a bus are shown executing over time. In figure 7.1(b), the same activities, each with doubled latency, are shown as they would execute if the bus could be shared without contention. Figure 7.1(c), the increase in execution time in the presence of contention is shown.

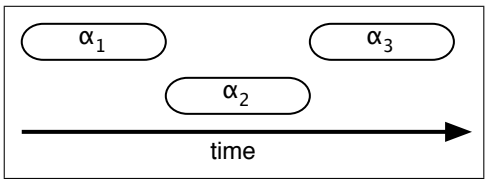
Due to the distributed nature of the sources generating bus traffic, it may not be possible, in many cases, for the bus to maintain global FIFO ordering of bus access. It may also be undesirable to adhere to a simple FIFO ordering, as certain bus masters may require high performance and could benefit from preferential access to the bus. Common bus implementations provide a variety of arbitration mechanisms for granting bus access,



(a) fast activity execution



(b) slow activity execution with no contention



(c) slow activity execution with contention

Figure 7.1: Bus Contention Examples

including round-robin, fixed priority arbitration, and various dynamic priority protocols. To model the effects of bus contention and arbitration mechanisms, a bus model can be created which draws on the concept of the arbitrated shared resource, described in section 5.4.

7.1 Bus Transfer Performance

In the arbitrated shared resource model, the service time of customers is left to the child class implementation to define. These service times are to be calculated based on properties of the resource usage and properties of the resource itself. A generalized model for calculating bus transfer time is required to implement a model of bus performance.

A technique proposed by Knudsen and Madsen [18] allows determination of the time for a single transfer on a variety of interconnect designs. The model takes into account:

- transmitting and receiving driver properties, including delays incurred by packing
- channel properties, including burst modes, synchronization cycles, and other properties.

For bus modeling, the main area of interest is the effect of the bus channel properties on transmission delays. Driver delays may be negligible, or may be considered as part of the workload. The consideration of driver delays and of driver modification of transmitted data is therefore omitted from the model in the following description.

The channel delay model [18] bases channel delay for a transfer on several parameters, listed in table 7.1

The first step in determining the transfer delay is to determine the division of the transfer into bursts. Each burst requires c_{sb} synchronization cycles, plus a number of

Parameter	Description
b_m	burst mode
f_c	channel frequency
s_b	channel burst size
c_{sb}	synchronization cycles per burst
c_{ss}	synchronization cycles per session (transfer)
n_c	number of channel words in transfer

Table 7.1: Channel Model Parameters

data transfer cycles dependent on the number of channel words in the burst. Four burst modes are described in the delay model [18]:

- b_m = **fixed** burst mode, in which each burst is of size s_b. If not enough channel words are available to fill the last burst in a transmission, the last burst will be padded to s_b
- b_m = **max** burst mode, in which bursts may be of size s_b or less
- b_m = **inf**inite burst mode, in which all channel words in the transmission are grouped into a single burst, and only one set of synchronization cycles is required
- no burst mode, in which each channel word must be sent in a separate burst, including any necessary burst synchronization cycles. This is equivalent to **fixed** or **max** burst mode with burst size $s_b = 1$.

The number of bursts required to transmit n_c channel words can be calculated:

$$n_b = \left\lceil \frac{n_c}{s_b} \right\rceil \tag{7.1}$$

These bursts are divided into $n_b - 1$ full bursts of length s_b, and a final remainder burst of length s_r. The length of s_r is dependent on the burst mode:

$$s_r = \begin{cases} n_c & b_m = \textbf{inf} \\ s_b & b_m = \textbf{fixed} \\ n_c - (n_b - 1)s_b & b_m = \textbf{max} \end{cases} \tag{7.2}$$

The number of synchronization cycles per transmission can be calculated as:

$$c_{cs} = c_{ss} + c_{bs}n_b \tag{7.3}$$

The number of actually transmitted words is:

$$n_t = (n_b - 1)s_b + s_r \tag{7.4}$$

and total delay of the channel is:

$$t_{cd} = \frac{c_{cs} + c_{ct} n_t}{f_c} \tag{7.5}$$

A consideration not included in the original model [18] is the two-phase nature of many bus transactions. For example, a memory read access may require a response from the memory slave before the bus is released. In this case, the response time of the memory slave becomes important to determining the bus transfer time. Certain bus implementations, however, may allow splitting of read transactions, in which case the entirety of the slave response time does not impact the bus transfer time. The bus model should be capable of accounting for both of these possibilities.

To account for this, the channel model described above is modified to include three additional parameters:

- transmission response mode r_m: either **no** response, **split** response, or **int**egrated response
- slave latency cycles c_{sl}
- slave lookahead cycles c_{la}
- slave device clock frequency f_{sl}.

Using these parameters, the channel transmission delay calculation can be modified to include the potential delay. Currently, **no** and **split** response modes operate identically, and require no modification to the channel transmission delay calculated in (7.5).

The no response mode corresponds to write transfers or other transfers which do not require a response.

Split response mechanisms allow the slave latency to be hidden from the bus, by releasing control of the bus and initiating a new bus transmission when the slave is ready to respond. It may be possible to model this form of operation by applying two concurrent activities to the bus model, one describing the properties of the request transmissions and one describing the properties of the response transmissions. A potential problem with this method is that the random processes defined by each activity would be modeled as non-correlated, whereas in reality request and response transmissions clearly are correlated. As the currently available simulators for producing baseline results do not perform split transfers, these effects cannot be investigated, and their study will therefore be deferred.

The integrated response mode indicates that the bus is held while the slave device prepares and transmits response data. This form of bus access and response corresponds to a read operation. To complete these forms of transfers, the bus must be held while the slave completes any operations required to produce a response. The slave latency c_{sl} indicates the number of slave device cycles required before a response can be transmitted. In some bus implementations, a slave device may be able to begin preparing a response early by observing address and control signals. To account for this possibility, a number of lookahead cycles c_{la} can be specified, which effectively reduces the latency of the slave device as seen by the bus. This results in a total bus transfer time of:

$$t_{bt} = t_{cd} + \frac{c_{sl} - c_{la}}{f_{sl}} \tag{7.6}$$

Currently, it is assumed that data transmission is unidirectional within a transfer. That is, a transfer either writes data to a slave device or reads data from it. The model could be extended relatively easily to handle bidirectional transfers over a single channel, by calculating transmission delays for each direction, but for the currently studied architectures, this form of communication is not prevalent.

7.2 Implementation

A model for performance prediction of buses is implemented in `BusModel`, a child class derived from the arbitrated shared resource model base class, `MCResourceModel`, described in section 5.4. To predict bus performance, the shared resource model determines the effects of resource contention based on properties of the arrival processes, and based on service times calculated from activity and architecture properties.

While the base class performs Monte Carlo simulation to determine the contention effects, the child class is responsible for determining the service time for customers. The service time calculation mechanism is based on the transmission delay model [18] described above.

7.3 Model Specification

To demonstrate the ability of the model to describe an actual bus implementation, the AMBA AHB bus [5] was studied. AMBA AHB is a high-speed on-chip system bus standard, developed by ARM and released as an open standard.

AMBA (Advanced Microcontroller Bus Architecture) is a specification of three different on-chip bus standards. The AMBA AHB (Advanced High-performance Bus) is defined for use in the most performance sensitive situations, such as the main backbone bus in a processor based platform. A cycle-accurate simulator of this bus is included in the MPARM multiprocessor simulator [23].

The features of the AHB which are important at the level of modeling performed here include:

- synchronization cycles
- arbitration
- burst transfer implementation
- split transactions
- pipelined transfers.

The AHB requires a one cycle address and control period. In the bus model, this corresponds to $c_{ss} = 1$. Arbitration occurs within this address cycle. The AHB standard does not define an arbitration algorithm. The choice of arbitration algorithm is left to the designer.

Several burst transfer types are defined by the AHB standard, including single data channel word transfers, fixed size bursts of 4, 8 or 16 channel words, and bursts of unspecified length. Each of these burst types can be modeled effectively by the appropriate selection of b_m and s_b, as shown in table 7.2.

The AHB standard allows for but does not require splitting of transactions. This would occur when a slave device requires additional cycles to respond to a request. Two

AMBA Burst Type	b_m	s_b
SINGLE	**fixed** or **max**	1
INCR	**inf**	-
WRAP4 or INCR4	**fixed**	4
WRAP8 or INCR8	**fixed**	8
WRAP16 or INCR16	**fixed**	16

Table 7.2: Burst Type Modeling

mechanisms are provided, RETRY and SPLIT. Both cause the master and slave to back off the bus; the SPLIT mechanism additionally causes the arbiter to adjust the priority of bus masters until the slave device indicates a response is ready. These mechanisms are not currently considered by the bus model, but as described previously, it may be possible to model them using separate activities defining request and response bus traffic.

The AHB supports pipelining of bus access. In particular, the address and control phase of one bus transfer can overlap with the data phase of another. Another way of viewing this capability is to consider the pipelined nature as allowing slave devices to begin response one cycle before their bus transfer begins. In other words, the slave devices have a lookahead capability, which can be modeled by setting $c_{la} = 1$.

The latency of slave devices is, naturally, dependent on the slave. In the SPAM methodology, the slave latency c_{sl} can be set using a property dependency.

Table 7.3 summarizes the parameterization of the bus model for the AHB bus.

Model Parameter	**Symbol**	**Value**
Burst Mode	b_m	see table 7.2
Channel Frequency	f_c	implementation dependent
Channel Burst Size	s_b	see table 7.2
Burst Synchronization Cycles	c_{sb}	1
Session Synchronization Cycles	c_{ss}	0
Transfer Size	n_c	workload dependent
Transmission Response Mode	r_m	workload dependent
Slave Latency	c_{sl}	implementation dependent
Slave Lookahead	c_{la}	1
Slave Clock Frequency	f_{sl}	implementation dependent

Table 7.3: Bus Model Parameter Values for AMBA AHB

7.4 Discussion

The arbitrated shared resource model from which BusModel is derived depends on Monte Carlo simulation of a queueing system which is intended to model the operation of the bus. There are three major potential sources for error in this method of performance estimation:

- model mapping error: if the queueing system model does not adequately represent the actions of the real bus, the results of the model will be inaccurate
- activity mapping error: similarly, if the model inputs describing the operation of the arrival processes do not match the operation of real bus masters, the results will be inaccurate. In particular, the actions of bus masters may not correspond closely to the modified Poisson distribution which is assumed
- random variations: since the results are obtained from a random process, they potentially may vary. The expected accuracy of the model will depend on the number of events simulated. Simulating more events will result in higher model accuracy, but also result in higher runtimes.

Verification of the bus model is described in chapter 8, due to difficulties in isolating buses for study. Since a bus is a communication mechanism, it requires both sources of and sinks for communication events. Bus contention depends materially upon the activities of event sources. Bus transaction times may depend on the response times of event sinks. For these reasons, the comparison of modeled results to actual results is discussed in the context of a complete platform model.

Inclusion of driver properties and driver delays currently is not included in the model, as concepts such as packet construction or message packing are not generally applicable to the high-speed system buses which likely are to be used within SoC platforms. For other applications, such as modeling interprocessor communication in large MPSoC designs, for example, such considerations may become important.

Chapter 8

Platform Modeling

The SPAM methodology described in chapter 3 allows the integration of component models performing performance estimation of various platform components into a single unified model. In chapters 4, 6 and 7, general component models for processors, memories and buses were presented. In this chapter, the usage of these component models in constructing integrated platform model is described. Two case studies are presented: a single processor platform, and a multiprocessor platform with a shared memory. Both platforms are based on ARM7 processors and an AMBA AHB system bus. The platform models' predicted results are compared against the MPARM [23, 21] cycle-accurate simulator.

8.1 Single Processor Platform

The single processor platform is simulated using MPARM, specifying one ARM7 processor core and one memory. The processor and memory are connected by the AMBA AHB bus. Figure 8.1 depicts this platform.

8.1.1 Model Construction

To construct the platform model, three component models are specified in the architecture input XML file.

A processor model component is defined to model the ARM7 processor. The ARM7 core [4] is a scalar, unpipelined processor. Since the ARM7 core as simulated in MPARM is not easily modified, its particular design parameters are not considered crucial when

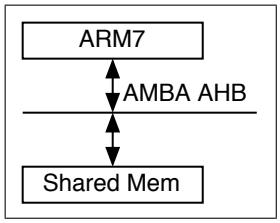


Figure 8.1: Single Processor Platform

modeling the MPARM platform. Furthermore, the processor model has been considered in detail in chapter 4. However, it is important to model the effects of the bus interconnect and memory latencies. To accomplish this, the processor is modeled with two instruction classes: memory instructions and ordinary instructions. The latency of ordinary instructions is set to 1 cycle, while the latency of memory instructions is set using a property dependency from the bus model. The remaining processor model properties are set as required to describe a scalar unpipelined processor, similar to those shown in section 4.6.2.

A bus model is defined to model the AMBA AHB bus. The bus model properties are based on the properties described in section 7.3. The particular burst mode used by the AMBA AHB bus in the simulator is **inf**. The slave latency for the bus model is determined via a property dependency from the memory model.

The memory in the MPARM-simulated system has a latency of 1 setup cycle and 1 cycle per transferred word. The size of the memory port is identical to bus word size. Since word size is not being varied in this test, a default size of 1 is used. For design space exploration involving various memory and bus configurations with different word sizes, it would be important to specify the relative difference in word size.

Activities must be defined to describe the action of the workload. The workloads used in testing were modeled with four parallel activities executing simultaneously on processor, bus and memory components. Two bus activities represent the separate action of cache misses (block reads) and cache write through (single word writes). Property dependencies exist between the memory model and the bus model, through which the latency of memory operations as determined by the memory model is transfered to the slave latency property of the bus model; and between the bus model and processor model, through which the average latency of memory requests made via the bus is transfered to the processor model's memory instruction functional unit latency. For the memory instruction latency, the cache miss penalty is incorporated with a measured cache miss rate to determine the latency of an average memory access. For this calculation, cache write through is assumed to have no effect (i.e. an ideal write buffer is assumed).

Activation and completion dependencies enforce the parallel execution of these activities. The activities for memory and bus accesses are assumed to be subordinate to the processor activity. That is, the usage of the bus would not terminate before the processor usage if the bus clock rate is increased; rather, the latency determined by the bus would change, which would in turn affect the operation of the processor. The processor would continue to issue memory requests throughout execution of its activity.

The resulting activity graph is shown in figure 8.2.

8.2 Multiprocessor Platform

The modeled multiprocessor platform is also based on the MPARM simulator. In this case, the simulator is executed with 4 ARM7 processors, a single shared memory, and a single AMBA AHB bus connecting these components. Figure 8.3 depicts this platform.

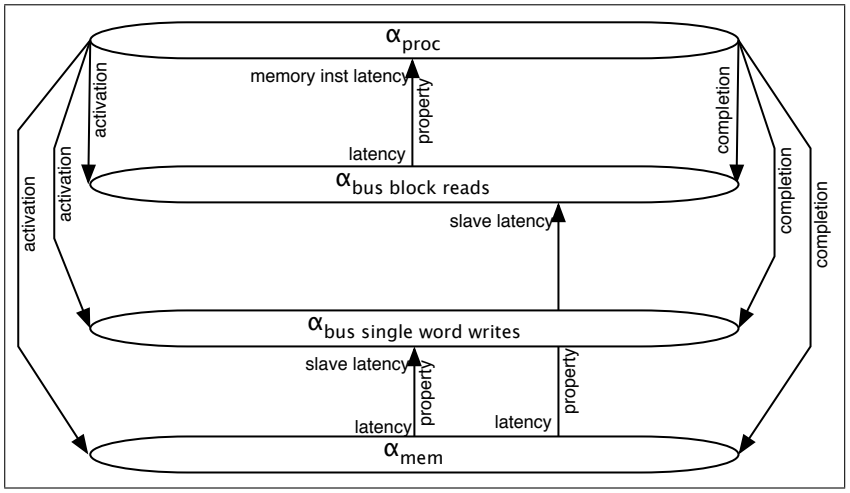


Figure 8.2: Single Processor Platform Activity Graph

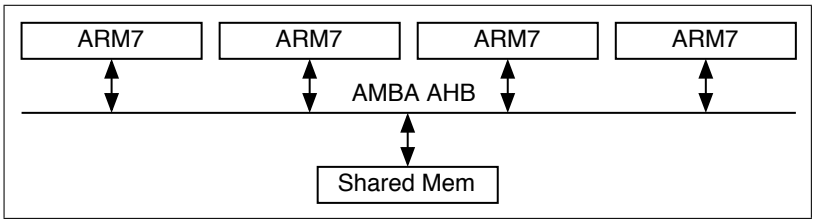


Figure 8.3: Multiprocessor Platform

8.2.1 Model Construction

The construction of the platform model is identical to that given in section 8.1.1, except four identical instances of the ARM7 processor model are instantiated instead of one. The primary difference in the model is that now four separate sets of activities are used, modeling the action of the workload executing on each of the four processors. One processor activity is allocated to each processor model; four activities representing block reads and four activities representing single word writes are assigned to the bus model; and a single activity is used to represent memory usage. The reason memory usage is still modeled by a single activity is because of the arbitrating effect of the bus interconnect between processors and memory. Any contention occurs at the bus level, while the memory sees an ordered, non-conflicting stream of memory accesses. If the AMBA AHB interconnect modeled in the MPARM simulator used transaction splitting, then it would be possible to see resource contention at the memory level.

The dependencies between activities is similar to that described in section 8.1.1. Each bus activity draws it slave latency property from the memory activity via a property dependency. Each processor activity similarly draws its memory instruction latency values from the corresponding bus activity.

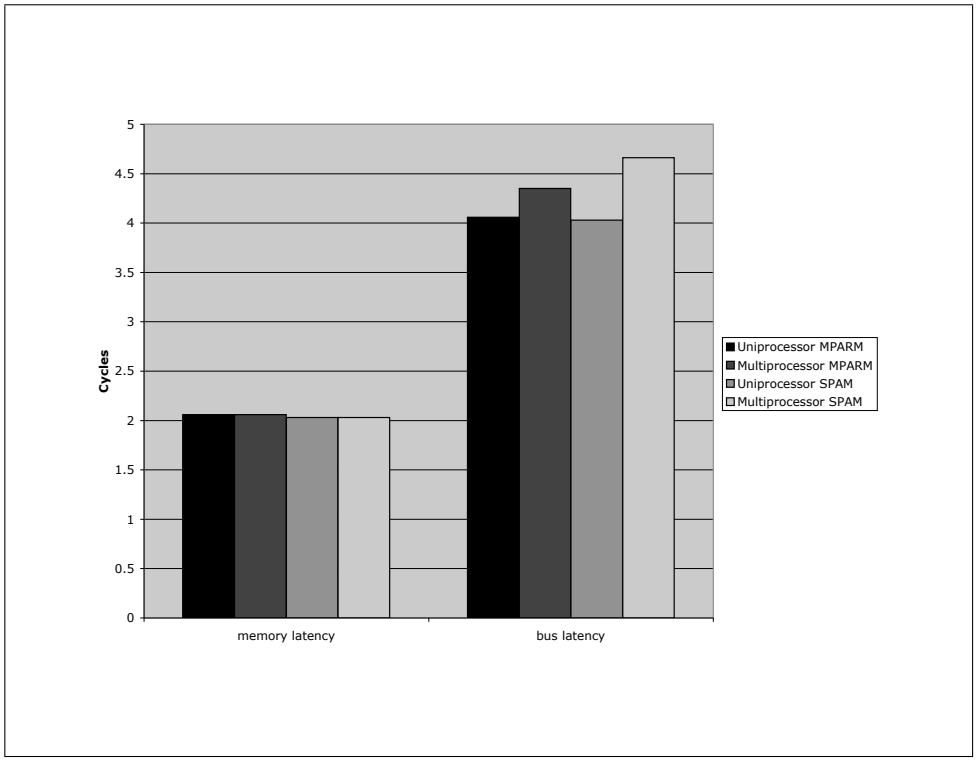


Figure 8.4: Resource Latencies for MPARM Platforms

8.3 Experimental Results

The models described in section 8.1.1 and 8.2.1 were executed, and results compared to simulation using MPARM. For these tests, the `asm-matrixind` workload was used. This workload has a relatively low usage of memory. Furthermore, the ARM processor is a scalar processor and its performance highly deterministic. Hence, the model and simulation results matched with a high degree of accuracy.

A more interesting result to study is the predicted latency of memory accesses, as seen by the processor core, and the resulting total memory overhead during execution. By comparing these results for the single processor and multiprocessor platforms, it is possible to confirm that the bus and memory models are providing accurate estimates.

Figure 8.4 shows the average latencies for memory, and for memory plus interconnect, as determined by MPARM simulation of the `asm-matrixind` workload on both 1 and 4 processor platforms. The SPAM memory model's latency prediction is also shown, along with the average bus latency predicted by the SPAM bus model. The results given for the 4 processor SPAM model are averages over the results for each processor. Since the bus model obtains estimates through Monte Carlo simulation, there is some small variation in the prediction for each processor.

Figure 8.5 shown the total number of bus busy cycles determined through MPARM simulation. The comparative figure shown for the SPAM model is the predicted total latency of bus accesses.

The results show a relatively close correspondence between actual and predicted

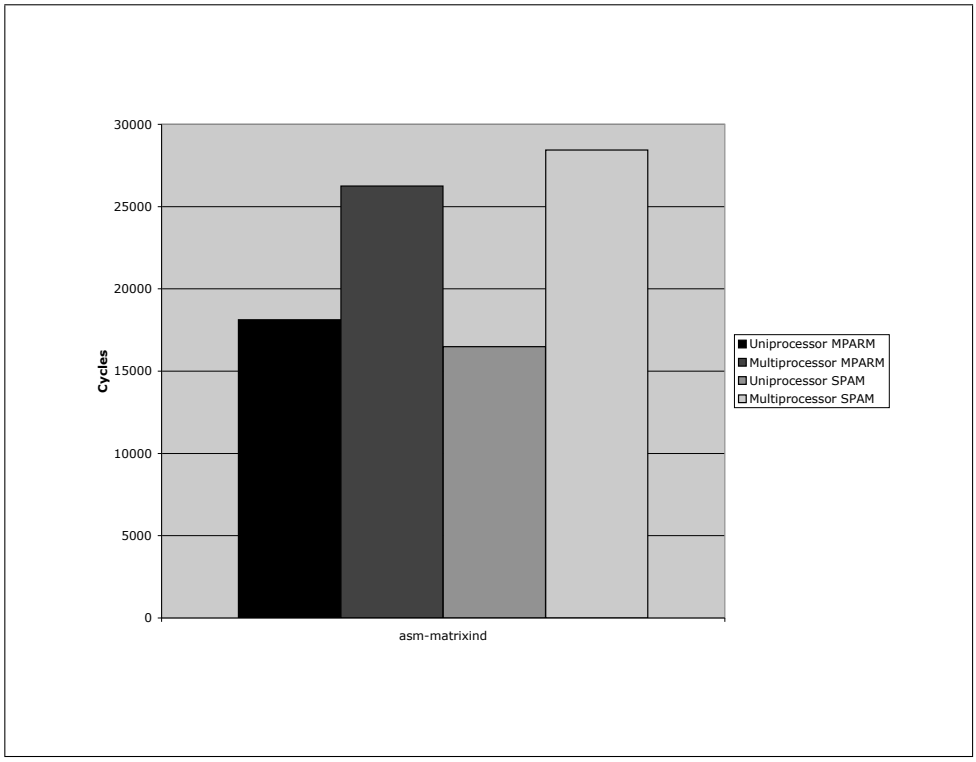


Figure 8.5: Total Cycles of Bus Usage

behavior. This provides confirmation of the applicability of the bus model described in chapter 7, along with further confirmation of the memory model presented in chapter 6.

8.4 Summary

The results presented in this chapter show that SPAM model can be applied to multiprocessor platforms, in addition to single processor platforms, and can produce accurate performance estimates. Furthermore, the results help confirm the applicability of the shared resource models, which are applied to memory and buses to predict the consequences of resource contention. These two case studies also help illustrate how the SPAM methodology can be applied to diverse platform configurations.

Chapter 9

Discussion

The previous chapters have described a method for combining diverse performance predicting models of platform components into a unified platform performance model. Results have been presented in the relevant chapters demonstrating the ability of the models to predict performance of:

- processors
- processors with memory
- single and multiprocessor platforms with shared memories accessed via a platform bus.

However, the flexibility of the SPAM methodology and its object-oriented design means that there are many ways in which the platform model could be extended, improved, or modified. Furthermore, there are features of the SPAM methodology that haven't yet been exercised in relation to a real world design problem. Further testing of the models could also be performed on a wider variety of platform designs.

9.1 Model Extensions

9.1.1 Platform Components

The applicability of the SPAM methodology could be extended in several ways. First, through the object-oriented component modeling architecture, additional models could be developed for platform components not yet considered. Among these are:

- coprocessors, such as DSPs or others
- dedicated, custom-designed hardware used to perform application-specific functionality
- communication interfaces, such as for connecting to NoCs, sensor networks or conventional networks
- I/O devices
- reconfigurable hardware.

Many of these components could be modeled as shared resources, in a manner similar to that proposed for memories in chapter 6. The latency of components such as I/O devices, or of dedicated logic performing small, frequently used computations, would

have to be calculated and provided on a component-specific basis. These latencies would then provide the service time for a shared resource model.

Reconfigurable hardware possibly could be modeled similarly. However, there would also have to be a method of expressing and modeling reconfiguration, and the delays required for runtime changes. Further study of reconfigurable hardware use in embedded systems and SoCs should be pursued to determine the best method for doing so. Similarly, proposing models for DSPs and other application-specific processors will require further research.

9.1.2 Component Model Extensions

While new component models for currently unconsidered components could be created, it could also be possible to extend the current models to consider a wider range of performance-affecting parameters, and thus to increase the flexibility of the model.

Processor Technology

Olivieri [26] described a processor power and performance model which investigated relationships between architectural parameters such as pipeline depth, and lower level details such as supply voltage scaling and process parameters. A multi-stage analysis is used to determine CPI and power dissipation given these design parameters.

Among the intermediate results obtained are values for cycle time. Currently, cycle time for a processor model is specified as a model input. By combining results from Olivieri's model with the existing processor model, it would be possible to extend the flexibility of the processor model to include process technology details in the analysis.

Memory Technology

Similarly, a more detailed analysis of memory core design could be performed. Amrutur and Horowitz [3] develop an analysis of SRAM delay and power dissipation, which is based on process parameters and the size and organization of the SRAM array. Delays for memory accesses are computed by considering separately gate and wire delays in address decoders and output multiplexers. The resulting model considers the performance effects of process parameters, memory size, and memory array organization.

For such a model to be useful in the SoC domain, it would be necessary to perform further study of on-chip RAM core implementations, and the possible use of memory controllers for accessing multiple RAM cores.

Power Modeling

As discussed in chapter 3, it should be possible to integrate power modeling into the SPAM methodology using the same component modeling mechanisms used for performance modeling. The benefit of combining power and performance modeling is that it gives an integrated picture of power consumption: while a design change may reduce average power consumption, it may also increase execution time, resulting in higher total energy consumption.

In the SPAM architecture, component-specific power models could be created, derived from `ComponentModel`. This gives them the built-in capabilities of:

- tracking running and preempted activities
- receiving property `Triggers`
- determining when the system state changes, such as property value changes or running activity set changes, require re-execution of power consumption estimation.

A power model derived from `ComponentModel` would set an average power consumption property for running activities, instead of setting their execution rate as is the case for performance models.

Details of how power estimation could be accomplished have not been researched sufficiently to propose component-specific methods.

9.1.3 Task Scheduling and RTOS

By integrating SPAM platform models with a system such as ARTS [22], the potentially correlated effects of platform design and operating system design could be investigated. Several changes would need to be made to both models to facilitate integration. First, the ARTS model's conception of task execution time would have to be changed: instead of pre-generating a random execution time, the execution time would be determined by the numerical integration of execution rate described in chapter 3. Second, the ARTS model's use of an allocator to determine resource assignment would need to be examined. If the locking of a resource was actually performed by a workload using operating system facilities, then the allocator could be used. Similarly, if a task actually utilized a resource exclusively for the entirety of its duration, the allocator could be used. However, if the resource was utilized for only a portion of the task duration in an unscheduled manner, this resource usage would be best modeled by describing activities executing on a platform component in the SPAM model.

Operating System Overhead

Since operating systems generally exist as software executing on a processor, it would be possible to model the execution time requirements of operating system functions as separate tasks, with their usage of platform components such as memory described as SPAM activities. For example, the overhead of running a scheduling algorithm could be defined by a task, which through task dependencies is specified as precedent to the scheduled task.

Task Preemption

For some RTOS scheduling algorithms, running tasks may be preempted by higher-priority tasks which become ready. The ARTS model considers this possibility by including a `preempted` state in the explicit state machine for tasks. Similarly, the SPAM model has been implemented with support for preemption `Triggers` and resumption `Triggers`, which allow a SPAM model to consider preemption and resumption of activities. Additionally, the `Arbitrator` and `ComponentModel` modules maintain separate lists of preempted activities.

In an integrated ARTS and SPAM model, when a task is preempted, a preemption `Trigger` could be sent to all components, and all activities in the task's `ActivityGraph`, including those waiting in the `Arbitrator` or those executing in a `ComponentModel` could be moved to the corresponding preempted list. When the RTOS allows the task to continue, a resumption `Trigger` could be sent which would then move the task's activities back to waiting or running state.

Boundary Conditions

When a task begins executing on a processor, there may be a certain associated overhead, such as context switching overhead. If tasks are relatively large and seldom preempted, context switching overhead may be safely ignored. If tasks switch frequently, it may be useful to model context switch overhead.

Similarly, when a task begins executing, the address space for its instruction stream and possibly for its working set of data will be different from that of the previous task. A working set change could also occur during task execution, depending on the workload and implementation. For a processor with cache, this change of address space for memory accesses may result in increased probability of cache misses, and hence an increased utilization of memory and interconnect, during the initial portion of task execution, or immediately after a working set change. The potential performance effects of this may be substantially higher than the few processor cycles required for a typical context switch. Again, the relative importance of this effect will depend on the size of tasks and the frequency of task switches.

The SPAM methodology does not explicitly support modeling of effects such as context switch or working set changes. It may be possible to model these effects as extra activities with different properties. The small instruction count of typical context switches, and special nature of processor behavior during context switch, may make it infeasible to model as a typical processor activity on the instruction throughput model described in chapter 4. However, an activity `mode` property defined for processor activities would allow the `ProcessorModel`'s `estimate` method to differentiate between standard activities and activities representing context switch. A simple method for calculating cycles for context switch, perhaps even by taking this delay as a model input, could then be used to determine the running time of context switch activities. Then, as shown in figure 9.1, a task's processor usage could be represented by a prefix context switch activity, followed by the actual task activities.

Working set changes could possibly be similarly modeled by specification of sequential activities: a set of activities could be defined to model the behavior of the workload in the period immediately after working set change, and a subsequent set of activities could be defined for the steady state behavior of the activity. However, activity execution rates and supplementary results are modeled as constant in the absence of changes in the running activity set. It is not currently possible to model time varying execution rates or memory latencies, for example.

Cache miss behavior after working set change may be expected to follow a characteristic curve approximately similar to that shown in figure 9.2(a). Using two sequential activities to describe this behavior would result in the curve shown in figure 9.2(b). If execution rates and properties could be specified as time varying functions, this would

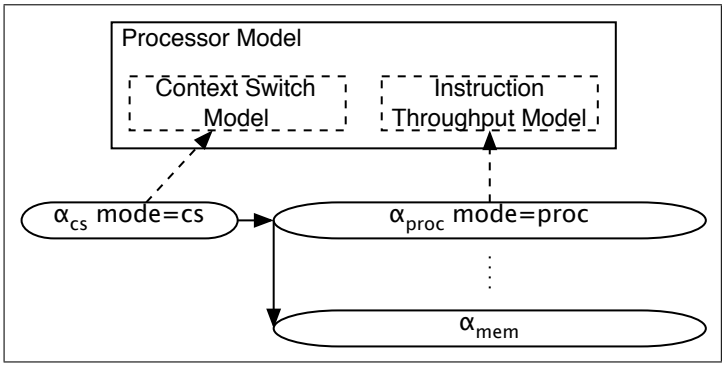
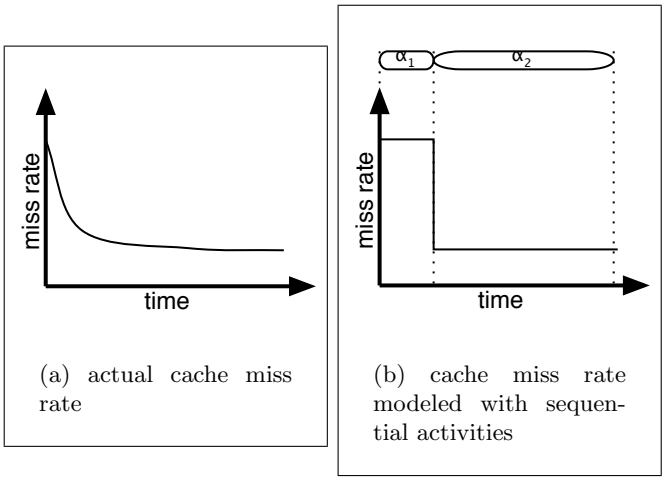


Figure 9.1: Modeling of Context Switch



(a) actual cache miss rate

(b) cache miss rate modeled with sequential activities

Figure 9.2: Modeling of Working Set Changes

allow more precise modeling of the actual behavior.

Another potential boundary condition would be modeling of delays incurred by power-saving functionality. Power-saving features could come in the form of reduced power modes in which devices do not perform their normal complete functionality, or it could come in the form of reduced frequency, reduced voltage operation.

In the case of power-saving mode changes, a component model would be required to track its power saving mode. A mechanism would be required to allow scheduled power-saving mode changes to be specified in the workload. Additionally, a method of modeling the delays incurred in shifting from one mode to another would be required. A possible solution for both of these issues is to use an activity, with a `mode` property such as that proposed for context switch changes, to model the delay, and to use this or another property to force the modeled component into a new power saving state.

In the case of reduced frequency operation, the model already supports specification of processor frequency. However, the frequency is currently specified as a component property. It may be useful to specify the component operating frequency as an activity property, allowing representation of changing frequency. Methods like Olivieri's [26], discussed in section 9.1.2, could provide a means of analytically determining power

consumption consequences of operating frequency changes.

9.2 Methodological Improvements

While the model can be extended in a variety of ways to consider different platform components and effects of operating systems, instruction sets and compilers, it may also be possible to improve model performance or accuracy through changes to the overall SPAM methodology.

9.2.1 Time Varying Values

As described previously, it may be useful to allow execution rates, property values or supplementary results to conform to a time varying function. This could allow more precise and flexible modeling of not only working set changes, but also varying workload behavior over time due to algorithmic or input data changes, or varying environmental conditions for systems with externally applied behavior, such as sensor networks.

Methods for describing linear piecewise functions have already been included in the model as a means of describing issue and retire dependence in the processor model. Since the SPAM simulation is a discrete time simulation, executing in steps, a sufficiently detailed linear piecewise function could describe any required time varying function.

However, use of time varying functions would also require some more fundamental changes in the modeling system. One possibility is to modify the `Generator` module to examine, calculate and then transmit new property values on every simulator clock cycle. However, this would result in an excessive amount of property `Triggers` being sent on the simulator's single link. Furthermore, time-varying supplementary results would require re-estimation for every property dependent activity, every cycle. While the component models presented in this paper are fast, repeating execution at each cycle defeats the original purpose of using analytical estimation methods.

A possible option is to use simple techniques such as linear interpolation to produce time-varying execution rate estimates based on time-varying model inputs. Using such a method, the performance estimating model would be run once for each distinct segment edge of a set of linear piecewise input functions. The results of the estimation would provide the endpoints for a new linear piecewise output function.

Figure 9.3 depicts an example in which an estimation method uses two time-varying linear piecewise functions as inputs. The resulting output linear piecewise function consists of segments time-delimited by the points in time in which either input function changes slope.

9.2.2 Statistical Distributions

The models presented in this paper produce what could be best described as typical case performance estimates. It may be interesting as well to examine worst case performance, best case performance, or a probability distribution of performance. Mechanisms to perform this analysis for the platform components have not been considered. However, from a methodological standpoint, this would require properties, execution rates and supplementary results to be specified as distributions, and would require modification

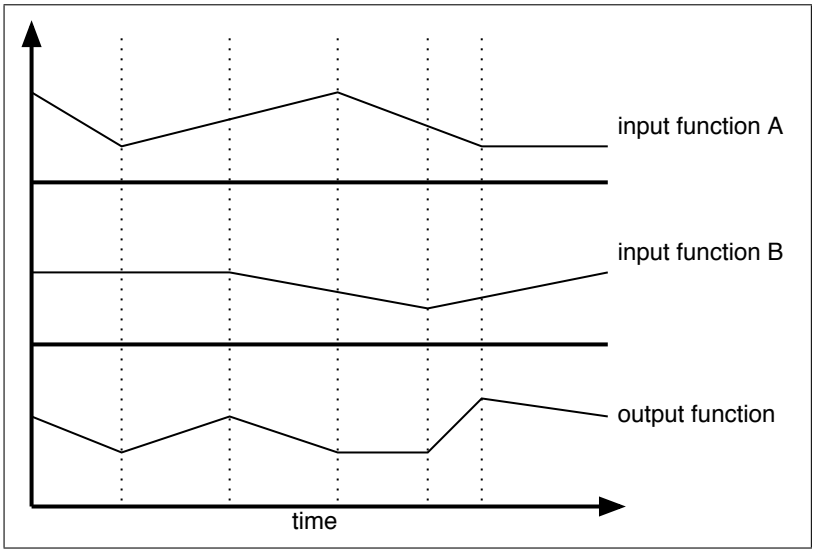


Figure 9.3: Use of Time-Varying Functions in Modeling

of the activity progress mechanism to track all statistically possible progress rates. The complexity of this requirement may mean that it would be necessary to run multiple simulations with random, static property inputs to compile a set of results, which could then be used to define a result distribution. An additional consideration would be the potential correlation of model inputs.

9.2.3 Event-Based Simulation

The SPAM methodology currently uses what is essentially a polling method of simulation: at each simulator clock tick, each running activity has its execution rate sampled and its progress updated. At each simulator clock tick, dependency information is analyzed and the appropriate `Triggers` are generated, sent, and processed. Such a method can be inefficient, particularly if the activity rates change infrequently.

The usual alternative to a polling simulation is to use event-based simulation. In an event-based simulator, time ordered lists of events are maintained, indicating when the state of the simulated system changes. The simulator clock can be advanced to the point of the next change in state. SimpleScalar provides mechanisms such as `sc_event` to perform an event-based simulation.

To take advantage of these mechanisms, some additional functionality would have to be included in the SPAM methodology. Instead of incrementing activity progress each cycle, calculation of the progress amount between consecutive events would have to be determined. Whether using constant execution rates or a time-varying rate specified by a linear piecewise function, this is not a particularly difficult task. Additionally, the timing of future events, including dependency trigger points and activity completion, could be determined relatively simply.

However, the major obstacle in using event-based simulation is that activity execution rates can be changed by one event, invalidating the timing of previously calculated future events. Two possible methods for overcoming this problem are:

- purging any invalidated future events and recalculating their timing
- retaining only the next future event, and calculating all event timings when this event fires.

The first method requires some support from the SystemC library for selective cancellation of events. Whether this capability is supported has not been investigated. If it is supported, a method would also be required for determining which future events are invalidated. This set of events would include all future dependency triggers, including the self-completion dependency, for any activity whose execution rate changes. After canceling the existing events, a new set of events could be issued, based on the newly calculated execution rates. The SystemC simulator then provides the mechanisms for maintaining a sorted list of events, and for advancing the simulator clock to the next event.

In the second method, the need for event cancellation is eliminated, since only the next future event is submitted to the SystemC simulator's event list. Note that the next event can never be invalidated, since the only changes to execution rate will occur in response to other events. This method places the responsibility on the SPAM system for determining event ordering, rather than allowing SystemC to track all events. Furthermore, this method requires recalculation of event timings and redetermination of the next event after each event firing. The benefit is that it is no longer necessary to invalidate future events.

Although the performance of the current model is generally very good compared to cycle-accurate simulation, it is nevertheless desirable to replace the current polling method of tracking activity progress and dependencies with some event-based mechanism. Doing so would eliminate any need for setting simulator resolution, and the corresponding potential for resolution errors. Using a simulator clock rate equal to the actual component clock rates would not affect model execution time, only the size of the clock advancements performed at each event firing.

9.3 Application and Testing

While application of the model was performed on three different processors, and on single and multiprocessor platforms, it is clear that there is a much wider array of platform configurations and components available. Further testing on more diverse test cases would be beneficial. Unfortunately, while specifying new platform configurations and setting model parameters for different components is relatively easy in the SPAM model, acquiring and setting up numerous simulators for comparison can be a difficult and time consuming process.

Additionally, it would be interesting to use SPAM models to perform a design space analysis of some simple platform designs, whether by integrating the model with some design space exploration algorithm, or by simply executing the model on a set of configurations and plotting results. Such an analysis has not yet been performed. It may be desirable to test the method more rigorously on a wider range of model inputs before considering the model results in detail.

9.4 Performance

No precise measurement of the SPAM model execution time has been performed, but generally platform performance analysis can be completed in a matter of a few seconds. Corresponding simulations on cycle-accurate simulators required between a few minutes and, for the most complicated workloads, several hours. The bulk of the SPAM model runtime is usually taken up by the polling method used for updating activity progress and analyzing dependencies.

Chapter 10

Conclusion

In this project, a framework for modeling SoC platforms has been developed. The framework, SPAM, provides a high degree of modeling flexibility. Through the use of object-oriented design techniques, SPAM models can be constructed for a variety of platform components, and new models can easily be added to the framework. An augmented directed graph-based description of workloads allows considerable freedom and precision in describing workloads applied to SoC platforms. A method for structurally composing platform performance models from arbitrary numbers and kinds of component models allows modeling of a wide variety of platform configurations. Parameterizable component models allow a range of components to be easily modeled, without resorting to source code changes. Through the use of the FEARS library, file-based input is used to control all aspects of platform model creation and execution, meaning the modeling flexibility provided by the framework is carried over to runtime.

Modeling of many SoC platforms may require analysis of contention for shared resources, and the resulting effects on platform performance. Queueing systems analysis, with considerations particular to the modeling of processor-based platforms, was used to predict resource performance for FIFO conforming resources. The more complex behavior of resources governed by non-FIFO arbitration schemes necessitated the use of Monte Carlo simulation to estimate performance. A Markov chain analytical method for modeling these resources was explored, but determined to be too computationally expensive in the absence of suitable heuristics.

Selection and development of component models focused on speed, flexibility, and accuracy. A superscalar processor model from the literature [31] was selected, and extended for application to scalar processors. A simple model for determining the latency of memory accesses was combined with the FIFO shared resource modeling methods to create a predictive model for memory cores. Similarly, a method for determining the latency of communication events [18] was extended and combined with the Monte Carlo-based model for arbitrated shared resources to provide a method for predicting bus performance.

The applicability of the presented models and the flexibility of the modeling framework was demonstrated by applying the models to a number of platform configurations operating under a number of workloads. These include:

- various MiBench workloads executing on the SimpleScalar simulator with ideal

cache hierarchy
- MiBench workloads executing on the SimpleScalar simulator with contention for unified L2 cache
- prime number workload executing on the Avrora AVR simulator
- matrix multiplication workload executing on a single processor MPARM platform
- matrix multiplication workload executing on a multiprocessor MPARM platform.

The results of these tests indicate that the collected component models can be combined to produce accurate estimates of performance for a wide range of platforms. Furthermore, the time required by the SPAM models to obtain an estimate is much smaller than the corresponding simulation time on a cycle-accurate simulator.

The modeling flexibility inherent in the SPAM methodology invites extension of the model in several ways. Models of other platform components, such as various coprocessors, reconfigurable hardware, communication interfaces, NoCs, and I/O devices could be created. Power models for components could be integrated with the performance model to obtain power estimates which reflect changes in execution time. Cache and branch predictor modeling could be added to the capabilities of the processor model. Through the property dependency mechanism, it may be possible to consider effects of non-execution time design parameters, such as instruction set design or compiler optimizations.

Finally, while the SPAM platform models are fast compared to cycle-accurate simulators, it is possible to improve performance further. Use of event-based simulation methods would reduce the amount of processing time spent polling completion and dependency information. Use of analytical methods for predicting arbitrated shared resource performance could potentially reduce model running time compared to statistical Monte Carlo simulation. The instruction throughput processor model could be accelerated by predicting and then calculating the effect of steady-state operation during macroblock execution.

Bibliography

[1] Santosh G. Abraham and Scott A. Mahlke. Automatic and Efficient Evaluation of Memory Hierarchies for Embedded Systems. *Proceedings of the 32nd Symposium on Microarchitecture*, pages 114–125, 1999.

[2] David H. Albonesi and Israel Koren. STATS: A Framework for Microprocessor and System-Level Design Space Exploration. *Journal of System Architecture*, pages 1097–1110, 1999.

[3] Bharadwaj S. Amrutur and Mark A. Horowitz. Speed and Power Scaling of SRAMs. *IEEE Transactions on Solid-State Circuits*, February 2000.

[4] ARM Ltd. ARM7 Processor Family. http://www.arm.com/products/CPUs/families/ARM7Family.html.

[5] ARM Ltd. AMBA Specification. http://www.arm.com/products/solutions/AMBA_-Spec.html, 1999.

[6] ATMEL Corporation. AVR Enhanced RISC Microcontrollers Data Book, May 1996.

[7] Doug Burger, Todd M. Austin, and Steve Bennett et. al. Evaluating Future Microprocessors: The SimpleScalar Toolset, 1996.

[8] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing State Loss for Effective Trace Sampling of Superscalar Processors. *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 468–477, 1996.

[9] Robert Cooper. *Introduction to Queueing Theory.* Elsevier North Holland, Inc., 1981.

[10] Michael Dales. Software ARM. http://www.cl.cam.ac.uk/ mwd24/phd/swarm.html.

[11] Lieven Eeckhout and Koen De Bosschere. Early Design Phase Power/Performance Modeling Through Statistical Simulation. *IEEE Internation Symposium on Performance Analysis of Systems and Software*, 2001.

[12] Lieven Eeckhout and Koen De Bosschere. Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces. *Proceedings of the Internation Conference on Parallel Architectures and Compilation Techniques*, 2001.

[13] Harold W. Cain et. al. Precise and Accurate Processor Simulation. *Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 13–22, February 2002.

[14] Joshua J. Yi et. al. A Statistically Rigorous Approach for Improving Simulation Methodology. *ARCTiC Technical Report 02-07*, 2002.

[15] M. R. Guthausch et. al. MiBench: A Free, Commercially Representative Embedded

Benchmark Suite. *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[16] Youngchul Cho et. al. Scheduling and Timing Analysis of HW/SW On-Chip Communication in MP SoC Design. *Design Automation and Test in Europe Conference*, March 2003.

[17] Jesper N. R. Grode. Component Modeling and Performance Estimation in Hardware/Software Codesign, March 1999.

[18] Peter Voigt Knudsen and Jan Madsen. Integrating Communication Protocol Selection with Partitioning in Hardware/Software Codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 111–116, December 1998.

[19] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Fast Performance Analysis of Bus-Based System-on-Chip Communication Architectures. *Proceedings of the International Conference on Computer-Aided Design*, pages 566–573, 1999.

[20] Kanishka Lahiri, Anand Raghunathan, and Sujit Dey. Performance Analysis of Systems with Multi-Channel Communication Architectures. *Proceedings of the International Conference on VLSI Design*, pages 530–537, January 2000.

[21] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing On-Chip Communication in a MPSoC Environment. *Design and Test in Europe Conference*, February 2004.

[22] Jan Madsen, Kashif Virk, and Mercury Gonzalez. Abstract RTOS Modelling in SystemC. *Proceedings of the 20th IEEE NORCHIP Conference*, pages 43–49, November 2002.

[23] Microelectronics Research Group, University of Bologna. MPARM Project. http://www-micrel.deis.unibo.it/ angiolini/mparm.html.

[24] Randolph Nelson. *Probability, Stochastic Processes, and Queueing Theory.* Springer-Verlag New York, Inc., 1995.

[25] Derek B. Noonburg and John Paul Shen. A Framework for Statistical Modeling of Superscalar Processor Performance. *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 298–309, February 1997.

[26] M. Olivieri. Theoretical System-Level Model for Power-Performance Trade-off in VLSI Microprocessor Design. *Workshop on Complexity-Effective Design*, June 2001.

[27] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1996.

[28] SimpleScalar LLC. SimpleScalar. http://www.simplescalar.com.

[29] Elizabeth S. Sorenson. Cache Characterization and Performance Studies Using Locality Surfaces, February 2003.

[30] T. M. Taha. A Parallelism, Instruction Throughput, and Cycle Time Model of Computer Architectures, November 2002.

[31] T. M. Taha and D. S. Willis. An Instruction Throughput Model of Superscalar Processors. *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping*, June 2003.

[32] UCLA Compilers Group. Avrora: The AVR Simulation and Analysis Framework. http://compilers.cs.ucla.edu/avrora.

[33] Liqiang Zhang and Vipin Chaudhary. On the Performance of Bus Interconnection for SoCs. *Proceedings of the 4th Workshop on Media and Stream Processors*, November 2002.

[34] Y. Zhu and W. F. Wong. Performance Analysis of Superscalar Processor Using a Queueing Model. *Proceedings of the 2nd Australian Conference on Computer Architecture*, pages 147–158, 1997.

[35] Y. Zhu and W. F. Wong. Sensitivity Analysis of a Superscalar Processor Model. *Australian Computer Science Communications*, pages 109–118, 2002.

Appendix A

Source Code

Source code for the SPAM modeling system and presented models is provided on a supplementary CD.