

Interrupt Modeling for Efficient High-Level Scheduler Design Space Exploration

F. RYAN JOHNSON
Carnegie Mellon University
and
JOANN M. PAUL
Virginia Tech

Single Chip Heterogeneous Multiprocessors executing a wide variety of software are increasingly common in consumer electronics. Because of the mix of real-time and best effort software across the entire chip, a key design element of these systems is the choice of scheduling strategy. Without task migration, the benefits of single chip processing cannot be fully realized. Previously, high-level modeling environments have not been capable of modeling asynchronous events such as interrupts and preemptive scheduling while preserving the performance benefits of high level simulation. This paper shows how extensions to Modeling Environment for Software and Hardware (MESH) enable precise modeling of these asynchronous events while running more than 1000 times faster than cycle-accurate simulation. We discuss how we achieved this and illustrate its use in modeling preemptive scheduling. We evaluate the potential of migrating running tasks between processors to improve performance in a multimedia cell phone example. We show that by allowing schedulers to rebalance processor loads as new tasks arrive significant performance gains can be achieved over statically partitioned and dynamic scheduling approaches. In our example, we show that system response time can be improved by as much as 1.96 times when a preemptive migratory scheduler is used, despite the overhead incurred by scheduling tasks across multiple processors and transferring state during the migration of running tasks. The contribution of this work is to provide a framework for evaluating preemptive scheduling policies and task migration in a high level simulator, by combining the new ability to model interrupts with dramatically increased efficiency in the high-level modeling of scheduling and communication MESH already provides.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**]: Computer-aided Design (CAD); C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems, cellular architecture*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]: *Modeling techniques*

This work was supported in part by ST Microelectronics and the National Science Foundation under grants 0607934 and 0606675. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Authors' Addresses: F. R. Johnson, Department of Electrical and Computer Engineering, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; J. M. Paul, Department of Electrical and Computer Engineering, 302 Whittemore Hall, Virginia Tech, Blacksburg, VA, 24061.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2008 ACM 1084-4309/2008/01-ART10 \$5.00 DOI 10.1145/1297666.1297676 <http://doi.acm.org/10.1145/1297666.1297676>

ACM Transactions on Design Automation of Electronic Systems, Vol. 13, No. 1, Article 10, Pub. date: January 2008.

General Terms: Design, Performance

Additional Key Words and Phrases: Heterogeneous chip multiprocessors, MESH, scenario oriented design

ACM Reference Format:

Johnson, F. R. and Paul, J. M. 2007. Interrupt modeling for efficient high-level scheduler design space exploration. *ACM Trans. Des. Autom. Electron. Syst.* 13, 1, Article 10 (January 2008), 22 pages. DOI = 10.1145/1297666.1297676 <http://doi.acm.org/10.1145/1297666.1297676>

1. INTRODUCTION

Single chip heterogeneous multiprocessor (SCHM) systems combine tight integration and low cost with high performance execution of a wide variety of workloads. This makes them an increasingly attractive platform on which to design modern consumer electronics. Offerings such as the Texas Instruments OMAP [TexasInstruments], Philips Nexperia [Philips], and Sony/Toshiba/IBM Cell [Pham et al. 2006] bring heterogeneous hardware to products ranging from cell phones and flexible multimedia players to home entertainment and gaming consoles.

Historically these devices were truly “embedded.” They were closed systems that executed statically-scheduled software on a single processor, with the single goal of meeting all deadlines. The scheduler design space was limited because real-time scheduling was very effective. Their modern counterparts, on the other hand, execute dynamically scheduled software, are more open to post-design time programming, and often exhibit highly data-dependent behavior. In addition, the user’s metric of performance is often “how fast it runs my program,” though deadlines also remain important.

Under these circumstances pure real-time scheduling is overly conservative and limits performance. Previous work [Paul et al. 2004; Kumar 2005; Paul et al. 2003] has shown that architecture- and application-aware schedulers can significantly improve performance by exploiting heterogeneity in the system. This adds a new dimension to the design space of these devices: the choice of scheduling strategy.

The Modeling Environment for Software and Hardware (MESH) [Paul et al. 2005; Cassidy et al. 2003] provides a foundation for high-level design exploration of SCHMs, including software and hardware configurations as well as scheduling strategies. However, neither MESH nor other high-level simulators have previously supported the modeling and simulation of the asynchronous events behind preemptive scheduling and task migration, without sacrificing simulation speed.

There are two primary challenges to efficiently capturing the behavior of preemptive scheduling on a SCHM. First, a software task could be scheduled on various hardware resources of differing speeds and capabilities. A common high-level simulation approach involves executing annotated software models on a native host, but these time-based annotations typically used are inadequate for capturing task performance in a heterogeneous multiprocessor. MESH

already addresses this shortcoming by representing tasks in terms of their computational complexity rather than time and resolving time from task-resource mappings at runtime [Paul et al. 2005]. This allows software models to specify the performance impact of hardware features such as floating point support or vector instructions, for example, allowing the designer to easily explore the impact of changing the hardware tasks execute on. It is also required to model task migration in a heterogeneous system.

Second, asynchronous events such as interrupts are difficult to capture precisely at a high level [Lavagno et al. 2005] because high level models do not capture individual cycles or instructions; simulations might advance hundreds or thousands of instructions between scheduler invocations, delaying the simulator's response to interrupt arrivals. However, the timing of these events can have a significant impact on the overall timing of the system, especially response time [Rhodes and Wolf 1999], and cannot be ignored. Simply increasing the level of detail to improve precision (using finer-grained annotations) significantly increases simulation time, negating the speed benefits of high-level modeling.

To address this challenge we extended the MESH kernel to speculatively execute software models, rolling them back when asynchronous events force unanticipated scheduling decisions. In addition, we allow software models to avoid unnecessary communication with the simulator through “lightweight” annotations. These eliminate most or all interactions between software models and their schedulers, greatly reducing simulation overhead and allowing the designer to use more accurate models. Together these features allow MESH to capture asynchronous concurrent interactions, including arbitrary preemptive schedulers, without sacrificing simulation speed.

We use our approach for efficiently including high-level modeling of interrupts to illustrate the benefits of task migration coupled with preemptive scheduling. We compare several static and preemptive schedulers in a multimedia cell phone to evaluate the potential performance impact of migrating running tasks between processors by aborting and restarting them. We build on previous work that demonstrated the effectiveness of MESH in evaluating task-processor mappings [Paul et al. 2003]. We apply Scenario Oriented benchmarking [Paul et al. 2004] to determine whether the overhead imposed by increased scheduler complexity, wasted work, and migration overhead is justified for the expected uses of the system.

The contribution of this work is the ability to model interrupts along with scheduling and communications overhead in a high-level simulator, ultimately permitting the evaluation of the cost-benefit of task migration at the chip level for single chip heterogeneous multiprocessors.

Our experimental results show that dynamic schedulers—which redistribute non-deadline tasks among processors as new tasks arrive or complete—achieve significant performance increases over statically partitioned scheduling schemes, improving response time by 1.96 times without impacting the system's ability to meet deadlines. They also underscore the effectiveness of high level modeling in finding the best scheduling strategy for a system.

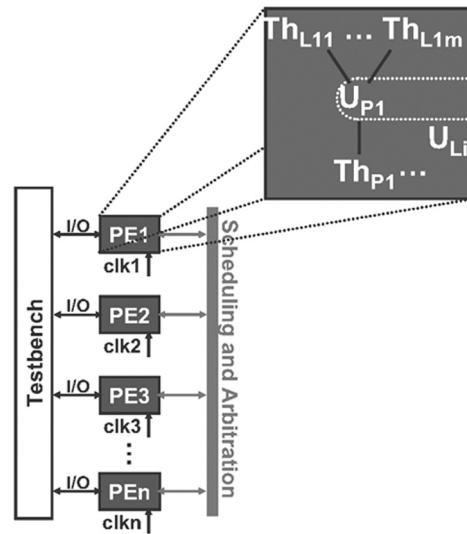


Fig. 1. Layered logical and physical design elements in MESH.

2. OVERVIEW OF MESH

The foundation for this work is the Modeling Environment for Software and Hardware (MESH) [Paul et al. 2005; Cassidy et al. 2003], a high-level simulator that permits the evaluation of the cost-benefit of high-level scheduling decisions, but previously without the capability to efficiently model interrupts.

MESH can be thought of as a thread-level simulator for a SCHM instead of an instruction-level simulator for a microarchitecture. Using MESH, we have had success exploring SCHM designs well above the level of the ISS; designers manipulate threads, processors, scheduling and communications strategies instead of instructions, functional units, and registers [Paul et al. 2006; Meyer et al. 2005; Bobrek et al. 2004; Paul et al. 2003]. We have compared our performance results with ISS-level models and found reasonable accuracy while execution is typically two orders of magnitude faster than ISS-level models.

MESH represents systems as layers or services: hardware, scheduling, and software. Scheduling forms the boundary between the hardware and software domains, and serves not only to coordinate software tasks, but also to convert logical computational complexity into timing information tied to a specific hardware model.

Figure 1 illustrates the primitive modeling elements of MESH at a high level. This view provides the essential modeling elements of design of SCHMs at the thread level as opposed to the instruction or cycle level. At the lower left of the figure an SCHM is illustrated interacting with some external testbench. The SCHM includes n PEs, with n separate clocks. The cooperation of the PEs in the SCHM is illustrated as a layer of “scheduling and arbitration,” shown as a solid, thick line. Many overlapping layers of scheduling and arbitration may exist in SCHMs; the single domain is shown to simplify the illustration of the main design elements of MESH. Also, the label “scheduling and arbitration” may

represent networked communications as well as shared busses; the label in the figure simply illustrates processors grouped into a logical, cooperative domain.

In Figure 1, $PE1$ is expanded into a MESH “thread stack.” The thread stack illustrates all of the design elements contributed by $PE1$ (each PE contributes a similar thread stack to the SCHM). At the bottom is a model of the physical capabilities of the processor resource, modeled as physical thread Th_{P1} . Each unique PE represents a different physical capability within the SCHM, which can be programmed to carry out concurrent behavior. This programming is captured as a collection of logical threads, $Th_{L11}, \dots, Th_{L1m}$, shown at the top of the expanded view. The logical threads represent the untimed, logical sequencing of software, along with the computational resources it requires, through annotations known as “consume calls.” These consume calls capture the effects of software executing on hardware, where physical timing is not determined until the software executes on a model of (or a real) physical machine. Each PE may execute an unbounded number of software (logical) threads, with scheduling decisions potentially made dynamically in response to different datasets. This per-processor, thread-level decision making is modeled by a physical scheduler, U_{P1} .

Cooperation among the various PEs, including the exchange of state across private memory spaces as well as communications and task arbitration, is modeled by the logical scheduling domain, U_{L1} . Because U_{L1} represents a cooperative scheduling domain, the threads that execute on any given resource may also be eligible to execute on any other resource that belongs to the same logical scheduling domain. Significantly, this is true even if the processors are heterogeneous.

U_{L1} captures the penalty of resource sharing and cooperation, allowing MESH to model what is perhaps the key challenge of designing of SCHMs at a high level: the trade-off between few powerful, complex, PEs that execute more threads locally, or more, less powerful, simpler, PEs with the cost of global cooperation.

Internally, MESH consists of a simulation kernel that executes software task models as threads on the host machine. Task threads can execute software directly on the host in order to generate realistic control flow and outputs; consume calls serve to annotate computational complexity in a machine-independent way. Tasks communicate with the MESH kernel using the thread synchronization primitives supplied by the host operating system; whenever a task generates a consume call control switches to the kernel thread, which applies scheduling and translates the annotations into timings for the target processing element.

This paper describes two improvements to MESH that allow it to efficiently model asynchronous events such as device interrupts and preemptive scheduling: interrupt modeling (Section 4.1) and *lightweight consume calls* (Section 4.2). For more detailed explanations of other aspects of the MESH environment we refer the interested reader to Paul et al. [2005].

3. PRIOR WORK IN INTERRUPT MODELING

Interrupt arrivals in a system can have a profound impact on performance [Regehr and Duongsaa 2005; Jones and Regehr 1999]. [Rhodes and Wolf 1999]

evaluated how precise interrupt modeling impacts the accuracy of simulations for predicting schedulability of hard real-time task sets.

There are several existing methods for modeling interrupts at a high level. Some approaches apply high level RTOS and interrupt models to lower-level SystemC [systemC] simulations in order to improve simulation speed, either at the instruction accurate [Yi et al. 2003; Passerone et al. 1997] or basic block level [Honda et al. 2004; Bouchhima et al. 2004]. This preserves the accuracy of detailed simulation while significantly reducing scheduling overhead. Though this takes advantage of high-level scheduling models, these approaches still simulate software (and hardware) at a much lower level than MESH.

Yu et al. [2003] model RTOS services, including interrupt handling, using transaction level modeling (TLM) in order to integrate with TLM communication models. This results in a high-level model of the system but also reduces precision in interrupt handling; interrupts are forced to only occur at transaction boundaries.

The approach in Lavagno et al. [2005] models time-sliced scheduling at the basic block level using the Metropolis [Balarin et al. 2003] framework. However, the Metropolis scheduler must be able to predict all upcoming scheduling points; it does not capture “preemptions caused by unpredictable events, like when the occurrences of nonperiodic interrupts are to be modeled” [Lavagno et al. 2005].

“Virtual Synchronization” [Yi et al. 2003; Kim et al. 2005] aims to reduce the overhead of synchronizing component clocks in distributed HW/SW cosimulation. The overhead of clock synchronization is similar to the scheduling overhead in MESH. However, Virtual Synchronization targets cycle-accurate HW/SW simulation, and interrupt modeling is handled by individual component simulators; the simulation backplane does not participate.

Another approach [Kempf et al. 2005] represents tasks as sequences of (time, priority, delay, state) tuples, and uses a set of “timed communicating extended finite state machines” joined with priority queues to model the system. While effective for modeling the timing characteristics of a particular system configuration, the method does not provide a straightforward way to vary the type of hardware each task executes on (i.e. without reimplementing the software model), or to experiment with different scheduling strategies, both important features MESH provides.

Finally, all of these approaches focus on the problem of scheduling tasks within a single processor using static schedules. While tasks may be partitioned among multiple processors at design time each is statically mapped to a single processor for execution. In contrast, prior work using MESH [Paul et al. 2003] has shown that schedulers form a vital design element of SCHM systems and that breaking down static partitions can improve performance by allowing schedulers to exploit idle resources that would otherwise go unused, despite increased scheduling complexity.

Similarly, none of these works considers the performance effects of migrating running tasks. The schedulers considered in Paul et al. [2003] did not attempt to migrate running tasks, and there was no discussion of starvation due to preemptions. Tasks were initially mapped to candidate resources upon arrival

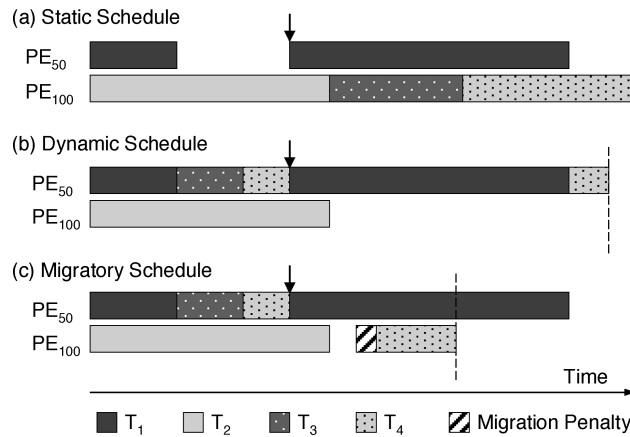


Fig. 2. Influence of scheduling strategy on system response time. (a) Waiting tasks T_3 and T_4 cannot run on idle PE_{100} . (b) Preemption starves dynamically scheduled thread T_4 . (c) Migration exploits idle resources while avoiding starvation.

and executed—uninterrupted—to completion. Up to this point task migration has not been widely considered in the embedded domain.

An extensive survey of the performance improvements achievable through task migration can be found in Milojicic et al. [2000]. Though this work discusses task migration in the context of distributed computing, we anticipate that it will extend to the SCHM case for two reasons. First, the cost of on-chip state transfers is orders of magnitude lower than going off-chip or over a network, and second, the memory footprints of tasks in an SCHM will typically be much smaller than those encountered in distributed computing. These factors make migration attractive even for short-lived tasks by significantly reducing its cost.

4. TASK MIGRATION AND PRE-EMPTIVE SCHEDULING

Modern SCHM systems feature dynamic workloads and highly data-dependent execution, even among tasks with deadlines. For example, the average and worst-case times to decode one frame of MPEG video can differ by more than an order of magnitude [Duda and Cheriton 1999]. Under these circumstances real-time scheduling is highly conservative and results in overprovisioned hardware resources. The traditional approach of statically partitioning all tasks among processing elements at design time can prevent other tasks in the system from taking advantage of scheduling slack in such resources.

Consider the simple SCHM depicted in Figure 2(a). It consists of two processing elements PE_{100} and PE_{50} , running at 100 and 50 MHz, respectively. PE_{100} is reserved for real-time task T_1 , while tasks T_2 through T_4 are statically assigned to PE_{50} . Due to data-dependent execution, T_1 finishes early, leaving PE_{100} idle until its next activation. Because the system is statically partitioned, waiting tasks assigned to PE_{50} cannot take advantage of the superior processing power PE_{100} offers.

In Figure 2(b), a dynamic scheduler eliminates the static partition by assigning tasks to resources as they become available, but does not move tasks once they begin execution. Previous work in MESH has explored the benefits of this strategy in the absence of preemption [Paul et al. 2003]. In this case T_1 preempts T_4 before it completes, causing starvation. Because T_4 migrated, PE_{50} now remains idle and unused. The net result is very little improvement in response time over the static scheduler.¹

Finally, Figure 2(c) depicts a migratory scheduler that is capable of moving tasks even after they begin execution. Like the dynamic scheduler it exploits PE_{100} when it becomes idle, but a short time after T_1 reactivates, it detects that T_4 is no longer making forward progress and migrates it to PE_{50} . In spite of the time delay and the cost of transferring task state between processors, this strategy improves response time significantly.

Modeling task migration at a high level in a SCHM requires the ability to accurately model the performance of tasks for every resource they might execute on. MESH provides this functionality by representing tasks in terms of the computational resources they consume rather than their execution time. It can also provide equivalent sets of resource consumption, to specify how hardware features such as floating point support would affect its performance. This ability can be applied successfully to divisions as fine as individual instruction types [Meyer et al. 2005], though this is usually unnecessary.

4.1 Interrupt Modeling

Preemptive scheduling and interrupts present challenges for high level modeling because of their asynchronous nature. A simulation environment can easily provide API calls to capture synchronous events such as task creation and use of synchronization primitives. These API calls then explicitly identify scheduling points, allowing the scheduler to model them properly. In contrast, unanticipated external events like preemption and interrupt handling are transparent to the software tasks they affect and will generally not occur near conveniently marked scheduling points. Similarly, a synchronous event on one processing element could easily trigger an asynchronous even on some other processor. For example, a semaphore “post” operation on processing element A might unblock a high-priority task, resulting in a preemption on processing element B .

Consider the situations in Figure 3. In time line (a) two software tasks, T_1 and T_2 , practice cooperative multitasking on a single processor. Each executes for some time then yields, allowing the other to run. All context switches occur at well-defined points marked by consume calls (marked by heavy lines). At each consume call boundary, the scheduler resolves computational complexity to physical timing, and simulation proceeds efficiently and accurately. (b) shows the preemption points that would be imposed by a time-slicing scheduler. Context switches are no longer correlated with consume call boundaries, and multiple interrupts can occur during the span of a single consume call. The last time line shows T_1 and T_2 properly interleaved by time slicing, with the original consume call boundaries marked by heavy lines. We note that although MESH

¹In fact, response time could actually worsen if T_1 were to run longer after preempting T_4 .

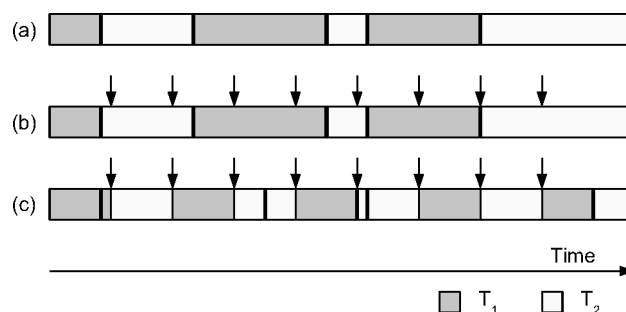


Fig. 3. Timing effects of asynchronous event arrivals: (a) Cooperative multitasking, (b) Preemption points dictated by a time-slicing scheduler, (c) Accurate time-sliced multitasking. Heavy lines mark consume call boundaries.

models the cost of servicing interrupts, that cost is negligible compared to the length of a time slice and is not shown in the figure.

Clearly the scheduler cannot atomically advance by consume call-dictated intervals: an interrupt could arrive at any moment and cause a context switch. Because the scheduler is generally not aware of future interrupt arrivals, it cannot plan for them in advance.² While it is straightforward to process only part of a consume call, how much or little must be processed? Advancing simulation by fixed intervals would bound the delay before the scheduler becomes aware of an interrupt. However, since even small delays in responding to interrupts can cause significant changes in the timing behavior of a system [Rhodes and Wolf 1999], the designer must then choose between accuracy and simulation speed.

Fine-grained scheduling is highly undesirable when interrupts do not occur very frequently in the system. In this case the simulator will process many small time steps, very few of which actually contribute to accuracy.

The key to working around this difficulty comes by observing that most interactions between tasks in a system affect only their timing, rather than their behavior. For example, time-sliced execution or contention for a bandwidth on a shared bus will not affect the final output of a task. Given a set of relevant behavior traces (e.g., computational demands or memory accesses) the simulator can easily resolve timing-only interactions after the fact.

MESH takes this post-mortem approach to scheduling. It allows the host threads representing tasks in the system to generate full consume calls before returning control to the simulator kernel. MESH resolves timing conflicts among tasks, detects and processes interrupts, then commits consume calls once they are known to be free of interruptions. Each scheduler in the system advances the simulation as quickly as possible without reducing accuracy, often hundreds or thousands of cycles at a time. When interrupts do occur, MESH employs a roll-back mechanism to recover from the misspeculation (see Section 4.1.1).

Speculative scheduling provides two significant benefits. First, even though it accurately captures asynchronous interrupts, simulation is still highly

²The periodic timer tick in this example could easily be predicted, but not aperiodic interrupts caused by peripherals or communication between schedulers.

efficient because time advances in the largest time steps possible by avoiding unnecessary scheduler invocations between interrupts. Second, it allows the designer to create software models without concern for timing-only interactions between tasks; she need only consider those scheduling interactions that might actually affect task behavior, such as accessing lock-protected memory. This flexibility allows quick evaluation of arbitrary preemptive scheduling strategies without imposing repeated changes to the software models involved.

4.1.1 Task State Rollback in MESH. The internal state of MESH consists of two parts. A set of host threads models the software and hardware in the system (as described in Section 2), while the simulator kernel coordinates the host threads, resolves computational complexity into timing, and makes scheduling decisions.

The MESH kernel stores a global time as well as one pending (speculative) consume call for each runnable task in the system. One simulation step then consists of the following actions:

- (1) Ensure that a pending consume call is available for each runnable task, allowing the corresponding host thread to run and return one if necessary. We note that, conceptually, test benches are tasks which do not compete for processor time and whose consume calls usually end by raising an interrupt on some simulated processing element.
- (2) Convert the consume calls into time delays by combining each one with the hardware model of the processing element it executed on³ and advance global time to the end of the shortest consume call.
- (3) Apply any side effects dictated by the completed consume call (synchronization primitive access or raised interrupts), potentially triggering scheduling decisions. Any scheduling decision that impacts tasks running on other processing elements triggers a rollback: Split the affected consume calls at the new global time and commit the part that completed; the other part remains pending.

To illustrate how MESH advances simulation speculatively and applies rollback, consider the simulation depicted in Figure 4. It consists of three tasks (T_x , T_y , T_z) executing on two processing elements (PE_1 , PE_2). In addition, a test bench task raises an interrupt at time t_2 , indicated by a large arrow.

At time t_1 (Figure 4(a)), only T_x and the test bench (not shown) are runnable. MESH allows both host threads to execute and return consume calls, then begins to apply post-mortem scheduling. It first combines the consume call from T_x with the processing power of PE_1 to determine its length. MESH then advances global time to t_2 because the test bench thread's consume call (ending with an interrupt) is shorter.

The interrupt causes T_y to unblock and be scheduled on PE_2 (Figure 4(b)). The consume call for T_x does not commit or roll back because no new

³In reality, MESH only needs to resolve timing once per consume call and cache the result—as long as the task doesn't migrate to a different processing element.

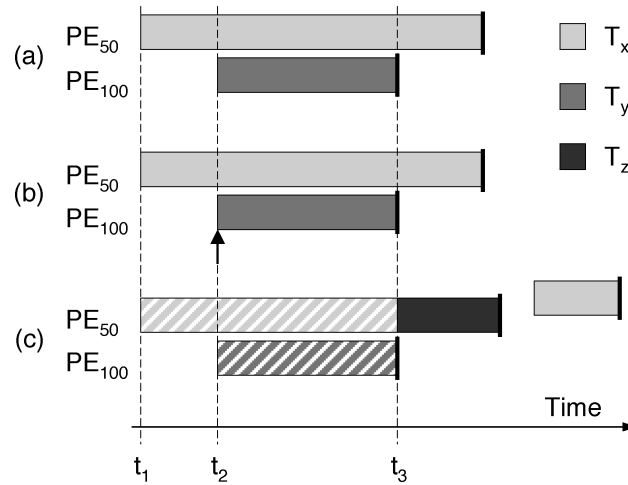


Fig. 4. Example of interrupt handling and rollback in MESH: (a) Software model T_x runs at t_1 , speculatively executing to the next consume call (heavy line). (b) An interrupt later arrives at t_2 , causing T_y to also execute. (c) T_y unblocks T_z at t_3 , which in turn preempts T_x . This triggers a rollback, allowing the preemption to occur at the proper (simulated) time. Hatched consume calls have been committed.

scheduling decision has affected it so far. Because MESH does not have a speculative consume call for T_y , it allows the host thread to execute and return one. MESH again resolves timing and determines that the consume call for T_y will commit before that of T_x .

The consume call from T_y ends with an access to a synchronization primitive which unblocks high-priority task T_z . MESH applies the designer's scheduling policy and determines that T_z should preempt T_x at time t_3 . However, the host thread for T_x has already executed past the preemption point, which triggers a rollback (Figure 4(c)). To implement the rollback, MESH splits the consume call for T_x at t_3 and commits the earlier piece, as well as the full consume call for T_y (committed consume calls are marked with a hatching pattern in the figure). The remainder of the consume call is stored as part of the internal state for T_x and will not be considered again until T_x gets rescheduled at some future point in the simulation. MESH also speculatively executes the host thread for T_z to obtain a consume call, and the simulation continues.

One potential concern with the approach of splitting consume calls is that the host thread executes each consume call all at once, but MESH could split it repeatedly due to preemptions. The consume call may then experience an arbitrarily long delay before it commits. This disconnect between model execution and timing resolution must not affect the correctness of the resulting simulation. We note that, for a well-behaved program, transparently time-shifting task execution will not change system behavior as long as no synchronization points are speculatively passed. Pieces of code that are sensitive to delays or interleaving with other tasks (i.e. critical sections) must be protected by some

form of synchronization to ensure correctness.⁴ Because all explicit synchronization in MESH occurs at consume call boundaries, it is always safe for the simulator to speculate up to the next such boundary without rolling back host thread state.

4.1.2 ISR-Task Communication. Interrupts typically interact with tasks either through synchronization primitives (semaphores, queues, etc.) or by background polling. While MESH supports both types of communication, use of polling is discouraged because it reduces simulation speed.⁵ Fortunately, polling operations can usually be converted into blocking ones, especially for the kinds of applications we consider in this paper.

As with task-task communication, ISR-task communication through OS primitives results in timing-only interactions subject to post-mortem scheduling. For example, in Figure 4(b), T_y was blocked when the interrupt fired; the ISR used some OS primitive to unblock it. Had T_y been running (or blocked on some unrelated operation) at the time, the ISR would not have impacted its behavior until it accessed the primitive at some later point. Therefore, we see that MESH consume calls already support the most common ISR-task communication pattern.

Occasionally, an ISR must interact directly with a task through some form of polling. In this case, the task disables interrupts for a short time while it tests the value of some indicator variable. When the interrupt fires, its ISR updates the indicator, triggering some action when control returns to the polling task. MESH provides a set of API calls that permit software tasks to disable and enable particular interrupts as well as interrupts in general. ISR tasks can also disable interrupts to prevent themselves from being interrupted in turn by higher-priority interrupts. Once interrupts are reenabled the interrupt scheduler fires any pending interrupts that arrived during the critical section.

4.2 Lightweight Consume Calls

Overly fine annotations impact simulation speed in two ways. Each consume call boundary causes a context switch to the simulation kernel in order to perform timing resolution and scheduling. As Figure 5 shows, the overhead of context switching quickly reduces simulation speed; at the basic block level (<10 instructions) performance in MESH is comparable to fast ISS simulation. A similar effect arises while synchronizing component clocks in distributed cosimulation [Yi et al. 2003], where overhead approaches 90% even if synchronization is achieved through simple function calls instead of context switches. Unfortunately, the most natural approach to annotation marks basic blocks to capture data-dependent execution, leading to many small simulation steps and high overhead.

MESH introduces the concept of *lightweight consume calls* to address the overhead of context switching. Normal consume calls, though computationally

⁴MESH will faithfully reproduce data races that result from unprotected critical sections.

⁵Polling impacts performance because it generates precisely the kind of fine-grained consume calls we wish to avoid.

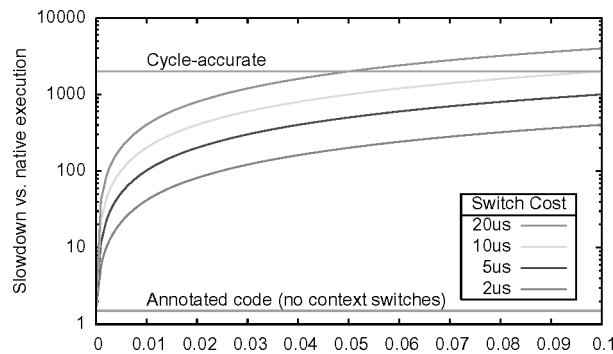


Fig. 5. Impact of synchronization overhead on simulation time. Annotation overhead in MESH is observed to be between 50 and 100%; ISS simulation is assumed to process one million instructions per second on a host executing two billion instructions per second.

inexpensive to produce, cause an immediate context switch to the simulator’s scheduling layer for processing. Lightweight consume calls instead allow the host software task to aggregate the information supplied by these lightweight consume calls into an equivalent “heavyweight” consume call that will be delivered to the simulator, either when the task requests it or when the trace grows too large. Traces of lightweight consume calls are stored as simple lists associated with the simulator state of each task. The task also maintains an aggregate consume call that represents the running total at any time. If no interrupts arrive during the interval, MESH simply commits the aggregate consume call. Should an interrupt trigger rollback, MESH examines the list of lightweight consumes to determine exactly which logical computations occurred before the interrupt, and splits the list at that point before processing the interrupt.

Lightweight consume calls can replace any normal consume call that does not mark a synchronization point such as a critical section. In the common case of no intertask communication the aggregate consume call may span the entire lifetime of the task. The resulting simulation has the same accuracy as one using all heavyweight consume calls (the original MESH), but with virtually no overhead due to context switching. It is important to note that lightweight consume calls do not affect the outcome of the simulation—they only improve simulator performance by eliminating unnecessary context switch overhead.

The roll-back mechanism is based on the assumption that roll-backs are rare events; it adds virtually no performance overhead for workloads that do not show unpredictable interrupts. Lightweight consume calls provide the maximum performance boost with such workloads while context switching always dominates the cost of heavyweight consume calls. Lightweight consumes do impose a minor memory overhead on the order of 10 to 100 kilobytes of storage per task. This overhead is well worth the performance boost lightweight consumes offer, given the abundant main memory available on most host machines.

As tasks experience interrupts more and more frequently the benefits of lightweight consume calls diminish; at some point the cost of repeatedly splitting lists will outweigh the reduction in context switching; should this occur the designer can simply return to using heavyweight consume calls. In the

```

1 // Real-valued FFT on a bit-reversed input.
2 // Exploits conjugate symmetry about N/2
3 void fft(complex *array, int size) {
4     int j, k, N;
5
6     // pass once for each positive power of two <= N
7     for(N=2 ; N <= size; N *= 2) {
8
9         // special cases for k=0 and k=N/4
10        for(k=0; k < size; k += N) {
11            // k = 0
12            .
13            .
14            .
15            lightweight_consume("ADD:2");
16
17            // k = N/4
18            if(N >= 4) {
19                .
20                .
21                .
22                lightweight_consume("ADD:1");
23            }
24        }
25
26        // handle 0 < k < N/4
27        for(k=1; 4*k < N; k++) {
28
29            // handle each butterfly at this k
30            for(j=0; j < size; j += N) {
31                .
32                .
33                .
34                lightweight_consume("ADD:6 , _MUL:4");
35            }
36        }
37    }
38
39    lightweight_commit();
40 }

```

Fig. 6. Skeleton code of an FFT kernel, annotated for use with MESH.

workload we examine (see Section 5) interrupts are so rare that the overhead of list-splitting is negligible. Workloads featuring tasks with high interrupt rates (e.g., due to intense I/O activity) will benefit less from lightweight consume calls. However, tasks scheduled on other processing elements continue to benefit from lightweight consumes.

We illustrate the use and effect of lightweight consumes with the following simple example: an FFT kernel. Figure 6 shows skeleton code annotated at lines 15, 22, and 34. The transform is functionally independent of other tasks in the system and therefore a good candidate for lightweight consume calls; any timing dependencies (such as contention for memory) can be resolved post-mortem by the simulator. As this code executes in MESH each call to `lightweight_consume` adds some number of add and multiply operations to the trace for this task. Finally, at the completion of the routine (line 39) a call to `lightweight_commit` makes the trace available to the simulator and requests scheduling.

In our example the FFT task's host thread completes with only one context switch to the simulator kernel. Without lightweight consumes the designer would have been forced to accept $O(N \lg N)$ context switches (the asymptotic

complexity of an FFT with N inputs), or annotate at a coarser granularity. For tasks with data-dependent execution the latter alternative could easily lead to unacceptably low accuracy.

5. EXAMPLE: MULTIMEDIA CELL PHONE

We used the enhanced version of MESH described in the previous section to evaluate scheduling strategies for a hypothetical SCHM multimedia cell phone with the following features:

- Internet Access.* A built-in wireless connection provides access to several Internet services including e-mail, online photo albums, ebook downloads and music services. The cost of decoding many small images—potentially in parallel—dominates Web browsing.
- Text to Speech.* As an alternative to reading from the device’s small screen, speech synthesis allows the user to listen to ebooks, and text and email messages.
- Online Photo Management.* Users can upload snapshots from the built-in camera and download images from an online photo management service. Significant overhead stems from encoding and decoding large images one at a time.

The goal in these explorations was to demonstrate the use of interrupt modeling in MESH while comparing the performance of task migration in a preemptive dynamically scheduled system to that of a statically partitioned baseline. We chose a Scenario Oriented benchmarking [Paul et al. 2004] approach to provide realistic multiprogrammed workloads, with response time of nondeadline applications as the metric of performance; real-time tasks must not miss any deadlines but are otherwise ignored.

5.1 Scenarios

Scenario Oriented benchmarking [Paul et al. 2004] is based on the observation that a system’s performance can only be determined by evaluating the kinds of multiprogrammed workloads it is (being) designed for. The performance of isolated benchmarks does not provide a complete picture due to contention for shared resources such as processor time and bus bandwidth. At the same time, it is vital that the workloads used to evaluate the system be reasonable representations of those the system would encounter in real life.

Figure 7 shows the set of tasks which form each of the cell phone’s features. All tasks are members of the MiBench [Guthaus et al. 2001] benchmark suite. Rsynth and ADPCM are real-time tasks with periodic 2.5 sec and 10 ms deadlines, respectively. JPEG encoding and decoding are provided on a best-effort basis, while CRC and AES are either real-time or best-effort tasks, depending on where they are used. In addition, rsynth and JPEG exhibit strongly data-dependent execution. Finally, web browsing is highly parallel as multiple images can be decoded independently.

This application mix reflects the highly variable workloads that are starting to appear in portable and handheld devices. Task migration in heterogeneous

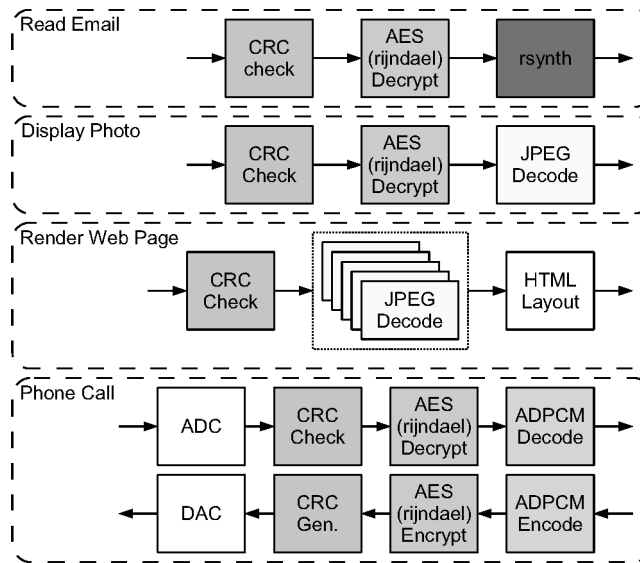


Fig. 7. Scenario application compositions. Tasks with white backgrounds are not modeled.

systems is a potential way to balance unpredictable computing demand with other constraints such as power utilization [Kumar et al. 2003]. While task migration within a single ISA is straightforward, many consumer electronics sport multiple ISAs within a single chip, complicating task migration.

We note that the applications presented earlier can all be divided into small, largely independent tasks for scheduling. For example, the real-time applications divide their inputs into periodic tasks, each with a single deadline. This simplifies system design and also eases task migration in heterogeneous systems: Important kernels can be optimized for each target in the system. Splitting each kernel invocation into a separate task then allows the system to change the target without having to migrate working state between architectures. Tasks with little impact on performance can be pinned to a single architecture or core to avoid unnecessary design-time complexity: There is little reason to port a low-priority background task to a powerful DSP chip, for example.

Section 6 shows that scheduling small subtasks, combined with a kill-and-restart migration strategy, is quite effective for this workload.

All applications may be used simultaneously, providing a rich set of potential use cases to explore. A full evaluation would involve dozens of different scenarios intended to exercise the system in a variety of ways, but due to limited space we only present results for one representative scenario. Figure 8 depicts a variable use pattern that could arise as a business traveler waits for a delayed flight. Activities include reading email, browsing web pages, downloading product photos, and reading an email to a coworker over the phone. It presents mixed real-time and best-effort workloads ranging from idle to single tasks to several concurrent tasks. Dotted lines indicate task arrivals that change system

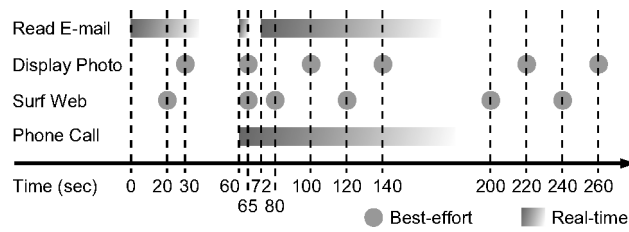


Fig. 8. Example scenario: a business traveler listening to email messages, downloading product photos, and visiting several web sites. A phone call also spans much of the scenario.

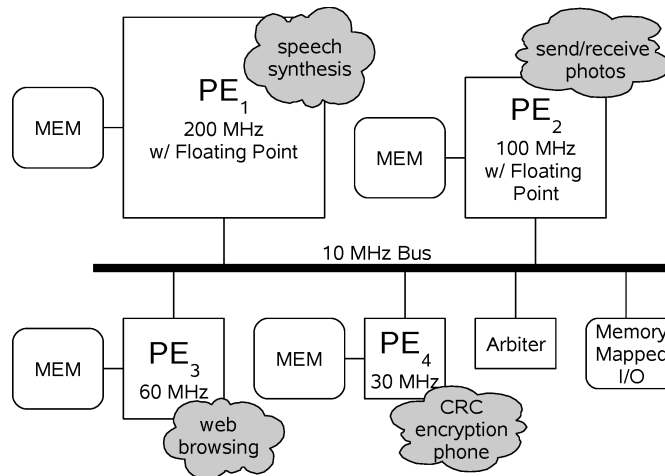


Fig. 9. Multimedia cell phone hardware configuration and static task partitions.

load. Performance depends not only on the data set of each task, but also on concurrent interactions arising from task arrival times. The scenario spans nearly five minutes of simulated time and more than 25 billion target instructions.

5.2 Hardware Configuration and Static Scheduler

The example cell phone is powered by an SCHM containing four ARM processors running at 200, 100, 60, and 30 MHz, respectively. The two fastest processors provide hardware floating point support while the other two do not. Each processor performs all computations in its own private memory; data in off-chip memory or Flash must be transferred to local storage over a shared 10 MHz bus (arbitrated in 100 cycle bursts) before use. As shown in Figure 9, each application is statically mapped to a single processor; the number and types of processors in the system are based on the computational complexity of those tasks.

Rsynth is the most demanding application in the system and must also meet deadlines in spite of its unpredictable execution time; it therefore requires the most powerful processor in the system (200 MHz), though in the average case it is over-provisioned. JPEG encoding and decoding are mapped to more modest processors (100, 60MHz), while the remaining low-complexity tasks are relegated to the weakest processor in the system (30MHz).

5.3 Dynamic Scheduling Strategies

We compared the statically partitioned baseline against three migratory schedulers, each more aggressive than the last. The first scheduler uses a conservative work-stealing strategy to distribute work during periods of load (*passive work stealing*). As tasks arrive in the system they are added to the queue of their statically assigned processor to await its services. Each processor services best-effort tasks one at a time, preempting them whenever real-time tasks arrive; once tasks begin executing they cannot be migrated. When any processor becomes idle it attempts to steal work from the queues of other processors in the system; when tasks arrive at loaded processors the scheduler will raise interrupts on all idle processors to trigger work stealing. This approach improves performance by exploiting parallelism in the workload.

The second scheduler expands work-stealing to also allow idle processors to usurp tasks from less powerful resources even when those resources are idle (*active work stealing*). This more aggressive approach improves performance over passive work stealing by attempting to schedule tasks on the most powerful resources possible.

The third scheduler adds migration of running tasks to reduce the cost of bad scheduling decisions that can result from active work stealing (*migratory work stealing*). System events such as task arrivals or completions can cause inconvenient preemptions or free up previously unavailable resources. Task migration allows the scheduler to redistribute system load by moving existing tasks. Our implementation monitors tasks for starvation caused by preemptions and moves them to idle resources to complete execution.

Migrating running tasks is significantly more expensive than changing their processor assignment before execution begins because all task state must be transferred to the new processor before execution can resume. We note that, like many embedded applications, the tasks in our system do not allocate significant amounts of memory; task state is roughly equivalent to the tasks's input data set. Because tasks in the system are generally short-lived and must already transfer their working data to processor-local memory before beginning work, we chose to model migration by simply aborting and restarting the task to be migrated. This also avoids the complexity of transparently migrating between heterogeneous processors. More sophisticated approaches for saving and restoring task state during migrations exist but are highly complex; Milojevic et al. [2000] note that aborting short-lived tasks often requires less work than moving them. Our results confirm that this simple strategy significantly improves response time without the complexity of fully transparent migration.

6. EXPERIMENTAL RESULTS

We applied all four scheduling strategies to the scenario described in Section 5.1 and simulated them using MESH. In addition to the software tasks we modeled the overhead of arbitrating 100-cycle bus burst transfers. We also considered the overhead of the various schedulers as well as task steals and aborts. However, in practice scheduling and migration occur so rarely (roughly once per task) that they have virtually no impact on response time.

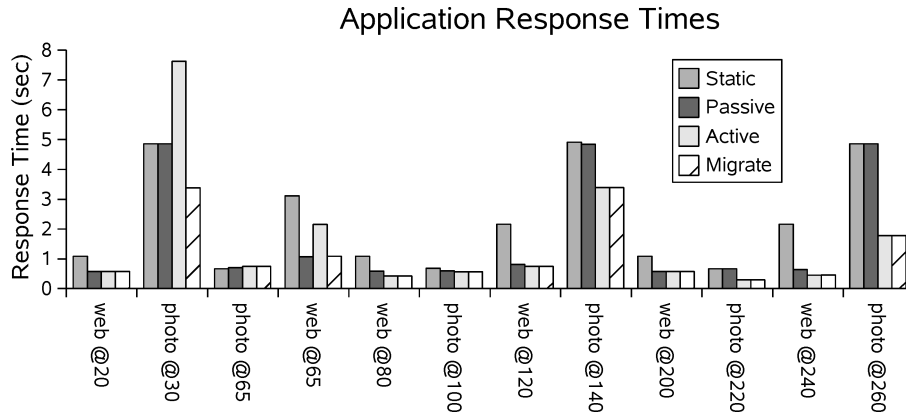


Fig. 10. Job response times resulting from each of the four scheduling strategies.

Figure 10 shows the resulting response times of the best-effort tasks. The response time of each task depends both on its own processing requirements, and on scheduling decisions made in the context of system load. There are four bars for each application, corresponding to the four schedulers. As discussed in 5.1, each simulated scenario spanned approximately five minutes on a multiprocessor ARM target system. Each simulation required between 15 and 30 seconds to complete on a 2.4 GHz Pentium 4 host with 1 GB of memory. Disabling lightweight consume calls increased simulation time to between 18 and 20 *minutes*, approximately 50–60 slower. The difference in simulation speed is entirely due to the overhead (or lack thereof) from context switching in the simulator at consume call boundaries. Cycle-accurate simulation would have required more than 7 hours, roughly 1000 times slower.⁶

As expected, the passive work stealing scheduler effectively distributed tasks of the parallel web page rendering application, improving system response time by 1.32 times. However, it achieved minimal speedup for the monolithic photo decoding application because it conservatively respected the original static mapping rather than exploiting PE_1 .

Active work stealing improved the performance of photo decoding by allowing idle processors (PE_1 and PE_2 in this case) to steal jobs from weaker processors. However, the strategy backfired for the jobs at $t = 30$ and $t = 65$ when the speech synthesis task preempted them. The jobs were then forced to wait until the speech segment completed, or nearly three extra seconds for $t = 30$. In spite of this shortcoming, however, the strategy improved response time by 1.41 times overall.

The fully migratory scheduler achieved the best overall performance, improving system response time by 1.96 times. It successfully detected starvation in the $t = 30$ and $t = 65$ photo decoding tasks and migrated them to other processors, for a speedup of 1.39 times over active work stealing.

⁶This optimistically assumes one million simulated instructions per second, with no additional overhead for coordinating four processors or modeling the bus.

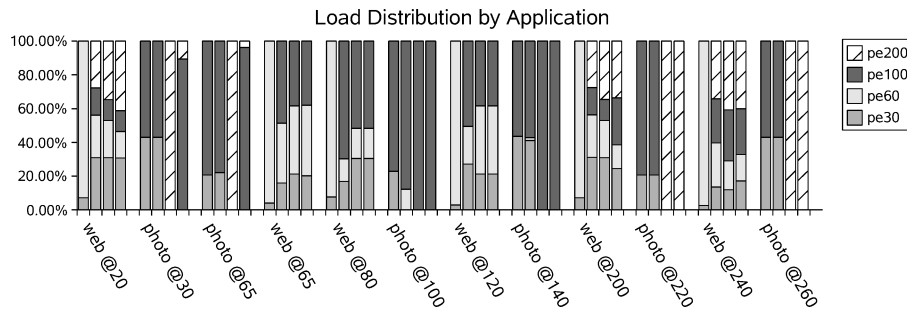


Fig. 11. Load-balancing effects of the static, passive, active, and migratory schedulers for each job.

It is interesting to note that the migratory scheduler achieved superior performance over the static scheduler for $t = 30$. In both cases PE_2 handled the entire JPEG decoding process, but the migratory scheduler also allowed partial JPEG decoding to occur on PE_1 before aborting and restarting it on PE_2 , increasing processing time for the task. However, this loss was offset by the gains of performing the CRC check and decryption on PE_1 instead of PE_4 (see Figure 11).

Finally, Figure 11 provides a detailed breakdown of load distribution for each scheduler and application. The best-effort tasks are lined up along the x-axis, with the y-axis showing the percentage of time the task spent on each type of processor in the system. Each task has four bars, corresponding to the four schedulers. We note that the performance improvement is not solely due to executing tasks on more powerful resources; many tasks make significant use of three or even all four processors in the system ($t = 20$, $t = 200$, and $t = 240$). It further shows that the best task-processor mappings depend on the state of the system, such as whether speech synthesis is active (compare $t = 30$ and $t = 240$, for instance). This demonstrates that merely adjusting static task partitions could not achieve the same results because it limits the number of processors each task can utilize and cannot adjust to runtime conditions.

7. LIMITATIONS AND FUTURE WORK

Full support for interrupt modeling in MESH opens up a wide range of design space explorations. This article demonstrates one use by modeling a particular family of preemptive schedulers. Fully exploring this and other preemptive scheduling strategies such as time slicing is the subject of future work.

Similarly, this article explored one form of task migration, implemented by aborting and restarting tasks. While quite effective for the short tasks considered here, aborting long-running tasks could potentially waste significant amounts of work. A full exploration of task migration would require the ability to transparently migrate the state of running tasks between processors instead of restarting them. Transparent migration would be straightforward to model in MESH, but the results would only be meaningful if based on accurate overhead estimates. Task migration is highly complex to implement in a real

system [Milojicic et al. 2000], especially if it contains heterogenous processors with multiple instruction set architectures.

Finally, this article did not examine how interrupt modeling impacts the accuracy of high-level simulations. A next step would be to compare the results presented in this paper with those generated by a multiprocessor cycle-accurate simulation, in order to determine whether interrupt modeling improves accuracy or, conversely, if its precision can be relaxed to a “higher level” without lowering accuracy further. A similar exploration would evaluate whether lightweight consumes could improve accuracy through finer annotations, without undue performance penalty.

8. CONCLUSIONS

This paper described how interrupt modeling and lightweight consume calls allow MESH to efficiently model preemptive schedulers at a high level by overcoming the challenge of capturing asynchronous scheduling events. We improved simulation speed in MESH by more than 50 times, allowing it to run over 1000 times faster than ISS simulation.

We also showed how these features allow effective scheduler design space exploration by using MESH to evaluate several scheduling strategies on a SCHM-based multimedia cell phone. Finally, we show that dynamic load distribution and migrating running tasks can reduce response time by as much as 1.96 times over statically partitioned scheduling in the system we evaluated. This result underscores the importance of using high-level modeling to explore a wide variety of scheduling approaches to maximize system performance.

ACKNOWLEDGMENTS

We express appreciation to the other members of the MESH group, Alex Bobrek and Brett Meyer, for their many helpful comments and insights. Finally, we thank the reviewers for their time and helpful suggestions.

REFERENCES

- BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., AND SANGIOVANNI-VINCENTELLI, A. 2003. Metropolis: An integrated electronic system design environment. *Comput.* 36, 4, 45–52.
- BOBREK, A., PIEPER, J. J., NELSON, J. E., PAUL, J. M., AND THOMAS, D. E. 2004. Modeling shared resource contention using a hybrid simulation/analytical approach. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*. Washington, DC, IEEE Computer Society, 21144.
- BOUCHHIMA, A., YOO, S., AND JERAYA, A. 2004. Fast and accurate timed execution of high level embedded software using HW/SW interface simulation model. In *Proceedings of the Conference on Asia South Pacific Design Automation*. 469–474.
- CASSIDY, A. S., PAUL, J. M., AND THOMAS, D. E. 2003. Layered, multi-threaded, high-level performance design. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 10954.
- DUDA, K. J. AND CHERITON, D. R. 1999. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 261–276.
- GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Workshop on Workload Characterization*. 1–12.

- HONDA, S., WAKABAYASHI, T., TOMIYAMA, H., AND TAKADA, H. 2004. RTOS-centric hardware/software cosimulator for embedded system design. In *Proceedings of the International Conference on HW/SW Codesign and System Synthesis*. 158–163.
- JONES, M. B. AND REGEHR, J. 1999. The problems you're having may not be the problems you think you're having: Results from a latency study of windows nt. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*. 287.
- KEMPF, T., DOERPER, M., LEUPERS, R., ASCHEID, G., MEYR, H., KOGEL, T., AND VANTHOURNOUT, B. 2005. A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE'05)*. Washington, DC, IEEE Computer Society, 876–881.
- KIM, D., YI, Y., AND HA, S. 2005. Trace-driven HW/SW cosimulation using virtual synchronization technique. In *Proceedings of the Conference on Design Automation*. 345–348.
- KUMAR, R. 2005. Heterogenous chip multiprocessors. *IEEE Comput.* 38, 32–38.
- KUMAR, R., FARKAS, K. I., JOUPPI, N. P., HY RANGANATHAN, P., AND TULLSEN, D. M. 2003. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. Washington, DC, IEEE Computer Society, 81.
- LAVAGNO, L., PASSERONE, C., SHAH, V., AND WATANABE, Y. 2005. A time slice based scheduler model for system level design. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 378–383.
- MEYER, B. H., PIEPER, J. J., PAUL, J. M., NELSON, J. E., PIEPER, S. M., AND ROWE, A. G. 2005. Power-performance simulation and design strategies for single-chip heterogeneous multiprocessors. *IEEE Trans. Comput.* 54, 6, 684–697.
- MILOJICIC, D. S., DOUGLIS, F., WHEELER, R., AND ZHOU, S. 2000. Process migration. *ACM Comput. Surv.* 32, 3, 241–299.
- PASSERONE, C., LAVAGNO, L., CHIDO, M., AND SANGIOVANNI-VINCENTELLI, A. 1997. Fast hardware/software co-simulation for virtual prototyping and trade-off analysis. In *Proceedings of the Conference on Design Automation*. 389–394.
- PAUL, J. M., BOBREK, A., NELSON, J. E., PIEPER, J. J., AND THOMAS, D. E. 2003. Schedulers as model-based design elements in programmable heterogeneous multiprocessors. In *Proceedings of the 40th Conference on Design Automation*. 408–411.
- PAUL, J. M., THOMAS, D. E., AND BOBREK, A. 2004. Benchmark-based design strategies for single chip heterogeneous multiprocessors. In *Proceedings of the International Conference on HW/SW Codesign and System Synthesis*. 54–59.
- PAUL, J. M., THOMAS, D. E., AND BOBREK, A. 2006. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Trans. VLSI Syst.* 14, 868–880.
- PAUL, J. M., THOMAS, D. E., AND CASSIDY, A. S. 2005. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.* 10, 3, 431–461.
- PHAM, D., ANDERSON, H.-W., BEHNEN, E., AND ET AL., M. B. 2006. Key features of the design methodology enabling a multi-core SoC implementation of a first-generation cell processor. In *Proceedings of the Conference on Asia South Pacific Design Automation*. 871–878.
- PHILIPS. <http://www.semiconductors.philips.com/products/nexperia/>.
- REGEHR, J. AND DUONGSAA, U. 2005. Preventing interrupt overload. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. 50–58.
- RHODES, D. L. AND WOLF, W. 1999. Overhead effects in real-time preemptive schedules. In *Proceedings of the International Workshop on HW/SW Codesign*. 193–197.
- SYSTEMC. <http://www.systemc.org/>.
- TEXASINSTRUMENTS. <http://focus.ti.com/omap/docs/omaphomepage.tsp>.
- YI, Y., KIM, D., AND HA, S. 2003. Virtual synchronization technique with os modeling for fast and time-accurate cosimulation. In *Proceedings of the International Conference on HW/SW Codesign and System Synthesis*. 1–6.
- YU, H., GERSTLAUER, A., AND GAJSKI, D. 2003. RTOS scheduling in transaction level models. In *Proceedings of the International Conference on HW/SW Codesign and System Synthesis*. 31–36.

Received August 2006; revised June 2007, August 2007; accepted September 2007