MULTI-OBJECTIVE DESIGN SPACE EXPLORATION OF EMBEDDED SYSTEM PLATFORMS

Jan Madsen, Thomas K. Stidsen, Peter Kjærulf, Shankar Mahadevan
Informatics and Mathematical Modelling
Technical University of Denmark
{jan,tks,sm}@imm.dtu.dk

Abstract In this paper we present a multi-objective genetic algorithm to solve the problem of mapping a set of task graphs onto a heterogeneous multiprocessor platform. The objective is to meet all real-time deadlines subject to minimizing system cost and power consumption, while staying within bounds on local memory sizes and interface buffer sizes. Our approach allows for mapping onto a fixed platform or onto a flexible platform where architectural changes are explored during the mapping.

We demonstrate our approach through an exploration of a smart phone, where five task graphs with a total of 530 tasks after hyper period extension are mapped onto a multiprocessor platform. The results show four non-inferior solutions which tradeoffs the various objectives.

1. Introduction

Modern embedded systems are implemented as heterogeneous multiprocessor systems often realized as a single chip solution, System-on-Chip (SoC). Given the high development cost and often short time-to-market demands, these systems are developed as domain specific platforms which can be reconfigured to fit a particular application or set of applications. They are typically designed under rigorous resource constrains, such as speed, size and power consumption. Determining the right platform and efficiently mapping a set of applications onto it, requires hardware/software partitioning, hardware/software interface, processor selection and communication planing.

In this paper, we address the following problem:

> Given a set of applications with individual periods and deadlines, and a heterogenous multiprocessor architecture on which to execute the applications, determine a *mapping* of all tasks on processors and all communications on communication links, such that all deadlines are met subject to power consumption, memory size, buffer sizes of network adapters and overall component cost.

By mapping we mean the allocation of tasks in space and time, i.e. the determination of which tasks to execute on a given processor as well as the detailed time schedule of each task, and likewise for the communications on communication links.

We address two different variations of the problem;

1 *Fixed* platforms, i.e. no changes of type or number of processors nor any interconnection topology. Hence, the focus is on mapping the applications onto the platform. This variation corresponds to the case where we want to *re-use* an existing platform, which is often the case when moving from one generation to the next of a product family.

2 *Flexible* platforms, i.e. the types and/or number of processors may be changed and the interconnection topology may be changed by adding or removing buses and bus bridges. This variation corresponds to the case where we may change the platform to better fit the requirements of the application.

To demonstrate the capabilities of our approach, we will explore the design of a smart phone, i.e., a heterogeneous multiprocessor platform running five applications with a total of 114 tasks: MP3, JPEG Encoder, JPEG Decoder, GSM Encoder and GSM Decoder. We will demonstrate how our approach can lead to improved solutions for both variations of the optimization problem and in particular for the co-exploration of the architecture selection and application mapping.

The rest of the paper is organized as follows; Section 2 discusses related work. Section 3 presents the application and architecture models. In Section 4 and 5 we present details of our exploration framework. Section 6 present the design space exploration case study of a smart phone. Finally, we present the conclusions in Section 7.

2. Related Work

Static scheduling algorithms for mapping task graphs onto multiprocessor platforms have been studied extensively. A good survey of various heuristic scheduling methods can be found in [5].

Recently, Genetic Algorithms (GA) have been applied to multiprocessor co-synthesis problems due to their property to escape local optima [3, 6–8]. In [6], the goal of the GA-based scheduler is to minimize completion time of all tasks. Although some processor characteristics are taken into account, the approach only addresses homogeneous platforms. In [7] the objectives are to minimize the number of processors required and the total tardiness of tasks for real-time task scheduling. In MOCAG [3] the objectives are extended to also include power consumption beside system price (cost) and task completion time. The approach showed very good results in particular for large systems. The approach described in [2] minimizes schedule length (i.e. the sum of computation, communication and processor wait times) in mixed-machine distributed heterogeneous computing platforms executing up to 200 tasks. The approach uses a fast heuristic with the GA optimization, thereby reducing the exploration time as compared to traditional GA. The approach presented in [8] emphasize energy minimization through the use of dynamic voltage scaling provided by the processors. It is applied to heterogeneous multiprocessor SoC platforms.

Our approach is similar to [2] and [8], but we use a more detailed communication exploration, and in addition to cost, completion time and energy, we explore memory and buffer constrains - with true multi-objective optimization.

3. Models

In this section we present the application model and the architecture model on which to execute the application. Both are inputs to our exploration environment.

3.1 Application Model

We consider a real-time application to be modelled as a task graph (expressed as a directed acyclic graph) $G_T = V_T, E_T)$, where $V_T = \{\tau_i : 1 \leq i \leq n\}$ is the set of schedulable tasks, and $E_T = \{e_j : 1 \leq j \leq k\}$ is the set of directed edges representing the data dependencies between the tasks in V_T, i.e., if $\tau_i \prec \tau_j$ then $(\tau_i, \tau_j) \in E_T$. The weight of an edge indicates the size of the message to be transferred between two tasks. Figure 1a shows a example of an application task graph. Each task $\tau_i \in V_T$ is characterized by a five tuple $\langle d_i, T_i, c_i, e_i, m_i \rangle$, i.e. the exact functionality of the task is abstracted away. The relative deadline, d_i, and the period, T_i, are given by external requirements of the application and, hence, are independent of runtime input values, intermediate results or configurations of processing elements. However, the execution time, c_i, the consumed energy, e_i, and the memory usage, m_i, are all determined by the actual mapping of the task onto a particular processor.

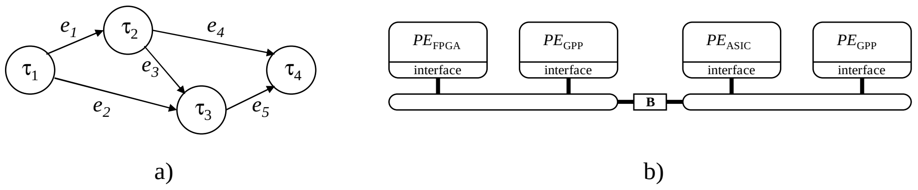
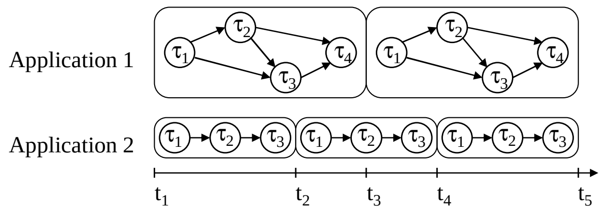


Figure 1. Models, a) Application task graph, b) Architecture graph with 4 PEs, 2 busses and a bus bridge.

The deadline of a real-time application, D_T, is represented by the deadline of the task(s) in V_T with no successors, i.e. no outgoing edges. The task graphs for the different applications are unfolded to cover the hyper period of the complete application. If the different task graphs have different deadlines the period of the hyper period is the least common multiple of all task graphs periods. Figure 2 shows a case of two applications unfolded to fit the hyper period. Application 1 has two copies and Application 2 has three copies.

An instance of a task graph cannot start before the preceding instance has completed its execution. The table in figure 2 shows the earliest start time for each task graph instance.



App.	inst.	EST	d_i
1	1	t_1	t_3
	2	t_3	t_5
2	1	t_1	t_2
	2	t_2	t_4
	3	t_4	t_5

Figure 2. Hyperperiod of two task graphs, and Table with earliest start time and deadline for all task graph instances.

3.2 Architecture Model

We consider a heterogeneous multiprocessor architecture to be modelled as an architecture graph $G_A = (V_A, E_A)$. The vertices represent three different types of components, $V_A = V_{PE} \cup V_L \cup V_B$, where $V_{PE} = \{PE_q : 1 \leq q \leq m\}$ is the set of processing elements (PEs), $V_L = \{l_v : 1 \leq v \leq l\}$ is the set of buses which makes up the interconnection network,and $V_B = \{b_k : 1 \leq k \leq r\}$ is the set of bus bridges. Processing elements can be any of dedicated hardware accelerators (PE_{ASIC}), reconfigurable devices (PE_{FPGA}), or general purpose processors (PE_{GPP}). Each PE is characterized by a tuple $\langle f_i, m_i \rangle$, where f_i is the operating frequency of the processor and m_i is the maximum size of the local memory of the processor. Figure 1b shows a example of an architecture graph.

The mapping of the individual tasks, determines if a task will be implemented as hardware logic, ASIC and/or FPGA, or as software running on a GPP. Consequently, by choosing a different processor, the execution characteristics of the task may be changed, which in turn will affect the scheduling of the succeeding tasks; and eventually the completion time of the application.

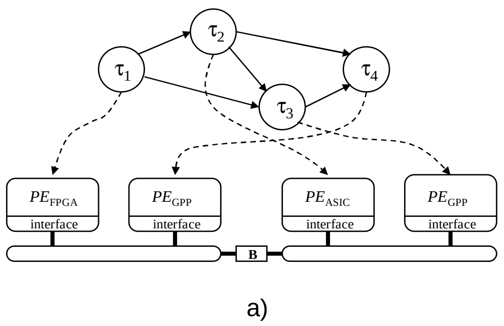
The interconnections are formed by a (possible hierarchical) network of buses connected through bridges. The communication between two tasks mapped to the same PE is done via accessing shared memory, i.e. we assume that each processing element has local memory, and its access time is negligible. The communication delay between two tasks mapped to different PE's is the property of the size of the message, the sizes of the interface buffers, and the bandwidth of the bus.

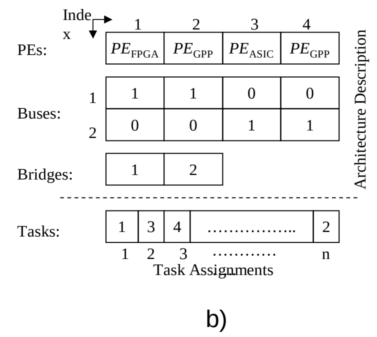
Processing elements are connected to buses through network adapters. A network adapter may include buffers, allowing for communication to take place concurrently with computation.

4. Design Space Exploration

To solve the presented multi-objective optimization problem, we have used the PISA framework [1] to create a multi-objective Genetic Algorithm (GA). We take as input the set of application task graphs and an architecture graph as described in Section 3. The GA is responsible for design instantiations, i.e. the selection of V_A, and the assignment of the set of tasks V_T onto the set of processing elements $V_{PE} \in V_A$. The selection process can be skipped if the user is only interested in a mapping onto a *fixed* platform, otherwise the platform will be regarded as flexible.

A GA is an iterative and stochastic process that operates on a set of individuals (the population). Each individual represents a potential solution to the problem being solved, and is obtained by decoding the genome of the individual. Initially, the population is randomly generated (in our case based on the input graphs). Each individual in the population is assigned a fitness value which is a measure of its goodness with respect to the problem being considered. This value is the quantitative information used by the algorithm to guide the search for a feasible solution. The basic genetic algorithm consists of repeated execution of three major stages: selection, reproduction, and replacement. Each iteration is called a generation. During the selection stage, individuals with a high fitness value has a higher probability of being selected to create of spring through crossover. A new population is then created by performing crossover followed by mutation. Finally, individuals of the original population is substituted by the newly created individuals in such a way that the most fit individuals are kept deleting the worst ones. A thorough description of genetic algorithms may be found in [4]. There are two important issues which have to be addressed when formulating a problem to be solved by a GA; the representation, i.e. the encoding/decoding mechanism of the genom of an individual, and the evaluation of the fitness of an individual. These issues will be explained in the following sections.





		1	2	3	4
PEs:		PE_{FPGA}	PE_{GPP}	PE_{ASIC}	PE_{GPP}
Buses:	1	1	1	0	0
	2	0	0	1	1
Bridges:		1	2		

	1	2	3	………..	n
Tasks:	1	3	4	……………..	2

Figure 3. a) Example of a mapping, and b) the corresponding GA representation.

4.1 Design Representation

In order for the GA to optimize the designs, each design must be represented as an individual. Figure 3 shows a mapping of an application graph onto an architecture, and the corresponding representation. Each individual consists of two parts: A part specifying the architecture and a part specifying the task assignment. In Figure 3b the architecture representation part contains an array of the deployed processing elements, in this example four PEs of three different types (GPP, $ASIC$ and $FPGA$). The connection between the PEs is given by the $2D$ matrix. Each row corresponds to a bus and each element in the row indicates if the corresponding PE is connected to the bus ('1') or not ('0'). The bridges which are connecting the busses, are defined as a bridge matrix, where each row represents a bridge and the elements indicates which busses the bridge connects. The task assignment is given as an array, where each index identifies a task and the corresponding element identifies the index of the PE to which the task is assigned.

The chosen representation is problem specific and uses internal references. The tasks do *not* identify which PE to use, but rather the index of the PE in the PE array. Hence, if the type of a PE is changed for an entry, all tasks referring to this index, will have their executing PE changed.

4.2 Genetic Operators

Initially, a set of individuals are instantiated with unique architecture and application mapping in order to form a population. During each generation we can apply one or more of the following five types of genetic operators,

- *Change* PE: Randomly select an existing PE and change it's type, and randomly select a bus and change its type.
- *Add* PE: Add a new PE to a randomly selected bus, and assign $\lceil \frac{|V_T|}{|V_{PE}|} \rceil$ tasks randomly selected from the other PEs.
- *Remove* PE: Remove a PE from a randomly selected bus, and distribute its tasks among the remaining PEs.
- *Crossover*: Crossover on PE types and tasks mapped to PE. This operator copies the mapping and PE-type from one individual to a PE in another individual.
- *Randomly Re-assign Task*: Move [1;4] randomly selected tasks from a PE to another randomly chosen PE.
- *Heuristically Re-assign Task*: Identify the task graphs which have tasks missing their deadlines, and select a task from these and move it to a PE with no deadline violation.

The first four of the genetic operators enables the GA to find any solution in the problem space. The fifth mutation operator adds a more focused search regarding deadlines and workload balancing. Neither of these operators change the cardinality of V_L, however the GA has full flexibility to reorganize the existing interconnect topology. After applying these operators to individuals, the outcome needs to be evaluated. This is done by a scheduling algorithm which is responsible for determining the start- and the end-times of the computation and communication activities. The scheduling algorithm will be presented in the next section.

5. Scheduling

The scheduling task is NP-hard, and it has to be performed for each individual constructed by the GA algorithm. Hence, a fast scheduling method is central for good performance. For a survey of different scheduling algorithms see [5]. We have chosen to use a static list scheduling algorithm which requires a priority for each task. We use a mix of the so called t-levels and b-levels: The t-level of a task is the earliest start time of that task whereas the b-level is the latest start time if time limits are to be satisfied. We use a linear combination of the two measures to produce a task priority-list.

During scheduling tasks are selected from the start of the priority-list but with two important sub conditions

1 For a task to be selected for scheduling, all of its preceding tasks have to have been scheduled already.

2 Tasks with smallest 'earliest start time' is scheduled before other tasks.

5.1 Scheduling algorithm

In Figure 4 we outline the pseudo code for the list scheduling algorithm. The list scheduling algorithm initially calculates the t- and b-levels to initialize the $Priority\_List$ (1). Then the list $Num\_Unschedueld\_Predecessors$ is initialized (2). Then the current task to schedule τ_y is set to the task with the highest priority which also satisfies sub-condition 1) and 2) (3). In the main loop, first the earliest possible starting time for the task is found (5). Then τ_y is scheduled to start at this time (6). Afterwards the $Num\_Unschedueld\_Predecessors$ is updated (7). Then the task with the highest priority satisfying sub-condition 1) and 2) is selected as the next task τ_y to schedule (8). Finally the *Earliest Communication Time* (ECT) for all predecessors to τ_y are found, in order to find earliest ready communication resources for mapping and scheduling (9).

```
1: Calculate $Priority\_List$.
2: Initialize $Num\_Unschedueld\_Predecessors[..]$
3: Set $\tau_y$ to the first task in $Priority\_List$ satisfying sub condition 1) and 2)
4: repeat
5:    Find earliest starting time for $\tau_y$
6:    scheduled $\tau_y$
7:    Update $Num\_Unschedueld\_Predecessors[..]$
8:    Set next ready task in $Priority\_List$ to $\tau_y$
9:    Calculate $ECT$ to $\tau_y$
10: until All tasks scheduled
```

Figure 4. Scheduling Algorithm

Example: Consider a given inter-task communication: $(\tau_x, \tau_y) \in V_T$ (Figure 5a), such that $\tau_x \prec \tau_y$, and $(PE_1, PE_3) \in V_{PE}$, where $\tau_x \rightarrow PE_1$ and $\tau_y \rightarrow PE_3$. Further we assume that the network adapter in PE_3 has no buffers, while PE_1 has both input and output buffers. For the schedulable resources and their interconnectivity, we associate $l_v \in V_L$ a vector of items in the topology set i.e. direct bus (one item) or bridged bus (3 or more items) connecting PE_1 with PE_3. In this case, l_v consists of 3 items: local buses of PE_1 and PE_3, l_1 and l_2, and the bridge, b_1, between l_1 and l_2. Further, we assume the bandwidth of $l_2 > l_1$. Let the message size to be transferred be m. Figure 5b shows a snapshot of the scheduling profile during the communication of interest. For clarity, we assume the transfers over the bridge to be instantaneous and hence ignored in the figure. The shaded portions, imply that the shared resource is busy.

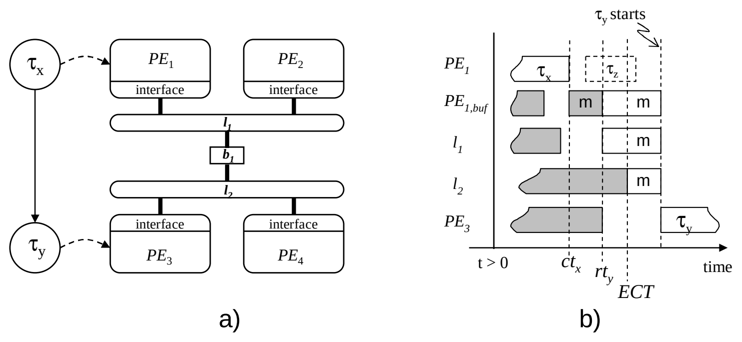


Figure 5. a) Mapping of two tasks, and b) calculation of Earliest Communication Time.

In the following, we are showing how ECT is calculated for the example in Figure 5a. First we calculate the completion time, (ct_x), of τ_x on PE_1. For PE_1, the space in the output buffer, $PE_{1,buf}$, is found to be available, thus the message is moved to $PE_{1,buf}$, freeing PE_1 to start executing another task τ_z. Knowing the precedence constraints and the ordering in the $Priority\_List$, we calculate the earliest possible start time, rt_y, for τ_y on PE_3. ECT is set to the furthest time when either the communication is possible (τ_x completes on PE_1) or required (τ_y ready to start on PE_3), i.e. $ECT = rt_y$. Then we find the topology set, l_v, connecting PE_1 with PE_3, which is $\{l_1, l_2\}$. We evaluate the availability of each of the busses of l_v. Although l_1 is available, the earliest time at which communication can be scheduled is when l_2 is also available. This dictates the ECT. Overall the communication speed is dictated by the slower bus, keeping the output buffer, $PE_{1,buf}$ occupied. The actual start time of τ_y is after the message m has been received. $\Box$

5.2 Memory issues

During scheduling both interface buffers and local memory are taken into account.

Interface buffers of a processor can be used in two ways 1) to store data coming from the bus to the processor and 2) to store data going from the processor to the bus. It is assumed that buffers can not block. This means that even if a communication task can not be stored in the buffer (e.g. buffer is full), the buffer can still send data to the bus.

When tasks are mapped to processors, the static and dynamic memory consumption of the tasks are taken into account. This assures that the number of tasks mapped to a PE will always fit within the available size of the local memory. The local memory size for each PE is specified as a constraint in the input. However, during scheduling data waiting to be sent to the bus may have to be saved in the local memory of the processors, for instance in the case where the corresponding buffer is full. This can cause a violation of the memory constraint on a given processor. This memory violation is one of the objectives optimized in the multi-objective GA algorithm.

6. Case study

In this section we explore a smart phone [8] running 5 applications (JPEG encoder and decoder, MP3, and GSM encoder and decoder) with a total of 114 tasks. After expanding the task graphs into a hyper period, we have a total of 530 tasks to schedule. The GA was run for 100 generations which corresponds to approximately 10 min of run time. In each generation 100 individuals was evaluated. Hence, 10.000 solutions were explored, resulting in four interesting architectures (see figure 6) on the approximated pareto front. Table 1 lists the cost, energy consumption and memory violation for each of the four architectures.

id	price	Energy consumption	Memory violation
id : 166	1396	2.20746e+07	336
id : 171	1048	2.89746e+07	0
id : 184	1396	2.45602e+07	0
id : 187	1596	2.19572e+07	153

Table 1. Characteristics of four solutions on the approximated pareto front. Memory violation is measured in 32-bit words.

The two architectures id 166 and id 184 are identical, but with a different mapping of tasks to processors. This gives id 166 a smaller energy consumption with the cost of a memory violation. The cheapest architecture is id 171, this is however the solution with the largest energy consumption. With regard to energy consumption id 187 is the cheapest but at the same time the most expensive architecture.

As there is no guarantee for optimal solutions the selection of architectures will only be an approximation to the pareto front. However, the experiment shows how the algorithm is a powerful tool to explore the design space for embedded system architectures with both one and multiple busses.

7. Conclusions

The design of a heterogenous multiprocessor system, is accomplished either by design reuse or incremental modification of existing designs. In this paper, we have presented a multi-objective optimization algorithm which allows to optimize the application mapping on to an existing architecture, or optimize the application mapping and architecture during development. Our algorithm couples GA with list scheduler. The GA allows to instantiate multiple designs which are then evaluated using the scheduler. The outcome is an approximated pareto front of latency, cost, energy consumption and buffer and memory utilization. The case study has shown, that maximum gains are achieved when optimizing both architecture and application simultaneously.

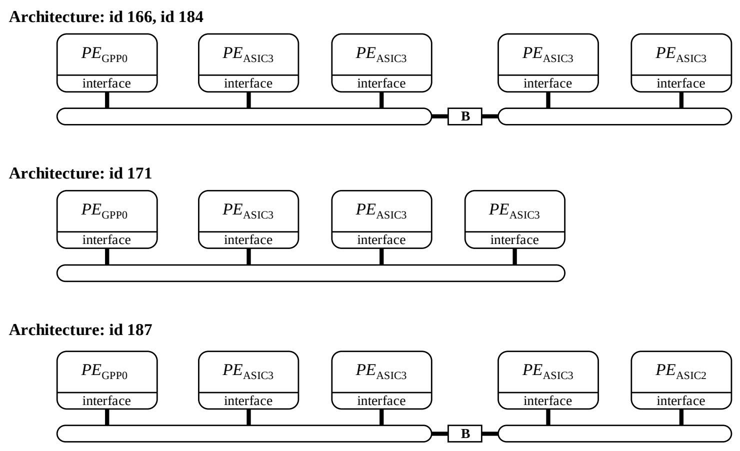


Figure 6. Non-inferior architectures from the optimization runs.

8. Acknowledgement

This work has been supported by the European project ARTIST2 (IST-004527), Embedded Systems Design.

References

[1] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. PISA — a platform and programming language independent interface for search algorithms. In Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508, Berlin, 2003. Springer.

[2] Muhammad K. Dhodhi, Imtiaz Ahmad, Anwar Yatama, and Ishfaq Ahmad. An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. In *Journal of Parallel and Distributed Computing*, pages 1338–1361. Elsevier Science, 2002.

[3] Robert P. Dick and Niraj K. Jha. MOGAC: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 920–935. IEEE, 1998.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison-Wesley, 1989.

[5] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[6] Ceyda Oguz and M.Fikret Ercan. A genetic algorithm for multi-layer multiprocessor task scheduling. In *IEEE Region 10 Conference (TENCON)*, pages 168–170. IEEE, 2004.

[7] Jaewon Oh and Chisu Wu. Genetic-algorithm-based real-time task scheduling with multiple goals. In *Journal of Systems and Software*, pages 245–258. Elsevier, 2004.

[8] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Kluwer Academic Publishers, 2004.