# A Parallel Simulation Technique for Multicore Embedded Systems and Its Performance Analysis

Dukyoung Yun, Sungchan Kim, *Member, IEEE,* and Soonhoi Ha, *Senior Member, IEEE*

***Abstract*—A virtual prototyping system is constructed by replacing real processing components with component simulators running concurrently. The performance of such a distributed simulation decreases drastically as the number of component simulators increases. Thus, we propose a novel parallel simulation technique to boost up the simulation speed. In the proposed technique, a simulator wrapper performs time synchronization with the simulation backplane on behalf of the associated component simulator itself. Component simulators send null messages periodically to the backplane to enable parallel simulation without any causality problems. Since excessive communication may degrade the simulation performance, we also propose a novel performance analysis technique to determine an optimal period of null message transfer, considering both the characteristics of a target application and the configurations of the simulation host. Through intensive experiments, we show that the proposed parallel simulation achieves almost linear speedup to the number of processor cores if the frequency of null message transfer is optimally decided. The proposed analysis technique could predict the simulation performance with more than 90% accuracy in the worst case for various target applications and simulation environments we have used for experiments.**

***Index Terms*—Distributed simulation, embedded system, multiprocessor, parallel simulation.**

## I. Introduction

A VIRTUAL prototyping system is commonly used to validate the expected performance as well as functionality of an embedded system before a hardware prototype is built. It enables us to develop software without real hardware and to explore the design space of the target architecture to find an optimal design satisfying the required design constraints on performance, power, cost, and so on. A virtual prototyping system can be constructed by replacing real processing components with simulation models. For this purpose, an instruction set simulator (ISS) is used as a processor simulator and a SystemC simulator as the communication architecture simulator. Thus, a virtual prototyping system becomes a distributed event-driven simulation system.

A key problem of distributed simulation is scalability. As more processing elements are integrated into a system, the simulation speed degrades worse than inverse proportionally since we have to pay the overhead of communication and synchronization between processor simulators. To support third party simulators without modification, we use interprocess communication (IPC) mechanism based on pipe or socket for intersimulator communication. Then a single transaction takes around $1\,\mu$s inside the same host machine, which is long enough to simulate several hundred machine instructions of the target processor. If simulators are running on different host machines, the communication overhead easily becomes ten times larger.

In this paper, we propose a *parallel* simulation technique that boosts the simulation performance significantly. The key idea is to attach a simulator wrapper to each processor simulator as shown in Fig. 1. A simulator wrapper is a module that synchronizes the simulation backplane and the corresponding processor simulator. The simulator wrappers and the communication architecture simulator are implemented in a single process, which is called a simulation backplane. Processor simulators and the simulation backplane run concurrently. Each processor simulator communicates with the associated wrapper only when the processor accesses shared memory. A simulation host deploys multiple processor cores to enable parallel simulation. We map each processor simulator to a dedicated core on the host machine for maximum exploitation of parallel simulation. The simulation backplane is also mapped to a dedicated core as illustrated in Fig. 1.

The simulation backplane is a conductor of the overall simulation. It monitors the progress of each simulator as it receives a bus request to access shared memory or an interrupt signal from the processor simulator. The backplane collects progress information from all simulators, and replies to the simulator. Thus, shared memory access and interrupt check belong to synchronous communication, IPC that requires a response. If a simulator does not send any request to the backplane, the backplane may not advance the global simulation clock. To overcome this problem, each processor simulator periodically

Manuscript received April 21, 2011; revised July 28, 2011; accepted August 11, 2011. Date of current version December 21, 2011. This work was supported by research funds of the Chonbuk National University in 2010, by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology, under Grant 2011-0026105, by the BK21 Project and Acceleration Research sponsored by the Korea Science and Engineering Foundation Research Program, under Grant R17-2007-086-01001-0, by the Seoul R&BD Program, under Grant JP090955, by the System IC 2010 Project of the Korean Ministry of Knowledge Economy, and by the Technology Innovation Program (Industrial Strategic Technology Development Program, KI002168, Development of Configurable Device and SW Environment) funded by the Ministry of Knowledge Economy (Korea). The Institute of Computer Technology and the Inter-University Semiconductor Research Center at Seoul National University and the IC Design Education Center provided research facilities for this paper. This paper was recommended by Associate Editor Y. Xie.

D. Yun and S. Ha are with the School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-600, Korea.

S. Kim is with the Division of Computer Science and Engineering, Chonbuk National University, Jeonbuk 561-756, Korea (e-mail: sungchan.kim@chonbuk.ac.kr).

Digital Object Identifier 10.1109/TCAD.2011.2167329

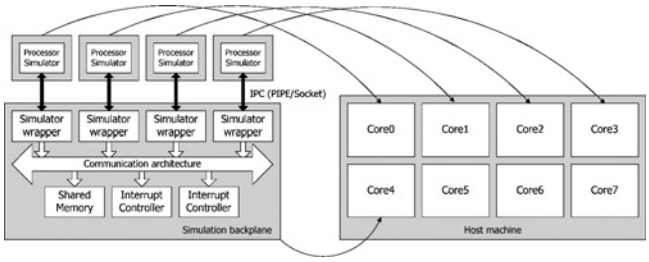



Fig. 1. Organization of the proposed parallel simulation framework.

transmits a *null* message to notify the progress of the processor simulation in time to the backplane, which is commonly used in distributed event-driven simulation [2]. Since a null message transmission does not require a response, it is referred to as asynchronous communication.

The main benefit of the proposed technique comes from the fact that time synchronization between processor simulators is accomplished within the backplane, which is very fast without any IPC overhead. Since the backplane contains the simulation model of the underlying communication architecture, additional overhead such as bus conflicts can be readily considered. Note that the simulator wrapper, not the processor simulator, manages the global time information. One obstacle to this scheme is handling of interrupts. In the base scheme, a processor simulator is synchronized with the rest of the system at shared resource accesses. But an interrupt may occur at any time. Therefore, we may have to use lock-step synchronization to guarantee cycle-accuracy of interrupt services. Instead, we propose to sacrifice the timing accuracy for interrupt services as long as it does not harm the functional correctness of the design, preserving the speed benefit of parallel simulation.

In such a parallel simulation environment, the IPC frequency has significant effects on the simulation performance: excessive communication and synchronization overhead between simulators running concurrently may degrade the simulation performance. Even though the *null* message transmission improves the parallelism effectively, too frequent message transmissions will result in degradation of simulation performance. Hence we also propose a novel analysis technique for simulation performance estimation, considering both the characteristics of a target application and the configuration parameters of the simulation host. It can be used to determine the optimal frequency of null message transfers for achieving the maximal simulation performance.

The rest of this paper is organized as follows. Section II reviews the related work on parallel simulation of embedded systems. Section III explains the synchronization scheme that promises significant performance boost. In Section IV, the proposed analysis technique to estimate simulation performance is presented. In Section V, the experimental results show the effectiveness of the proposed technique. Finally, we conclude this paper with future work in Section VI.

## II. Related Work

Most research on distributed event-driven simulation for multicore embedded systems has been focused on performance improvement by raising the abstraction level of the simulation model and/or reducing the synchronization overhead between the component simulators. We first review the previous studies from the time synchronization perspective.

### A. Time Synchronization

A lock-step method synchronizes processor simulators every cycle, which is commonly used due to its simplicity and high accuracy. Even though this method assures causality that no past event arrives at the simulator, it degrades the simulation performance significantly because of excessive IPC overhead. There are two kinds of approaches to reduce the synchronization frequency: conservative and optimistic [2].

A conservative approach guarantees that no past event will occur by advancing the local clock of a simulator only up to the minimum timestamp of the possible input events. An optimized conservative approach [8] was proposed assuming that a simulator notifies globally of the next earliest event, which is usually not feasible. In [9], time synchronization with equidistant interval has been suggested for HW/SW cosimulation. The communication overhead between HW and SW components can be reduced by increasing the interval. However, the functionality of target application may be corrupted if the synchronization interval is not carefully chosen considering the characteristics of target application. For instance, shared memory consistency will not be preserved. In [5], the time synchronization point is predicted based on a static analysis of application software running on each processor. The predicted minimum time, called *lookahead*, should be large in order to get more gain. A similar approach to [5] is employed in [6] where the lookahead is obtained by dynamic execution path prediction and hardware prediction. The performance of these approaches depends on the accuracy of prediction.

An optimistic approach, on the other hand, allows each simulator to advance its local clock optimistically assuming that no past event will arrive [4], [10]. Once the assumption is broken, current simulation should roll back to the latest checkpoint to handle the events safely without any causality problem. If a processor simulator does not support a roll back mechanism, as is the usual case, this approach is not applicable.

Recently, a technique, called virtual synchronization, has been proposed where each processor simulator does not synchronize its own local clock to the global clock [11]. The central idea is to separate functional simulation and timing simulation. A processor simulator performs function simulation and collects the traces of special events, shared memory accesses and context switches. A simulator is blocked if it meets a shared memory access until the access is served. The simulation backplane receives the event traces from all processor simulators, aligns the events, and services the events in the chronological order. Since a processor simulator communicates with the simulation backplane only when it is blocked, the frequency of intersimulator communication is reduced. In the virtual synchronization technique, however, the simulated order of task executions may be different from the actual order. In this case, it is not possible to port an operating system (OS) on an ISS. Instead, an OS modeler is used to emulate the OS activities, which is the main source of timing inaccuracy.

### *B. Parallel Simulation*

Another approach to boost the simulation speed is to parallelize the simulation. Parallel distributed event simulation (PDES) can be divided into time-parallel simulation and space-parallel simulation [2]. Time-parallel algorithms divide the simulated time[1] axis into intervals, and assign each interval to different processors. However, they have "state-matching" problem that the states at the boundary of the time intervals must be matched and rely on specific properties of the system being modeled. On the other hand, space-parallel algorithms assign logical processes to different processors. They offer greater flexibility and wider applicability and the proposed simulation technique falls into this category.

Time synchronization has been a major concern in the context of PDES. In Wisconsin wind tunnel approach [7], multiple processor simulators are executed in parallel based on a conservative time bucket synchronization scheme where simulators are synchronized per a predefined interval. This way sacrifices simulation accuracy, but increases parallelism [17]. Also, an adaptive technique that adjusts the quantum size is proposed in [16]. By dynamically relaxing accuracy over less interesting computational phases, it increases performance with a marginal loss of precision. However, the technique is not generally applicable to the simulation of embedded MPSoC where a timing error may incur serious functional corruption.

A common PDES approach is to let simulators run in parallel, preserving the causality condition locally. If an input event queue is empty, the simulator has to wait until it receives any event from the other simulators to compute the minimum timestamp. If there is a cyclic dependency between simulators, this can lead to deadlock. One solution to resolve the deadlock is that simulators exchange null messages, which contain only a timestamp without any value [3]. A null message is a promise that the sender simulator will not send an event whose timestamp is smaller than the one in the null message. Since the null message transfer contributes to the time synchronization overhead, reducing the number of null messages affects directly the performance of simulation. If a null message is exchanged at every cycle, it degenerates to a lock-step simulation.

In [12], a parallel simulation technique based on the virtual synchronization has been proposed using a virtual buffer that can be regarded as a pipeline buffer between processor simulators. However, the OS modeling problem of the virtual synchronization still remains unresolved.

From the time synchronization point of view, the proposed technique resembles the virtual synchronization technique in that the processor simulators are not synchronized to the global clock. However, our technique executes an OS directly as a part of the application so that we do not need to model the OS and there is no restriction on the programming model. Also the proposed technique is a conservative approach that guarantees free from causality error by sending null messages dynamically. Therefore, it does not need any prediction of future events. The proposed technique is extended from our previous work [1]. We have extended the previous study with in-depth analysis to estimate the simulation performance. This leads to the maximal simulation performance by predicting the optimal period of null message transfers for a given application and simulation environment.

[1]The host machine time and the target processor time should be distinguished. We represent the host machine time as simulation time, and the target processor time as simulated time as in [2].

## III. Proposed Techniques for Parallel Simulation

In this section we explain two key techniques to enable parallel simulation that supports asynchronous interrupts: period update and imprecise interrupt modeling.

### *A. Parallel Execution with Periodic Update*

In the proposed parallel simulation, when a processor simulator requests a shared resource access, it sends the request message along with the elapsed time from the last request to the current request time to the connected simulator wrapper. Then the simulator wrapper increases its simulated time by the elapsed time, and posts the request to the communication architecture module in the backplane to simulate the request. Note that the time stamp associated with this request is the simulated time of the processor simulator. Once a processor simulator issues a request to the wrapper, it waits until the request is serviced by the simulation backplane.

Shared memory accesses are processed by the communication architecture module in the simulation backplane. More importantly, the simulation backplane keeps track of the simulated times of all component simulators, and manages the global clock of the target system: the global clock is the minimum simulated time of the component simulators. On receiving a request from a processor simulator, the backplane should wait until all processor simulators issue requests to check whether it is safe to process the request. This policy guarantees the causality. But this base synchronization policy hinders parallel execution of simulators unfortunately.

Fig. 2 shows a scenario of simulation progress with two component simulators $P_0$ and $P_1$. The simulation backplane is omitted for simple illustration in the figure. In the ideal case, both simulators should run concurrently as shown in Fig. 2(a). In reality, however, the progress of simulation will be as illustrated in Fig. 2(b); it is serialized after the initial computation period. $P_0$ should wait after it sends a memory access event to the backplane at time 25 until it is signaled the completion of the memory access from the simulation backplane. Then, the backplane waits until it receives an event from $P_1$ at time instance 40, at which it signals $P_0$ to continue. $P_1$ should now wait in turn until it is signaled to proceed from the backplane that again waits for an event from $P_0$, making the execution of two processor simulators serialized.

If a processor does not generate any requests for shared resource access, the simulation backplane cannot progress its global clock until the processor finishes its execution. To solve this problem, we make a processor simulator notify the current simulated time of the processor to the simulation backplane periodically. We call this notification "periodic update." The

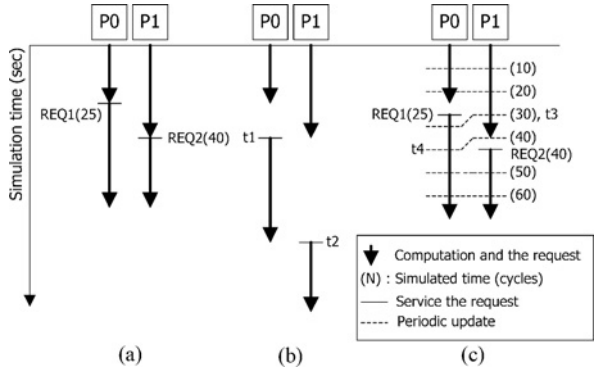



Fig. 2. Simulation behaviors according to three approaches. (a) Ideal case, parallel execution. (b) Without periodic update. (c) With periodic update.

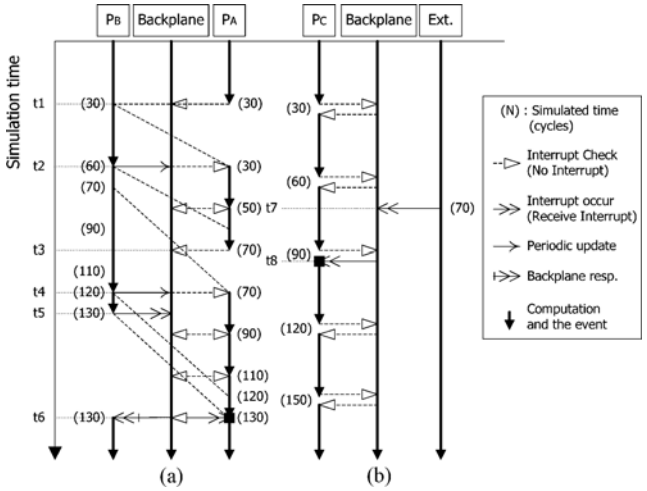



Fig. 3. (a) Interrupt interrupt. (b) External interrupt.

periodic update can be accomplished by asynchronous communication. The role of the periodic update resembles that of *null* messages in PDES to help other simulators proceed.

By periodically notifying the time advance of each processor simulator to the simulation backplane, the backplane can start processing the requests without waiting requests from all simulators as shown in Fig. 2(c). In this example, we assume that the update period of all processor simulators is set to ten cycles identically. At the beginning, $P_0$ and $P_1$ proceed in parallel, and send the update messages to backplane to report their advance to simulated time 10 and 20 without ceasing execution. When the simulated time of $P_0$ reaches 25, $P_0$ issues a shared memory access and waits for the response. The backplane waits until $P_1$ sends another notice at time 30. This notice implies that $P_1$ is now ahead of $P_0$ and so enables the backplane to serve the request of $P_0$ earlier than the base scheme of Fig. 2(b). Similarly when $P_1$ issues a request at time 40, it waits until $P_0$ sends a periodic update at its simulated time 40. In this way, the periodic update efficiently exploits parallel execution of two simulators. For each request, waiting time is as large as the update period in the worst case, while it is the maximum interval between two subsequent requests in the base approach.

Note that the period of notification has significant effects on the simulation performance. If we ignore the communication overhead of periodic update, the proposed technique will converge to the ideal case as we shorten the update period. In reality, however, it degenerates to a lock-step approach that is very costly if we send the periodic update every cycle. It means that there is a tradeoff between parallel execution and communication overhead depending on the update period. Therefore, we perform in-depth analysis on this tradeoff to determine an optimal update period in Section IV.

### B. Imprecise Interrupt Modeling

Since interrupt is an important mechanism of interprocessor communication and I/O event from an external source, it is essential to support the interrupt to be a useful simulator. In this section, we explain how the proposed simulator serves interrupts.

The proposed technique of interrupt handling is based on the observation that the accuracy of interrupt service can be sacrificed if the error does not affect the application behavior. When to serve an interrupt is highly nondeterministic. For instance, if the operating system operates in a supervisory mode, interrupts at the user level are usually disabled or suspended for nondeterministic amount of time. As another example, if an interrupt is caused by external sources such as I/O devices or other external environments, the arrival itself is nondeterministic. As a result, we recognize that we do not need to model the interrupt arrival at the accuracy of instruction cycles. Instead, we increase time granularity of interrupt reception to accelerate the simulation in the proposed technique in order to propose a proper tradeoff between accuracy and performance. The proposed scheme is called "imprecise interrupt modeling."

The time resolution of interrupt reception determines the frequency of synchronization between processor simulators and the backplane. In the proposed scheme, we synchronize the processor simulators periodically to support interrupts, which is the critical factor of tradeoff between accuracy and performance. The shorter period makes the simulation more accurate but slower. This synchronization is a synchronous communication. In other words, the synchronization for interrupt pauses the component simulator until the request is serviced by the backplane like the shared resource access. The desirable period varies according to the purpose and type of interrupt. We distinguish two different kinds of interrupts, internal interrupt and external interrupt, according to the required simulation accuracy and whether the start and end of interrupt are known *a priori*.

1) *Internal Interrupt:* When a processor requests a computation service to another processing component like a library call, the processor expects to receive a response sometime in the future. Not to waste the polling overhead, we may resort to interrupt to know when the response arrives at the processor. Since an interrupt is caused by the request of the client processor, it is referred to as an internal interrupt. In this case, the interrupt handling requires high accuracy and its activated duration is known to the processor.

Fig. 3(a) shows an illustrative example of an internal interrupt. In the example, we assume that the period of the periodic

update is 60 cycles for both two processor simulators $P_A$ and $P_B$, and $P_A$ checks internal interrupts every 20 cycles after $P_A$ requests a service to $P_B$ at simulated time 30. Since the interrupt check is a synchronous communication, the backplane and $P_A$ wait for a message from $P_B$. It sends a periodic update to the backplane when $P_B$ reaches its simulated time 60. Then the backplane notifies $P_A$ that no relevant interrupt has arrived until time 30. Similarly, the backplane handles the next interrupt check request of $P_A$ at time 50 with no success. When $P_A$ performs next synchronization for interrupt at time 70, the backplane is delayed again until the periodic update of $P_B$ arrives at time 120. Afterwards, $P_B$ finishes computation and generates an interrupt to signal the completion at time 130. Then $P_B$ is suspended until $P_A$ catches up the simulated time to 130. Finally when simulation time becomes $t_6$, the interrupt is delivered to $P_A$ and interrupt check is deactivated.

2) *External Interrupt:* Another use case of interrupt is that the system waits an asynchronous interrupt from an external device such as the timer interrupt for task scheduling of OS. Thus the time resolution can be usually as large as milliseconds. Even though a processor does not predict the time of interrupt occurrence unlike the internal interrupt, a jitter due to late detection of such an event does seldom harm the simulation accuracy. Taking this observation into account, the proposed technique performs synchronous communication with a given global period that is the latency requirement as shown in Fig. 3(b). In this example, the synchronization period for the external interrupt is 30 cycles. When the external interrupt is asserted at time 70, the interrupt is queued in the simulation backplane until the corresponding processor simulator $P_C$ detects it at the next synchronization point, time 90. It incurs a jitter or accuracy error of 20 cycles on sensing the interrupt. On average, an interrupt detection jitter is half the interrupt synchronization period.

## IV. Performance Analysis of the Proposed Technique

In this section, we estimate the simulation time of the proposed technique analytically based on the models of application behavior and architecture parameters. This analysis reveals how the periodic update and the shared resource access affect the simulation performance. Thus, we can obtain the optimal period of the periodic update from this analysis. As discussed earlier, there are three kinds of interactions between processor simulators and the simulation backplane, which are periodic update, shared resource access, and interrupt check. We aim at analyzing their interactions to quantify the exploited parallelism and synchronization overhead. We first formulate the estimated simulation time considering the communication dependencies and frequencies of the target application.

We denote by $t_{\mathrm{ISS}_i}$ the simulation time that a processor simulator $PS_i$ takes to progress a unit simulated time (in this paper, it is a single bus cycle); $t_{\mathrm{BP}}$ corresponds to the simulation time of the backplane. To consider the effect of periodic update, we define $n_i$ as the update period for a processor simulator $PS_i$ in simulated time. For IPC, a buffer is commonly used to improve communication efficiency. Hence

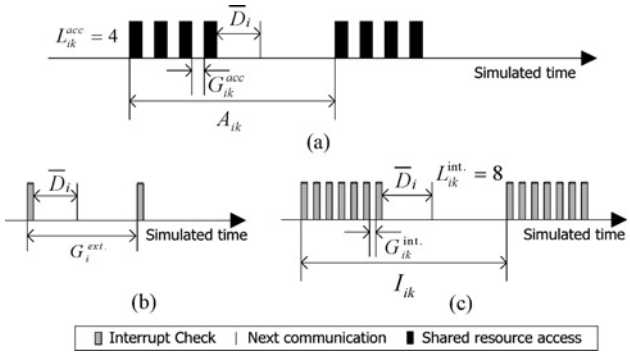



Fig. 4. Three kinds of synchronous communications. (a) Burst shared memory access. (b) External interrupt. (c) Internal interrupt.

we take into account the effect of the buffer assuming that it takes $t_{\mathrm{send}}$ to store the message into the IPC buffer. On the other hand, the associated wrapper in the backplane receives the message and advances the simulated time on the backplane side. It is assumed to take $t_{\mathrm{recv}}$ to fetch the message from the buffer. Therefore, the communication delay per message becomes a sum of $t_{\mathrm{send}}$ and $t_{\mathrm{recv}}$, represented as $t_{\mathrm{comm}_i}$ for $PS_i$

Unlike the update period, an interval between synchronous communication requests varies at runtime. For $PS_i$, we denote a random variable associated with the interval of the consecutive synchronous communications by $S_i$ and its average by $\overline{S}_i$, respectively. That is

$$\overline{S}_i = \sum_{\forall j} P_i(S_i = j) \cdot j \tag{1}$$

where $P_i(S_i = j)$ is the probability that the interval is $j$.

In order to obtain $P_i(\cdot)$, we model each type of synchronous communications. We first describe the model of shared memory accesses. Fig. 4(a) shows a shared memory access pattern for a processor simulator $PS_i$, where a black rectangle corresponds to a single burst transaction and four synchronous communications constitute the memory access pattern. $\overline{D_i}$ is the average simulated time to the next communication, which will be explained later in this section. A burst transaction consists of a series of bus accesses once granted a bus. Shared memory access is represented as a group of dense burst transactions. There may be more than one memory access patterns for $PS_i$, which is defined as a set $AP_i = \{ap_{ik}\}$. To model each of memory access patterns $ap_{ik}$ for $PS_i$, we define two random variables $L_{ik}^{\mathrm{acc}}$ and $G_{ik}^{\mathrm{acc}}$ that are associated with the number of burst transactions and their gap in a transaction group. We also define a random variable $A_{ik}$ to model the interval with which shared memory access pattern $ap_{ik}$ is repeatedly performed.

To obtain the statistics information to define memory access patterns, we perform initial functional simulation of a target application assuming ideal communication architecture. Thus memory access latency is identical to the length of a burst transfer. Then, based on the collected memory traces, we define memory access patterns by analyzing the characteristics of memory access behavior, compromising the accuracy and the complexity. The memory access patterns approximate the actual memory access behaviors. For example, if intensive

reads and writes from/to the shared memory appear at the beginning and end of task execution, respectively, two memory access patterns may be defined. Note that we do not need to repeat the profiling of a task as long as the mapping of task to a processor does not change.

Next, for external interrupts, we use $G_i^{\text{ext}}$ to denote the period of checking interrupts for $PS_i$ as shown in Fig. 4(b). Unlike the external interrupt, internal interrupts are modeled similarly to the case of shared memory access. A random variable associated with the interval of interrupt activation for a processor simulator $PS_i$ is given as $I_{ik}$ as depicted in Fig. 4(c). We denote the number of interrupt checks and their interval for an interrupt pattern $ip_{ik} \in IP_i$ by $L_{ik}^{\text{int}}$ and $G_{ik}^{\text{int}}$, respectively.

The model parameters associated with an external interrupt are decided manually considering the desired accuracy of external interrupt handling. If a designer wants to keep the jitter of the external interrupt below 10 000 cycles, then period $G_{ik}^{\text{int}}$ should be set to 10 000. For internal interrupt checking, we consider both the profile result of the application and the timing requirement: $I_{ik}$ and $G_{ik}^{\text{int}}$ come from the profile result while $G_{ik}^{\text{int}}$ is determined by the required timing accuracy.

The frequencies of the three access types are $\frac{L_{ik}^{\text{acc}}}{A_{ik}}$, $\frac{1}{G_i^{\text{ext}}}$, and $\frac{L_{ik}^{\text{int}}}{I_{ik}}$, respectively. So the total access frequency $\overline{F_i}$ becomes

$$\overline{F_i} = \sum_{\forall ap_{ik} \in AP_i} \frac{L_{ik}^{\text{acc}}}{A_{ik}} + \frac{1}{G_i^{\text{ext}}} + \sum_{\forall ip_{ik} \in IP_i} \frac{L_{ik}^{\text{acc}}}{I_{ik}}. \tag{2}$$

The average access period $\overline{S_i}$ is the inverse number of the access frequency $\overline{F_i}$. Thus

$$\overline{S_i} = \frac{1}{\overline{F_i}}. \tag{3}$$

To formulate the simulation time to advance a unit simulated time, delay associated with each synchronous communication should be estimated. We consider the effects of asynchronous and synchronous communication separately as illustrated in Fig. 5(a) and (b) with two processor simulators ($PS_0$ and $PS_1$). In the figure, we assume that the processor simulators send update messages every ten cycles and the starting time of $PS_1$ is 5.

In Fig. 5(a), when $PS_0$ sends a notice (step 1), the backplane has to schedule the associated wrapper module $W_0$ (step 2). $W_0$ receives the notice and advances itself from 0 to 10 cycles using "*wait*()" function in SystemC (step 3). Now, the backplane recognizes $W_1$ as the slowest module and schedules it (step 4). Then $W_1$ receives the notice that $PS_1$ sent (step 5), advancing itself from 5 to 15 cycles. The backplane schedules the slowest module $W_0$ (step 6). Afterwards, $PS_0$ sends another periodic update message at 20, delivered to the $W_0$ (step 7). As demonstrated in this example scenario, the simulation time of the backplane corresponding to a single periodic update can be written as $n_{\text{avg}} \cdot t_{\text{BP}} + M \cdot t_{\text{recv}}$ where $\eta_{\text{avg}}$ is the average of the update periods of participating $M$ processor simulators.

The amount of simulation time required to progress $PS_i$ by $n_i$ cycles becomes $n_i \cdot t_{\text{ISS}_i} + t_{\text{send}}$. Since the backplane and a processor simulator can be executed concurrently, the overall simulation time is determined by the slowest component between processor simulators and the backplane. Then, the

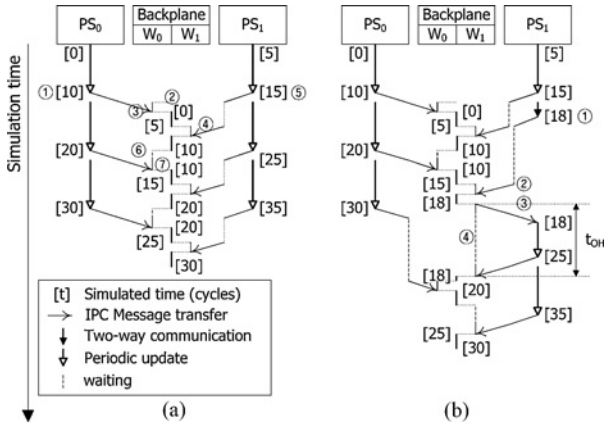



Fig. 5. Simulation time with two processor simulators for two kinds of communication. (a) Periodic update. (b) Other synchronous communications.

average time to simulate a unit simulated time, denoted by $T_{\text{update}}$, is approximated as the following equation:

$$T_{\text{update}} \cong \max\left\{ \max_{\forall i}\left( t_{\text{ISS}_i} + \frac{t_{\text{send}}}{n_i} \right), t_{\text{BP}} + \frac{M \cdot t_{\text{recv}}}{n_{\text{avg}}} \right\}. \tag{4}$$

Next, we consider synchronous communications, shared memory access and interrupt check. Since all communications are serialized in simulation time by the simulation backplane, we examine the behavior of the simulation backplane. In the example of Fig. 5(b), $PS_1$ issues a shared memory access in time 18 (step 1). The requested access is then processed by the backplane after ensuring that the simulated times of all other processor simulators are not behind $PS_1$ (step 2) and the backplane sends the response to $PS_1$ (step 3). In the figure, it is assumed that the service time of communication request is zero for the sake of simple illustration. After sending the response at step 3, $PS_1$ is considered as the slowest component during step 4 and, then, the simulation in the backplane is blocked until $PS_1$ sends the next message. We define $T_{OH_i}$ as such blocking time of the simulation backplane owing to the slowest processor simulator $PS_i$, $PS_1$ in the example. We distinguish three types of synchronous communications by defining $T_{OH_i}^{\text{sh}}$, $T_{OH_i}^{\text{ext}}$, and $T_{OH_i}^{\text{int}}$ as the expected blocking time by shared memory access, external interrupt, and internal interrupt, respectively. Accordingly, $T_{OH_i}$ becomes

$$T_{OH_i} = T_{OH_i}^{\text{sh}} + T_{OH_i}^{\text{ext}} + T_{OH_i}^{\text{int}}. \tag{5}$$

First, we calculate $T_{OH_i}^{\text{sh}}$ as follows:

$$T_{OH_i}^{\text{sh}} = \sum_{\forall ap_{ik} \in AP_{ik}} \frac{d_{ik} \cdot t_{\text{ISS}_i} + L_{ik}^{\text{acc}} \cdot 2t_{\text{comm}_i}}{A_{ik}} \tag{6}$$

where $d_{ik}$ is the amount of simulated time consumed by $PS_i$ while the simulation backplane is blocked. Therefore, the numerator on the right-hand side of (6) corresponds to the blocking interval of the backplane, which is labeled as step 4 in Fig. 5(b). Referring the shared memory access model in

Fig. 4(a), the above equation is rewritten as

$$T_{OH_i}^{\text{sh}} = \sum_{\forall ap_{ik} \in AP_{ik}} \frac{\left\{\left(L_{ik}^{\text{acc}} - 1\right) \cdot G_i^{\text{acc}} + \overline{D_i}\right\} \cdot t_{\text{ISS}_i} + L_{ik}^{\text{acc}} \cdot 2t_{\text{comm}_i}}{A_{ik}}. \tag{7}$$

As defined in Fig. 4(a), $\overline{D_i}$ is the elapsed simulated time from the end of shared memory access to the next communication. If the next communication is a periodic update request, it is bounded by $n_i$. Otherwise, it becomes the average access period $\overline{S_i}$ of (3). Therefore, we determine $\overline{D_i}$ by taking the minimum of these two

$$\overline{D_i} = \min(\overline{S_i}, n_i). \tag{8}$$

In the same way, we can formulate $T_{OH_i}^{\text{ext}}$, and $T_{OH_i}^{\text{int}}$ as

$$T_{OH_i}^{\text{ext}} = \frac{\overline{D_i} \cdot t_{\text{ISS}_i} + 2t_{\text{comm}_i}}{G_i^{\text{ext}}} \tag{9}$$

and

$$T_{OH_i}^{\text{int}} = \sum_{\forall ip_{ik} \in IP_{ik}} \frac{\left\{\left(L_{ik}^{\text{int}} - 1\right) \cdot G_i^{\text{int}} + \overline{D_i}\right\} \cdot t_{\text{ISS}_i} + L_{ik}^{\text{int}} \cdot 2t_{\text{comm}_i}}{I_{ik}}. \tag{10}$$

Finally, let us define $T_S$ as the simulation time required for the backplane to progress by a unit simulated time. Then, by adding (4) and (5), it becomes

$$T_s = T_{\text{update}} + \sum_{\forall i} T_{OH_i}. \tag{11}$$

The estimated performance is simply a reciprocal of $T_s$.

## V. Experiments

In this section, we first explain the simulation environment and methodology to validate the proposed parallel simulation and its performance estimation technique. Then, experimental results will be discussed in the following order; scalability of the proposed parallel simulator, accuracy of the performance analysis method, and its application to some real-life examples.

### A. Simulation Environment and Methodology

As a target system, we assume a multicore system that consists of ARM926EJ-S processor cores because real view development suit (RVDS) is readily available as an ISS to simulate the processor [14]. The extension interface provided by RVDS is used to realize the communication between the ISS and the backplane. Each processor has a local memory and its own interrupt controller. The backplane is implemented based on SystemC 2.1 [13]. In the current implementation, we assume that processors communicate through a global memory that is connected to a global shared bus while other communication architectures can be modeled in SystemC. The simulation host consists of three workstations, each of which has two 3.0 GHz quad-core processors (thus eight cores in total) and 4 GB main memory, and runs Linux of the kernel version 2.6. The workstations are connected through 100M-bit LAN. Socket is used for interworkstation communication while pipe is used for intra-workstation communication.

TABLE I
Model Parameters for Shared Memory Access in the Applications

| Application | Default | Prime Num. | JPEG | MPEG | Synthetic |
|---|---|---|---|---|---|
| # of ISSs | 4 | 4 to 14 | 3 | 6 | 3, 5 |
| # of access patterns | 1 | 4 to 14 | 3 | 12 | 12, 20 |
| Period (cycles) (min, avg, max) | 100 | (2 360 000, 2 360 000, 2 360 000) | (3000, 3293, 3442) | (18, 21, 24) | (1000, 87 793, 900 000) |
| Length (words) (min, avg, max) | 32 | (1, 1, 1) | (66, 99, 133) | (1, 1, 1) | (1, 7, 30) |
| Gap (cycles) | 10 | N/A* | (8, 8, 8) | N/A* | (4, 6, 16) |

*A shared memory access consists of a single burst transfer.

The real-life applications used in the experiments are a prime number test, a JPEG encoder, and the MPEG decoder of ALPBench [15]. In the prime number test, the range of integer numbers is partitioned into the same size of sub-ranges. Initially, each processor simulator is allocated a partition and finds the prime numbers within the partition. At the end of computation it stores the results to the shared memory and gets another partition again. The arbitrary number of participating processors can be chosen according to the number of partitions. The JPEG encoder is divided into three sub-tasks that are mapped to different processors and executed in a pipelined fashion. A subtask processes a macro block (8×8 pixels) for each invocation. So the pipelined execution of three subtasks is iterated many times to process a single image frame. This application incurs frequent communication between subtasks with a unit of macro block. In the MPEG decoder application, a master processor splits an encoded input stream in a unit of slice of an image frame. Then each processor decodes the allocated slice and writes the results back to the shared memory space. Similarly to the prime number test, we may use any number of processors by increasing the number of slices.

Besides we use synthetic applications to model various application characteristics. The simulation performance is heavily dependent on the various parameters such as inter/intra-communication time, the update period, and so on. Therefore, we examine the sensitivity of simulation performance according to the simulation parameters with the synthetic applications. Through the sensitivity test, we validate the accuracy of the proposed analysis technique by comparing the analytical results with the simulated results. Table I summarizes the model parameters corresponding to shared memory access for the applications used in our experiments.

### B. Scalability of the Proposed Simulation Technique

We examine the scalability of the proposed simulation technique according to the number of processor cores in the simulation host with two examples, the prime number test and MPEG decoder. Fig. 6 shows the simulation performance varying the number of processor simulators. Note that the simulation performance measured in kilo-simulated cycles per second (KCPS) indicates the simulation speed to advance a unit simulated time of the entire target system. The horizontal axis indicates the number of processors in the target system. Since each ISS runs on a separate processor core in the simulation host, it also represents the degree of parallelism.

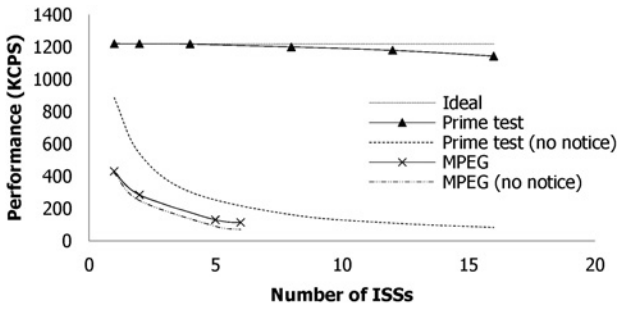



Fig. 6. Scalability of the proposed parallel simulation.

In the ideal case, the simulation performance should remain constant independently of the number of processor cores.

In the prime number test, if we do not perform periodic update ("no notice" in the figure), the simulation performance decreases inverse proportionally to the number of ISSs because the ISSs are serialized. By using periodic update, however, we can parallelize the time advance of the simulation backplane and the processor simulators. Thus, the simulation performance approaches to the ideal case as can be observed in the figure since the prime number test application involves infrequent communication. On the other hand, the MPEG decoder example suffers from heavy communication overhead. It requires frequent communication, once per about every 16 bus cycles. Even though it is still better to use periodic update than not, the performance improvement is not as outstanding as the prime number test, meaning that the simulation performance heavily depends on the application behavior. Thus, it is necessary to estimate the simulation performance considering the application characteristics, which is the issue addressed in the next experiment. This experiment proves that the proposed technique shows scalable performance for the applications where the processors incur infrequent communication.

### C. Accuracy of the Proposed Performance Estimation

1) *Sensitivity Analysis Based on Random Memory Access Generation:* To validate the accuracy of the proposed estimation technique, we compare the estimated performance from the proposed analysis with the simulation result. First, we perform a variety of sensitivity analysis by varying each factor that affects the simulation performance. We consider three parameters associated with the simulation environment, which are the update period, the number of processor simulators, and IPC overhead. In addition, since the memory access pattern of a target application affects the simulation performance, we also consider the frequency of shared memory accesses.

Since it is not easy to find real applications that show the intended memory access behaviors in the sensitivity analysis, we use a pattern generator instead of a processor simulator. The pattern generator requests shared memory accesses to the simulation backplane following a scenario of predefined access patterns. At the same time it also generates notification messages with a given notification period. To define a representative and realistic memory access pattern, the default values are determined considering the profiling results of the aforementioned real-life examples, JPEG, prime number test, and MPEG. We show the default parameter values in the second column of Table I.

TABLE II
MEASURED COMMUNICATION OVERHEAD

| Communication Method | Parameters | | |
|---|---|---|---|
| | $t_{ISS}$ ($\mu$s) | $t_{send}$ ($\mu$s) | $t_{comm}$ ($\mu$s) |
| Pipe | 0.86 | 1.00 | 10.0 |
| Socket | | 2.50 | 230 |

We measured the following parameter values and set them as constant during simulation: $t_{ISS}$, $t_{send}$, and $t_{comm}$, which are shown in Table II according to the communication method. To measure $t_{ISS}$, we ran several real-life applications and then took the average of their $t_{ISS}$. $t_{send}$ and $t_{comm}$ are measured with a simple benchmark program that sends and receives a packet repetitively through pipe and socket. Afterwards, we perform the sensitivity analysis based on two different communication methods, pipe and socket.

As shown in Fig. 7(a), the simulation performance is decreasing as more processor simulators are involved because of the increase of communication and synchronization overhead. The performance varies depending on the notification period as shown in Fig. 7(b); the simulation performance appears to increase as the notification period becomes longer due to the reduction of the IPC overhead. However, the further growth of the notification period causes the loss of parallelism between processor simulators to degrade the performance. As the period increases, the simulation performance approaches to the case of no-notification, meaning the serialized simulation performance. Also, it is obvious that the wider period of shared memory accesses increases simulation performance as shown in Fig. 7(c) since the amount of IPC decreases accordingly.

In all experiments of Fig. 7, the estimated performance matches well with the simulated performance with at most 10% error, which proves that the effect of each factor in the proposed performance analysis is accurately modeled.

Now we assign a different memory access behavior to each processor to make a realistic model. Table III shows the memory access patterns assigned to three processor simulators. We assign multiple memory access patterns to each processor simulator. The parameters for each of access patterns are randomly given. Also we consider both pipe and socket-based IPC. The period of update however is set to identically for all processor simulators. In this way, we synthesize three examples. Since we use a separate processor for the simulation backplane, the simulation host consists of four processors. As shown in the table, the error of estimated performance grows as the number of patterns increases and the IPC overhead is bigger (using sockets).

For a given application and a simulation host, the simulation performance depends on the update period. In the previous sensitivity analysis, we set the notification period of all processor simulators identical. Now we consider seven examples that randomly assign different notification periods to four processors, and examine how the estimation accuracy is affected by the notification period. The ratio of update periods between processor simulators is made up to almost 1000 to make

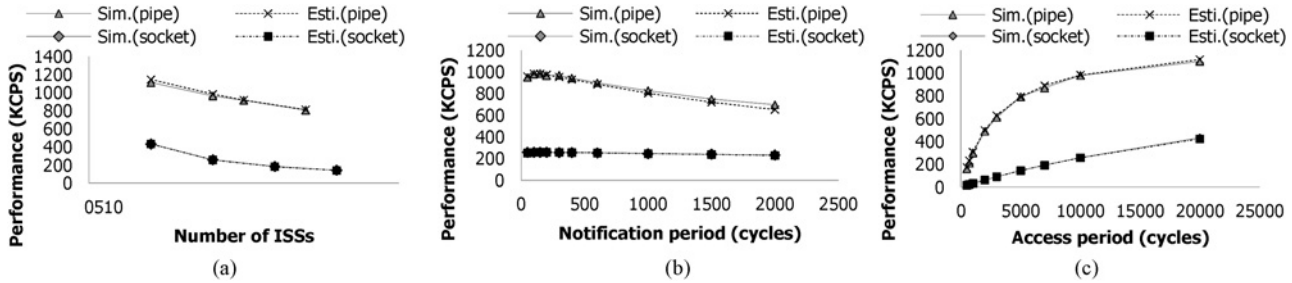



Fig. 7. Sensitivity analysis according to interprocessor communication methods varying (a) number of processor simulators, (b) period of notification, and (c) average period of shared accesses.

TABLE III
ACCURACY OF THE PERFORMANCE ESTIMATION WITH COMPLICATED MEMORY ACCESS PATTERNS

| Example | Case 0 | Case 1 | Case 2 |
|---|---|---|---|
| Number of memory access patterns | 12 | 20 | 20 |
| Communication method | PIPE | PIPE | Socket |
| Estimation (KCPS) | 160 | 640 | 67 |
| Simulation (KCPS) | 161 | 662 | 74 |
| Estimation error (%) | 0.6 | 3.4 | 9.5 |

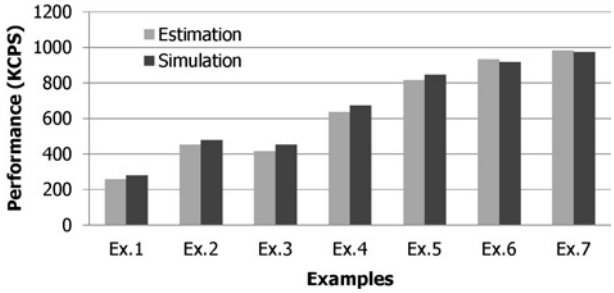



Fig. 8. Estimated simulation performance with different notification periods given to processor simulators.

unfavorable simulation configuration. The results depicted in Fig. 8 shows that the proposed estimation technique robustly predicts the simulation performance with high accuracy even when processor simulators have different notification periods. The estimation error is about 5% on average and is below 8% in the worst case.

2) *Validation Using Real Applications:* In this experiment, we apply the proposed estimation technique to the real-life applications, a prime number test, a JPEG encoder, and the MPEG decoder of ALPBench.

Fig. 9 shows the estimated performance and the actual simulation performance for the prime number test, varying the number of processor simulators to 4, 10, and 14, respectively. When we use 4 processor simulators, they are mapped to the same simulation host. In case of 10 processor simulators, they are split into two hosts identically. We use one more host for additional 4 processor simulators when 14 processor simulators are used. From the graph, our observation is that the simulation performance is kept over 1200 KCPS regardless of the number of processor simulators when the notification period is small. It indicates that the proposed simulator is scalable for this application as explained above. As the notification

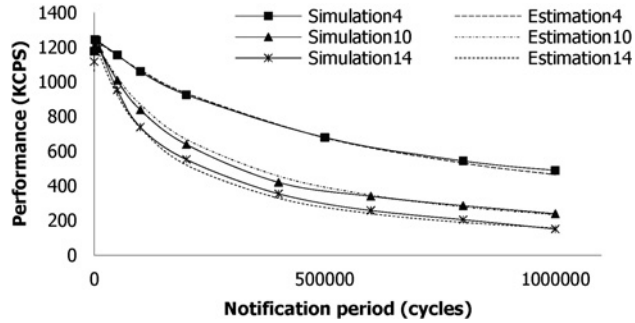



Fig. 9. Simulation performance of the prime number test varying notification period.

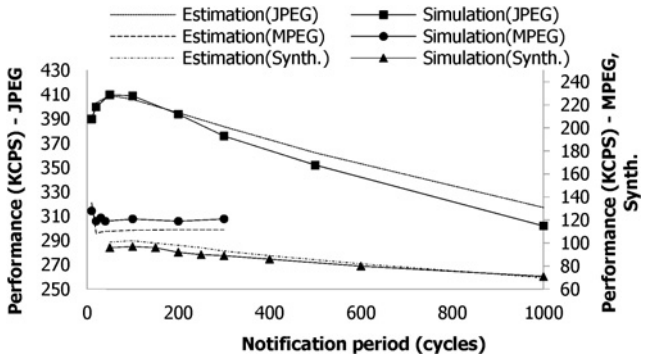



Fig. 10. Simulation performance of JPEG decoder, MPEG decoder, and a large synthetic application varying notification period.

period increases, the simulation performance continues to drop as we use more simulators since the degree of parallelism is reduced. In all cases, however, the estimation accuracy is small, 8% at most.

Fig. 10 shows simulation performance estimation of three other applications by varying the notification period. In this figure, we zoom up the short range of notification period as opposed to Fig. 9. The JPEG decoder has the similar performance trend to the case of the prime number as displayed in Fig. 10. The performance curve forms a convex whose maximum point occurs around the notification period of about 50 in the JPEG decoder whereas the maximum point occurs around 5000 in the prime number test. Such difference is caused by the frequency of the shared memory accesses issued by the processor simulators. As the frequency becomes smaller, the optimal notification period grows larger since it reduces the communication overhead while not sacrificing the parallelism significantly.

In case of MPEG decoder, we use five target processors to decode 704 × 480-sized frames. Each processor is in charge of decoding a slice that consists of 44 × 6 macro blocks, and the size of macro block is 16 × 16 pixels. This application incurs quite frequent communication, which takes place about every 16 bus cycles. Therefore, the notification period does not have significant effect on the simulation performance. The best notification period from our analysis is ten cycles. The estimation error is slightly bigger, 10% in the worst case. This is mainly due to the inaccuracy of the simplified model of the memory access patterns. Recall that our performance estimation technique is based on the average behavior. Therefore as the actual behavior is deviated from the mathematical distribution that we assume, the estimation error may grow. Nonetheless, the tendency of performance variations is predicted accurately even in this case.

Lastly, in order to validate the proposed estimation technique with unfavorable cases, we created a large synthetic example where 16 target processors are given different access patterns. Each target processor has complicated memory access behavior with three to four different patterns. The average access period of each pattern ranges from 1000 to 900 000 cycles randomly and the burst length is set to 4 to 30 per shared memory access. The target processors are partitioned into three simulation hosts. They run 6, 6, and 4 processor simulators, respectively. The estimated simulation performance is quite close to the actual simulation performance as shown in Fig. 10, keeping the estimation error below 5% in the worst case.

In summary, the optimal notification period heavily depends on the characteristics of the application. In particular, the effect of notification period becomes more significant as the application has less communication as in the case of the prime number test.

## VI. Conclusion

In this paper, we proposed a novel parallel simulation technique that shows scalable performance as the number of component simulators increases. Two key techniques, periodic update and imprecise interrupt modeling, enabled parallel execution of component simulators without suffering from the increased synchronization overhead as the number of component simulators increases.

Also, we proposed a novel estimation technique to predict simulation performance with a set of parameters to configure the parallel simulation. The proposed technique formulates the simulation performance as a function of notification period based on the memory access behavior of a target application. Through intensive experiments, we confirmed that the proposed estimation technique is capable of accurately predicting simulation performance with the wide variation of simulation configurations. The estimation error does not exceed 10% in the worst case in all our experiments. Therefore, the proposed estimation technique can be used to configure the parallel simulation environment in an optimal way for a given target application. As future work, we planned to release the assumption that each processor is assigned a single processor simulator. As more processors are used in a target system, we may have to assign multiple simulators to a single host processor. Then we will have a wider choice of simulation host configurations.

## References


[1] H. Kim, D. Yun, and S. Ha, "Scalable and retargetable simulation techniques for multiprocessor systems," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, Oct. 2009, pp. 89–98.

[2] R. M. Fujimoto, "Parallel and distributed simulation systems," in *Proc. Winter Simul. Conf.*, Dec. 2001, pp. 147–157.

[3] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 4, pp. 198–205, Apr. 1981.

[4] D. R. Jefferson, "Virtual time," *ACM Trans. Programming Languages Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[5] J. Jung, S. Yoo, and K. Choi, "Performance improvement of multiprocessor systems cosimulation based on SW analysis," in *Proc. Des. Autom. Test Eur.*, Mar. 2001, pp. 749–753.

[6] M. Chung and C. Kyung, "Enhancing performance of HW/SW cosimulation and coemulation by reducing communication overhead," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 125–136, Feb. 2006.

[7] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. Hill, J. Larus, and D. Wood, "Fast and portable parallel architecture simulators: Wisconsin wind tunnel II," *IEEE Concurrency*, vol. 8, no. 4, pp. 12–20, Oct.–Dec. 2000.

[8] W. Sung and S. Ha, "Efficient and flexible cosimulation environment for DSP applications," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E81-A, no. 12, pp. 2605–2611, Dec. 1998.

[9] F. Fummi, M. Loghi, S. Martini, M. Monguzzi, G. Perbellini, and M. Poncino, "Virtual hardware prototyping through timed hardware-software cosimulation," in *Proc. Des. Autom. Test Eur.*, Mar. 2005, pp. 798–803.

[10] S. Yoo and K. Choi, "Optimistic distributed timed cosimulation based on thread simulation model," in *Proc. Int. Workshop Hardw./Softw. Codesign*, Mar. 1998, pp. 71–75.

[11] D. Kim, C. Rhee, and S. Ha, "Combined data-driven and event-driven scheduling technique for fast distributed cosimulation," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 10, no. 5, pp. 672–678, Oct. 2002.

[12] Y. Yi, D. Kim, and S. Ha, "Fast and accurate cosimulation of MPSoC using trace-driven virtual synchronization," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 12, pp. 2186–2200, Dec. 2007.

[13] *Open SystemC Initiative (OSCI), SystemC 2.2*. (2006) [Online]. Available: http://www.systemc.org/home

[14] ARM Ltd. (2005). *RealView ARMulator* [Online]. Available: http://www.arm.com/products/DevTools/RealViewDevSuite.html

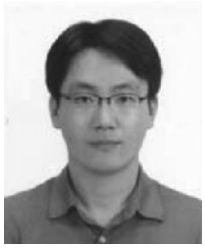
[15] M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes, "The ALPBench benchmark suite for complex multimedia applications," in *Proc. Int. Symp. Workload Characterization*, Oct. 2005, pp. 34–45.

[16] A. Falcon, P. Faraboschi, and D. Ottega, "An adaptive synchronization technique for parallel simulation of networked clusters," in *Proc. Int. Symp. Performance Anal. Syst. Softw.*, Apr. 2008, pp. 22–31.

[17] D. Burger and D. Wood, "Accuracy versus performance in parallel simulation of interconnection networks," in *Proc. Int. Symp. Parallel Process.*, Apr. 1995, pp. 22–31.


**Dukyoung Yun** received the B.S. degree in computer science and engineering from Seoul National University, Seoul, Korea, in 2004. He is currently working toward the Ph.D. degree in electrical engineering and computer science with the School of Electrical Engineering and Computer Science, Seoul National University.

His current research interests include the system level design and the parallel simulation for multiprocessor embedded systems.



**Sungchan Kim** (M'10) received the B.S. degree in material science and engineering, the M.S. degree in computer engineering, and the Ph.D. degree in electrical engineering and computer science from Seoul National University, Seoul, Korea, in 1998, 2000, and 2005, respectively.

He is currently an Assistant Professor with the Division of Computer Science and Engineering, Chonbuk National University, Jeonbuk, Korea. His current research interests include the hardware/software codesign and the system level design of multiprocessor system-on-chip.

**Soonhoi Ha** (S'87–M'94–SM'07) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, Korea, in 1985 and 1987, respectively, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1992.

He is currently a Professor with the School of Electrical Engineering and Computer Science, Seoul National University. His primary research interests include various aspects of embedded system design, including HW/SW codesign, design methodologies, and embedded software design for MPSoC.

Dr. Ha was the Program Co-Chair of CODES+ISSS'2006, ASP-DAC'2008, and ESTIMedia'2005–2006. He has been a member of the technical program committees of several technical conferences, including DATE, CODES+ISSS, and ASP-DAC.