A Time slice based scheduler model for System Level Design

Luciano Lavagno, Claudio Passerone, Vishal Shah
Politecnico di Torino – Dept. Electronics
10129 Torino - Italy
{vishal.shah,passerone,lavagno}@polito.it

Yosinori Watanabe
Cadence Berkeley Laboratories
Berkeley, CA, USA
watanabe@cadence.com

Abstract

Efficient evaluation of design choices, in terms of selection of algorithms to be implemented as hardware or software, and finding an optimal hw/sw design mix is an important requirement in the design flow of Embedded Systems. Time-to-market, faster upgradability and flexibility are some of the driving points to put increasing amounts of functionality as software executed on general purpose processing elements. In this scenario, dividing a monolithic task into multiple interacting tasks, and scheduling them on limited processing elements has become very important for a system designer. This paper presents an approach to model time-slice based task schedulers in the designs where the performance estimate of hardware and software models is less than time-slice accurate. The approach aims to increase the simulation efficiency of designs modeled at system level. We used Metropolis [1] as our codesign environment.

1. Introduction

One of the main objectives of an effective hw/sw design flow is to provide the system designer with the ability to model a system at varying levels of abstraction, and with tools and techniques to analyze and verify his models. Integrated simulation of hardware and software components is an important step of a codesign flow as it allows a designer to explore the design space and evaluate the performance of an algorithm on a particular hw/sw partition choice. Changing from one partitioning decision and algorithm implementation to another is less time consuming if the models are still abstract and less detailed, which is usually the case early on in the design stages. Hence codesign tools from companies such as Mentor Graphics, Cadence, Synopsys and CoWare, and from university groups such as Roses, Metropolis, and others, support hw/sw cosimulations at various levels of abstraction.

Raising the abstraction level makes both design activities, such as creating and changing components of a design, and verification activities, such as simulation and formal verification, much faster. It is generally considered the only means to cope with the increased number of gates that Moore's law offers to designers whose number and whose productivity in terms of "design objects per day" are almost constant over time. Debugging and optimization become much easier when the number of objects is reduced, as long as "referential transparency" is preserved, meaning that design decisions have an obvious and controllable impact on the quality of the final results. Higher abstraction levels also increase the modularity of design, by reducing the number of implementation details, and thus further increase productivity by means of re-use. Designers can focus on functionality and constraints, rather than on implementation details.

In this paper we focus on a design approach based on the separation of concerns [2, 3], in which high-level functional models are annotated, by hand or automatically, with the results of mapping decision to a variety of architectural components, such as processors, buses, memories, standard cells, IP blocks, FPGAs, and so on. Annotating performance directly at the algorithmic level makes design space exploration much faster, even though often such high-level time and power estimates are only approximate. This can however help identify algorithmic and communication blocks which are bottlenecks in a given mapping choice, and hence drive the designer to better solutions.

The goal of the paper is to show that one need not annotate performance information at a very fine-grained level (e.g. for software, instruction level or below), but only coarser-grained information is really needed to make choices with a good level of fidelity. We do so by using as an example the Metropolis modeling and simulation environment [3], and by focusing on the representation of the functionality and performance of a set of software tasks mapped to a processor under control of a time-sliced real-time executive. While the results are specific to this application, the techniques and the conclusions that we can draw from this exercise are much broader, and can be extended to other types of software execution environments and hardware architectures.

2. Related Work

To cope up with the pressures of time-to-market and design complexity of hw/sw systems, several attempts have been made to raise the level of modeling abstraction [1, 4, 5]. However, a design with abstract models introduces several limitations on the accuracy of performance estimates. Scheduling policies tend to have a significant effect on the performance of a system and hence, capturing scheduler overheads in system level design is important.

In the past, there have been several efforts expended in the direction of modeling schedulers, preemptive and otherwise, for coarse grain models of software tasks [2, 6, 7, 8]. The scheduler model we present, is closest to [8] and it can be seen as an optimization to it for the case of modeling time-slice based scheduling policies. While the authors of [8] describe the scheduler model, they do not detail the approach to be used for time-slice based scheduling and dynamic scheduling in general.

Polis [2] represents each component of a design using a finite state machine based representation. It can then synthesize an RTOS that schedules the models of software tasks on the hardware model. While it can generate time-slice based schedulers, it does not provide the designer any control over modifying the behavior of a particular scheduler, and hence the designer cannot exploit the efficiency that is on offer by models of software that can be less than time-slice accurate. A similar approach is adopted in [7]. Yi et.al. [6] annotate a software model with a preemptive scheduler overhead by running the software tasks on an instruction set simulator. While this increases accuracy and is helpful towards the final stages of the design, it is of no help when a designer only needs a rough estimate on his software models also capturing the scheduler overhead.

The basic advantage that our model aims to capture is to exploit the level of abstraction, both of hardware and software models, available in System Level Design frameworks and use it to improve simulation efficiency. We used Metropolis because it gave us an explicit control over how a scheduler is modeled and how it interfaces with the hardware and software components modeled at varying levels of abstraction.

3. The Metropolis Framework

Metropolis [1] is a system-level design infrastructure based on a model with precise semantics that is general enough to support existing computation models and accommodate new ones. It can represent all the key ingredients in the design flows: function, architecture, mapping, refinement, abstraction and platforms. To allow better reuse of models, the focus is on separation of independent aspects, such as:

- Computation and communication.
- Functionality and architecture.
- Behavior and performance.

The system to be developed is hierarchically decomposed into several objects, with two distinct networks at the top level. The *scheduled network*, which contains all blocks related to function, architecture and mapping, and the *scheduling network*, which is used to drive the execution of the scheduled network and assign cost and performance using special objects called *quantity managers*.

Function is defined by a network of concurrent processes that communicate through communication media (similar to SystemC and SpecC channels). Processes and media are both described using a sequential language called *metamodel*, which is similar to Java^{TM}, but has been extended to support modular specifications using ports and interfaces. The behavior of the network is a sequence of event vectors, where each event in a vector represents the execution of a computation or communication step (e.g. an assignment, or reading a shared memory location, or writing a FIFO queue) by a particular process.

The architecture provides computation and communication services to the function. It is a network of processes and media as well, and services are specified using the same metamodel used in the function side. Services are decomposed into a sequence of events, and each event can be annotated with a value representing its cost and performance, in terms of a number of physical and abstract quantities.

Mapping is used to associate functions to services provided by the architecture. It is achieved by synchronizing events occurring within function processes with events occurring in the architecture. The annotations defined for the architectural events are thus inherited by those in the function side, allowing to determine the costs and performance of the entire system.

Performance annotation and the modeling of scheduling policies is done using the abstraction of *quantity managers*. Figure 1 shows a high level anatomy of a quantity manager. Its input can be seen as a set of annotation requests from the software tasks, queued in using the *request(...)* method, and its output can be seen as either a single result or a sequence, indicating the execution order of the tasks, constructed by the *resolve(...)* method.

Simulation of a hw/sw design is divided into two phases, the *request phase* and the *resolve phase*. In the *request phase*, the software tasks (as well as the hardware-mapped processes, which are however outside the scope of this paper) instantiated in the *scheduled network* execute their behavioral description till they reach a point where a request for performance annotation is made. At this point they will generate an event and queue in their annotation request to the quantity managers which control the physical quantity that

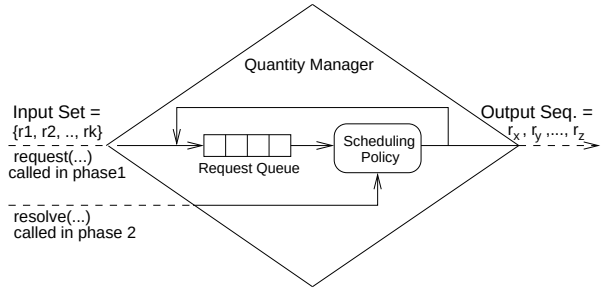

Figure 1. Quantity Manager Anatomy

the tasks want to be annotated with. After all the tasks have made such requests, the second phase, the *resolve phase* starts. In this phase a quantity manager will have a set of annotation requests from the tasks. Based on its policies, the quantity manager will annotate the selected requests and notify the owner tasks. The request phase now starts again. The notified tasks will further execute their behavioral descriptions till they reach the next annotation point. This alternation between the *request* and *resolve* phase is comparable to the two-stage simulation cycle semantics of VHDL simulators.

Figure 2 shows a design that contains more than one quantity managers. It models a shared bus architecture. As seen in this case, one quantity manager might need to know the decision of other quantity managers before it can finalize on its own decision. In our example, as the bus is shared between Cpu1 and Cpu2, before a CpuQM schedules a task it has to know whether it has the bus ownership. In such cases, the resolve phase might have to be run iteratively for an implicit communication between the quantity managers. In the interest of a modular, loosely coupled design specification, the modeling semantics of quantity managers do not allow them to communicate with each other explicitly. An iterative execution of the resolution phase allows all the quantity managers to reach a mutually agreeable decision. For instance, when Task A and Task C request cpu functionality which also needs bus access, in the first execution of the resolution phase each quantity manager, in our case the bus-time quantity manager and the two cpu-time quantity managers, determine their decision. In the next execution of the resolution phase each cpu-time quantity manager can check if the decision of the bus-time quantity manager conflicts with their own decision. If it does, the cpu-time quantity manager can be modeled to do a rollback or make a different choice and the resolution phase can be run again.

The main reason for using Metropolis as our design framework was the ease and flexibility it gives in design space exploration and rapid systems prototyping, and its modeling abstractions being based on formal execution semantics, all key requirements for Embedded Systems design.

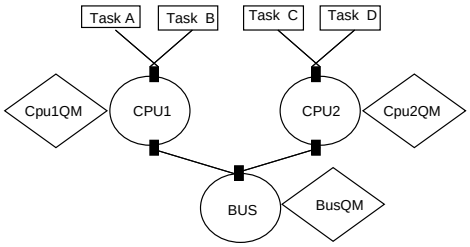

Figure 2. Shared Bus Architecture

4. The Scheduler Model

Our approach to model schedulers is based on the principle of exploiting the availability of abstract hardware and software models in system level design frameworks. It aims to increase the simulation efficiency by reducing the amount synchronization required between a scheduled network and a scheduling network. It should be noted that, though we discuss our modeling technique in regard to Metropolis framework, it is general enough to be applied to any simulation environment that is based on two phase execution semantics between functionality and scheduling. To generalize, a scheduled network can be seen just as a group of components capturing functionality in the system and the scheduling network can be seen as a group of components capturing the scheduling aspects of the system. Ptolemy [5], is one such environment with analogous modeling and execution semantics in terms of *actors* and *directors*.

In the following we will assume that the performance of application software tasks is modeled at the level of basic blocks. Several methods have been proposed to estimate the CPU cycles required for an instruction [9, 10, 11]. Using these methods, a designer can construct a model for software, in which the cycle count of all the instructions in a basic block is aggregated, and a single request for the time annotation of this aggregated amount of cpu cycles is made to the quantity manager. It should be noted that the aggregated cycle count for a basic block is not yet annotated with the overhead caused by scheduling effects, it is the quantity manager model that does this annotation.

In this research we focused on modeling a time-slice based task scheduler, improving simulation efficiency when the annotation granularity of the processes to be executed is coarser than the time-slice, as can happen in the case where the designer wants to annotate software performance at the basic-block level. The model is applicable to different kind of shared resources, such as buses and memories with burst-mode and DMA access, but in the following we will concentrate, without loss of generality, only on CPUs.

Our goal is to avoid starting a new resolution phase each time a time-slice elapses, otherwise we would lose possible gains in simulation performance that can be achieved by using coarser annotations at the process level. When the annotation granularity is fine, e.g. at the instruction level, our technique, while still applicable, does not provide significant advantages, since the performance bottleneck is created by the high-detailed models.

Algorithm 1 details the resolution mechanism that we implemented in a Metropolis quantity manager. At the start of each resolve phase all quantity managers instantiated in the *scheduling network* will have a set of requests. These requests are from the software tasks that wish to use the resource controlled by a quantity manager. Based on the scheduling policy modeled, the quantity manager will decide which task should be allowed to use the resource. *Step 1* of the algorithm selects the request ***R*** from the input set ***X***, containing the task requests.

Our quantity manager model, instead of choosing a *single* software task as its scheduling decision, will choose a *sequence* of tasks. This sequence corresponds to the alternation of execution slices of the various tasks mapped to the shared resource and it is this sequence that captures the time-sliced execution of the tasks. The sequence stops, and the resolve() method terminates, when the last slice selected completely fulfills the request of a task. This task is signaled to proceed, while the other tasks in the sequence stay suspended since they need more slices to finish.

Step 2 models this. It first checks whether the resource amount requested by the selected software task exceeds the *time_slice* defined for the resource. If it does then the request is updated to reflect that the task was alloted resource amount = *time_slice*, and is put back in the request set ***X***. It then records the selected task in a sequence ***Y*** and the control goes back to step 1. When a task is selected whose requested resource amount $<=$ *time_slice*, the task is recorded in the sequence ***Y*** and control is transferred to *Step 3*, which exits with ***Y*** as its scheduling decision.

If $t_1, t_2, ..., t_n$ denote ***n*** software tasks respectively, and the value of ***Y*** is t_1, t_2, t_1, t_2, t_1 then the scheduling decision can be seen as, t_1 got the entire amount of resource it had requested for and that it can proceed, while t_2 got resource amounting to two time slices and is waiting to get more resource. In reaching this scheduling decision, t_1 and t_2 were each preempted twice. The value of ***Y*** represents the execution order of tasks. In this case the order of preemption of tasks is t_1, t_2, t_1, t_2.

4.1. Example

Consider the hw/sw model of Figure 3. Here 3 tasks *A*, *B* and *C* request cpu cycles. Suppose *A* requests 10 cycles, *B* 20 and *C* 30 cycles and the time-slice for using the cpu is defined as 10 cycles. Figure 4(a) shows the state of request queue of cpu time quantity manager after the execution of *phase 1* . The number next to each task name indicates the number of cycles the task is waiting to be allocated. In this case, our model of preemptive scheduler, based on FCFS selection policy, will allocate first 10 and the next 10 cycles to *C* and *B* respectively, and then put them back in the queue since their entire quantity request has not been filled. Figure 4(b-d) show the state of the queue at the end of each time slice. At the end of 3 time slices the scheduler has with it at least one request, that of task *A* whose entire request for quantity has been satisfied. The cpu quantity manager will remove the request of task *A* from the queue and transfer the execution control to it. Figure 4(d) shows the status of the queue after this scheduling decision.

It should be noted here that in theory, control should have transferred to tasks *C* and *B* at the end of 10 and 20 cpu cycles respectively, but the coarseness of the annotation requested by the designer did not make this a requirement. If the designer wants control to be transferred to the software task at the expiry of each time slice, his annotation requests should be always smaller than the time slice defined for the resource. This allows him to introduce more details in his

Algorithm 1: Select resource owners

Input: Set ***X*** of requests from software tasks

Output: Sequence ***Y*** representing the execution order of software tasks

1. ***R*** = remove(***X***, selection_policy)
   where selection_policy = FCFS, priority based, etc.
2. ***if*** requested_amount(***R***) $>$ time_slice
   requested_amount(***R***) =
   requested_amount(***R***) - time_slice
   insert(***X***, ***R***)
   insert(***Y***, task_owner(***R***))
   go to 1
   else
   insert(***Y***, task_owner(***R***))
   go to 3
3. return ***Y***

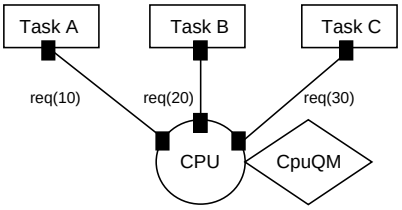

Figure 3. Shared Cpu

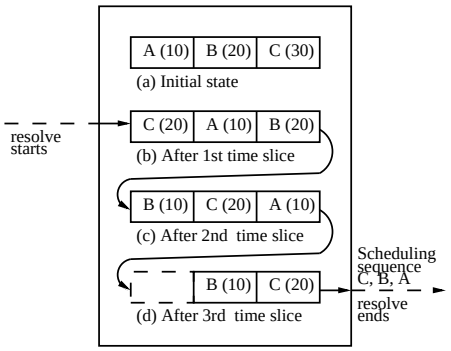

Figure 4. CpuQM Request Queue state

model at the cost of simulation efficiency. Our quantity manager can handle this case without requiring any changes in the modeling code.

4.2. Limitations

This approach to modeling time-slice based scheduling, while reducing the number of calls to the scheduling network, introduces some limitations. The foremost limitation that we can see is that we cannot directly extend this approach to handle preemptions caused by unpredictable events, like when the occurrences of non periodic interrupts are to be modeled. This is because, to exploit the technique, the quantity manager should be able to make assumptions on when to assign resources that are requested. This is clearly impossible when the instant in which the event occurs is not known in advance. The solution, that was first discussed in [2], is to modify the Quantity Manager for the CPU so that it does not allow any requests to be made to the Global Time Manager (another quantity manager in Metropolis) until it is known that global time has reached that point, and hence no preemption can occur for the executing task any longer. In case the task gets preempted before, such pending requests are postponed in time, by the amount corresponding to the duration of the preempting task (and possibly of other tasks preempting it).

Also in a system modeling multiple schedulers as in Figure 2, where there exists dependencies between scheduling decisions of each scheduler, in order to reach a global stable state of the system at the end of each time-slice, a scheduler might need to know the decision of scheduler lower in the hierarchy before it can finalize its own decision. While this increases the accuracy, it does introduce significant amounts of overhead as we show in the experimental results. For the example in Figure 2, a CPU scheduler before deciding on a result sequence will need to know the scheduling decision of the bus scheduler at the end of each time slice. This is because each run of the *resolve()* method of the time-slice scheduler outputs a sequence of task executions, rather than a single task as its result. Therefore to achieve results accurate at time-slice level the resolution of bus requests from the very same tasks should follow an equivalent execution order for the two schedulers to be consistent.

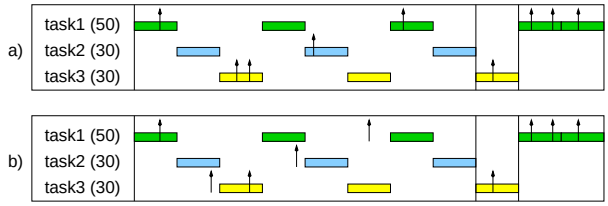

Figure 5. CPU and bus scheduler (a) synchronized, (b) not synchronized

In case the synchronization between schedulers cannot be achieved, either because the models do not support it, or because it may imply reduced simulation performance, the events scheduled in the architecture might be in conflict between each other. For instance, the trace of the simulation might show a bus access from a process that has made a request for CPU cycles while it is not actually running on the CPU, because the current slice is assigned to another process. If one is just interested in aspects not involving the perfect synchronization of the events, such as the total bus load or its switching activity, then the model may still valid and the results that are obtained are often correct, albeit approximate. For a more refined simulation at a lower level of abstraction, on the other hand, the models should allow the information sharing between schedulers, or the time-slice scheduler should limit the sequences always to a single slice.

Figure 5 shows an example of what happens when two schedulers are synchronized, and when they are not. It shows a Gantt chart of three processes, that have requested a certain amount of CPU cycles, shown in parenthesis. The arrows in the chart represent bus accesses. In both cases, three runs of the *resolve()* method of the time-slice scheduler are shown. In Figure 5-(a), the bus and the CPU scheduler are synchronized; thus all bus accesses happen when the corresponding process is actually running on the processor. On the other hand, in Figure 5-(b), the two schedulers are not synchronized. Without any other information, the bus scheduler will schedule accesses to the bus at its own convenience (typically earlier than the previous case), so the result is inconsistent. However, the number of bus accesses is the same in both cases.

This model trades off these limitations with the efficiency it can achieve at the simulation level, without compromising on the accuracy of the functional behavior.

Design Environment	% reduction in simulation time
Metropolis	23
SpecC	27

Table 1.

5. Experimental Results

In this section we compare the results obtained by two time-slice accurate scheduler models, *S1* which optimizes the simulation performance, using the technique presented in Section 4, and model *S2*, that is based on a naive approach of invoking the scheduler at the end of each time-slice.

The system we model and simulate has 4 software tasks. Each of them models a behavior shown in Figure 6. It reads a 1000 data words from memory, does some computation on it and then writes back a 1000 words to memory.

```
Process   {
    for (i=0 to 1000) {
        read (1000);
        compute();
        write (1000);
    }
}
```

```
Cpu {
    load (n words) { annotate with
                        n*estimate_for_1_word_memory_load;}
    store (n words) { annotate with
                        n*estimage_for_1_word_memory_store;}
}
```

Figure 6. Process and Cpu Model

On the architecture side, we have a cpu model that has performance estimates for load and store instructions. Our RTOS model only contains the scheduler. To simulate this hardware software combination, we map the *read* and the *write* instructions of the software tasks to the *load* and *store* instructions of the cpu.

In order to illustrate the generality of our approach, we also model the same system specification in SpecC environment, where we model the 4 software tasks, the Cpu and the Scheduler as a network of behaviors communicating by exchanging events.

As Table 1 shows, in the Metropolis framework, the scheduler model *S1* outperformed model *S2* by reducing the simulation time by upto 23%, and in the SpecC framework by 27%. The difference in the performance numbers on the two environments can be attributed to different modeling mechanisms and code generation backends. We achieve performance improvements in simulation time by decreasing the number of scheduler invocations. The results are reported on the particular hardware/software combination we modeled. Depending on the coarseness or the fineness of the models the results may vary and one can achieve more or less improvements.

6. Conclusion

System Level Design facilitates modeling at high-level of abstraction as well as mixing highly detailed and low detailed models. There is a lot of opportunity to exploit the presence of abstract models in order to improve the performance of design tools. This paper presented an approach to model time-slice based schedulers in such environments exploiting this advantage. Our future work aims to address the limitations of this technique.

References

[1] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe. Metropolis: An integrated environment for electronic system design. *IEEE Computer*, 36(4):45–52, April 2003.

[2] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli. *Hardware-Software Co-Design of Embedded Systems - The Polis Approach*. Kluwer Academic Publishers, 1997.

[3] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogenous systems. In Jordi Cortadella, Alex Yakovlev, and Grzegorz Rozenberg, editors, *Concurrency and Hardware Design*, pages 228–273. Springer, 2002. LNCS2549.

[4] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: specification language and methodology*. Kluwer Academic Publishers, 2000.

[5] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[6] S. Ha Y. Yi, D. Kim. *Fast and Time-Accurate Cosimulation with OS Scheduler Modeling*. Kluwer Academic Publishers, 2003.

[7] J. Cockx. Efficient modeling of preemption in a virtual prototype. In *Proceedings of 11th IEEE International Workshop on Rapid System Prototyping*, pages 14–19, 2000.

[8] Andreas Gerstlauer, Haobo Yu, Daniel D. Gajski. RTOS modeling for system level design. In *Proceedings of the Design Automation and Test in Europe(DATE)*, pages 31–36, March 2003.

[9] S. Malik, M. Martonosi, and Y.T.S. Li. Static timing analysis of embedded software. In *Proceedings of the Design Automation Conference(DAC)*, pages 147–152, June 1997.

[10] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proceedings of the DAC*, pages 605–610, June 1996.

[11] V. Zivojnovic and H. Meyr. Compiled hw/sw co-simulation. In *Proceedings of the DAC*, pages 690–695, June 1996.