A New Approach to Solving the Hardware-Software Partitioning Problem in Embedded System Design

Daniel W. Engels Srinivas Devadas
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA USA 02139
(dragon,devadas)@lcs.mit.edu

Abstract

We present a new approach for solving the hardware-software partitioning problem in embedded system design. Our approach is based on transforming an instance of the hardware-software partitioning problem into an instance of a deterministic scheduling with rejection problem that minimizes a function of the completion times of the tasks. A solution to this real-time scheduling problem yields a partition of the system functionality and provides valuable feedback to the system designer. Experimental results indicate that the simplicity and effective solution techniques of our approach make it ideally suited as an automated design analysis tool in embedded system design.

1 Introduction

In recent years, small embedded systems have evolved into complex multifunction systems implemented using a mix of both hardware and software. The allocation of the system functionality into hardware and software components has a significant impact on the the total system cost. For this reason, several approaches have been presented to automate the exploration of possible functional partitions, thereby aiding in the solution of the Hardware-Software Partitioning Problem [1] [2] [4]. These approaches typically target either a single-processor single-ASIC System On a Chip (SOC) or a multiple Processing Element (PE) distributed heterogeneous system. In this paper, we present a simple, effective, and efficient approach to solving the hardware-software partitioning problem. Our approach assumes the SOC target architecture, but its simplicity and efficiency allow it to be used for distributed heterogeneous target architectures as well.

Regardless of the target architecture, there are three main subproblems that must be solved in determining the hardware-software partition of a system:

- **Functional Clustering:** Cluster the system functionality into a set of tasks.
- **Allocation:** Allocate the tasks to either hardware or software (or to a subset of the given PEs).
- **Scheduling:** Schedule the allocated tasks to determine timing correctness of the partitioned system.

These problems are interdependent; thus, they must be solved simultaneously to determine an optimal solution. The allocation and scheduling subproblems are known to be $\mathcal{NP}$-hard [3], and there are an exponential number of possible functional clusters. Therefore, determining a guaranteed optimal solution is computationally intractable for all but the simplest systems. Consequently, automated heuristic approaches to solving the hardware-software partitioning problem should be used by the designer to search the solution space.

The most common heuristic approaches assume a fixed functional clustering and use either stochastic search, iterative improvement, or constructive algorithms to solve the allocation subproblem. The tasks are scheduled to evaluate the allocations. Stochastic search based algorithms, such as simulated annealing and genetic algorithms, have been found to yield optimal or near optimal solutions [2]. The random search nature of these algorithms causes them to have long running times, while their solutions often yield little insight for the designer as to how the system may be changed for the better. Iterative improvement algorithms have been found to yield suboptimal solutions [4] [11]. The greedy nature of these algorithms makes their final solution dependent upon the quality of the initial solution. Constructive algorithms, unlike stochastic search and iterative improvement algorithms, incrementally build a solution. Despite a lack of global knowledge, constructive algorithms have

0-7695-0843-X/00 $10.00 © 2000 IEEE

been found to yield good solutions [1] with a reasonable execution time. Although many constructive algorithms perform software task scheduling while the solution is being built, task scheduling is never the primary problem to be solved.

In contrast to prior hardware-software partitioning problem formulations that emphasize the allocation of tasks, our approach, referred to as SHaPES (Software-Hardware Partitioning for Embedded Systems), simultaneously solves the allocation and scheduling subproblems as a pure deterministic real-time scheduling problem. This scheduling problem formulation assumes that the tasks are given as input in the form of a set of periodic and sporadic real-time tasks.

Our real-time deterministic scheduling problem formulation schedules instances of the tasks through the Least Common Multiple (LCM) of their periods. The resultant deterministic schedule provides valuable feedback to the designer, allowing system timing problems and other system design problems to be identified. Furthermore, the generality of this approach allows the hardware-software partitioning problem to be solved at any level of abstraction and at any level of functional granularity.

The remainder of this paper is organized as follows. Section 2 describes our deterministic scheduling problem formulation and how it is related to the hardware-software partitioning problem. The algorithm for solving the scheduling problem is presented in Section 3, and results are presented in Section 4. Section 5 presents our conclusions and directions for future research.

2 Formulating the Scheduling Problem

We formulate the allocation and scheduling subproblems of the hardware-software partitioning problem as a scheduling with rejection problem that is capable of scheduling periodic and sporadic real-time tasks. We assume the SOC target architecture consisting of a single microprocessor to execute the software functionality and a single Application Specific Integrated Circuit (ASIC) to implement the hardware functionality. The system functionality is given as a set of tasks, where a task is to be implemented completely in either hardware or software.

2.1 Modeling Implementation Costs

The main system implementation costs of the SOC target architecture are due to hardware area, power consumption, and timing constraints. Much of the hardware elements, such as the microprocessor and input/output devices, have a fixed size. The memory requirements of the software tasks and the ASIC area requirement of the hardware tasks are the primary variable area requirements. The power consumption is primarily a function of the clock frequency (the microprocessor and the ASIC share the same clock) and the power supply voltage. Slower clock frequencies reduce the switching activity and allow for lower power voltages (and reduced power consumption) but at the cost of increasing the time required to execute the tasks. The timing requirements for the system functionality address system performance and feasibility issues. Timing constraints on the functionality of the system are violated at some cost, and timing constraint violations must not make the system infeasible (such violations have an infinite cost associated with them).

A scheduling problem is able to model some, but not all, of the implementation costs. Size limitations, such as software memory constraints and system power constraints, cannot be modeled by a scheduling problem. Instead, these constraints must be checked after a schedule has been generated. Thus, a pure scheduling problem formulation must be used in an iterative methodology when hard size and power constraints exist.

While a scheduling problem is not able to model hard size and power constraints, it is capable of modeling hard timing constraints. To model these constraints, the timing requirements are specified using four task parameters: processing time p_j, release time r_j, deadline d_j, and weight w_j. The processing time p_j indicates how long the task will take to complete if it executes without interruption on the microprocessor. The release time r_j indicates the first time at which the task may begin execution. The deadline d_j indicates the time by which the task should be finished, and the weight w_j indicates the importance of the task. Violating the release time of a task in some schedule incurs an infinite cost, while violating the deadline of a task incurs a cost that is a function $f(w_j, C_j)$ of the completion time C_j of the task in the schedule and the weight w_j of the task.

Under this model, scheduling a task corresponds to implementing the task in software. Furthermore, the task incurs no cost if it is implemented in software and completes by its deadline. It follows that not scheduling a task, or rejecting the task, corresponds to implementing the task in hardware. Since we assume that the hardware is 'fast enough,' timing constraints are always met by rejecting the tasks. To prevent all tasks from being implemented in hardware, rejecting a task

incurs some rejection cost e_j. Thus, the hardware requirements are modeled using a single parameter: the rejection cost e_j.

2.2 The Scheduling Problem Formulation

Given the five task parameters, a scheduling with rejection problem formulation can be stated for the hardware-software partitioning problem.

> Given a set of n tasks $\mathcal{T} = \{1, 2, \ldots, n\}$, each with a processing time p_j, a release time r_j, a deadline d_j, a weight w_j, and a rejection cost e_j, schedule a subset $S \subseteq \mathcal{T}$ of the tasks on a single microprocessor such that the objective function $f(w_j, C_j) = (\sum_{j \in S} w_j T_j + \sum_{j \in \bar{S}} e_j)$ is minimized, where $\bar{S} = \mathcal{T} - S$ and $T_j = \max\{0, C_j - d_j\}$.

A solution to this scheduling problem allocates the scheduled tasks S to software and the rejected tasks $\bar{S}$ to hardware. Furthermore, the deterministic schedule may be examined to identify areas of the design that can be improved.

This problem formulation may be extended by the addition of *precedence* constraints between tasks. A precedence constraint between task i and task j, $i \prec j$, requires task i and all of its predecessors to complete execution (or be rejected) before task j and all of its successors begin execution in the schedule. Precedence constraints arise from data dependencies and control dependencies between tasks.

Communication delays between tasks are modeled with the use of *separation* constraints. A separation constraint k is associated only with a precedence constraint $i \prec j$, and it requires that task i and all of its predecessors complete execution (or be rejected) at least k time units before task j and all of its successors begin execution in the schedule. Communication delays arise between tasks implemented in different partitions.

3 Solving the Scheduling Problem

Given a set of tasks, the scheduling with rejection problem is solved to create a solution to the partitioning problem. Efficient algorithms exist to solve scheduling problems closely related to our problem formulation. In this section, we review the most effective algorithm for the weighted tardiness objective function and describe how it is extended to handle rejection and separation constraints.

3.1 The Apparent Tardiness Cost

Scheduling a set of jobs so as to minimize their total weighted tardiness is a problem that has been actively investigated for more than thirty years. The continuing interest in this problem stems from its accurate modeling of the manufacturing problem where a set of jobs must be completed by their respective deadlines, and each job incurs some penalty if it is tardy. The practical versions of this problem are $\mathcal{NP}$-hard [3]. Hence, a considerable number of heuristic algorithms have been proposed to solve scheduling problems involving the tardiness objective function. Most of these algorithms are based upon a greedy constructive algorithm differing only in their dispatch rules.

The simplest dispatch rules, such as Earliest Due Date (EDD) first and Shortest Processing Time (SPT) first, have been found to yield near optimal solutions only under certain conditions. To overcome the deficiencies in these simple dispatch rules, more complex dispatching rules have been developed. The most successful of these heuristics is the Apparent Tardiness Cost (ATC) rule introduced by Rachamadugu and Morton [5].

The ATC rule is based on the structure of an optimal schedule when no precedence constraints exist between tasks. In these schedules, the tasks are sequenced in non-increasing priority order where the priority of task i is equal to

$$\frac{w_i}{p_i}\left(1 - \frac{\max\{0, (d_i - t - p_i)\}}{p_j}\right),$$

where p_j is the processing time of the task j scheduled immediately after task i.

Instead of trading off the slack of task i against the processing time of task j, the ATC rule uses a standard reference. A piecewise linear reference may be obtained by replacing the unknown p_j in job i's priority by a factor kp, where p is the mean processing time of the unscheduled tasks and k is a look-ahead parameter related to the number of competing tardy or near-tardy tasks. However, an inverse of allowance is actually closer to the 'apparent cost' of tardiness implied by the break-even priority of tardy tasks with processing times exceeding their slack. With this in mind, the ATC dispatching rule is defined as

$$ATC_j(t) = \frac{w_j}{p_j} \exp\left(-\frac{\max\{0, (d_j - t - p_j)\}}{kp}\right).$$

Intuitively, the exponential look-ahead works by ensuring timely completion of short tasks (with a steep

increase of priority close to its deadline), and by extending the look-ahead far enough to prevent long tardy tasks from overshadowing clusters of shorter tasks. The look-ahead parameter can be adjusted based on the expected number of competing tasks to reduce weighted tardiness costs during high processor load. Experiments have found that a reasonable range of values for k is $1.5 \leq k \leq 4.5$ with $k = 2$ yielding good results over a wide range of load conditions [10].

Empirical experiments have found that the ATC rule yields close to optimal schedules for single machine schedules [5] and outperforms all other dispatch rules for multiple machine schedules [10]. *Additionally, the ATC dispatch rule has been found to be robust in the presence of errors in the estimated processing times of the tasks [9].* The robustness of the ATC dispatch rule in the presence of errors in processing time estimates is essential for its use in solving our scheduling with rejection problem formulation.

3.2 Inserted Idleness

Simply using the ATC dispatching rule yields a non-preemptive schedule without any inserted idle time. However, in the presence of release times, or, similarly, separation constraints, allowing inserted idle time can yield better schedules with minimal additional computational expense. Morton and Ramnath [6] showed that for all problem instances and for any regular objective function, including the (weighted) tardiness objective function, there exists an optimal schedule such that no job is scheduled next on a given machine unless its release time is at most the current time plus the processing time of the shortest job that was released by the current time. Based on this fact, they proposed a modification of the ATC rule for the single machine problem. The priorities of the jobs are multiplied by a penalty proportional to the inserted idleness caused by scheduling that job next. In this way, the set of candidate jobs to be scheduled next is extended to include jobs that will arrive in the near future.

The priorities of the yet to be released jobs are reduced proportional to the idleness that would be incurred by scheduling them next. The proportionality multiplier α may be a constant, or it may be variable to allow it to increase linearly with the machine utilization as suggested by Morton and Ramnath [6]. The ATC rule that allows inserted idle time is then defined as

$$ATC_j(t)' = ATC_j(t)\left(1.0 - \alpha\frac{\max\{0,(r_j - t)\}}{p_{\min}}\right),$$

where $p_{\min}$ is the processing time of the shortest job that is ready at time t. This new ATC rule degrades the original ATC priority by a term proportional to the induced idleness as a fraction of the minimum of the processing times of the waiting jobs. If the reduced priority of a yet to be released task is greater than all other task priorities, then the machine is kept idle until this job is released. We use this dispatching rule with a constant proportionality multiplier to allow for inserted idleness in the algorithm described in the following section.

3.3 The Scheduling Algorithm

Our greedy constructive algorithm to generate a solution to our scheduling with rejection problem is simply stated as follows. At each time t that the processor becomes free, compute $ATC_j(t)'$ for all tasks j that are ready to execute at time t or become ready to execute during the interval $t + p_{\min}$, where $p_{\min}$ is the minimum processing time of the tasks that are ready to execute at time t. Let i be the task with the largest computed ATC. Let $z_i = w_i T_i$ be the weighted tardiness of task i if it is scheduled as soon as possible at or after time t. If $z_i \geq e_i$, then reject task i and repeat the task selection process at time t; otherwise, schedule task i as soon as possible at or after time t. Let the completion time of task i be the new current time t. Repeat until all jobs have been either scheduled or rejected. This algorithm runs in time $\mathcal{O}(n^2)$.

4 Experimental Results

SHaPES has been implemented as a prototype in Java and applied to several examples from the literature. These examples schedule a given task graph on a subset of given Processing Elements (PEs). Although SHaPES assumes a target architecture consisting of a single processor and a single ASIC, it can be applied to these systems in a straightforward manner. Choose an initial PE and call it the 'processor.' Run SHaPES with this target processor. The solution to this problem yields a scheduled set of tasks that are to be implemented on the chosen PE and a rejected set of tasks. The release times and deadlines of the rejected tasks can be recalculated based on their dependencies with the scheduled tasks. The rejected tasks then form the input to a second iteration of the scheduling with rejection problem formulation. A second PE is chosen to act as the processor, and the scheduling problem is solved with only the rejected tasks as input. This process is repeated until either no feasible schedule can be

Table 1: Prakash and Parker's examples.

Example (makespan)	No. of Tasks	SOS		MOGAC		Oh & Ha		SHaPES	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
Prakash & Parker 1 (4)	4	7	28	7	3.3	7	0.01	7	0.026
Prakash & Parker 1 (7)	4	5	37	5	2.1	5	0.01	5	0.026
Prakash & Parker 2 (8)	9	7	4,511	7	2.1	7	0.01	7	0.014
Prakash & Parker 2 (15)	9	5	385,012	5	2.3	5	0.01	5	0.014

Table 2: Hou's examples.

Example (clustering)	No. of Tasks	Yen		MOGAC		Oh & Ha		SHaPES	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
Hou 1 & 2 (u)	20	170	10,205.3	170	5.7	170	0.1	170	0.083
Hou 1 & 3 (u)	20	240	11,550.2	N.A.	N.A.	170	0.1	170	0.084
Hou 3 & 4 (u)	20	210	7,135.0	170	8.0	170	0.1	170	0.083
Hou 1 & 2 (c)	8	170	14.96	170	5.1	170	0.01	170	0.045
Hou 1 & 3 (c)	7	170	4.92	N.A.	N.A.	N.A.	N.A.	170	0.042
Hou 3 & 4 (c)	6	170	3.34	170	2.2	N.A.	N.A.	170	0.035

found, or all tasks have been scheduled on some PE.

Our results were obtained on a 300 MHz Pentium II system with 128 MB of main memory running the Windows NT 4.0 operating system and Java JDK 1.2. We compare our results with those of MOGAC [2], Oh and Ha [7], SOS [8], and Yen and Wolf [11]. MOGAC was implemented in C++, and its results were obtained on a 200 MHz Pentium Pro system with 96 MB of main memory running the Linux operating system. Oh and Ha [7] implemented their algorithm in C++, and their results were obtained on an Ultrasparc I with a 200 MHz processor and 256 MB of main memory. SOS's results were obtained on a Solbourne Series5e/900 (similar to a SPARC 4/490) with 128 MB of main memory. Yen and Wolf [11] implemented their algorithm in C++, and their results were obtained on a Sun Sparcstation SS20.

Table 1 compares the performance of SHaPES to that of SOS, MOGAC, and Oh and Ha when they are applied to Prakash and Parker's task graphs [8]. The performance number shown by each task graph is the worst-case finish time, or makespan, of the task graph. For example, "Prakash & Parker 1 (4)," refers to Prakash and Parker's first task graph with a makespan of 4 time units. The cost of a solution is determined by the price of the PEs used in the solution, plus 1 for each communication link required.

Table 2 compares the performance of SHaPES to that of Yen's system, MOGAC, and Oh and Ha's approach when each is run on the clustered and unclustered versions of Hou's task graphs [4]. Hou ran Yen's system on the clustered and unclustered versions of his graphs. We use the same clusters as Hou, MOGAC, and Oh and Ha when comparing our results with theirs.

Table 3 compares the performance of SHaPES to that of Yen's system, MOGAC, and Oh and Ha's approach when each is applied to Yen's large random task graphs [11].

Table 4 compares the performance of SHaPES to that of MOGAC and Oh and Ha's approach when each is applied to MOGAC's very large random task graphs [2]. MOGAC's random 1 contains eight independent task graphs, each containing approximately sixty-three tasks. There are eight PE types and five link types. MOGAC's random 2 contains ten independent task graphs, each containing approximately ninety-nine tasks. There are twenty PE types and ten link types.

For all of the examples, SHaPES was able to determine the optimal solution. It was able to do this despite the fact that it was designed assuming the SOC target architecture, not the multiple PE target architectures of the examples. The simplicity of the SHaPES approach is apparent in the extremely short running times required to solve the examples. The running time of 2.486 seconds required to find the optimal solution to MOGAC's Random 2 example illus-

Table 3: Yen's large random examples.

Example	No. of Tasks	Yen		MOGAC		Oh & Ha		SHaPES	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
Yen Random 1	50	281	10,252	75	6.4	51	2.1	51	0.699
Yen Random 2	60	637	21,979	81	7.8	81	3.6	81	0.826

Table 4: MOGAC's very large random examples.

Example	No. of Tasks	MOGAC		Oh & Ha		SHaPES	
		Cost	CPU time (s)	Cost	CPU time (s)	Cost	CPU time (s)
MOGAC Random 1	510	39	2,454	39	17.6	39	1.302
MOGAC Random 2	990	35	12,210	13	299.8	13	2.486

trates the applicability of SHaPES to large examples. It also illustrates the applicability of SHaPES as a fast analysis tool for the designer.

5 Conclusions

We have presented a new approach to solving the hardware-software partitioning problem in embedded system design. Formulating both the allocation and scheduling subproblems of the hardware-software partitioning problem as a scheduling with rejection problem forms the cornerstone of SHaPES. SHaPES is amenable to multi-rate, real-time systems and systems targetting complex hardware architectures.

The SHaPES approach to solving the hardware-software partitioning problem is a simple, fast, and effective approach to performing automated analysis of a system design and quick exploration of the solution space.

References

[1] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *Proceedings of the 34th Design Automation Conference*, pages 703–708, June 1997.

[2] R. P. Dick and N. K. Jha. MOGAC: A multiobjective genetic algorithm for hardware-software co-synthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):920–935, October 1998.

[3] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[4] J. Hou and W. Wolf. Process partitioning for distributed embedded systems. In *Proceedings of the Fourth International Workshop on Hardware/Software Codesign*, pages 70–76, March 1996.

[5] T. E. Morton and R. M. V. Rachamadugu. Myopic heuristics for the single machine weighted tardiness problem. Technical Report CMU-RI-TR-83-09, Robotics Institute, Carnegie Mellon University, November 1982.

[6] T. E. Morton and P. Ramnath. Guided forward tabu/beam search for scheduling very large dynamic job shops, i. Technical Report 1992-47, Graduate School of Industrial Administration, Carnegie Mellon University, 1992.

[7] H. Oh and S. Ha. A hardware-software cosynthesis technique based on heterogeneous multiprocessor scheduling. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, pages 183–187, May 1999.

[8] S. Prakash and A. C. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–351, 1992.

[9] A. P. Vepsalainen. State dependent priority rules for scheduling. Technical Report CMU-RI-TR-84-19, The Robotics Institute, Carnegie-Mellon University, 1984.

[10] A. P. Vepsalainen and T. E. Morton. Priority rules for job shops with weighted tardiness costs. *Management Science*, 33(8):1035–1047, August 1987.

[11] T.-Y. Yen. *Hardware-Software Co-Synthesis of Distributed Embedded Systems.* PhD thesis, Princeton University, June 1996.