Host-Compiled Multicore RTOS Simulator for Embedded Real-Time Software Development

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

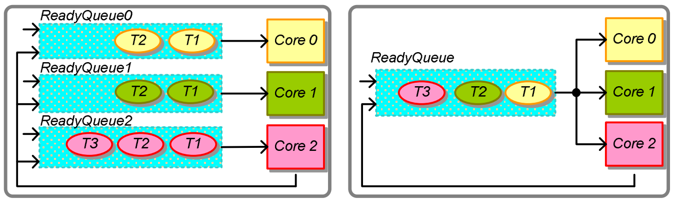
***Abstract*—With increasing demand for higher performance under limited power budgets, multicore processors are rapidly becoming the norm in today's embedded systems. Embedded software constitutes a large portion of today's systems and real-time software design on multicore platforms opens new design challenges. In this paper, we introduce a high-level, host-compiled multicore software simulator that incorporates an abstract real-time operating system (RTOS) model to enable early, fast and accurate software exploration in a symmetric multi-processing (SMP) context. Our proposed model helps designers to explore different scheduling parameters within a framework of a general SMP execution environment. A designer can easily adjust application and OS parameters to evaluate their effect on real-time system performance. We demonstrate the efficiency of our models on a suite of industrial-strength and artificial task sets. Results show that models simulate at up to 1000 MIPS with 1-3% timing error across a variety of different OS configurations.**

I. Introduction

With increasing system complexities, embedded software forms a large portion of today's systems. Software provides a high degree of flexibility and code reuse at reduced development times and costs. To achieve continued high software performance while managing power budgets, multicore processors are rapidly becoming a permanent part of embedded systems. As we are reaching limits in technology and frequency scaling, multicore architectures provide concurrent resources and high performance at lower clock frequencies and power consumption [1].

Multicore processor architectures open new software development challenges, such as ensuring correct communication and synchronization between tasks running concurrently on different cores [2]. In traditional multi-processor setups, tasks are assigned to different processors in an asymmetric multi-processing (AMP) fashion. Each processor has an independent address space and is managed by a dedicated operating system (OS). With each processor acting like a conventional single-core machine, this approach allows for easy adaptation of legacy applications. By contrast, in a multicore setup, each processor can in turn contain multiple cores that share a common set of resources and are managed by a single OS. Tasks execute on different cores under a shared memory model in a symmetric or bound multi-processing (SMP or BMP) context, where the latter model allows designers to limit task migration and to control which cores are allowed to execute a particular task.

A crucial component for the performance of real-time software on multicore processors is the OS scheduler that assigns tasks to cores. There are two general classes of multicore scheduling schemes distinguished by the number of task queues associated with each core (Fig. 1). In a partitioned scheme, each core has a separate ready queue and tasks are initially assigned to a fixed core and queue. The OS picks tasks for a core only from the associated queue, but it can perform load balancing to migrate tasks between queues and cores either at regular intervals or whenever a task leaves a core. By contrast, in a global scheme the OS 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 core utilization but is less scalable and may result in degraded performance due to cache pollution when tasks move between cores too frequently [3]. Both schemes are further characterized by the policy used to organize the tasks queues (e.g. round-robin or by priority), as well as scheduling parameters such as time slices or affinities that restrict a task to a subset of cores.



(a) Partitioned queue (b) Global queue

Figure 1. Multicore OS scheduling schemes.

Combined, there is a large number of scheduling options that determine overall real-time behavior. Complex interactions and the highly dynamic nature make static analysis difficult. Instead, designers rely on simulations to validate software and optimize the implementation for specific application requirements like real-time constraints. With ever growing complexities and software content, fast and accurate simulations of multicore platforms are therefore essential. Traditional micro-architecture or instruction set simulators (ISS) can be very accurate but tend to be slow due to a high level of detail and fine granularity required for modeling interactions, esp. in a multicore or multi-processor context.

In this paper, we introduce a novel host-compiled multicore real-time software simulator. Our main objective is to address the need for rapid yet precise design space exploration of embedded multicore software at early stages of design process. The proposed approach uses a fast, high-level host-compiled application model that incorporates an abstract real-time OS (RTOS) to accurately simulate software execution in a multicore context. The model integrates into existing transaction-level modeling (TLM) backplanes using standard system-level design languages (SLDLs) for co-simulation with external hardware and the rest of the system, e.g. in a multi-

978-3-9810801-7-9/DATE11/©2011 EDAA

processor context. Furthermore, the model is parametrizable, and software developers can easily evaluate the effect of a wide range of design decisions on real-time performance of their multicore software applications.

The rest of the paper is organized as follows: in the following subsections, we review related work and present an overview of our proposed simulator. Section II then describes the supported programming model for hosting of applications in the simulator. Details of our abstract RTOS model are discussed in Section III, and we evaluate the proposed approach on an industrial-strength design example and random task sets in Section IV. Finally, Section V concludes the paper with a summary and outlook.

A. Related Work

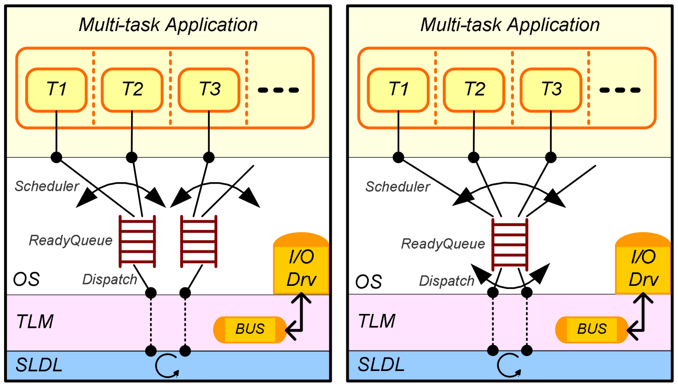
Traditionally, software is simulated using implementation-level micro-architecture or interpreted instruction-set simulators (ISSs) [4]. Such models can reach cycle accuracy at the expense of slow simulation speeds. More recently, virtual platform simulators using binary translation have become popular [5][6]. Such code morphing ISSs can provide significant speedups (reaching simulation throughputs of several hundred MIPS), but only focus on fast functional simulation with limited or no timing information. At an early and abstract level, several real-time simulation environments allow evaluation based on idealized task delay models without execution of actual task functionalities [7][8]. Other approaches evaluate different multicore scheduling strategies directly on a real OS and architecture [9][10]. However, such low-level implementations are time-consuming and labor-intensive.

Host-compiled (sometimes also called native or computational TLM) approaches have recently received widespread attention as solutions that can provide both fast and accurate simulations [11][12]. In such approaches, application code is natively compiled and executed on the simulation host to achieve fastest possible functional simulation. The code is further back-annotated with timing estimates that are typically obtained by compiling to an intermediate representation [13][14]. Finally, back-annotated code is wrapped into an abstract model of the software execution environment built and integrated on top of a proprietary or standard SLDL environment, such as SystemC [15] or SpecC [16].

Some of the earliest host-compiled approaches were centered around models of the OS itself [17][18]. Later, these approaches were extended into complete processor models that include timing-accurate descriptions of interrupt chains and TLM-based bus interfaces [19][20]. Such processor models have been shown to simulate at speeds beyond 500 MIPS with more than 95% timing accuracy. In all cases, however, existing models are for single core processors only. To the best of our knowledge, no host-compiled real-time multicore OS and processor modeling approach currently exists.

B. Host-Compiled Multicore RTOS Simulator

We propose a high-level, host-compiled multicore RTOS simulator that is based on the approach presented in [17]. Fig. 2 illustrates the structure of the simulator, which is constructed in a layered-based fashion. At the highest level, the user application consists of a set of sequential and concurrent tasks that are controlled by and interact with an underlying OS model through an abstract OS interface. We extend existing approaches by developing a novel multicore SMP OS model that supports both partitioned and global queue structures. In both cases, the OS layer manages the scheduling and dispatching of application tasks across available queues and cores. In doing so, the OS model wraps around the basic SLDL event handling mechanism, replacing SLDL primitives with calls to the OS interface instead and ensuring that at any time only as many SLDL threads as there are cores are active. The complete simulator is developed on top of a SLDL simulation kernel, which provides basic concurrency and event handling models and can be integrated with standard TLM libraries to provide an environment for fast system-wide co-simulation. In the following sections, we will describe each of the layers of our simulator in more detail.



(a) Partitioned queue structure (b) Global queue structure

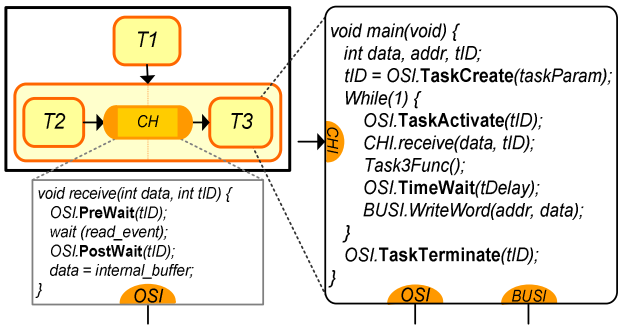
Figure 2. Host-compiled multicore RTOS simulation.

II. Multi-task Application Model

We provide a simple yet powerful programming model to mount applications on top of our simulator and the underlying C-based SLDL (Fig. 3). As mentioned before, an application model is composed out of concurrent and sequential high-level SLDL processes. Tasks can communicate and synchronize with each other using high-level inter-process communication (IPC) primitives provided by the standard SLDL channel library. For external communication with the rest of the system, tasks can access externally provided TLM bus interfaces.

Internally, task code is provided in standard C-based form. For timing accuracy, we assume that the execution delays of the task functionality are back-annotated from measurements or estimations. User application tasks are integrated into and access services of the OS model via an abstract OS interface shown in Fig. 4. Fig. 3 demonstrates the general structure of a multi-task application and shows how a software designer utilizes the OS interface to run the application on the multicore simulator. The OS interface thereby provides the facilities to configure the OS and integrate an application with APIs for OS initialization and startup, task management, execution delay modeling, and event synchronization.

During the system startup phase, the *Init()* method, initializes the OS model data structures and defines the OS parameters such as the number of cores (see Section III). The *Start()* method triggers multicore scheduling after all tasks have been attached to the model. Tasks are added to the OS kernel by calling *TaskCreate()*, which allocates an internal



```
void receive(int data, int tID) {
  OSI.PreWait(tID);
  wait (read_event);
  OSI.PostWait(tID);
  data = internal_buffer;
}
```

```
void main(void) {
  int data, addr, tID;
  tID = OSI.TaskCreate(taskParam);
  While(1) {
    OSI.TaskActivate(tID);
    CHI.receive(data, tID);
    Task3Func();
    OSI.TimeWait(tDelay);
    BUSI.WriteWord(addr, data);
  }
  OSI.TaskTerminate(tID);
}
```

Figure 3. High-level multi-task application model.

representation inside the OS. At the start of simulation, task threads are spawned by the SLDL and 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()*. In addition, tasks can suspend themselves (*TaskSleep()*) or can be resumed or killed by other tasks (*TaskResume()* and *TaskKill()*). Finally, 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()*).

During creation of tasks, the designer can identify task properties and characteristics. Specifically, the supported task parameters are:

1) *Affinity:* a bitmap representing the set of cores allowed to execute the task;
2) *Initial Core:* the initial core for the task in a partitioned queue structure;
3) *Priority:* the priority level of the tasks;
4) *Time Slice:* time interval the task is allowed to execute without preemption by other tasks with the same priority.

During execution, tasks communicate with the OS interface to model delays and synchronization points. The designer utilizes *TimeWait()* methods for modeling task delays, which in turn define possible preemption points. Likewise, *PreWait()* and *PostWait()* methods are used to encapsulate regular wait for event statements that implement event and task synchronization. Similar to forking and joining, these methods remove and return the task from/to OS scheduling while it waits for an external SLDL event that can in turn be notified by another task through regular SLDL mechanisms. Note that the designer will typically not have to deal with event handling directly. Instead, we provide a reimplementation of the SLDL channel library that is properly hooked into the OS.

III. Abstract Multicore OS Model

We have designed and developed a general model of a multicore RTOS with comprehensive configuration through task and OS parameters. The designer can adjust the OS model for a desired application to meet system requirements. The OS model parameters are as follows:

1) *Number of Cores*: Our simulator supports a configurable number of cores in the OS model.
2) *Scheduler Structure*: We support both partitioned and global ready queue structures.

```
/* OS initialization and startup */
void Init(OSParam param);
void Start(void);
/* Task management */
proc TaskCreate(TaskParam param);
proc ParStart(proc tID);           /* fork */
void ParEnd(proc tID);             /* join */
void TaskActivate(proc tID);
void TaskSleep(proc tID);
void TaskResume(proc ID);
void TaskTerminate(proc tID);
void TaskKill(proc tID);
 /* Delay modeling and event handling */
void TimeWait(long long nsec, PROC p);
proc PreWait(proc tID);
void PostWait(proc me);
```

Figure 4. Multicore OS interface.

3) *Scheduling Policy*: In general, the designer can assign different priorities and time slices to each task. For first come-first serve (FCFS or FIFO) scheduling, tasks are assigned the same priority and an infinite (-1) time slice. For round-robin (RR) scheduling, task with the same priority are assigned a fixed time slice instead.

Overall, our OS model helps designers to evaluate the real-time behavior of applications across different scheduling policies and facilities.

A. OS Scheduler

At the core of the OS model is the multicore scheduler, the body of which is shown in Fig. 5. The scheduler is an internal function of the OS model and is called by the OS interface 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. Note that suspended tasks, e.g. when waiting for an event wrapped via *Pre/PostWait()*, will have been previously removed from all cores and ready queues, placing them in a wait state and separate wait queue instead.

We utilize a time slice notion to model FIFO or RR scheduling on tasks that have the same priority. When the RTOS runs the scheduler for a specific core, it calculates the remaining time slice of the current active task on the desired core (lines 4-5 in Fig. 5). Then, the current task is moved to the corresponding ready queue based on the new value of the time

```
1   function Scheduler(int CoreID):
2   QueueID = GLOBAL_SCHEDULING? 0 : CoreID;
3   if ((oldcurrent = current[CoreID]) != NIL) then
4     TaskList[current[CoreID]].TimeSlice -=
5             (now() - TaskList[current[CoreID]].StartTime);
6     TaskList[current[CoreID]].State =  READY;
7     if (TaskList[current[CoreID]].TimeSlice == 0 ) then
8       FIFO(&ReadyQueue[QueueID], current[CoreID]);
9     else
10      LIFO(&ReadyQueue[QueueID], current[CoreID]);
11    endif
12    current[CoreID] = NIL;
13  endif
14  Dispatch(CoreID);
15  if (oldcurrent) wait4sched(oldcurrent);
```

Figure 5. Multicore RTOS scheduler.

```
function Dispatch(int coreid):
LoadBalance(coreid);
if (!empty(&ReadyQueue[coreid]) then
  ActiveTaskID = GetFirst(&ReadyQueue[coreid]);
  current[coreid] = ActiveTaskID;
  TaskList[current[coreid]].State = run;
  SendSched(ActiveTaskID);
else
  current[coreid] = NIL;
endif
```

(a) Partitioned queue.

```
function Dispatch(int coreid):
ReadyQueue_Mutex.Acquire();
if (!empty(&ReadyQueue[coreid]) then
  ActiveTaskID = GetFirst(&ReadyQueue, coreid).
  if (ActiveTaskID != NIL) then
    current[coreid] = ActiveTaskID;
    TaskList[ActiveTaskID].State = run;
    SendSched(ActiveTaskID);
  else
    current[coreid] = NIL;
  endif
else
  current[coreid] = NIL;
endif
ReadyQueue_Mutex.Release();
```

(b) Global queue.

Figure 6. Multicore task dispatcher.

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. Otherwise, it will be placed back at the beginning of the priority list and 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, while setting an infinite time slice value will result in a FIFO schedule.

At the end of the scheduler, an OS-specific *Dispatch()* function will be called to assign a new task to the current core. Fig. 6 shows the implementation of *Dispatch()* function for both global and partitioned queue structures. This function selects the highest priority task in the ready queue and assigns it to run on the current core. Ready queue are sorted by tasks priorities, and tasks with the same priority are arranged based on time slices. In case of partitioned queues (Fig. 6(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 (Fig. 6(b)), a semaphore controls all accesses to the shared queue. In both cases, the choice of tasks allowed to be migrated or picked for a given core is further restricted based on task affinities.

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 implements 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.

B. OS Kernel

The abstract model of the complete OS kernel implements OS interface methods by calling the scheduler at appropriate preemption points and task switch events. The *TimeWait()* method models task delays by calling the underlying SLDL to advance time, followed by a call to the OS scheduler. The latter gives any higher priority tasks that are asynchronously triggered (e.g. by a different core or an external interrupt) a chance to preempt the currently running task. The *PreWait()* and *TaskSleep()* methods remove the current task from all cores and ready queues, put it into a wait or sleep state and queue, and call the OS scheduler to assign a new task to the now idle core. At the other end, the *PostWait()* and *TaskResume()* methods return the calling or given task back into the ready queue after it has been externally woken up. From there it will be picked up for scheduling on the next scheduler call. Finally, both *TaskActivate()* and *TaskSleep()* suspend the calling task and wait until the *SendSched()* function activates the schedule event of the corresponding task.

IV. Experimental Results

To validate our models and demonstrate their efficiency for fast and accurate design space exploration, we have applied our approach to a set of randomly generated artificial tasks and an industrial-strength design example. For our experiments, we implemented models in SpecC, but we are also in the process of transferring results to other SLDLs, such as SystemC.

A. Random task sets

To evaluate the speed and accuracy of the proposed models, we compared the execution behavior of artificial task sets running on our processor model and on a virtual reference platform. We evaluate the response time of randomly generated periodic tasks on a dual-core MIPS34Kc Malta platform running a 2.6.24 Linux SMP kernel, which realizes a partitioned queue scheduling scheme and is configured with preemption and high resolution timers. We compare our model of this platform against a reference ISS running the actual Linux binary [6]. Following the setup presented in [9], task periods are uniformly distributed over [10, 100] ms, while task utilizations are distributed over [0.001, 0.1], [0.1, 0.4], and [0.001, 0.4], i.e. in the small (S), large (L) or whole/medium (M) range of execution delays. For each task set, we generate tasks until a maximum is reached or the core utilization falls into the range [0.3, 0.6), [0.6, 0.8) or [0.8, 1) for light, medium, or heavy task loads, respectively. Tasks priorities were assigned inversely to their periods following a rate-monotonic scheduling strategy. Actual task delays were measured on the reference simulator and back annotated into our model at different levels of timing granularity.

Generated task sets and resulting modeling accuracies are summarized in Table I, II and Fig. 7. Model error was measured as the average absolute difference in individual task response times over all tasks and task iterations. Results show that with a timing granularity of 1 µs and 10 µs the average timing error across all task sets is less than 0.4% and 1%, respectively. In general, timing error grows with increasing timing granularity. In addition, the timing error is a function of the number of task switches (Fig. 8), and it is higher for task sets with a large number of small tasks. This is due to

TABLE I. AVERAGE ERROR FOR SETS WITH SMALL TASKS UNDER DIFFERENT TIMING GRANULARITIES.

Task Set	S1		S2		S3		S4		S5	
Core ID	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1
# of Tasks	7	6	11	10	9	9	12	15	13	16
Total Util.	0.33	0.30	0.47	0.45	0.56	0.50	0.70	0.69	0.84	0.87
Avg. Task Util.	0.047	0.050	0.043	0.045	0.062	0.056	0.058	0.046	0.064	0.054
Avg. Err. (1 μs)	0.58%	0.30%	0.39%	0.73%	0.64%	0.38%	1.02%	0.88%	1.10%	1.12%
Avg. Err. (10 μs)	0.54%	0.29%	0.33%	0.68%	0.57%	0.35%	0.88%	0.81%	0.98%	0.98%
Avg. Err. (100 μs)	0.94%	0.59%	1.51%	1.78%	1.17%	0.58%	2.74%	1.77%	2.04%	3.57%
Avg. Err. (1 ms)	8.56%	3.55%	10.0%	10.1%	8.58%	4.09%	16.8%	12.6%	13.2%	15.6%

TABLE II. AVERAGE ERROR FOR SETS WITH LARGE AND MEDIUM TASKS UNDER DIFFERENT TIMING GRANULARITIES.

Task Set	L1		L2		L3		M1		M2		M3		M4	
Core ID	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1	C0	C1
# of Tasks	3	4	3	2	3	3	4	4	4	4	4	3	3	3
Total Util.	0.63	0.44	0.64	0.63	0.88	0.92	0.54	0.56	0.69	0.64	0.71	0.70	0.86	0.69
Avg. Task Util.	0.209	0.109	0.212	0.315	0.294	0.306	0.137	0.139	0.173	0.160	0.176	0.235	0.286	0.230
Avg. Err. (1 μs)	0.14%	0.09%	0.08%	0.14%	0.07%	0.11%	0.65%	0.14%	0.25%	0.55%	0.35%	0.16%	0.12%	0.09%
Avg. Err. (10 μs)	0.16%	0.08%	0.08%	0.13%	0.04%	0.13%	0.57%	0.1%	0.25%	0.53%	0.23%	0.15%	0.11%	0.09%
Avg. Err. (100 μs)	0.26%	0.23%	0.50%	0.14%	0.43%	1.02%	0.88%	1.1%	0.26%	1.97%	1.05%	0.44%	0.31%	0.12%
Avg. Err. (1 ms)	2.40%	2.91%	4.70%	0.63%	4.30%	9.05%	8.4%	12.7%	1.44%	19.3%	11.4%	3.16%	2.75%	0.81%

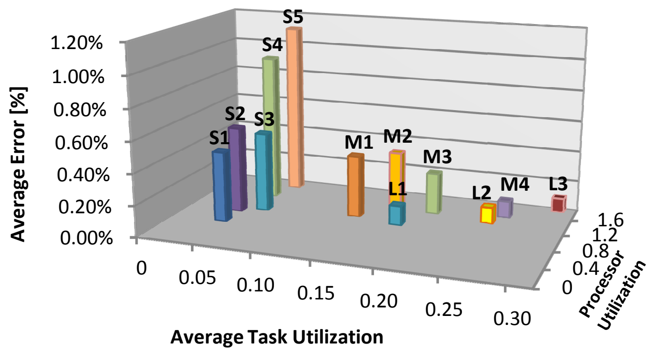


Figure 7. Average error in average response time of 1μs task sets.

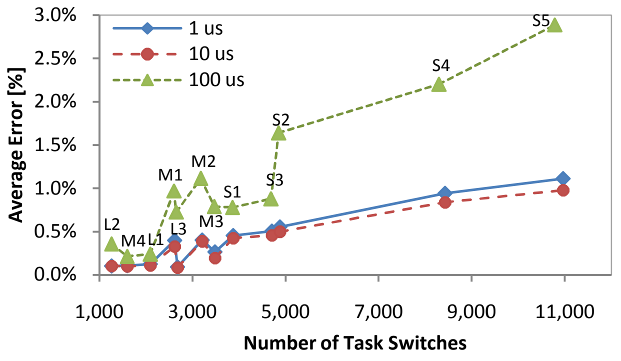


Figure 8. Average timing error over number of task switches.

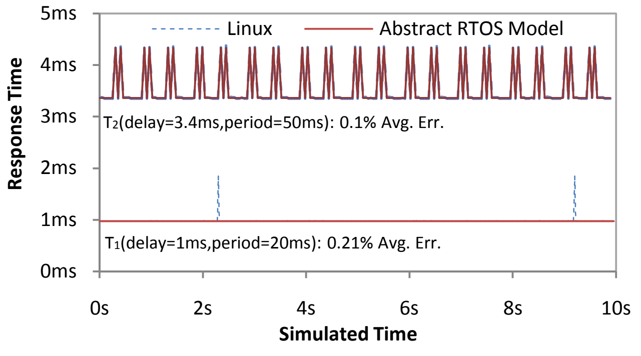


Figure 9. Response time traces for M1 task set.

TABLE III. SIMULATION TIME PER TASK SET.

Set	1 μs	10 μs	100 μs	1000 μs
S1	4.1s (240MIPS)	0.36s (2800MIPS)	0.07s (14.3GIPS)	0.03s (33GIPS)
S2	5.0s (200MIPS)	0.59s (1700MIPS)	0.08s (12.5GIPS)	0.05s (20GIPS)
S3	6.1s (160MIPS)	0.60s (1700MIPS)	0.08s (12.5GIPS)	0.04s (25GIPS)
S4	12.0s (80MIPS)	0.85s (1200MIPS)	0.14s (7.1GIPS)	0.04s (25GIPS)
S5	10.5s (90MIPS)	0.95s (1000MIPS)	0.16s (6.2GIPS)	0.08s (12GIPS)
M1	7.6s (130MIPS)	0.61s (1600MIPS)	0.10s (10GIPS)	0.03s (33GIPS)
M2	7.5s (130MIPS)	0.78s (1300MIPS)	0.12s (8.3GIPS)	0.03s (33GIPS)
M3	7.6s (130MIPS)	0.86s (1200MIPS)	0.12s (8.3GIPS)	0.04s (25GIPS)
M4	9.3s (110MIPS)	0.89s (1100MIPS)	0.16s (6.2GIPS)	0.03s (33GIPS)
L1	5.9s (170MIPS)	0.74s (1300MIPS)	0.12s (8.3GIPS)	0.03s (33GIPS)
L2	7.4s (130MIPS)	0.72s (1300MIPS)	0.10s (10GIPS)	0.02s (50GIPS)
L3	10s (100MIPS)	1.08s (920MIPS)	0.13s (7.7GIPS)	0.04s (25GIPS)

context switch delays, which are not yet included in our model. Finally, another source of errors is the non-ideal behavior of the real Linux kernel. Fig. 9 plots the response times of the two highest-priority tasks in the M1 task set on the Linux kernel and our simulator. As can be seen, on the Linux kernel the highest priority task is interrupted at regular intervals additional, unknown background activities. Table III shows the simulation runtimes of the models for each task set. We ran each task set for 10 s of simulated time. At a nominal rate of 100 MIPS simulated by the reference ISS, this corresponds to 1000 million NOP instructions on each core for artificial delay loops and any idling. Simulations each ran for about 30 s of wall time on the reference ISS. By contrast, our model simulates such a non-functional, delay-only setup in faster than real time with a throughput of more than 1000 MIPS per core (or 2000 MIPS for the whole dual-core system) at timing granularities of 10 μs and above.

B. *Industrial-strength example*

To demonstrate the benefits of our models for design space exploration, we implemented a cellphone example running concurrent control, MP3 decoding, and Jpeg encoding tasks on a model of a dual-core ARM system running at 100 MHz. Tasks communicate with external hardware and the rest of the system via an AHB bus. Task delays are back-annotated from measurements obtained on a cycle accurate ISS [21]. We explored both single-core and global and partitioned dual-core

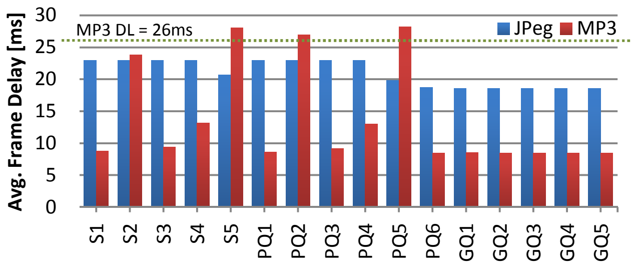


Figure 10. Simulated frame delays across different OS configurations.

scheduling structures with different task priorities, core assignments and time slices (FIFO or RR with different intervals). The complete system is modeled in 32,000 lines of SLDL code and effects of different scheduling policies on system performance were evaluated using model simulations.

Fig. 10 and Table IV compare average and maximum MP3 and Jpeg frame delays for all possible configurations. Results generally match expectations. When running on different cores (PQ6) or in a global queue (GQ*x*), each task is effectively assigned to a dedicated core and frame delays are consistently minimal. When running on a single (S*x*) or the same core (PQ1-PQ5), the MP3 task misses deadlines if its priority is lower than that of the Jpeg encoder (S5 or PQ5). Likewise, a FIFO strategy can lead to unpredictable blocking of the MP3 decoder by the Jpeg task (S2 and PQ2). Since the order of tasks executions highly depends on the order in which tasks become ready, results can vary widely, e.g. depending on the background load on the corresponding core. By contrast, a RR strategy can provide the necessary fairness in task accesses to a single core. Overall, such effects are hard to predict statically.

All in all, we were able to easily cycle through different system configurations of our model within seconds. Furthermore, system models simulate at an average speed of 520 Mcycles/s. Assuming a nominal CPI of 1, this corresponds to 520 MIPS per core or 1040 MIPS total. We simulated decoding of 55 MP3 frames and encoding of one 640x480 picture. Not counting idle periods and hardware interactions, this translates into roughly 200 million simulated instructions for an average of 610 MIPS. All in all, models provide rapid feedback for evaluation and analysis of scheduling options.

V. Summary and Conclusion

In this paper, we presented a configurable and high-level host-compiled multicore software simulator that integrates an abstract SMP RTOS model into a complete SLDL- and TLM-based multi-processor and system environment. Simulations are fast and accurate at speeds of more than 1000 MIPS with less than 3% timing error, supporting rapid and early embedded software development and design space exploration. A medium timing granularity of 10µs thereby seems to represent the best compromise. In the future, we plan to further investigate tradeoffs and improvements in accuracy and speed, e.g. through automatic adjustment of timing granularities, by including OS-internal timing models, and by broadly supporting other scheduling algorithms, such as PFair. Furthermore, we are currently working on expanding the OS model into a complete parametrizable processor simulator that includes models of interrupt chains and caches for evaluation of synchronization, task migration and cache pollution effects.

TABLE IV. Cellphone Example Experimental Results

Configuration			Avg. / Max. Frame Delay		MCPS	Sim. Time
			Jpeg [ms]	**MP3 [ms]**		
Single Core	**S1**	MP3 > Jpeg	23.01/25.36	8.84/9.99	500	0.35s
	S2	MP3 = Jpeg, FIFO	23.01/25.45	23.84/43.43	583	0.30s
	S3	MP3 = Jpeg, RR(1us)	23.01/25.36	9.42/10.79	514	0.34s
	S4	MP3 = Jpeg, RR(.5ms)	23.01/25.36	13.16/15.99	530	0.33s
	S5	Jpeg > Mp3	20.70/23.97	28.12/79.69	667	0.32s
Dual Core-PQ	**PQ1**	1:MP3 > Jpeg	22.97/25.29	8.65/9.72	427	0.35s
	PQ2	1: MP3=Jpeg, FIFO	22.97/32.25	26.97/43.70	546	0.32s
	PQ3	1: MP3=Jpeg, RR(1us)	22.97/25.29	9.23/10.61	605	0.29s
	PQ4	1: MP3=Jpeg, RR(.5ms)	22.97/25.28	13.05/16.22	546	0.32s
	PQ5	1:Jpeg > MP3	19.82/23.01	28.24/115.8	529	0.33s
	PQ6	1:MP3, 2:Jpeg	18.73/21.20	8.52/8.59	724	0.31s
Dual Core-GQ	**GQ1**	MP3 > Jpeg	18.66/21.21	8.56/9.80	426	0.35s
	GQ2	MP3 = Jpeg, FIFO	18.62/20.81	8.52/8.67	414	0.36s
	GQ3	MP3 = Jpeg, RR(1us)	18.62/21.20	8.52/8.67	438	0.34s
	GQ4	MP3 = Jpeg, RR(.5ms)	18.62/20.90	8.52/8.67	438	0.34s
	GQ5	Jpeg > MP3	18.62/20.81	8.52/8.66	438	0.34s

References

[1] G. Blake, R.G. Dreslinski, T. Mudge, "A survey of multicore processors," *IEEE Signal Processing Magazine*, vol.26, no.6, Nov. 2009.
[2] R. Goodman, S. Black, "Design challenges for realization of the advantages of embedded multi-core processors," *AUTOTESTCON*, Sept. 2008.
[3] S. Lauzac, R. Melhem, D. Mosse, "Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor," *Euroicro Workshop on Real-Time Systems*, Jun. 1998.
[4] L. Benini, D. Bertozzi, A. Bogoliolo, F. Menichelli, M. Olivieri, “MPARM: Exploring the multi-processor SoC design space with SystemC,” *Journal of VLSI Signal Processing*, vol. 41, no. 2, 2005.
[5] F. Bellard, “QEMU, a fast and portable dynamic translator,” *USENIX*, 2005.
[6] Open Virtual Platforms [online]. Available: http://www.ovpworld.org
[7] Xenomai: Real-Time Famework for Linux [online]. Available: http://www.xenomai.org
[8] VirtualTime: Simulation of Real-Time Systems [online]. Available: http://www.rapitasystems.com/virtualtime
[9] J. Calandrino, H. Leontyev; A. Block, U. Maheswari, C. Devi, J. H. Anderson, "LITMUS^RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," *RTSS*, Dec. 2006.
[10] RTSIM [online]. Available: http://rtsim.sssup.it
[11] J. Schnerr, O. Bringmann, A. Viehl, W. Rosenstiel, “High-performance timing simulation of embedded software,” *DAC*, Jun. 2008.
[12] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, H. Meyr, “A high-level virtual platform for early MPSoC software development,” *CODES+ISSS*, Sep. 2009.
[13] Z. Wang, A. Herkersdorf, “An efficient approach for system-level timing simulation of compiler-optimized embedded software,” *DAC*, Jul. 2009.
[14] Y. Hwang, S. Abdi, D. Gajski, “Cycle approximate retargettable performance estimation at the transaction level,” *DATE*, Mar. 2008.
[15] SystemC [online]. Available: http://www.systemc.org
[16] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer, 2000.
[17] A. Gerstlauer, H. Yu, D. Gajski, "RTOS modeling for system-level design," *DATE*, Mar. 2003.
[18] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, F. Escuder, “RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model,” *DAES*, vol. 10, no. 4, Dec. 2005.
[19] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A. Jerraya, “Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration,” *ASPDAC*, Jan. 2005.
[20] G. Schirner, A. Gerstlauer, R. Dömer, "Fast and Accurate Processor Models for Efficient MPSoC Design," *TODAES*, vol. 15, no. 2, article no. 10, Feb. 2010.
[21] M. Dales, SWARM 0.44 Documentation. Available: http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html