A

Host-Compiled Multi-Core System Simulation for Early Real-Time Performance Evaluation

PARISA RAZAGHI and ANDREAS GERSTLAUER, The University of Texas at Austin

With increasing complexity and software content, modern embedded platforms employ a heterogeneous mix of multi-core processors along with hardware accelerators in order to provide high performance in limited power budgets. To evaluate real-time performance and other constraints, full-system simulations are essential. With traditional approaches being either slow or inaccurate, so-called source-level or host-compiled simulators have recently emerged as a solution for rapid evaluation of complete system at early design stages. In such approaches, a faster simulation is achieved by abstracting execution behavior and increasing simulation granularity. However, existing source-level simulators often focus on application behavior only while neglecting effects of hardware/software interactions and associated speed and accuracy tradeoffs.

In this paper, we present a host-compiled simulator that emulates software execution in a full-system context. Our simulator incorporates abstract models of both real-time operating systems (RTOSs) and multi-core processors to replicate timing-accurate hardware/software interactions and enable full-system co-simulation. An integrated approach for automatic timing granularity adjustment (ATGA) uses observations of the system state to automatically control the timing model and optimally navigate speed versus accuracy conditions. Results as applied to industrial-strength platforms confirm that OS- and system-level effects can significantly contribute to overall accuracy and simulation overhead. By providing careful abstractions, our models can achieve full-system simulations at equivalent speeds of more than a thousand MIPS with less than 3% timing error. Coupled with the capability to easily adjust simulation parameters and configurations, this demonstrates the benefits of our simulator for early application development and design space exploration.

Categories and Subject Descriptors: I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms: Design, Performance

Additional Key Words and Phrases: Native simulation, abstract modeling, system-level design

ACM Reference Format:
Razaghi, P., Gerstlauer, A., Host-Compiled Multi-Core System Simulation for Early Real-Time Performance Evaluation. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 25 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

In todays embedded systems, software content is growing continuously to deal with increased complexities and tight development cycles. During early design exploration, system developers are interested in evaluating an application behavior on a particular architecture. However, system-wide interactions and dynamic behavior in complex parallel systems make static analysis challenging. Efficient full-system simulations therefore play an important role in the design process.

Multi-core processors have become popular both in general-purpose as well as in embedded computing, since they provide higher performance with less power con-

Author's addresses: P. Razaghi and A. Gerstlauer, Electrical and Computer Engineering Department, The University of Texas at Austin.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

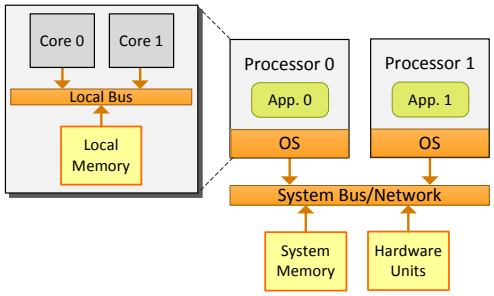


Fig. 1: Generic multi-processor/multi-core SoC (MPCSoC) architecture.

sumption [Blake et al. 2009]. In practice, such multi-core processors are integrated into a multi-processor platform in order to provide a heterogeneous system that meets all real-time and design constrains. Figure 1 shows a generic architecture of typical multi-processor and multi-core systems-on-chip (MPCSoCs). Each processor in the system can have one or more cores, which share a single memory address space and are managed by a single operating system (OS) in a symmetric multi-processing (SMP) context. In other words, application tasks are able to run on any core and can easily migrate between processor cores. By contrast, each multi-core processor in an overall multi-processor platform has its own memory address space managed by a dedicated or distributed operating system. As such, processors are organized in an asymmetric multi-processing (AMP) manner, where application tasks are partitioned among the processors and task migration rarely happens.

The complexities of the MPCSoC design space have made traditional cycle- or instruction-accurate simulators inefficient. Cycle-based simulators are highly accurate, but very slow, especially in a multi-core or multi-processor context. By contrast, virtual platform prototypes that employ binary translation coupled with abstract modeling of system peripherals can establish fast functional simulation, but provide little to no timing information. More recently, source-level and host-compiled simulators have emerged as an alternative that aims to address the need for fast and accurate simulation. In pure source-level approaches, application code is natively compiled and executed on a host machine to achieve the fastest possible functional simulation. For accuracy, the source code is further back-annotated with target-specific timing information obtained through estimation or measurement. To achieve full host-compiled simulation, back-annotated source code is then wrapped into abstract models of operating systems and processors, which integrate into existing transaction-level modeling (TLM) backplanes [Cai and Gajski 2003] on top of standard system-level design languages (SLDLs), such as SpecC [Gajski et al. 2000] or SystemC [Ghenassia 2005].

Management of intra- and inter-processor interactions in the OS kernels, device drivers and interrupt handling chains of complex MPCSoCs can carry a large overhead. As such, OS- and system-level interactions can contribute significantly to overall system performance. Furthermore, execution of associated detailed code in traditional instruction set simulations (ISSs) can lead to a large simulation overhead. However, in reality, designers may only care about the effect on application performance, and they are not concerned with the OS internals, for example. This provides an opportunity to abstract such details and develop host-compiled OS and processor models that faithfully replicate such effects without including any of the associated simulation overhead. Existing approaches thus far, however, have paid little attention to the question of how to optimally exploit speed and accuracy in such integrated abstractions and models.

In this paper, we present an efficient host-compiled multi-core processor simulator to support fast and accurate system-level exploration at early design stages. In previous work [Razaghi and Gerstlauer 2011], we have developed a host-compiled software simulator that focused on multi-core OS and scheduler models. Furthermore, we introduced a novel approach for automatic timing granularity adjustment (ATGA) in single-core OS simulations [Razaghi and Gerstlauer 2012a]. In this approach, the OS model uses knowledge about the system state to dynamically and automatically calibrate the simulation granularity and optimally navigate fundamental speed and accuracy tradeoffs. The contributions of this paper are twofold: (1) we adapt the ATGA approach for use with our multi-core OS model such that speed and accuracy tradeoffs are optimally maintained across a configurable number of cores, and (2) we extend the simulator to provide a full multi-core processor model that integrates automatic timing granularity adjustment in the OS model with timing-accurate yet fast simulation of external system interfaces and interrupt handling chains. We have developed flexible models that can be parametrized and configured to simulate a range of OS and processor combinations. We have implemented our simulator library in both SpecC and SystemC. A SystemC version is available for download at [HCSim 2014].

The rest of this paper is organized as follows: after presenting an overview of related work, we introduce the general structure of our host-compiled simulator in Section 3. In Section 4, we show how an application can be integrated into the simulator. In Section 5 and Section 6, details and internals of OS and processor models are discussed. We evaluate the efficiency of our simulator for early system exploration in Section 7. Finally, we present a summary and an outlook on future work in Section 8.

2. RELATED WORK

Conventional ISS-based software simulators using micro-architectural or interpreted simulation [Austin et al. 2002; Benini et al. 2005; Renau et al. 2005; Magnusson et al. 2002] can reach cycle accuracy at a speed of several kHz. At the other end of the spectrum, virtual platform simulators using dynamic binary translation can provide significant speedups (reaching simulation throughput of several MIPS), but only focus on functional simulation with no or very limited timing accuracy [Schnerr et al. 2005; Bellard 2005; Binkert et al. 2011; OVP Co]. Although the aforementioned types of simulators offer multi-core support, including CPU modeling at different levels of abstraction ranging from instruction-accurate models to fully cycle-accurate, micro-architectural ones, the need to simulate cross-compiled applications running on top of the complete binary code of an operating system kernel makes these simulator inefficient for fast and early integration and evaluation of complete systems.

Instead, source-level simulators aim to provide a fast yet accurate simulation platform by integrating instrumented source code of applications with a coarse-grain timing model that is obtained from the target architecture [Meyerowitz et al. 2008; Schnerr et al. 2008; Ceng et al. 2009; Lin et al. 2010]. For accurate performance evaluation, several approaches back-annotate the code with timing estimates that are obtained by compiling to an intermediate representation [Hwang et al. 2008; Bouchhima et al. 2009; Wang and Henkel 2012].

Source-level approaches can provide accurate simulation of single-task application behavior, but lack support for modeling of parallel applications and architectures. Host-compiled simulators further extend source-level simulation to include abstract models of the software execution environment [Gerstlauer 2010]. Originally, host-compiled simulators only focused on modeling of OS effects [Gerstlauer et al. 2003; Posadas et al. 2005; Miramond et al. 2009]. Later, those approaches were expanded into complete processor models that include timing-accurate interrupt chains and TLM-based bus interface [Gerin et al. 2007; Schirner et al. 2010]. Such host-

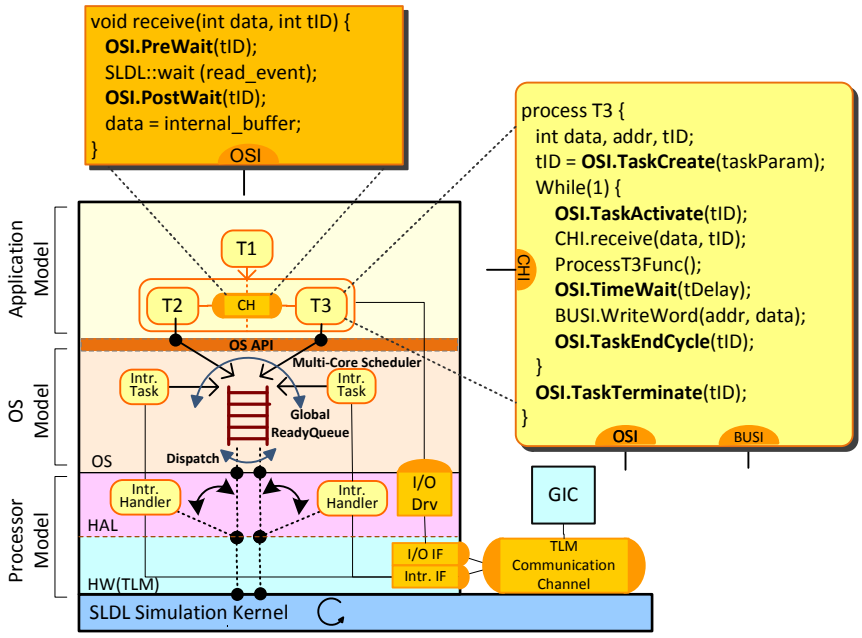


Fig. 2: Host-compiled simulation model.

compiled single-core simulators have been shown to run at speeds beyond 500 MIPS with more than 95% timing accuracy. In this paper, we propose novel approaches for host-compiled simulation of complete multi-core platforms at significantly improved speed and accuracy.

Improving the accuracy of high-level simulation while maintaining high performance has been the focus of many researchers. [Krause et al. 2008] present a hybrid ISS and RTOS modeling approach to combine cycle-accurate application simulation with fast OS scheduling and context switching. [Salimi Khaligh and Radetzki 2010] present an adaptive TLM simulation kernel, which changes the level of accuracy during simulation to the level expected by designers. [Stattelmann et al. 2011] propose an approach that precisely models the execution time of access conflicts in shared resources by using a proactive quantum allocator in a temporally decoupled simulation. [Schirner and Domer 2008] introduce a result-oriented method for accurate simulation of interrupts on host-compiled processor models by applying optimistic prediction and correction. In all cases, however, fundamental, statically determined speed and accuracy tradeoffs remain. By contrast, we propose approaches for adjusting granularities automatically, optimally and dynamically to achieve fastest possible simulation at highest possible accuracy.

3. HOST-COMPILED SIMULATION TECHNOLOGY OVERVIEW

Figure 2 shows our host-compiled multi-core simulator, which is based on a layered organization as introduced for single-core processor models in [Schirner et al. 2010]. At the top layer, the user application consists of a set of sequential and concurrent high-level, C-based processes. Tasks running on the same processor interact with each other via inter-processor communication (IPC) primitives, while they communicate with the external world using high-level bus transactions. For timing accuracy, the application code is back-annotated with sequential execution delays estimated or measured based on a selected target platform.

Applications are mounted on top of the simulator using a canonical, high-level OS interface. At the core of the simulation engine, an OS layer implements this interface and replicates a typical multi-core OS architecture to emulate the execution of application tasks on top of underlying SLDL kernel. The OS model thereby schedules, queues, dispatches and executes the application tasks according to a chosen SMP scheduling policy. Underneath, a hardware abstraction layer (HAL) in conjunction with a hardware layer constitute the processor model. The HAL integrates the software into the processor hardware models and includes necessary description of I/O drivers and interrupt handlers. Combined with interrupt processing in the OS layer, this replicates and emulates an accurate interrupt handling mechanism. The hardware layer also provides interrupt and bus interfaces to the external communication infrastructure. Generally, the bus interface can be developed at an arbitrary level of abstraction; here, a transaction level model (TLM) of communication is used to establish a fast simulation environment. A high-level, generic interrupt controller (GIC) model collects interrupts from the hardware side and manages their distribution across processor cores.

Finally, at the base, the complete simulator is implemented over a standard system level design language (SLDL) that provides the required concurrency, timing and event handling infrastructure on a host machine.

3.1. Automatic Timing Granularity Adjustment (ATGA)

In a conventional simulation mode, back-annotated delays define the base granularity of the simulation. In such a setup, the scheduler is only called after advancing the simulation time to allow for preemption of the current task by any higher priority task that became available in the meantime. As a result, errors in the task preemption model are a direct function of back-annotated granularities. However, as we have been able to show previously, under certain circumstances, errors in discrete preemption models can potentially exceed the bounds set by the timing granularity by a large amount [Razaghi and Gerstlauer 2012b].

By contrast, in our automatic timing granularity adjustment (ATGA) approach, the OS kernel internally monitors the state of the system and automatically controls the timing model of the simulation to invoke the scheduler whenever a task preemption is required. Thus, simulation speed and accuracy is independent of the granularity of back-annotated delays, which frees designers from having to settle on a particular, difficult to evaluate and predict tradeoff. Instead, the OS kernel itself accumulates or breaks delays into a number of smaller steps as needed, automatically providing the best timing granularity for fully accurate and fast results.

Generally, timing errors happen when a task is running and, while advancing simulation time, a higher priority tasks becomes ready without the scheduler getting a chance to immediately preempt the current one. This situation can occur in the following cases: (a) a periodic task reaches its next iteration time, (b) the interrupt handler triggers an interrupt task, or (c) a blocked or sleeping task returns to the ready state when the running task notifies an event or resumes it. In such cases, the start of the newly released task is delayed until the expiration of the current time granule.

In the ATGA approach, the OS kernel eliminates task preemption errors by switching between two timing modes: predictive mode and fallback mode. In predictive mode, the OS monitors the state of periodic tasks running on the system and uses this information to predict the next possible preemption point specifically for situations in case (a). If a back-annotated delay is larger than the predicted interval, the OS kernel divides the delay into smaller intervals in order to invoke the scheduler at the predicted time. Conversely, the exact next preemption point is unknown for cases (b) and (c), i.e. whenever a task is waiting for an internal or external event. In these cases, the OS model falls back to a fine preemption check until all events are captured.

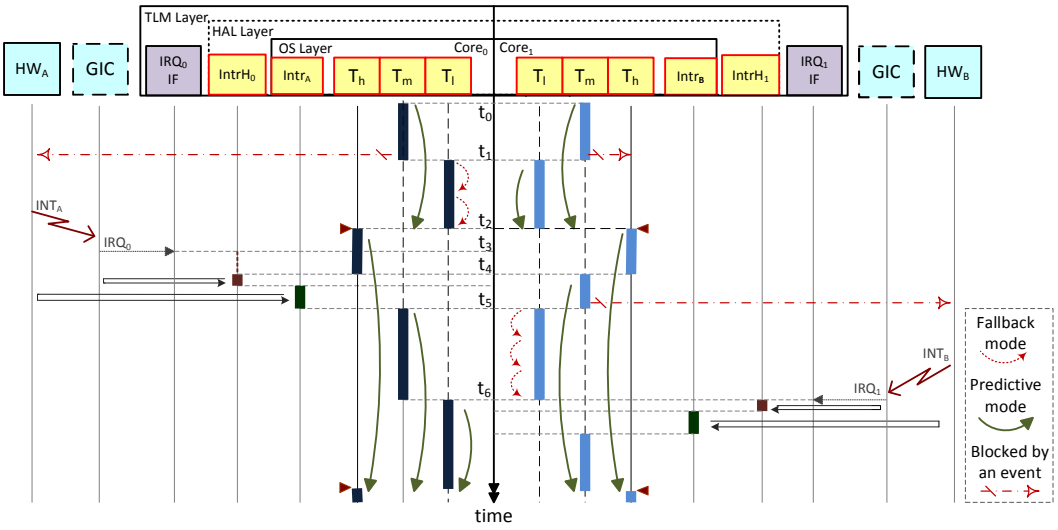


Fig. 3: Host-compiled simulation trace.

3.2. Host-Compiled Simulation Example

To illustrate the execution sequence of the integrated host-compiled model, we show a simulation trace of two task sets running on a dual-core platform under a partitioned scheduling strategy (see Figure 3). Each set contains three tasks with high, medium and low priorities (named T_h, T_m, and T_l respectively). Sets are mapped to run on separate cores. The highest priority task is modeled as a periodic task. Tasks may communicate with each other or external hardware via interrupt signals.

At the beginning of the simulation, T_h is in *Idle* state and T_m is therefore scheduled on both cores. At time t_1, T_m running on $core_0$ is blocked on an external event, and T_l is scheduled on that core. From this point forward, the OS switches to fallback mode on $core_0$, since a higher priority task is waiting for an interrupt. At the same time, T_m on $core_1$ waits for a response from T_h, and T_l is scheduled accordingly. Both cores can be in predictive and fallback modes independently, and the OS remains in predictive mode for $core_1$, since it is aware that neither T_m nor T_h can be scheduled before the next release time of the periodic task T_h (time t_2).

At time t_3, the hardware sends an INT_A interrupt to the GIC, which is programmed to route that interrupt to $core_0$. Accordingly, $core_0$'s interrupt interface activates the corresponding interrupt handler. However, we can delay the execution of the interrupt handler until the finish time of T_h. Such a situation allows the OS to stay in predictive mode, where it will not call the scheduler until time t_4, the finish time of T_h. The interrupt handler then communicates with the GIC and activates the corresponding interrupt task for further interactions with the hardware. Although such an interrupt modeling approach introduces a small timing error in the execution of T_h, the interrupt task and the interrupt handler, the simulation is kept fast while these errors are typically negligible. Finally, the interrupt task $Intr_A$ releases T_m, such that T_m and subsequently T_l execute in predictive mode until the start of the next period of T_h.

Now consider the simulation trace on $core_1$: after releasing T_m again once T_h has run, T_m blocks on an external interrupt at time t_5 while T_h is *Idle*. Task T_l is therefore scheduled and the OS switches to fallback mode in order to continuously monitor for possible interrupts. As such, when the interrupt request is captured by $core_1$ at time t_6, the OS schedules the activated interrupt handler immediately and provides a fully accurate interrupt handling sequence.

```
1   /* OS initialization and startup */
2   void Init(OSPARAM param);
3   void Start(void);
4   /* Task management */
5   int TaskCreate(TASKPARAM param);
6   void ParStart(int taskID); /* fork */
7   void ParEnd(int taskID); /* join */
8   void TaskActivate(int taskID);
9   void TaskSleep(int taskID);
10  void TaskResume(int taskID);
11  void TaskEndCycle(int taskID);
12  void TaskTerminate(int taskID);
13  /* Delay modeling and event handling */
14  void TimeWait(long long nSec, int taskID);
15  void PreWait(int taskID);
16  void PostWait(int taskID);
17  void PostNotify(int taskID, int blockedTaskID);
18  /* Interrupt handling */
19  void IntrTrigger(int intrID);
20  int CreateIntrHandler(int coreID);
21  void IEnter(int coreID, int handlerID);
22  void IReturn(int coreID);
```

Fig. 4: High-level OS interface.

4. APPLICATION MODEL

In host-compiled simulators, application code is captured at the source level in order to achieve a fast simulation and eliminate low-level implementation details. In the following, we present the simple and canonical parallel programming model used to develop applications and integrate them into our host-compiled simulator. In this approach, a designer only need to describe the functionality of application tasks. To describe inter-task communication, the simulator provides a comprehensive library of standard communication primitives and channels. Referring to Figure 2, a model of a typical application task and an internal implementation of a communication channel are shown, which are further detailed in the rest of this section.

4.1. Task Modeling

As discussed above, the complete simulator is developed over a standard SLDL. Accordingly, application tasks are modeled as high-level, hierarchical SLDL processes, which are connected to the simulator via the underlying OS application program interface (API), shown in Figure 4. Task behavior is described by conventional C functions and their target-specific execution delays are back-annotated into the code once at compile time. The source code is instrumented with the required timing information using `TimeWait()` methods of the OS API.

During the system startup phase, the `Init()` method initializes the OS model data structures and defines the OS parameters, including the number of supported cores and the default simulation quantum. The `Start()` method is then used to enter multi-core scheduling after all tasks have been attached to the model.

During the task creation phase, tasks are added to the OS by calling `TaskCreate()`, which allocates an internal representation inside the OS model. Furthermore, this method allows application designers to explore a wide range of tasks properties and behavior. The currently supported parameters are listed as follows:

(1) *Type:* each task can be defined exclusively as an aperiodic, a periodic, or a kernel special task model;
(2) *Period:* valid only for periodic tasks;
(3) *Priority:* the static priority level of tasks;
(4) *Time Slice:* time interval that a task is allowed to execute without preemption by other tasks with the same priority;
(5) *Affinity:* a bitmap representing the set of cores allowed to execute the task;
(6) *Initial Core:* the initial core that is allowed to run the task at start time.

The `TaskCreate()` method returns a unique ID, which is passed to the OS kernel in all following task-related API calls.

At the start of simulation, task threads are spawned via the SLDL. They then register themselves with the OS via a call to the `TaskActivate()` method at the beginning of their execution. This allows the OS model to collect all threads and enter them into the scheduler. At the end of their execution, tasks remove themselves from the OS kernel by calling `TaskTerminate()`. If supported by the underlying SLDL, tasks can fork children and temporarily remove themselves from OS scheduling (`ParStart()`) until all children are collected on the SLDL level (`ParEnd()`).

Finally during the execution, a task can remove itself from the active core temporarily by either calling `TaskEndCycle()` or `TaskSleep()`. The former moves the calling periodic task into idle mode until its next release time. The latter puts the task in sleep mode until another task calls a corresponding `TaskResume()` method.

4.2. Channel Library

Inter-task communication is implemented by one-way or two-way message passing channels. Channels are described using the underlying SLDL event/notify primitives, which are wrapped into OS APIs that will allow the OS to accurately control the execution order of tasks connected to the channel. Designers will typically not have to deal with event handling directly. Instead, the simulator provides a reimplementation of a rich library of communication channels and primitives that are properly hooked into the OS model.

As shown in Figure 2, each "wait for event" statement is encapsulated by `PreWait()` and `PostWait()` methods. `PreWait()` simply removes the running task from the OS kernel and lets the scheduler start the next ready task on the current core. As soon as the event is captured, `PostWait()` returns the blocked task into the ready state and waits until the task is scheduled by the OS kernel.

In point-to-point communication channels, where sender and receiver of messages are fixed during the simulation, "notifying an event" is followed by a call to an OS `PostNotify()` method. If the just unblocked task has a priority higher than the current one, `PostNotify()` will immediately perform a task switch and call the OS scheduler. In this way, a more accurate task scheduling is modeled, since high priority unblocked tasks are not required to wait until the next point that the scheduler is called.

5. OS MODEL

Figure 5 depicts an overview of our abstract multi-core OS model, which is designed to perform three main functionalities: task management, multi-core scheduling emulation, and coordination of the simulated timing model.

During its execution, each simulated task can be in five states, and tasks move to different states by calling a corresponding OS API method. In order to emulate the state of the system, the OS model maintains tasks in five internal queues: a *Ready* queue holds tasks that are ready to execute and is sorted based on a user-defined scheduling policy. An *Idle* queue holds periodic tasks that have called the kernel's `TaskEndCycle()`

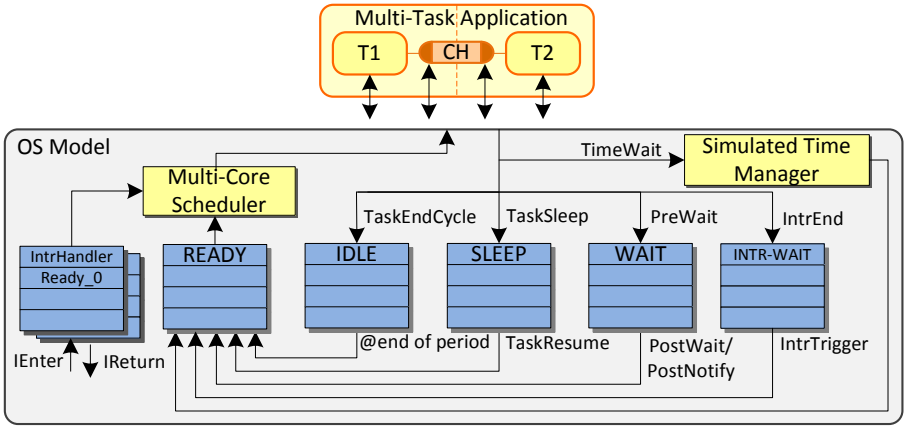


Fig. 5: Abstract OS model.

method. The *Idle* queue is ordered based on the release time of each task's next iteration. Idle tasks are retrieved from the head of queue and placed in the *Ready* queue by the OS kernel at the start time of their next period. A *Sleep* queue holds tasks that have been suspended until they are resumed again. Tasks waiting for an event are suspended and transferred to a *Wait* queue. As part of modeling the top half of the OS interrupt handling chain, an *IntrWait* queue holds special interrupt tasks until a core-specific interrupt handler calls the `IntrTrigger()` method to move them into the *Ready* queue. Since the core-specific interrupt handlers are treated as special high-priority tasks by the OS scheduler, a separate *IntrHandlerReady* queue is dedicated to each core. Interrupt handlers are activated and moved into a corresponding queue when the `IEnter()` method is called by the processor interrupt interface. After triggering interrupt tasks, interrupt handlers deactivate and remove themselves from the *IntrHandlerReady* queue by calling the `IReturn()` method.

In the context of SMP scheduling, there are two major task scheduling schemes distinguished by the number of *Ready* queues associated with each core: partitioned and global scheduling schemes. In a partitioned scheme, each core has a separate *Ready* queue and tasks are initially assigned to a fixed queue. The scheduler picks tasks for a core only from the associated queue. In a global scheme, the scheduler maintains only a single *Ready* queue and tasks can be freely assigned to the next available core. A global queue can lead to a better utilization but is less scalable and may result in degraded performance due to cache pollution when tasks move between cores too frequently [Lauzac et al. 1998]. Our OS model supports both scheduling schemes, such that designers can explore the right structure for a particular application. Depending on the scheduling scheme, a single *Ready*, *Idle*, *Sleep*, *Wait* and *IntrWait* queue is either shared among all cores as depicted in Figure 2 and Figure 5, or multiple such queues are replicated one per core [Razaghi and Gerstlauer 2011]. In the following subsections, we present further details on the internals of the OS' scheduler and time management blocks.

5.1. Multi-Core Task Scheduling

The core component of the OS model is a replicated, generic multi-core scheduler, the body of which is shown in Figure 6. The scheduler is an internal function of the OS model and is called by the OS API methods whenever a task switching is possible or required. The main functionality of the scheduler is to retire the currently active task on a core, if any, and place it in a proper place in the right *Ready* queue. We utilize a

```
    Function Scheduler (int coreID):
1     queueID := GLOBAL_SCHEDULING ? ø : coreID
2     oldTask := runningTask[coreID]
3     runningTask[coreID].RemainingTimeSlice − = (CurrentTime() - runningTask[coreID].StartTime)
4     runningTask[coreID].State := Ready
5     if runningTask[coreID].RemainingTimeSlice ≤ ø then
6       runningTask[coreID].RemainingTimeSlice := runningTask[coreID].TimeSlice
7       FIFO(ReadyQueue[queueID], runningTask[coreID])
8     else
9       LIFO(ReadyQueue[queueID], runningTask[coreID])
10    endif
11    Dispatch(coreID)
12    Wait4Sched(oldTask)
```

Fig. 6: Multi-core OS scheduler.

```
    Function Dispatch (int coreID):
1     if !Empty(IntrHandlerReadyQueue[coreID]) then
2       IntrHandlerID := PeekFirst(IntrHandlerReadyQueue[coreID])
3       runningTask[coreID] := IntrHandlerID
4       IntrHandlerList[IntrHandlerID].State := RUN
5       SendSched(IntrHandlerID)
6     else
7       OSDispatch(coreID)
8     endif
```

Fig. 7: Multi-core task dispatcher.

time slice notion to model FIFO or round-robin (RR) scheduling among tasks that have the same priority. When the OS runs the scheduler for a specific core, it calculates the remaining time slice of the current active task on the desired core. This is done by subtracting the time consumed by the task, which is computed as the difference between the current simulated time and the time the task was last put onto the core (line 3 in Figure 6). Then, the current task is moved to the corresponding *Ready* queue based on the new value of the time slice. If the time slice reaches zero, the task is added at the end of its priority list where it will be scheduled after all current ready tasks with the same priority. In addition, the task's remaining time slice value is reset back to its configured value. Otherwise, the task will be placed back at the beginning of the priority list from where it will be scheduled immediately again, right before any other ready tasks with the same priority. Consequently, in RR scheduling the value of the time slice defines the portion of time that every task is allowed to be executed without any preemption by tasks of the same priority, while setting an infinite time slice value will result in a FIFO scheduler.

At the end of the scheduler, a `Dispatch()` function will be called to assign a new task to the current core. Figure 7 shows the implementation of `Dispatch()`. As a first step, if there is an active interrupt on the core, a corresponding interrupt handler (see Section 6.1) will be assigned to that core (lines 2-5) instead of a regular task. The aforementioned core-specific interrupt handler queues (*IntrHandlerReady*) thereby manage the priorities and selection among pending interrupts if multiple interrupt vectors are supported by the modeled processor.

If there are no pending interrupts, an OS-specific dispatcher will be called to assign normal tasks to the core. We present the implementation details of the `OSDispatch()` function for both global and partitioned queue models (Figure 8). In both cases, the

```
    Function OSDispatch (int coreID):
1     queueID := coreID
2     LoadBalance(coreID)
3     if !Empty(ReadyQueue[queueID]) then
4       runningTask[coreID] :=
5               getFirst(ReadyQueue[queueID])
6       runningTask[coreID].State := RUN
7       runningTask[coreID].StartTime :=
8                                   CurrentTime()
9       SendSched(runningTask[coreID])
10    else
11      runningTask[coreID] := Null
12    endif
```

(a) Partitioned-queue scheme.

```
    Function OSDispatch (int coreID):
1     ReadyQueue_Lock.Acquire()
2     runningTask[coreID] := Null
3     if !Empty(ReadyQueue[ø]) then
4       activeTask := getFirst(ReadyQueue[ø], coreID)
5       if activeTask ! = Null then
6         runningTask[coreID] := activeTask
7         runningTask[coreID].State := RUN
8         runningTask[coreID].StartTime := CurrentTime()
9         SendSched(runningTask[coreID])
10      endif
11    endif
12    ReadyQueue_Lock.Release()
```

(b) Global-queue scheme.

Fig. 8: OS dispatcher.

function selects the highest priority task in the *Ready* queue and assigns it to run on the current core. Ready queues are sorted by task priorities, and tasks with the same priority are arranged based on time slices, as described above. In case of partitioned queues (Figure 8 (a)), a load balancing to optionally, e.g. at regular intervals, migrate tasks between queues is performed before dispatching. In case of a global queue structure (Figure 8 (b)), a semaphore controls all accesses to the shared queue. Note that in both schemes, the tasks to be migrated or the first task allowed to run on a particular core are selected (in the queue's `getFirst()` method, line 4) based on the task affinity. In other words, the current core can be idle even if the *Ready* queue is not empty.

After selecting a new task, the dispatcher releases it by calling `SendSched()` to notify an SLDL event associated with the chosen task. After returning from the `Dispatch()` call at the end of the scheduler, the current task on the given core in turn suspends itself on its own event. Leveraging SLDL events assigned to each task, this emulates actual context switches. Note that if no higher priority or other sibling task is available, the current task may simply dispatch itself and be immediately triggered again.

5.2. Timing Model Management

In addition to basic OS services, the OS kernel simulates task execution delays using underlying SLDL primitives. As we explained in Section 3.1, the OS model adjusts the granularity of task delays by switching between predictive and fallback modes. Figure 9 shows the algorithm for predicting the next preemption time among periodic tasks. Since the *Idle* queue is sorted based on the tasks' next release times, the preemption point is defined by the first task with a priority higher than the currently running one. Note that for a global scheduling scheme, the `AllowedRun()` method checks if a core is allowed to run a task based on the task's user-defined affinity.

As mentioned before, the OS switches to fallback mode when it cannot predict the next preemption point, i.e. a high-priority task is waiting to be released. However, the OS kernel does not need to switch to fallback mode in all cases of tasks waiting for any event. For example, assume an inter-task communication chain in which a set of tasks are blocked waiting for other tasks in the chain. The task next to last in the chain has a higher priority than the running task, but is blocked by a lower-priority task. Since the lower-priority task cannot be scheduled while the current task is running, the whole chain can never be triggered and prediction is not affected. Even if the low priority task at the end of the chain is blocked on an external event, unblocking the lower-priority task can never preempt the running task. Assuming small interrupt handler delays

```
    Function PredictNextPreemptionTime (task runningTask):
1     predictedDelay := SIMULATION_QUANTUM
2     scheduledCoreID := GetSchedCoreID(runningTask)
3     queueID := GLOBAL_SCHEDULING ? ∅ : scheduledCoreID
4     for all idleTask in IdleQueue[queueID] do
5       if idleTask.Priority ≥ runningTask.Priority and AllowedRun(idleTask, scheduledCoreID) then
6         predictedDelay := idleTask.NextPeriodTime - CurrentTime()
7         return predictedDelay
8       endif
9     endfor
```

Fig. 9: OS predictive mode.

```
    Function FallbackMode (task runningTask):
1     currentCoreID := GetSchedCoreID(runningTask)
2     queueID := GLOBAL_SCHEDULING ? ∅ : currentCoreID
3     for all waitingTask in WaitQueue[queueID] do
4       if waitingTask.Priority ≥ runningTask.Priority and AllowedRun(waitingTask, currentCoreID) then
5         blockingTask := waitingTask.blockingTaskID
6         if (blockingTask == Unknown) or
7               (blockingTask.Priority ≥ runningTask.Priority and blockingTask.State == IntrWait) then
8           return true
9         endif
10      endif
11    endfor
12    return false
```

Fig. 10: OS fallback mode.

and thus ignoring preemptions by the interrupt handler itself, this means that with a minor interrupt timing error, the fallback mode can also be ignored in this situation.

Furthermore, situations in which a high-priority task is blocked on a periodic task in the *Idle* queue can be handled by simulating the system at the predicted level. Likewise, if a higher-priority task is waiting for the currently running or a sleeping task, the preemption and context switch can be performed directly in the event notification or `TaskResume()` kernel methods, at the point when they are called by the running task, as explained for the case of `PostNotify()` in Section 4.2. In conclusion, the OS only moves to fallback when a task with a higher priority is waiting for an external event, i.e. a higher priority task is in the *Wait* queue and is blocked by a task in the *IntrWait* queue. Since fallback mode degrades simulation performance significantly, such finer control can maintain simulation performance without losing accuracy. Figure 10 shows the detailed fallback mode algorithm. In order to determine inter-task dependencies, we annotate all IPC primitives in the provided channel library to record the ID of the sending task that a receiving task is blocked on. The OS only turns to fallback mode if a higher priority task is blocked by an *Unknown* task (line 6) or is waiting for an external event (line 7).

In a multi-core context, the OS model further monitors the state of application tasks that are driven by an interrupt that is handled by a different core, such that the OS can adjust the predicted time or switch to fallback mode accordingly. In such cases, if the core handling the interrupt is responsible for releasing a high-priority interrupt-driven application task on the other core, it may have to adjust its predicted time or go to fallback mode even if it otherwise would not. In a situation in which the other core's interrupt-driven application task is in *Idle* state, the interrupt handling can be

```
    Function IntrDependencyCheck (task runningTask):
 1    currentCoreID := GetSchedCoreID(runningTask)
 2    for all otherCoreID with otherCoreID ! = currentCoreID do
 3      for all intrTask in IntrWaitQueue[otherCoreID] do
 4        if intrTask.HandledCore == currentCoreID or intrTask.HandledCore == Unknown then
 5          (adjDelay, fallback) := (∞, true)
 6          blockedTask := intrTask.blockedTaskID
 7          if blockedTask.State == IDLE then
 8            adjDelay := min(adjDelay, (blockedTask.NextPeriodTime - CurrentTime()))
 9            fallback := false
10          elsif blockedTask.Priority < runningTask[otherCoreID].Priorirty then
11            adjDelay := min(adjDelay, (runningTask[otherCoreID].NextWakeupTime - CurrentTime()))
12            fallback := false
13          endif
14        endif
15      endfor
16    endfor
17    return (adjDelay, fallback)
```

Fig. 11: Inter-core interrupt dependency check.

delayed until the next release time of this periodic task without actually changing the execution sequence. Similarly, if the priority of the interrupt-driven task on the other core is lower than the priority of the task currently running there, the OS model can simply adjust the predicted time to become the next wake-up time on that other core. The scheduler wake-up time is determined by the OS model internally whenever the simulation time is advanced, i.e. it is guaranteed that no context switch can happen in the meanwhile. Note that the predicted time on the interrupt-handling core can not be advanced further than that. Even if there is no higher-priority task waiting on the other core right now, the task running there can choose to enter a wait state and thus change the scheduling mix at any time.

In all other situations, the OS cannot predict the next possible scheduling point and switches to fallback mode. Figure 11 shows the method that checks inter-core interrupt-dependencies and calculates a new predicted delay if needed. To keep track of interrupt dependencies, interrupt handler models send their core ID to interrupt tasks they trigger. The `IntrDependencyCheck()` method explores *IntrWait* queues on other cores to see if some interrupt tasks can be triggered by the current core's interrupt handler (line 4). If such a condition exists, the application task waiting for that interrupt is determined via the receiving task ID recorded using similar channel library annotations as described earlier (line 6). Finally, a new adjusted delay is calculated and fallback mode conditions are determined (lines 7-13). At the end, the adjusted delay and the fallback mode check are reported back to the OS model.

Finally, the overall timing model of our simulator is managed in the `TimeWait()` method of the OS API. This method is called by the user code to model execution delays of tasks and uses the underlying SLDL "wait for time" primitive to advance simulation time. The pseudo code of the `TimeWait()` method is shown in Figure 12. In order to achieve highest possible speed even with fine-grained back-annotated delays, the OS accumulates back-annotated delays of the current task until the next predicted preemption point is reached or the OS needs to switch into fallback mode (lines 1-4). The OS then advances the simulation time using the predicted delay, optionally utilizing an event-driven fallback check, and calls the scheduler to perform a context switch, if necessary (lines 7-15). This loop continues as long as the accumulated delay is greater than the predicted delay or the OS is still in fallback mode.

```
    Function TimeWait (long long nsec, task runningTask):
 1    runningTask.AccDelay + = nsec
 2    FB := FallbackMode(runningTask)
 3    (adjustedDelay, ID_FB) := IntrDependencyCheck(runningTask)
 4    predictedDelay := min(PredictNextPreemptionTime(runningTask), adjustedDelay)
 5    while runningTask.AccDelay > predictedDelay or FB or ID_FB do
 6      runningTask.NextWakeupTime := CurrentTime() + predictedDelay
 7      if FB or ID_FB then
 8        startTime := CurrentTime()
 9        SLDL::wait(predictedDelay, OS::scheduleEvent)
10        predictedDelay := CurrentTime() - startTime
11     else
12        SLDL::wait(predictedDelay)
13      endif
14      runningTask.AccDelay − = predictedDelay
15      Scheduler()
16      FB := FallbackMode(runningTask)
17      (adjustedDelay, ID_FB) := IntrDependencyCheck(runningTask)
18      predictedDelay := min(PredictNextPreemptionTime(runningTask), adjustedDelay)
19    endwhile
```

Fig. 12: ATGA timing model.

To realize fallback mode, an OS-internal event (*schedulerEvent*) is introduced that enables asynchronously interrupting long time consumption periods (line 9 in Figure 12). Since the fallback mode is only entered when a high-priority task is waiting for an interrupt, the *schedulerEvent* is triggered by interrupt handlers in the HAL whenever an interrupt occurs (see Section 6.1). This in turn will abort the `wait()` statement in the SLDL kernel, at which point control is returned to the OS model to perform a corresponding scheduling check.

All in all, this timing model provides error-free task scheduling at the highest possible speed by accumulating and dividing user-defined timing granularities to internally maintain an independently controlled, optimal strategy to advance simulation time.

6. MULTI-CORE PROCESSOR MODEL

In order to provide fast yet accurate feedback about timing-accurate HW/SW interactions beyond OS scheduling, we have developed a high-level processor model, which emulates both TLM bus accesses and a detailed multi-core interrupt handling chain. Figure 13 depicts the connections across different layers of our simulator. At the inner most layer, a user application is directly connected to the OS layer and accesses the scheduling services via the provided OS API. In addition to the OS services detailed in Section 5, the OS layer provides high-level communication primitives for sending and receiving inter-processor application-level messages via the HAL. Together, the OS and HAL thereby realize models of drivers that transform application-level messages all the way down to corresponding transaction-level bus accesses plus interrupt-driven or polling-based synchronizations, if required. Finally, the HW layer models external bus communication via a TLM bus channel. The HW layer also emulates monitoring of processor interrupt signals and associated processor exceptions. The HAL combined with the HW layer constitute the processor model. In this way, the simulator can be integrated into any standard TLM backplane for co-simulation in an overall multi-processor system environment. In the following, we will describe various aspects of the processor model in more detail.

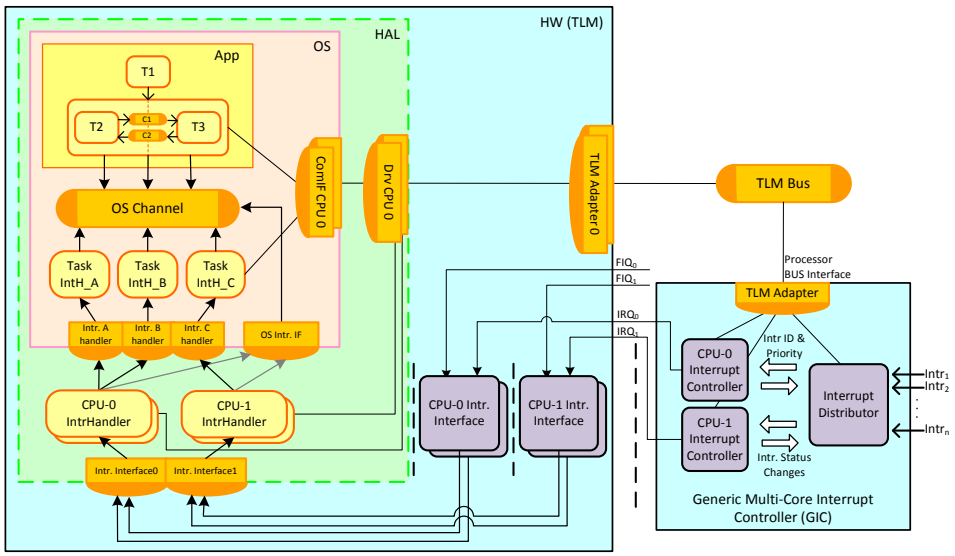


Fig. 13: High-level multi-core processor model.

6.1. Interrupt Handling Model

In addition to the overall structure of the processor model, Figure 13 depicts extra hardware components, inter-layer interfaces, and specialized OS-level tasks to replicate a general multi-core interrupt handling chain in our host-compiled model. From the hardware side, core-specific interrupt requests (IRQx) are generated by a generic multi-core interrupt controller (GIC) model, which manages the distribution of interrupt signals across processor cores. The internal structure of the interrupt controller will be discussed next in Section 6.2.

Inside the processor model, the HW layer contains core-specific SLDL processes (*IntrInterface*) that are sensitive to changes on the external interrupt inputs (IRQx). Whenever an interrupt request is asserted, the corresponding *IntrInterface* notifies the OS kernel by calling the `IEnter()` method of the OS API, exported to the HW layer via the HAL. `IEnter()` then moves the desired interrupt handler into the corresponding core's *IntrHandlerReady* queue (as shown in Figure 5) and triggers the OS-internal *schedulerEvent* to inform the OS of the recently activated interrupt. Depending on fallback versus predictive mode, the OS will terminate the current time-wait primitive and call the scheduler to perform a context-switch (as illustrated in Section 5.2). As discussed previously, the OS scheduler always first checks the *IntrHandlerReady* queue for active interrupt handlers in order to model processor suspension in response to external interrupt events.

In this approach, interrupt handlers are modeled as special, high priority tasks associated with each core. Interrupt handlers are created and added to the OS kernel by the HAL via the `CreateIntrHandler()` methods of the OS API. When an interrupt handler is scheduled by the OS kernel, it communicates with the GIC via a TLM bus channel to determine and acknowledge the interrupt source. It next triggers a special interrupt task associated with the interrupt source via a call to the `IntrTrigger()` method of the OS API. Before the interrupt handler returns to the OS kernel, it removes itself from *IntrHandlerReady* queue by calling the `IReturn()` method. Finally, user-supplied code in the interrupt tasks can communicate with external hardware, with application tasks or with the OS model, e.g. to spawn additional processing tasks.

6.2. Generic Multi-Core Interrupt Controller

We have developed a generic multi-core interrupt controller (GIC) model that manages interrupt distribution among the processor cores and generates interrupt request (IRQ) signals associated with each core. The high-level GIC model is a configurable, generic multi-core interrupt controller that is modeled after typical real-world components, such as the ARM Generic Interrupt Controller Architecture [ARM Co]. Our model supports up to 32 edge-triggered hardware interrupts and is able to manage the interrupts for an arbitrary number of processors/cores. A user can program the interrupt controller to define interrupt priority and target core for each interrupt source. The GIC model replicates a 1-N model for handling interrupts, i.e. it ensures that only one processor handles a captured interrupt.

Internally, our high-level GIC model is composed out of one centralized interrupt distributor and per-processor CPU interfaces (as shown in Figure 13). The distributor monitors incoming interrupts and dispatches the highest priority asserted interrupt to the associated CPU interface, which is determined by the programmed target core list. The CPU interface thereby asserts the IRQ signal to its connected core and takes care of the communication between the processor and GIC.

To manage interrupt detection and distribution, the GIC model identifies each interrupt by an ID and maintains the interrupts' state transitions. When an interrupt is asserted by a connected hardware, the GIC moves that interrupt to the *Pending* state. The distributor can then detect all pending interrupts and send the highest priority one to the corresponding CPU interface. A pending interrupt moves to the *Active* state whenever it is handled by the corresponding core, i.e. the associated interrupt handler reads the "Acknowledge" register. An active interrupt can move to *Inactive* (initial state) or *Active & Pending* states when the interrupt handler writes to the "End of Interrupt" register or another interrupt signal with the same ID is captured, respectively.

In such a detailed GIC model, three context switches are required in the simulator for an external interrupt to propagate until the actual interrupt handler is executed. Such a detailed model therefore carries a large simulation overhead. We have developed an alternative, lightweight GIC model that reduces the overhead by a factor of two. This model only contains core-specific processes, which identify the highest priority pending interrupt for the connected processor and communicate with the processors and the associated interrupt handlers directly. To achieve a faster interrupt routing and consequently a faster simulation, the priority of an interrupt is only defined by its ID and is not programmable. Furthermore, this model does not replicate a 1-N implementation. Hence, designers need to be careful when programming the GIC to ensure that each input interrupt is only mapped to a single target core.

On the whole, the presented processor and interrupt controller models and interconnections provide an infrastructure that enables accurate modeling of interactions between system hardware components and multi-core processors.

7. EXPERIMENTAL RESULTS

We evaluated different aspects of our simulator using a set of successive experiments. First, we evaluated the accuracy of our high-level interrupt handling models and the accuracy and the simulation performance of our OS and processor models on a suite of artificial task sets and compute-intensive multi-threaded benchmarks. We then also explored different architectures of an industrial-strength example to show the benefits of our simulation platform for early design space exploration. All experiments were performed on a 2.67 GHz Intel Core i7 workstation using the SpecC version of our simulator.

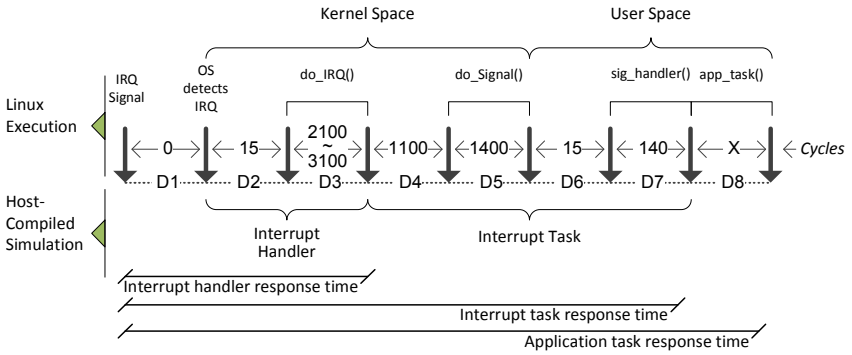


Fig. 14: Modeling of Linux interrupt handling.

We used the Open Virtual Platform (OVP) [OVP Co] as a reference ISS for evaluating the accuracy of our models. OVP consists of an instruction-accurate simulator (OVPSim), fast processor models, and behavioral peripheral modeling and programming APIs, which enable full-system modeling and simulation. Furthermore, we used Imperas Verification, Analysis and Profiling (M*VAP) tools to measure the execution cycles of applications in kernel and user space [Imperas Ltd]. Since OVP is based on a timing model of one cycle per instruction (CPI=1), execution delays were determined as the product of the number of executed instructions and the target processor clock period.

The reference platform consisted a quad-core Cortex-A9 ARM processor running a Linux 2.6.39 SMP kernel at a frequency of 1 GHz. OVP was configured to run at instruction-level granularity. The OVP peripheral modeling library was used to implement extra hardware components. Correspondingly, loadable kernel modules were developed to integrate drivers for the hardware into the platform and application. Hardware accesses were implemented using interrupts and memory-mapped I/O.

For timing-accurate interrupt modeling, we measured the number of instructions executed by Linux when handling an interrupt. Figure 14 shows the sequence of interrupt events in Linux and their mapping to the host-compiled simulation model. Since we modeled the system bus and GIC as untimed, high-level peripherals, the Linux kernel starts handling interrupts immediately after an interrupt is signaled. In our setup, the interrupt handler notifies an application process using Linux real-time signals. The execution trace shows that the interrupt handler delay varies between 2,100 to 3,100 cycles[1], depending on the system state and the internal implementation of signal queues in the Linux kernel. The Linux kernel then delivers the notified signal to the corresponding process by evoking an associated signal handler (`do_Signal()` in kernel space and `sig_Handler()` in user space). Note that in Linux, these signal handlers are executed in the context, i.e. at the same priority and time as their associated processes. Finally, the signal handler determines the interrupt source and calls a corresponding application service routine, which in turn communicates with the main process by releasing a POSIX semaphore to signal that the interrupt has been received.

To accurately model the Linux interrupt handling chain, we map the functionality of interrupt and signal handlers to interrupt handlers and interrupt tasks in our host-compiled model. Matching Linux behavior, interrupt tasks are thereby assigned the same priority as their associated user tasks. Finally, we back-annotate interrupt handler and task models with corresponding delays ($D2 + D3$ and $\sum(D4..D7)$, respectively).

[1]For the experiments presented in this paper, the number of cycles and instructions are interchangeable.

Table I: Interrupt Handling Response Time Errors

Interrupts	**E1: Identical Interrupt Load**				**E2: Random Interrupt Load**			
ID/Priority	Period	Intr. H	Intr. T	App. T	Period	Intr. H	Intr. T	App. T
1/*low*	$10ms$	0.86%	0.01%	0.01%	$15ms$	2.84%	0.27%	0.16%
2	$10ms$	0.20%	0.03%	0.02%	$12ms$	2.17%	4.97%	0.68%
3	$10ms$	0.39%	0.04%	0.03%	$11ms$	1.32%	5.36%	0.52%
4	$10ms$	0.84%	0.05%	0.03%	$8ms$	1.01%	9.12%	0.93%
5	$10ms$	1.03%	0.02%	0.01%	$7ms$	0.69%	7.01%	0.62%
6/*high*	$10ms$	0.23%	0.50%	0.10%	$5ms$	0.17%	4.01%	0.29%
Avg. Error		0.60%	0.11%	0.03%		1.37%	5.12%	0.53%

7.1. Interrupt Handling Evaluation

To verify the accuracy of the interrupt handling model, we compared the response time of interrupt handlers, interrupt tasks, and corresponding interrupt-driven application tasks to the reference simulation. Response times are defined as the delay between signaling an interrupt and completion of the corresponding handler or task (see Figure 14). Table I lists average absolute errors in response times for two different experiments, in which six interrupt-driven tasks are running on a single-core ARM processor. In experiment E1, all interrupts are triggered simultaneously with a fixed period of 10 ms. By contrast, in experiment E2, interrupts are generated at different rates in order to stress the experiment under random conditions. The interrupt controller was programed to assign different priorities to each interrupt signal. Experiments were run for a total simulated time of 5 sec.

Results show that for randomized interrupt behavior, average response time errors can be as high as 10% with maximum errors reaching 50% in some instances. Upon closer inspection, these errors stem from interference of the high-priority timer interrupt not being modeled in our setup. By synchronizing all interrupts with the fixed 10 ms rate of the Linux timer, such interferences are eliminated and both average and worst-case errors drop below 1%. Overall, assuming timer and other interrupt handlers to be generally short, results prove the accuracy of the interrupt controller and interrupt handling models.

7.2. Processor Evaluation

To evaluate overall simulation accuracy and performance, we generated 12 sets of random periodic tasks. Task periods were uniformly distributed over [1,100] ms and task weights randomly selected over [0.001, 0.1] for small (S), [0.1, 0.4] for large (L), and [0.001, 0.4] for medium/mixed (M) tasks. Task priorities are assigned inversely to their periods following a rate-monotonic scheduling scheme. Execution delays are modeled by a delay loop of No Operation (NOP) instructions. We ran each task set for 5 sec of simulated time. At a nominal rate of 1000 MIPS simulated by the reference ISS, this corresponds to 5 billion NOP instructions running on each core. Task sets were generated to cover various task weight ranges under different total system utilizations.

Experiments were executed on a simulated dual-core and a quad-core platform. Each task set was executed under four different interrupt conditions: periodic tasks plus a high, a medium, or a low priority interrupt-driven application task running on each core, or only periodic tasks running on the cores. Task weights for interrupt-driven application tasks were fixed at a value of 0.01, and their generation frequency/load was proportional to their priority, i.e. a higher load for the interrupt with a higher priority. For accuracy measurement, task response times were compared to the reference ISS.

Table II: Artificial Task Set Simulation Results

Set	**S1**	**S2**	**S3**	**S4**	**M1**	**M2**	**M3**	**M4**	**L1**	**L2**	**L3**	**L4**
Number of Tasks	13	18	29	58	8	8	8	16	7	5	6	12
Number of Cores	2	2	2	4	2	2	2	4	2	2	2	4
Avg. Task Weight	.05	.06	.06	.06	.14	.16	.19	.19	.19	.26	.28	.28
CPU Utilization	.65	1.03	1.7	3.4	1.1	1.3	1.5	3.0	1.3	1.3	1.7	3.4
Intr. High												
Error (periodic)	.33%	.26%	.31%	.33%	.11%	.05%	.02%	.04%	.02%	.02%	.04%	.13%
Error (intr-driven)	.37%	.23%	.21%	.23%	1.02%	.49%	.34%	.74%	.76%	.60%	.50%	.91%
Speed [GIPS]	24	40	60	53	46	54	63	53	47	46	70	62
Intr. Medium												
Error (periodic)	.15%	.11%	.13%	.13%	.11%	.05%	.01%	.01%	.06%	.02%	.04%	.04%
Error (intr-driven)	.13%	.06%	.16%	.19%	.12%	.12%	.10%	.10%	.13%	.19%	.10%	.11%
Speed [GIPS]	80	128	213	170	140	165	188	251	165	215	211	281
Intr. Low												
Error (periodic)	.15%	.11%	.13%	.13%	.07%	.05%	.01%	.01%	.02%	.02%	.04%	.05%
Error (intr-driven)	.23%	.17%	.25%	.27%	.13%	.11%	.13%	.13%	.08%	.07%	.03%	.05%
Speed [GIPS]	161	171	284	284	278	660	377	377	330	322	422	422
No. Intr.												
Error (periodic)	.15%	.11%	.13%	.13%	.07%	.05%	.01%	.01%	.03%	.02%	.04%	.04%
Speed [GIPS]	322	513	426	426	696	824	1,884	1,500	1,100	1,287	1,406	1,688

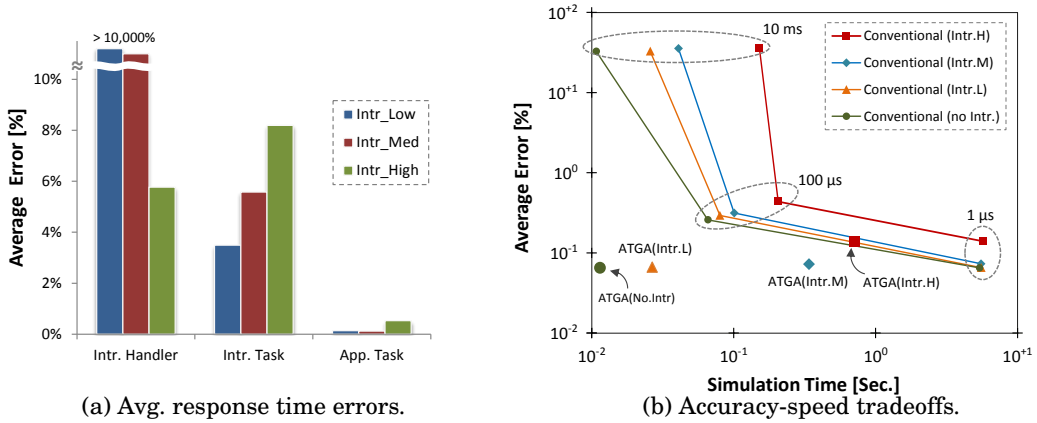
Table II lists the task set features and summarizes accuracy and speed results. Error was measured as the average percentage of absolute differences in individual task response times over all tasks and task iterations. To measure simulation performance, only the number of actually simulated instructions was considered. The number of simulated instructions was calculated based on the total simulated time and the nominal NOP instructions executed on the reference ISS. The idle time is eliminated by considering the CPU utilization. In other words,

$$Speed = \frac{Simulated\ time * 1000\,MIPS * CPU\ utilization}{Simulation\ time}.$$

Results show an average timing error of 0.16% and an average speed of 400 GIPS over all task sets and experiments. In all cases, error variance remains below $\pm$1.8%. Although we would expect to see zero errors using our ATGA approach, remaining errors are caused by non-modeled OS context-switch overheads, non-ideal behavior of a real Linux system, and errors in measured back-annotated delays. We can observe constant average errors for the periodic task sets in all experiments except when a high priority interrupt is running in the system. Reduced accuracy in these cases is caused by the error in estimated delays of interrupt and signal handlers, which have a larger effect when the interrupt load is high and tasks delays are small.

Further investigation of errors in the interrupt chain are shown in Figure 15 (a). High errors are measured for interrupt handlers in experiments with low and medium interrupt priorities. This is due to deliberate inaccuracies in the interrupt model. Since the OS model does not switch to fallback mode when the priority of the running task is higher than any interrupt-driven task, start times of interrupt handlers can be delayed until the next predicted scheduling point. This results in high response time errors of interrupt handlers. However, the total effect on response time of application-level tasks is negligible. Furthermore, larger errors are observed for high-priority interrupt and application tasks. Since high priority interrupts are generated at higher rates, non-modeled Linux back-ground tasks can introduce larger timing errors in such cases.

Finally, we compare our ATGA approach against a conventional host-compiled simulation at user-defined simulation granularity. Figure 15 (b) compares the average



(a) Avg. response time errors. (b) Accuracy-speed tradeoffs.

Fig. 15: Accuracy and speed analysis for artificial task sets.

simulation error and simulation time of ATGA and conventional models under different timing granularities. As can be seen, there is a fundamental accuracy and speed tradeoff in a conventional simulation, i.e. decreasing the timing granularity results in a higher accuracy but comes at a loss in simulation performance. Furthermore, there is a high variation in errors under large granularities. For example, with 10 ms simulation granularity, errors across task iterations vary between 0.1% and 1,200%. By contrast, our ATGA approach automatically and optimally provides fast simulation with the highest possible accuracy. The ATGA simulation provides the fastest possible speed when no interrupt is running in the system and the OS kernel runs only in predictive mode. By contrast, task sets with high-priority interrupt-driven tasks require the OS model to remain in fallback mode and thus simulate much slower.

To evaluate our models under more realistic multi-core conditions, we further applied our simulator to a set of compute-intensive multi-threaded applications from the ParMiBench suite [Iqbal et al. 2010]. ParMiBench applications are parallelized by data decomposition across threads that are synchronized via barrier channels. Task delays were back-annotated at the function level from measurements taken on the ISS. Pthread calls were mapped into corresponding OS model primitives, where a high-level inter-task channel was implemented to model POSIX-based barrier synchronization. Table III summarizes simulation accuracy and speed for a single, dual and quad core simulated platform. Results show that simulated application runtimes follow the reference simulation with an average error of 0.5%. Remaining errors are largely due to non-modeled execution delays in Pthread calls. Overall, when replacing artificial NOP instructions with real code, average simulation speed is degraded to 3,500 MIPS.

Table III: ParMiBench Accuracy and Speed Results

Application	**Simulated time (Error)**			**Simulation speed [MIPS]**		
	CPU=1	CPU=2	CPU=4	CPU=1	CPU=2	CPU=4
Bitcount (112500 iter.)	582ms (0.58%)	291ms (0.80%)	147ms (1.13%)	3,600	2,600	2,600
Basicmath (500K num.)	287ms (0.16%)	144ms (0.34%)	72ms (1.88%)	2,300	2,600	2,250
Susan-edge (2.8MB pic.)	9.134s (0.36%)	5.016s (0.64%)	2.928s (0.62%)	3,950	3,800	3,800
Susan-corner (2.8MB pic.)	1.80s (0.32%)	0.959s (1.48%)	0.532s (0.18%)	4,800	4,800	4,800
Susan-smooth (2.8MB pic.)	11.60s (0.11%)	5.843s (0.21%)	2.946s (0.73%)	2,700	2,500	2,600
SHA (16 input files)	348ms (0.15%)	218ms (0.19%)	156ms (0.27%)	2,800	3,200	3,200
Dijkstra (160 nodes)	45.58s (0.02%)	22.79s (0.01%)	11.39s (0.06%)	6,500	6,900	6,900
Patricia (5000 IP address)	917ms (0.73%)	459ms (0.43%)	229ms (0.42%)	2,200	2,100	2,100
Stringsearch (16MB in file)	59.11s (0.20%)	29.55s (0.35%)	14.77s (0.79%)	2,900	2,900	2,900

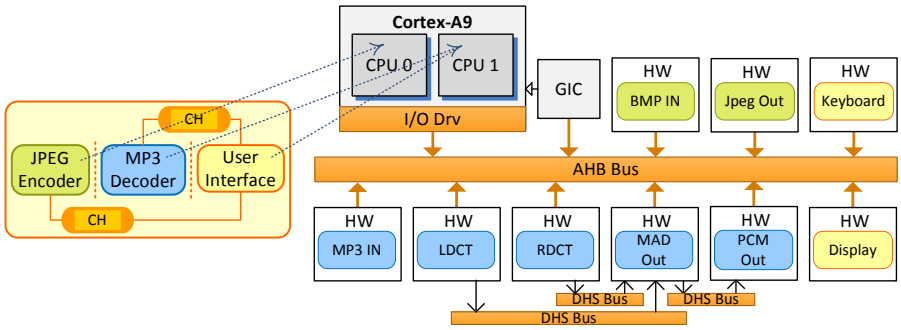


Fig. 16: Motion-JPEG example architecture.

7.3. System Evaluation

To finally demonstrate the benefits of our simulator for fast and accurate design space exploration, we applied our models to an industrial-strength Motion-JPEG (M-JPEG) example, running three concurrent MP3, JPEG and user interface tasks on a dual-core 650 MHz Cortex-A9 platform model. The overall architecture of the system is shown in Figure 16. The MP3 decoder uses hardware accelerators to perform audio decoding, and the JPEG encoder is completely implemented in software. Tasks communicate with external hardware and the rest of the system via an AHB bus and 12 interrupts. MP3 decodes 13 frames at a bitrate of 384 kbit/s, and JPEG encodes 10 frames of a 30 frames/s movie with 352×288 resolution.

For accuracy analysis, we compared the execution behavior of MP3 and JPEG tasks simulated on our host-compiled models to the reference OVP ISS. Task delays were back-annotated at the function level from measurements taken on the ISS. Moreover, average Linux context-switch overhead was measured and back-annotated into the OS model. We explored a wide range of architectures by applying different OS and processor configurations, including mapping of M-JPEG tasks and interrupts to different cores. Error was measured as the average percentage of absolute differences in individual frame delays over all frames. The simulation speed was calculated based on the number of application and Linux kernel instructions simulated by the reference ISS excluding Linux boot-up times. Instruction counts number between 800 and 1,040 million instructions depending on the system configuration. Note that the application-only instruction count was 160 million instructions, which means that a significant performance benefit comes from the OS abstraction approach. Finally, in order to achieve fast simulation, a TLM of the AHB bus at a granularity of user/application transactions [Schirner and Dömer 2009] and the lightweight model of the GIC were used.

Table IV shows the average error and the simulation speed for the explored architectures simulated by the ATGA model and the conventional model under three different granularities. In dual-core architectures with a *task-attached* interrupt model, application tasks are distributed among two cores and a task and its associated interrupts are mapped to the same core. By contrast, dual-core architectures with a *core-attached* interrupt model always handle and run all interrupts on core_1. Results show that with accurate interrupt modeling, the average error of MP3 and JPEG frame delays over all configurations is 0.71% at an average simulation speed of 1,400 MIPS. This translates into an average speed of 244 million application-only instructions per second. For configurations in which MP3 has a higher priority than JPEG, increasing the simulation granularity in a conventional approach degrades accuracy but increases simulation performance. By contrast, with the ATGA approach both fast and accurate results are achieved. This tradeoff is not observed in some of the other configurations. A further

Table IV: Motion-JPEG Example Simulation Results

Configuration	Average Frame Error				Simulation Speed [MIPS]			
	ATGA	Conventional			ATGA	Conventional		
		0.01μs	1μs	100μs		0.01μs	1μs	100μs
Single-Core								
1: C0: *Prty*(MP3>JPG>CTL)	0.53%	0.68%	0.66%	1.95%	1,000	50	1,170	1,500
2: C0: *FIFO*(MP3, JPG)>CTL	0.75%	0.62%	0.62%	0.62%	1,460	70	1,600	2,130
3: C0: *Prty*(JPG>MP3>CTL)	0.58%	0.65%	0.65%	0.65%	1,920	70	1,620	2,200
Task-attached interrupt model								
4: C0: CTL, C1: *Prty*(MP3>JPG)	0.48%	0.64%	0.60%	1.85%	1,000	50	1,200	1,400
5: C0: CTL, C1: *FIFO*(MP3, JPG)	0.67%	0.71%	0.71%	0.71%	1,740	70	1,500	2,200
6: C0: CTL, C1: *Prty*(JPG>MP3)	0.63%	0.64%	0.72%	0.72%	1,820	70	1,600	2,000
7: C0: *Prty*(MP3>CTL), C1: JPG	0.59%	0.72%	0.72%	0.72%	1,500	50	1,200	1,500
Core-attached interrupt model								
8: C0: *Prty*(MP3>JPG>CTL), C1: Intr	0.85%	0.93%	0.89%	3.75%	1,050	50	1,100	1,600
9: C0: *FIFO*(MP3, JPG)>CTL, C1: Intr	1.56%	1.92%	1.92%	33.1%	1,500	50	1,460	1,800
10: C0: *Prty*(JPG>MP3>CTL), C1: Intr	0.93%	0.35%	0.47%	35.9%	1,570	50	1,400	1,800
11: C0: *Prty*(MP3>CTL), C1: JPG,Intr	0.28%	0.46%	0.46%	11.9%	840	50	1,250	1,500
Average	0.71%	0.76%	0.77%	8.35%	1,400	57	1,373	1,785

investigation shows that in these configurations, the OS rarely switches to fallback mode and the running task is never preempted by the next release time of a higher priority one. As such, even with the conventional model, accurate results are always achieved, i.e. the accuracy is independent of the simulation granularity. However, this is hard to predict. By contrast, the ATGA approach always provides guaranteed accurate yet fast results. Note that in some cases, the conventional simulation accuracy is higher, because errors caused by the simulation granularity compensate inaccuracies for in back-annotated delays.

Figure 17 summarizes the average frame delays and average frame delay errors plus max. error bars of MP3 and JPEG tasks. In order to demonstrate the importance of accurate interrupt modeling, frame delays and errors are reported both with and without modeling the interrupt handling chain (in the latter case, by bypassing the complete interrupt handling model). As can be seen, the best MP3 performance is achieved when a higher priority is assigned to MP3, or MP3 and JPEG are running on separate cores. In other configurations, average MP3 frame delay is close to its deadline boundary (i.e. 26.1 ms). Minimized JPEG delay is obtained from configurations with FIFO scheduling, when JPEG has higher priority or when it runs on a separate core. In FIFO scheduling, MP3 behaves like a low-priority task. The reason is that MP3 is often blocked waiting on hardware, while JPEG completely runs in software. As such, whenever the MP3 task is blocked, the JPEG gets the highest priority and MP3 can only resume its execution after JPEG finishes encoding of the current frame. All combined, our explorations confirm that shortest-job-first or rate-monotonic scheduling guarantee that MP3 and JPEG meet their performance requirements.

Overall, optimized MP3 and JPEG performance is achieved when tasks run on separate cores. The performance degradation of JPEG in the last configuration compared to the same *task-attached* execution is caused by extra time periods that JPEG is preempted by MP3 interrupts. Finally, by mapping all interrupts to a separate core (core_1), we only see slight performance benefits in MP3 and JPEG delays. Since interrupt handlers are not running in parallel with their application tasks, putting them on a separate core does not reduce interrupt delays within a task but can minimize the influence of one task's interrupts on the other. This effect is more pronounced for

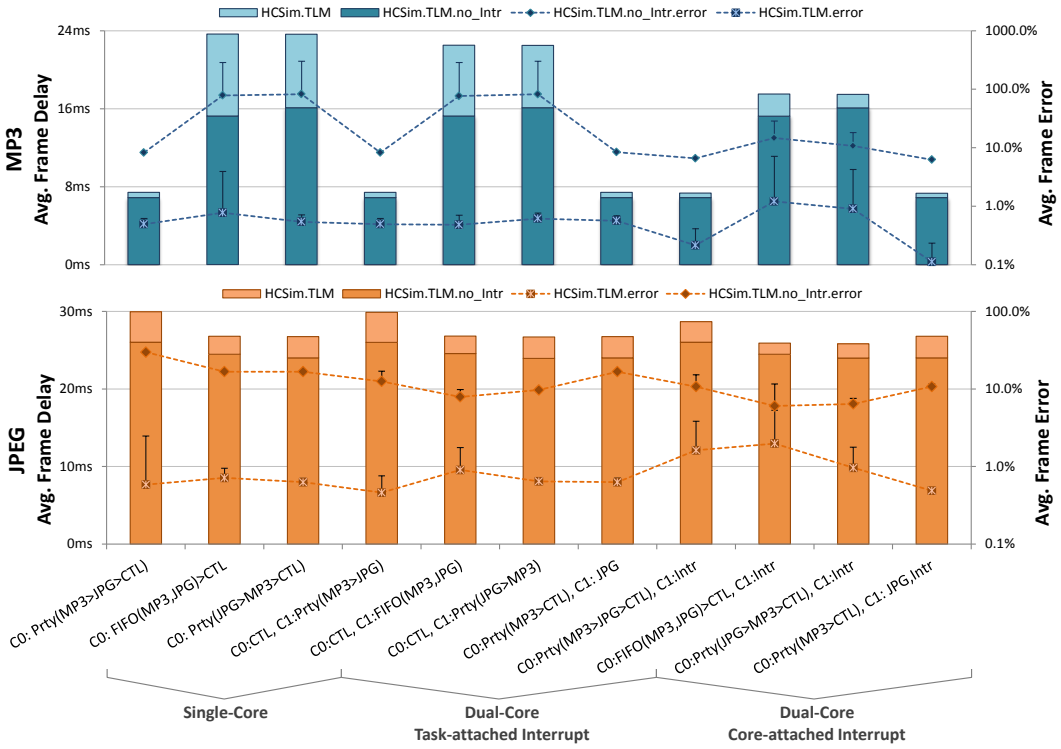


Fig. 17: Design space exploration results.

the JPEG task, but interrupt handlers have small execution delays and do not provide large speedup benefit.

In the end, results also confirm that interrupts can have a significant influence on overall system performance. When bypassing the interrupt handling model, some configurations exhibit a very large error that is caused by a wrong execution order of MP3 and JPEG tasks. By contrast, when including the model of the interrupt chain, average errors remain within 2%. Overall, the high accuracy and fast simulation of our host-compiled simulator including accurate OS, processor and interrupt models provides an efficient platform for early design space exploration.

8. SUMMARY AND CONCLUSION

In this paper, we presented a host-compiled multi-core system simulator designed for early real-time performance evaluation. At its core, the simulator consists of a configurable, abstract OS model, which emulates multi-core task scheduling. Our OS model transparently incorporates an automatic timing granularity adjustment (ATGA) approach in which the model automatically and optimally accumulates and adjusts back-annotated delays to provide an error-free task preemption model. Furthermore, the OS model is embedded in a high-level multi-core processor model that replicates a generic multi-core interrupt handling chain and supports standard TLM interfaces for integration into co-simulation backplanes to provide a fast and accurate full-system HW/SW co-simulation platform.

Experimental results demonstrate the efficiency of our simulator both on a suite of artificial task sets and an industrial-strength design example. Results show that compared to a reference ISS, simulations on the order of 1000 MIPS at less than 3% error

can be achieved. Although we would expect to see zero errors using our ATGA approach [Razaghi and Gerstlauer 2012b], remaining errors are caused by other orthogonal modeling issues, such as errors in back-annotation of task delays or the execution of back-ground tasks in real platforms, which are out of scope of this paper. Overall, experiments demonstrate the benefits of our configurable models for fast and accurate early design space exploration and software development.

In future work, we plan to extend our models to support a comprehensive set of multi-core scheduling policies and a larger range of target platforms. We also intend to provide a modeling interface in which designers are able to add custom scheduling algorithms to the OS kernel. Moreover, we plan to investigate integration of inter-core synchronization and multi-core cache hierarchy models [Razaghi and Gerstlauer 2013] into a simulator with temporal decoupling and automatic timing granularity adjustment in order to consider effects of the shared memory hierarchy on system performance.

REFERENCES

ARM Co. ARM Generic Interrupt Controller Architecture Specification. http://infocenter.arm.com.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. SimpleScalar: an infrastructure for computer system modeling. *Computer 35,* 2, 59 –67.

BELLARD, F. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*.

BENINI, L., BERTOZZI, D., BOGLIOLO, A., MENICHELLI, F., AND OLIVIERI, M. 2005. MPARM: Exploring the Multi-Processor SoC Design Space with SystemC. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology 41,* 2, 169–182.

BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News 39,* 2, 1–7.

BLAKE, G., DRESLINSKI, R. G., AND MUDGE, T. 2009. A survey of multicore processors. *IEEE Signal Processing Magazine 26*, 26–37.

BOUCHHIMA, A., GERIN, P., AND PETROT, F. 2009. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*.

CAI, L. AND GAJSKI, D. 2003. Transaction level modeling: an overview. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

CENG, J., SHENG, W., CASTRILLON, J., STULOVA, A., LEUPERS, R., ASCHEID, G., AND MEYR, H. 2009. A high-level virtual platform for early mpsoc software development. In *Proceedings of International Conference on Hardware/software codesign and system synthesis (CODES+ISSS)*.

GAJSKI, D. D., ZHU, J., DOMER, R., GERSTLAUER, A., AND ZHAO, S. 2000. *SpecC: Specification Language and Methodology*. Springer.

GERIN, P., SHEN, H., CHUREAU, A., BOUCHHIMA, A., AND JERRAYA, A. 2007. Flexible and executable hardware/software interface modeling for multiprocessor SoC design using SystemC. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*.

GERSTLAUER, A. 2010. Host-compiled simulation of multi-core platforms. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP)*.

GERSTLAUER, A., YU, H., AND GAJSKI, D. 2003. RTOS modeling for system level design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*.

GHENASSIA, F. 2005. *Transaction level modeling with systemc: Tlm concepts and applications for embedded systems*. Springer.

HCSim 2014. `http://www.ece.utexas.edu/~gerstl/releases`.

HWANG, Y., ABDI, S., AND GAJSKI, D. 2008. Cycle-approximate retargetable performance estimation at the transaction level. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*.

Imperas Ltd. Imperas Software Limited. http://www.imperas.com.

IQBAL, S., LIANG, Y., AND GRAHN, H. 2010. ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. *Computer Architecture Letters 9,* 2, 45–48.

KRAUSE, M., ENGLERT, D., BRINGMANN, O., AND ROSENSTIEL, W. 2008. Combination of instruction set simulation and abstract rtos model execution for fast and accurate target software evaluation. In *Proceedings of International Conference on Hardware/Software codesign and system synthesis (CODES+ISSS)*.

LAUZAC, S., MELHEM, R. G., AND MOSS, D. 1998. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Proceedings of the Euromicro Conference on Real-Time Systems*.

LIN, K.-L., LO, C.-K., AND TSAY, R.-S. 2010. Source-level timing annotation for fast and accurate tlm computation model generation. In *Proceedings of the Asia and South Pacific Design Automation Conference*.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Computer 35,* 2, 50–58.

MEYEROWITZ, T., SANGIOVANNI-VINCENTELLI, A., SAUERMANN, M., AND LANGEN, D. 2008. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*.

MIRAMOND, B., HUCK, E., VERDIER, F., BENKHELIFA, M. E. A., GRANADO, B., AICHOUCH, M., PRVOTET, J.-C., CHILLET, D., PILLEMENT, S., LEFEBVRE, T., AND OLIVA, Y. 2009. OveRSoC : a framework for the exploration of RTOS for RSoC platforms. *Intr. Journal on Reconfigurable Computing 2009*.

OVP Co. Open Virtual Platforms. http://www.ovpworld.org.

POSADAS, H., DAMEZ, J., VILLAR, E., BLASCO, F., AND ESCUDER, F. 2005. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *Design Automation for Embedded Systems*.

RAZAGHI, P. AND GERSTLAUER, A. 2011. Host-compiled multicore RTOS simulator for embedded real-time software development. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*.

RAZAGHI, P. AND GERSTLAUER, A. 2012a. Automatic timing granularity adjustment for host-compiled software simulation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*.

RAZAGHI, P. AND GERSTLAUER, A. 2012b. Predictive os modeling for host-compiled simulation of periodic real-time task sets. *Embedded Systems Letters, IEEE 4,* 1.

RAZAGHI, P. AND GERSTLAUER, A. 2013. Multi-core cache hierarchy modeling for host-compiled performance simulation. In *Proceedings of the Electronic System Level Synthesis Conference (ESLSyn)*.

RENAU, J., FRAGUELA, B., TUCK, J., LIU, W., PRVULOVIC, M., CEZE, L., SARANGI, S., SACK, P., STRAUSS, K., AND MONTESINOS, P. 2005. SESC simulator. http://sesc.sourceforge.net.

SALIMI KHALIGH, R. AND RADETZKI, M. 2010. Modeling constructs and kernel for parallel simulation of accuracy adaptive TLMs. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*.

SCHIRNER, G. AND DOMER, R. 2008. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *Proceedings of the Design, Automation and Test in Europe (DATE)*.

SCHIRNER, G. AND DÖMER, R. 2009. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *ACM Trans. Embed. Comput. Syst. 8,* 1, 4:1–4:29.

SCHIRNER, G., GERSTLAUER, A., AND DÖMER, R. 2010. Fast and accurate processor models for efficient mpsoc design. *ACM Trans. Des. Autom. Electron. Syst. 15,* 2, 10:1–10:26.

SCHNERR, J., BRINGMANN, O., AND ROSENSTIEL, W. 2005. Cycle accurate binary translation for simulation acceleration in rapid prototyping of socs. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*.

SCHNERR, J., BRINGMANN, O., VIEHL, A., AND ROSENSTIEL, W. 2008. High-performance timing simulation of embedded software. In *Proceedings of the Design Automation Conference (DAC)*.

STATTELMANN, S., BRINGMANN, O., AND ROSENSTIEL, W. 2011. Fast and accurate resource conflict simulation for performance analysis of multi-core systems. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*.

WANG, Z. AND HENKEL, J. 2012. Accurate source-level simulation of embedded software with respect to compiler optimizations. In *Proceedings of the Design, Automation Test in Europe (DATE) Conference*.