Automatic Timing Granularity Adjustment for Host-Compiled Software Simulation

Parisa Razaghi, Andreas Gerstlauer
Electrical and Computer Engineering, The University of Texas at Austin
{parisa.r, gerstl}@mail.utexas.edu

Abstract—**Host-compiled simulation has been widely adopted as a practical approach for fast and high-level evaluation of complex software-intensive systems at early stages of the design process. In such approaches, higher speed is achieved by coarse-grained simulation of the system, which also leads to a loss in timing accuracy. To eliminate the inherent speed and accuracy tradeoff, we present an adjustive software simulator, which automatically controls the timing model of the simulation platform to provide both fast and accurate results. At its core, we propose a novel RTOS model that permanently monitors the state of the system and optimally and automatically adjusts back-annotated timing granularities to provide an error-free task scheduling. We evaluated our approach on an industrial-strength example, and results show that the accuracy of a fine-grain simulation can be achieved while maintaining a speed of close to 900MIPS.**

Index Terms—**Real-time systems; host-compiled simulation; RTOS modeling;**

I. INTRODUCTION

In recent years, the complexity of embedded systems has increased dramatically. The challenge is to design such complex systems with constrains on design goals, especially real-time performance, at reduced development time and cost. Since software provides a high degree of flexibility and easy code reuse, the trend over the last years has been to shift more and more functionality into software. Hence, effective evaluation of such complex, software-intensive systems in early stages of the design process is essential.

Many studies have focused on methods to provide fast and accurate simulation by abstracting the software execution environment. For example, virtual platforms provide a high-level functional prototype of a target architecture, which allows designers to debug and simulate their software along with the rest of the system before the actual hardware is available. Virtual platforms execute the binary code of the software on the target architecture prototype at close to real-time speeds. However, such approaches only provide fast functional simulation, with limited or no timing information.

Recently, host-compiled approaches have been developed to provide fast simulation coupled with accurate timing execution. In such approaches, the software is natively compiled and executed on a host machine while an abstract model of the target architecture manages the execution order of user application tasks. For timing accuracy, the application code is instrumented with back-annotated execution delays. In host-compiled approaches, higher speed is achieved by coarse-grained simulation of the system, which inherently comes at a loss in timing accuracy. In other words, there is a fundamental tradeoff between simulation speed and timing accuracy.

In this paper, we aim to eliminate this tradeoff in host-compiled software simulation. We present a platform modeling approach for fully accurate yet fast simulation of real-time applications. At its core, this is enabled by a novel RTOS model, which is capable of permanently monitoring system state to automatically adjust simulated timing granularities and eliminate task scheduling errors while maintaining fast simulation speed. In such an approach, designers need not be concerned with manually selecting a proper granularity for optimizing the speed and accuracy tradeoff. Instead, the platform simulator automatically, continuously and dynamically adjusts to changing system conditions in order to achieve an optimal simulation.

The remainder of this paper is organized as follows: in the following subsections, we review related work and present an overview of our simulator. Then, we discuss the details of our approach in Section II. Results of our experiments are summarized in Section III. Finally, we conclude this paper with a summary and outlook on future work in Section IV.

A. Related Work

Recently, so-called host-compiled or source-level simulation approaches have received widespread attention as a solution for rapid evaluation of software at early design stages. Such approaches provide high performance by abstracting the simulation platform [1], [2], [3], [4]. The high-level source code of applications is back-annotated with timing estimates, which are typically obtained by compiling to an intermediate representation [5], [6]. Application execution is managed by an abstract model of the software execution environment, which is usually developed on top of standard system-level design languages (SLDLs) (e.g. SystemC [7] or SpecC [8]).

Some of the earliest host-compiled approaches were centered around models of the OS itself [9], [10], [11]. Later, these approaches were extended into complete processor models that include timing-accurate descriptions of interrupt chains and TLM-based bus interfaces [12], [13]. Such processor models have been shown to simulate at speeds beyond 500 MIPS with more than 95% timing accuracy.

Several researchers have focused on improving the accuracy of high-level simulators while maintaining similar performance. Krause *et al.* [14] present combined ISS and abstract RTOS model co-simulation. This approach replaces

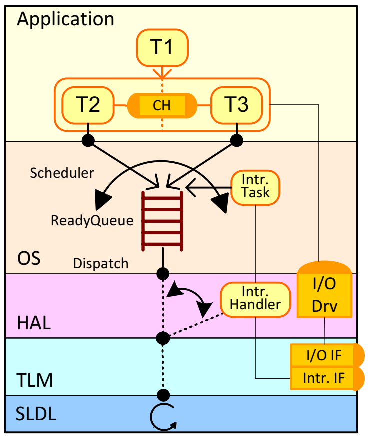


Fig. 1. Host-Compiled software simulator.

an actual RTOS binary code with an abstract model running outside the ISS and performs cycle-accurate thread switches. Khaligh *et al.* [15] present an adaptive TLM simulation kernel, which changes the level of accuracy during simulation to the level expected by designers. Schirner *et al.* [16] introduce a granularity-independent approach for accurate simulation of interrupts on host-compiled processor models by applying optimistic prediction and correction. In all cases, however, fundamental static speed and accuracy tradeoffs remain. By contrast, we adjust granularities automatically, optimally and dynamically to achieve fast and accurate simulation.

B. Host-Compiled Software Simulator

We have developed a high-level, host-compiled software simulator, details of which can be found in [17]. Figure 1 shows the structure of our simulator, which is designed in a layered-based fashion. A standard SLDL kernel provides a basic platform for running simulations on a host machine. In combination with the underlying SLDL, a TLM layer interfaces the software simulator with standard TLM backplanes that provide a fast system-wide co-simulation platform. A hardware abstraction layer (HAL) includes necessary I/O drivers and implements an abstract interrupt handling mechanism. When an interrupt is captured by the TLM layer, the HAL suspends application execution and lets the interrupt handler trigger the registered interrupt task in the OS. On top of the HAL, an OS layer replicates a typical OS architecture to manage the execution order of a multi-tasking application. The OS model thereby schedules, queues, dispatches and executes application and interrupt tasks according to a chosen scheduling policy. At the highest level, the application layer consists of concurrent and sequential high-level SLDL processes, which communicate with each other using abstract SLDL channels. The user application is integrated into the simulator and accesses services of the OS model via a canonical OS API.

At the core of the simulation engine is the OS model, which dynamically schedules concurrent application tasks to emulate their sequential execution in software. The structure of our OS kernel is shown in Figure 2. The key component of the kernel is a task scheduler, which is invoked by the OS API methods whenever a context switch is possible or

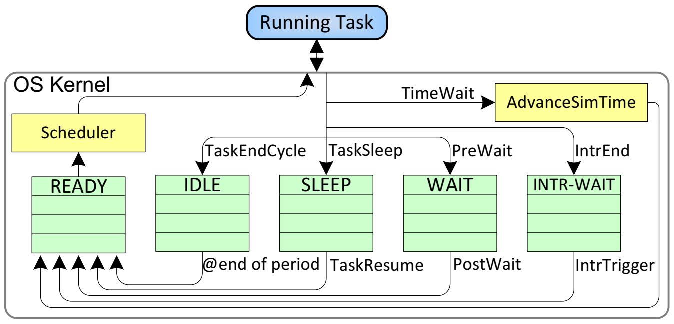


Fig. 2. Abstract RTOS model.

required. It decides on the next task to execute and preempts the currently running task if needed. In the OS model, each task can be in five states, and tasks move to different states by calling API methods of the OS kernel. In order to control 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()` method at the end of their iteration. 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. Tasks waiting for an event are suspended and transfered to a *Wait* queue upon calling a `PreWait()` method. Respectively, a blocked task will be placed back in the *Ready* queue when a `PostWait()` method is called to release it. To distinguish tasks that are waiting for an external event, an *IntrWait* queue holds interrupt tasks until the interrupt handler in the HAL calls the `IntrTrigger()` method to move them to the *Ready* queue. Finally, a *Sleep* queue holds tasks that have been suspended until they are resumed again.

In addition to basic OS services, the OS kernel simulates task execution delays using underlying SLDL primitives whenever the running task calls a `TimeWait()` method. Basic execution delays of the task code are back-annotated from estimations or measurements once at compile time. In traditional models, the granularity of delays is defined by the application code. 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 such, errors in the preemption model are a direct function of the back-annotated application-level timing model. Large granularities result in fast simulation, but may lead to preemption points being shifted by a large delay. On the other hand, accurate simulations require a fine granularity at slow simulation speeds. By contrast, we propose an approach that automatically adjusts timing granularities to the level needed. The OS model has complete knowledge of the system state at any given time. As such, we can develop a kernel that utilizes this knowledge to automatically control simulation timing such that an error-free scheduling mechanism is provided at the fastest possible speed.

II. Timing Granularity Adjustment

In the following, we describe our automatic timing granularity adjustment (ATGA) approach to eliminate task preemption errors in abstract RTOS models. In this approach, the OS kernel 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. As a result, 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 breaks delays into a number of smaller steps as needed, in order to automatically provide the best timing granularity for fully accurate results.

A. ATGA RTOS Model

Our RTOS model features an adjustable timing mechanism as shown in Figure 3. In this approach, the OS kernel switches between predictive and fall-back modes to call the scheduler at the right preemption points. In the following, we demonstrate the details of each mode and the mechanism that the OS model uses to automatically control the underlying timing model.

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 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 *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 the back-annotated granularity is larger than the predicted interval, the OS kernel adjusts the delay to invoke the scheduler at the predicted time. Figure 3 (a) shows the algorithm for predicting the next preemption time. 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 task.

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 kernel falls back to a user-defined default timing granularity. In this *fallback mode*, the OS divides back-annotated delays into very fine granules until all events are captured and no task remains in the *Wait* queues (Figure 3 (b)). Figure 3 (c) shows the method that advances the simulation time using the underlying SLDL `wait()` statement to advance simulation time. In fallback mode, only one user-defined fine-grain default granule is simulated.

For ultimate control of automatic timing granularity adjustment, the `TimeWait()` method is overloaded to monitor the OS timing mode and simulate back-annotated delays divided into proper intervals. Figure 3 (d) shows the pseudo code of the `TimeWait()` method in our ATGA RTOS model. It first computes the adjusted time delay by calling the method to predict the next preemption point (line 3). Then, it defines the OS timing mode by checking the fallback mode condition (line 7). After advancing the simulation time by the adjusted delay and updating the remaining delay with the consumed time value (line 8 and 9), the OS scheduler is called to perform a context switch and block the current task until it is scheduled again (line 10). This loop continues until the user-defined delay is consumed.

```
Function PredictNextPreemptionTime (task runningTask):
1   for tasks in Idle Queue do
2     if idleTask::Priority ≥ runningTask::Priority then
3       predictedTime := idleTask::nextPeriod - currentTime
4       return predictedTime
5     endif
6   endfor
```

(a) Preemption point prediction.

```
Function FallbackMode (task runningTask):
1   return !Empty(Wait) or !Empty(IntrWait)
```

(b) Fallback mode check.

```
Function AdvanceSimTime (long long nsec, bool fallBack):
1   if fallBack and nsec > defaultDelayGranularity then
2     consumedDelay := defaultDelayGranularity
3   else
4     consumedDelay := nsec
5   endif
6   SLDL::wait(consumedDelay)
7   return consumedDelay
```

(c) Simulation time advancement.

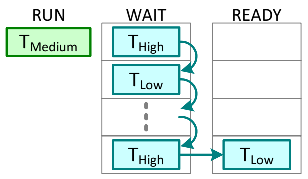
```
Function TimeWait (long long nsec, task runningTask):
1   remainedDelay := nsec
2   while remainedDelay > ø do
3     adjustedDelay := PredictNextPreemptionTime(runningTask)
4     if adjustedDelay > remainedDelay then
5       adjustedDelay := remainedDelay
6     endif
7     FB := FallbackMode(runningTask)
8     consumedDelay := AdvanceSimTime(adjustedDelay, FB)
9     remainedDelay := remainedDelay - consumedDelay
10    Scheduler()
11  endwhile
```

(d) Timing model.

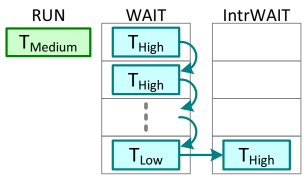
Fig. 3. Timing granularity adjustment.

B. Enhanced Fall-Back Mode

The adjustive RTOS model lets designers select coarse-grain back-annotated delays while achieving fast and still accurate results. However, when the OS switches to fallback mode, the performance of the simulation decreases dramatically. As such, we have developed techniques that exercise finer control over when to invoke fallbacks. Figure 4 illustrates two inter-task communication examples in which the OS kernel does not need to switch to fallback mode even though some



(a) Blocked by a *Ready* task.



(b) Blocked by an interrupt task.

Fig. 4. Inter-task communication examples.

tasks are waiting for an event. In Figure 4 (a), a inter-task communication chain is shown in which a set of tasks are blocked waiting for other tasks in the chain. The task at the end of the chain has a higher priority than the running task, but is blocked by a lower priority *Ready* task. Since the *Ready* task cannot be scheduled while the current task is running, remaining in predictive mode will not change the execution order of tasks. Similarly, in Figure 4 (b) the same chain is shown where the task at the end of the chain is a low priority task that is blocked on an external event, i.e. an interrupt. Even when the interrupt occurs and assuming small interrupt task delays, unblocking the lower-priority task can never preempt the running task, i.e. the fallback mode can also be ignored in this situation.

As illustrated by these examples, only the task at the end of the *Wait* chain needs to be examined to determine the fallback condition. Figure 5 lists all possible situations and required fallback conditions. Generally, the OS switches to the fallback mode due to unpredicted events. Therefore, the OS moves to fallback when a task with a higher priority is in the *Wait* queue and is blocked by an unknown task or a task in the *IntrWait* queue. In all other situations, granularity of the simulation will not affect the execution order of the application tasks. Lower-priority tasks in the *Wait* queue or tasks waiting for another task in the *Wait* queue can never affect execution of the current task. Likewise, the case of a lower-priority ready task has already been discussed, and a higher-priority task in the *Ready* queue should never exist. Situations in which a high-priority task is blocked on a periodic task in the *Idle* queue can be handled by switching to predictive mode and simulating the system at the predicted granule level. Similarly, 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 method, right at the point when it is called by the running task. Lastly, assuming a low execution delay of interrupt tasks, we can postpone any such task if it van only trigger a low priority task in the *Wait* queue. With a minor interrupt timing error, we can ignore fallback condition. All the aforementioned conditions are checked in the enhanced `FallBackMode()` function, which is shown in Figure 6. The OS only turns to fallback mode if a higher priority task is blocked by an *Unknown* task (line 4) or is waiting for an external event (line 7 and 8).

C. Accumulative Timing Mode

Using the new timing model, accurate simulation results with error-free task scheduling are achieved. This model optimally divides user-defined timing granularity to provide accurate task preemption. However, simulation speed still is limited by back-annotated granularities, which usually depend on various other factors, such as application code modularity. We introduce an accumulative timing approach to achieve highest possible speed even with fine-grained back-annotated delays, while maintaining overall accuracy. In this approach, temporal accumulation and decoupling is integrated into and controlled by the OS kernel itself.

In accumulative mode, the OS kernel lets the running task execute its code and accumulate back-annotated delays without calling the scheduler and advancing simulation time. Each task has a local counter to keep track of the amount of the delay that needs to be simulated. As long as this delay is less than the next predicted preemption point, the task continues to accumulate back-annotated delays. It only consumes delays whenever a task preemption point needs to be reached.

Figure 7 shows the pseudo code of the `TimeWait()` method in accumulative mode. This method is called from the application code with a user-defined timing granularity. At first, the OS increments the internal delay counter associated with the running task (line 1). Then, it predicts the next possible preemption point and determines the OS timing mode

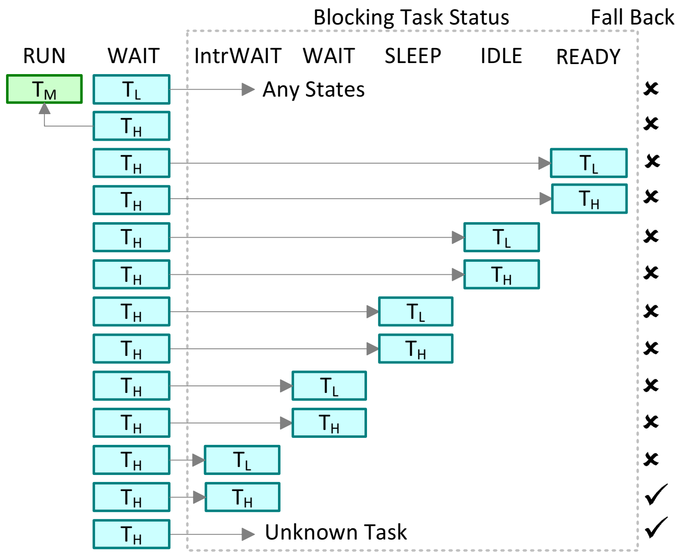


Fig. 5. Fallback mode conditions.

```
Function FallbackMode (task runningTask):
1  for all tasks in WAIT Queue do
2    if waitingTask::Priority ≥ runningTask::Priority then
3      blockingTask := waitingTask::blockingTaskID
4      if blockingTask == $Unknown$ then
5        return $true$
6      endif
7      if (blockingTask::Priority ≥ runningTask::Priority)
8                and (blockingTask::State == IntrWait) then
9        return $true$
10     endif
11   endif
12 endfor
13 return $false$
```

Fig. 6. Enhanced fallback mode checking.

```
Function **TimeWaitAcc** (long long nsec, task runningTask):
1  runningTask::accDelay + = nsec
2  predictedDelay := PredictNextPreemptionTime(runningTask)
3  FB := FallbackMode(runningTask)
4  while runningTask::accDelay > predictedDelay or FB do
5     consumedDelay := AdvanceSimTime(predictedDelay, FB)
6     runningTask::accDelay − = consumedDelay
7     Scheduler()
8     predictedDelay := PredictNextPreemptionTime(runningTask)
9     FB := FallbackMode(runningTask)
10 endwhile
```

Fig. 7. Accumulative timing mode.

```
Function AdvanceSimTime (long long nsec, bool fallBack):
1  consumedDelay := nsec
2  if fallBack then
3     startTime := currentTime
4     SLDL::wait(consumedDelay, OS::schedulerEvent)
5     consumedDelay := currentTime - startTime
6  else
7     SLDL::wait(consumedDelay)
8  endif
9  return consumedDelay
```

Fig. 8. Event-driven time model.

(lines 2 and 3). If the accumulated delay is larger than the predicted time or the OS is in fallback mode, simulation time is advanced until the predicted time is reached, calling the scheduler for possible task preemption (lines 4 to 9). In addition to preemption points, the OS kernel needs to consume accumulated delays at any task synchronization point. This is achieve by calling the original, non-accumulative `TimeWait()` method whenever a task accesses the bus, notifies an internal event, or switches to another state by calling `TaskEndCycle()` or `TaskSleep()` methods.

Timing accumulation and adjustment is only effective in predictive mode. Designers still need to decide on a timing granularity for fallback mode, which can affect speed and accuracy tradeoffs. Similarly to accumulation during predictive execution, we integrate an event-driven timing method into the fallback mode. In this setup, accumulated delays are executed even under fallback conditions, but an OS-internal event is introduced to be able to asynchronously interrupt long time consumption periods. Figure 8 depicts the event-driven time advancement method in the OS model. It uses underlying SLDL mechanisms to realize an interruptible time wait statement. Due to the high simulation overhead of such SLDL primitives, they are only utilized when in fallback mode. An OS-internal *schedulerEvent* is defined and triggered by the interrupt handler in the HAL whenever an interrupt occurs. This in turn will abort the `wait()` statement in the SLDL kernel, at which point control is returned to the OS model. The OS model computes the remaining unconsumed time and returns control to the scheduler, which then switches context away from the running to the high-priority interrupt task.

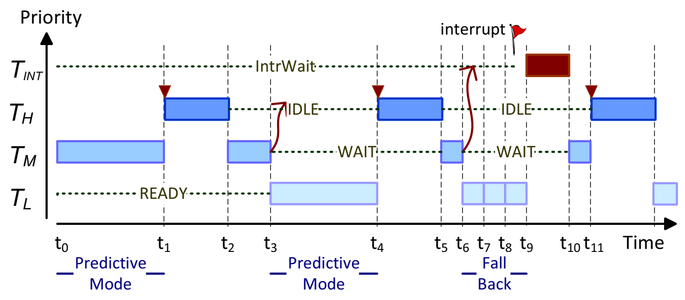


Fig. 9. Task execution example.

D. Model Execution

Figure 9 shows an example to illustrate the simulation of a system using our ATGA-RTOS model. The system contains four tasks. A high-priority interrupt task T_{INT} is located in the *IntrWait* queue and will be moved to the *Ready* queue whenever an interrupt is detected. Task T_H is a periodic task that has the highest priority among the application tasks. Tasks T_M and T_L have medium and low priorities respectively. At t_0, T_M starts execution of its functional code followed by a call to the `TimeWait()` method. Since T_H is in *Idle* state, the OS moves to the predictive mode, advances the simulation time to t_1, and calls the scheduler to start T_H. At time t_2, task T_H finishes its execution and task T_M resumes. At t_3, T_M moves to the *Wait* state and T_L gets a chance to run. At this point, T_M is blocked by T_H, which itself is in the *Idle* state. As such, the OS stays in the predictive mode and sets the scheduler to be called at t_4 when T_H is released again. T_H finishes its next iteration and T_M is resumed at time t_5. At t_6, T_M once more goes to the *Wait* state, blocking to receive an interrupt. Since T_H is in its *Idle* state, task T_L starts to execute. During this time, T_M is blocked by an interrupt task and the OS falls back to a very fine timing granularity. After receiving the interrupt (at time t_9), the OS switches to the interrupt task and exits the fallback mode.

III. Experimental Results

To demonstrate the benefits of the ATGA approach, we applied our RTOS models to an industrial-strength, ARM7-based cellphone example running concurrent MP3 decoding, JPEG encoding and control tasks. The MP3 decoder runs as a periodic task with the highest priority in the system. It uses a hardware accelerator to perform real-time audio decoding. The JPEG encoder runs as an interrupt-driven task with medium priority. The control task performs user-interface actions and runs at the lowest priority. Tasks communicate with external hardware and the rest of the system via an AHB bus and 14 interrupts. In this setup, the ATGA RTOS model can utilize predictive mode whenever the JPEG or control task are running and the MP3 is idle. On the other hand, fallback mode is triggered whenever MP3 or JPEG tasks are waiting for an external hardware interrupt while a lower-priority task is running. The application model and our simulator were developed in SpecC, but we are in the process of transferring results to other SLDLs, such as SystemC.

TABLE I
ACCURACY AND SPEED MEASUREMENTS FOR CELLPHONE EXAMPLE.

	RTOS 1μs	RTOS 10μs	RTOS 100μs	RTOS 1000μs	ATGA Fallback 1μs	ATGA/Acc Fallback 1μs	ATGA Event-driven	ATGA/Acc Event-driven	ISS
Avg. Err. (MP3)	0.73%	0.79%	1.40%	9.65%	0.73%	0.74%	0.73%	0.74%	0%
Avg. Err. (JPEG)	7.33%	7.33%	7.33%	7.35%	7.33%	7.32%	7.33%	7.32%	0%
Avg. Err. (MP3+JPEG)	4.18%	4.20%	4.49%	8.45%	4.18%	4.18%	4.18%	4.18%	0%
Simulation Speed	340MIPS	790MIPS	930MIPS	1080MIPS	554MIPS	684MIPS	621MIPS	892MIPS	0.13MIPS
Simulation Time	0.61s	0.26s	0.22s	0.19s	0.37s	0.30s	0.33s	0.23s	1580s

For accuracy analysis, we compared the execution of our host-compiled simulator using the proposed RTOS models to a cycle-accurate ISS [18]. Task delays were back-annotated from measurements obtained from the ISS. Our testbench performs encoding of 55 MP3 frames and JPEG decoding of a 680×480 picture divided into 60 stripes. This translates into a total of 200 million simulated instructions. Model error was measured as the average absolute difference in individual frame and stripe delays over all iterations.

Table I compares the accuracy and performance of our proposed RTOS models with that of a conventional one at four different back-annotated granularities. ATGA and ATGA plus accumulation models are simulated with both 1μs and event-driven (ED) fallback mode. We can observe that, regardless of the granularity of the back-annotated delays, the highest possible accuracy is achieved using our ATGA approach. This accuracy is equivalent to a conventional model at 1μs, which looses accuracy at coarser granularities. Although one would expect close to 100% accuracy in ATGA models, remaining errors are due to back-annotation inaccuracies and missing of model of OS effects like timer interrupts and task context-switch overhead. As a long-running low-priority task, the JPEG encoder is adversely affected by such basic errors. On the other hand, unlike high-priority tasks such as the MP3, it is not subject to preemption errors. As such, its errors are independent of the timing granularity.

Our measurements show that the highest speed of 890 MIPS is achieved using the accumulative ATGA/Acc with an event-driven fallback mode. This model is 2.6x faster than but as accurate as the conventional one at 1μs granularity. A conventional model achieves this speed with significantly reduced accuracy at a granularity of 100μs. Results clearly show the benefits of our ATGA approach. Furthermore, both timing accumulation and event-driven fallback help to increase simulation speed without adversely affecting accuracy.

Figure 10 plots the average error and accuracy for different RTOS configurations. As can be seen, there is a fundamental tradeoff using conventional RTOS models. By contrast, our ATGA RTOS models provide both accurate and fast simulation regardless of the granularity of back-annotated delays.

IV. SUMMARY AND CONCLUSIONS

In this paper, we presented an automatic timing granularity adjustment (ATGA) approach for accurate yet fast host-compiled software simulation. In this approach, the abstract RTOS model automatically and dynamically accumulates and adjusts back-annotated timing to provide an error-free scheduling. Our experiments show that high accuracy is achieved while maintaining fastest possible simulation speed. This makes host-compiled simulators suitable for rapid, early evaluation of the real-time performance of software-intensive embedded systems. In future work, we plan to extend the approach into a complete processor and system simulator integrated into a standard TLM framework.

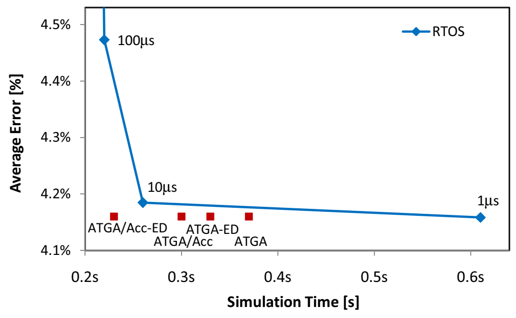


Fig. 10. Accuracy and speed tradeoffs.

REFERENCES

[1] J. Schnerr, *et al.*, “High-performance timing simulation of embedded software,” *DAC*, Jun. 2008.

[2] J. Ceng, *et al.*, “A high-level virtual platform for early MPSoC software development,” *CODES+ISSS*, Sep. 2009.

[3] K. Lin, C. Lo, R. Tsay, “Source-level timing annotation for fast and accurate TLM computation model generation,” *ASP-DAC*, Jan. 2010.

[4] T. Meyerowitz, *et al.*, “Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor,” *DATE*, Mar. 2008.

[5] Z. Wang, A. Herkersdorf, “An efficient approach for system-level timing simulation of compiler-optimized embedded software,” *DAC*, Jul.2009.

[6] Y. Hwang, S. Abdi, D. Gajski, “Cycle approximate retargettable performance estimation at the transaction level,” *DATE*, Mar. 2008.

[7] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Aplications for Embedded Systems.* Springer, 2005.

[8] D. Gajski, *et al.*, *SpecC: Specification Language and Methodology,* Kluwer, 2000.

[9] A. Gerstlauer, H. Yu, D. Gajski, “RTOS modeling for system-level design,” *DATE*, Mar. 2003.

[10] H. Posadas, *et al.*, “RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model,” *DAES*, 10(4), Dec. 2005.

[11] J.C. Prevotet, *et al.*, “A Framework for the Exploration of RTOS Dedicated to the Management of Hardware Reconfigurable Resources,” *Reconfigurable Computing and FPGAs*, 2008.

[12] A. Bouchhima, *et al.*, “Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration,” *ASPDAC*, Jan. 2005.

[13] G. Schirner, A. Gerstlauer, R. Dömer, “Fast and Accurate Processor Models for Efficient MPSoC Design,” *TODAES*, 15(2), Feb. 2010

[14] M. Krause, *et al.*, “Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation,” *CODES+ISSS*, Oct. 2008.

[15] R. S. Khaligh, M. Radetzki, “Modeling Constructs and Kernel for Parallel Simulation of Accuracy Adaptive TLMs,” *DATE*, Mar. 2010.

[16] G. Schirner, R. Dömer, “Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling,” *DATE*, Mar. 2008.

[17] P. Razaghi, A. Gerstlauer, “Host-Compiled Multicore RTOS Simulator for Embedded Real-Time Software Developement,” *DATE*, Mar. 2011.

[18] M. Dale, SWARM 0.44 Documentation. Available: http://www.cl.cam.ac.uk/∼mwd24/pdh/swarm.html