AIR: Application-Level Interference Resilience for PDES on Multicore Systems

JINGJING WANG, Binghamton University
NAEL ABU-GHAZALEH, University of California, Riverside
DMITRY PONOMAREV, Binghamton University

Parallel discrete event simulation (PDES) harnesses parallel processing to improve the performance and capacity of simulation, supporting bigger and more detailed models simulated for more scenarios. The presence of interference from other users can lead to dramatic slowdown in the performance of the simulation. Interference is typically managed using operating system scheduling support (e.g., gang scheduling), a heavyweight approach with some drawbacks. We propose an application-level approach to interference resilience through alternative simulation scheduling and mapping algorithms. More precisely, the most resilient simulators allow dynamic mapping of simulation event execution to processing resources (a work pool model). However, this model has significant scheduling overhead and poor cache locality. Thus, we investigate using application-level interference mitigation where the application detects the presence of interference and reacts by changing the thread task allocation. Specifically, we propose a locality-aware adaptive dynamic mapping (LADM) algorithm that adjusts the number of active threads on the fly by detecting the presence of interference. LADM avoids having the application stall when threads are inactive due to context switching. We investigate different mechanisms for monitoring the level of interference and different approaches for remapping tasks. We show that LADM can substantially reduce the impact of interference while maintaining memory locality.

Categories and Subject Descriptors: I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event, Parallel*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Interference, application adaptation, PDES, proportional slowdown

ACM Reference Format:
Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2015. AIR: Application-level interference resilience for PDES on multicore Systems. ACM Trans. Model. Comput. Simul. 25, 3, Article 19 (April 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2701420

1. INTRODUCTION

Discrete event simulation (DES) is a simulation methodology for systems where changes of state occur at discrete times. It is widely used in a range of application domains such as computer and telecommunication systems, war gaming,

This work is supported by the Air Force Research Laboratory under agreement number FA8750-11-2-0004. This work is also supported by National Science Foundation grants CNS-0916323 and CNS-0958501.
Authors' addresses: J. Wang and D. Ponomarev, Department of Computer Science, Thomas J. Watson School of Engineering and Applied Science, State University of New York at Binghamton, P.O. Box 6000, Binghamton, NY 13902-6000; emails: jwang36@binghamton.edu, dima@cs.binghamton.edu; N. Abu-Ghazaleh, Department of Computer Science and Engineering, 351 Winston Chung Hall, University of California, Riverside, 900 University Ave., Riverside, CA 92521; email: nael@cs.ucr.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 1049-3301/2015/04-ART19 $15.00
DOI: http://dx.doi.org/10.1145/2701420

transportation systems, operational planning, and biological simulations. Parallel discrete event simulation (PDES) [Fujimoto 1990a] harnesses the computational power and resources of parallel computing to improve the performance and capacity of DES, allowing the simulation of larger models, in more detail, and for more scenarios.

Parallel applications are commonly designed under the assumption of a homogeneous environment with no interference from other colocated applications. Interference from other applications, as well as other noise in the system, creates competition for the available resources, leading to slowdowns [Zhuravlev et al. 2010; Tsafrir et al. 2005]. Ideally, under interference, an application experiences a slowdown proportionately to the reduction in its share of the resources: a metric we call *proportional slowdown*. However, the impact of interference can be significantly worse, even when the amount of interference is small. For example, when evaluating a multithreaded PDES engine, we discover that even one external computationally bound thread results in a slowdown of up to 3.9 for an eight-way simulation on the Core i7 platform and up to 2.8 for a 48-way simulation on an AMD Magny-Cours platform.

A primary reason for the high cost of interference is the granularity of the operating system (OS) scheduler. When the OS schedules an interfering process on a core, it has to context switch one of the simulation threads out, making it inactive. As a result, this thread is stalled for an extended period of time related to the OS scheduling quantum. Meanwhile, any dependencies on the stalled thread are delayed, eventually causing other threads to stall, leading to an impact far beyond proportional slowdown.

Although the problem is common to most parallel applications, the impact is especially high for fine-grained applications such as PDES. In the context of PDES, assuming optimistic simulation, when a thread is descheduled, other simulation threads surge forward. Eventually, when the OS schedules the thread again, its late events cause rollbacks throughout the simulation: thus, most of the computation time on all threads is lost, and additional inefficiency results from the overhead of rollbacks. Since the dependency pattern is dense and dynamic, the simulation effectively fails to make progress whenever any of the threads is descheduled. Moreover, the dependency patterns make it difficult to apply traditional approaches to solve the problem such as fine-grained work sharing or work stealing [Blumofe and Leiserson 1999].

Gang scheduling is a standard solution to interference problems in parallel applications [Feitelson and Rudolph 1992]. Gang scheduling eliminates interference by separating applications in time. Threads belonging to an application are co-scheduled such that all threads are active, allowing application progress, or all are inactive, allowing other interfering applications to run. However, gang scheduling is a heavyweight approach that can lead to loss of application performance and system throughput. For example, applications may be able to coexist if some of the threads are I/O bound; they are forcibly separated by gang scheduling, reducing utilization and increasing runtime [Feitelson and Rudolph 1992; Wiseman and Feitelson 2003]. If interference is limited, the machine could be more efficiently shared by partitioning in space (using fewer processors) rather than in time. The bursty operation under gang scheduling may cause contention and challenge real-time applications. Finally, gang scheduling may not be available on many platforms such as networks of workstations [Arpaci et al. 1995] and clouds [Armbrust et al. 2010].

The goal of this article is to develop alternative organizations to PDES simulation that are more resilient to the impact of external interference. In conventional PDES implementations, a simulation model is partitioned across multiple processing elements (PEs), each responsible for executing the events destined to a subset of the simulation objects. Each PE is executed by a process (or thread).

In many existing PDES simulators such as ROSS [Carothers et al. 2000] and WarpIV [Steinman 2008], the mapping between PEs and processes (or threads) is established

at the initialization of simulation and does not change during the simulation (the so-called fixed-mapping, or FM, scheme). FM is an effective strategy in a load-balanced simulation in the absence of external interference, primarily because it promotes locality of memory references and because it incurs little overhead for scheduling [Fujimoto 2000]. However, FM suffers in the presence of interference because a stalled thread remains responsible for simulating its objects, leading to poor performance.

In this article, we explore application-level interference resilience for PDES on multicore platforms. We first propose a dynamic-mapping (DM) scheme that is capable of dynamically changing the mapping between PEs and threads during the simulation. In particular, each thread attempts to work on the next available PE in a round-robin fashion. For correctness, each PE can only be mapped to one thread at a time. As a result, DM has limited opportunities to solve the problem: a thread is often switched out while in the middle of processing events on a PE while holding its lock. Thus, we discovered that the baseline DM cannot effectively solve the interference problem.

We next investigate an adaptive DM scheme that reduces the number of active threads when interference is detected. As a result, the number of threads is again matched to the available hardware contexts, and the simulation does not have to suffer extended periods when one of its threads is switched out. The active threads have to service a number of PEs that are larger than them. Having the threads switch in round-robin fashion among the PEs promotes load-balanced operation but leads to poor locality as PEs move among threads, causing cache interrogation. To promote locality, the locality-aware adaptive DM (LADM) scheme creates a schedule where each thread is primarily associated with one PE but spends a portion of its time helping one or more other PEs whose primary thread has been disabled. The proportion of time is chosen so that the total active time each PE receives remains balanced. Since each thread works on a limited number of PEs (two under reasonable interference conditions), locality is kept high.

LADM has the following key characteristics:

(1) In the absence of external loads, LADM incurs small performance loss (less than 5%) compared with the optimal FM implementation on both the Intel Core i7 and AMD Magny-Cours machines. The loss includes the overhead of detecting interference but also the cost of rate misprediction of interference, a problem we hope to address with more careful design of the detector.
(2) LADM can substantially reduce the impact of interference, thus reducing the gap with proportional slowdown. For example, LADM is able to achieve 2 to 4× improvement in performance in the presence of interference on both a four-core (eight hardware threads) Intel Core i7 and a 48-core AMD Magny-Cours machine.

The remainder of the article is organized as follows. Section 2 provides background information regarding both the PDES simulator and two multicore platforms we used in our experiments. We then define *proportional slowdown* to quantify the PDES performance in the presence of external loads in Section 3. In Section 4, we show the actual impact of external loads on the performance of fixed-mapping PDES simulators. We then explain why the performance of the PDES simulator with FM implementation suffers considerably when external loads interfere with the simulation. In Section 5, we provide a design overview of the baseline DM mechanism. In Section 6, we provide the details of our LADM scheme that can address the limitations in the baseline DM implementation. Section 7 provides the details of the experimental setup and simulation benchmarks. Section 8 presents an experimental evaluation. In Section 9, we review some related work. Finally, in Section 10, we present some concluding remarks.

2. BACKGROUND

In this section, we first review the multithreaded simulator used in this article. We follow by providing an overview of the two multicore platforms used in the experiments: a quad-core Intel Core i7 system and a 48-core AMD Opteron Magny-Cours.

We use a recently developed multithreaded version [Jagtap et al. 2012b; Wang et al. 2014] of the Rensselaer's Optimistic Simulation System (ROSS) [Carothers et al. 2000]. ROSS is a state-of-the-art PDES simulation engine that supports both conservative and optimistic simulations. The multithreaded ROSS (ROSS-MT) encapsulates each group of objects as a PE and assigns each PE to a thread (i.e., it uses fixed mapping). The thread-based implementation allows optimizing communication using fast shared memory operations [Jagtap et al. 2012b; Wang et al. 2014]. Moreover, the simulator was evaluated on and optimized for a number of platforms, including conventional multicores [Jagtap et al. 2012b; Wang et al. 2014], the Tilera Tile64 many-core processor [Jagtap et al. 2012a], and clusters of multicores [Wang et al. 2013].

In PDES, a simulation model is partitioned across multiple PEs. Each PE processes the events in timestamp order. When an event is processed, it may update the state of the simulation object and/or schedule future events. These timestamped events are communicated to the destination PE. The events for each PE are continuously processed within a simulation loop until the simulation time of the PE reaches the simulation completion time. During each iteration, a designated number of events (batch size) can be processed before moving to the next iteration.

Each PE processes events in timestamp order to ensure correct causality of the simulation [Jefferson 1985]. Enforcing causality across multiple PEs requires the implementation of a synchronization protocol. PDES simulators use either conservative or optimistic synchronization [Fujimoto 1990a]. In conservative simulation, a model property called lookahead is used to allow PEs to communicate safe processing distances to other PEs and guarantee correct execution. In contrast, optimistic simulation allows PEs to process events without synchronization, advancing their local simulation time (LVT). Thus, it is possible to receive a remote event with an earlier timestamp than the current simulation time, indicating a causality error; such an event is called a *straggler*. Correct execution requires a rollback to a time earlier than the straggler time, restoring the simulation state, and cancelling any generated events after the checkpoint. ROSS-MT leverages efficient reverse computation [Carothers et al. 1999], instead of the more conventional state saving [Palaniswamy and Wilsey 1993], to restore the simulation state in the case of a rollback. In order to be able to commit events and to reclaim rollback checkpoint information, global virtual time (GVT) is computed periodically to measure the overall progress of the simulation. GVT computation is a form of the classical distributed global checkpoint computation problem [Koo and Toueg 1987].

We use two multicore platforms with significantly different CPU and memory organizations. The first is a quad-core Intel Core i7 system. In this platform, each core has private 32KB L1 and 256KB L2 caches and shares 8MB L3 cache with other cores. With hyperthreading enabled, each core can simultaneously execute two hardware threads that share both L1 and L2 caches. The second architecture we use is an AMD 48-core machine. It consists of four AMD Opteron 12-core chips, connected with hypertransport links. Each chip has two dies, with each die holding six cores. Each core has private 64KB L1 and 512KB L2 caches and shares a 6MB L3 cache with other cores on the same die. In addition, the memory accesses to different memory regions on this platform have nonuniform memory access (NUMA) latencies [Conway et al. 2010].

3. IDEAL SLOWDOWN UNDER INTERFERENCE

Consider a primary application, such as our PDES simulation, running with N_p threads on a multicore platform. Let N_c be the total count of hardware threads such that all

these threads can execute concurrently; hardware threads refer to cores, or hardware contexts in the case of simultaneous multithreaded (SMT) processors. Suppose that one or more external interfering loads can start or terminate at any time during the simulation. Thus, to measure performance more accurately, we divide the simulation into n small intervals $[X_{j-1}, X_j]$ indexed by j. In addition, let $N_{total,j}$ be the total number of software threads executing on the machine (i.e., the number of primary application threads, as well as the number of external loads running concurrently) during the interval j. In typical conditions, the operating system scheduler fairly allocates its CPU resources to each thread (e.g., the Linux Completely Fair Scheduler (CFS) with SCHED_NORMAL scheduling policy [Jones 2009]). In other words, each load obtains $(\frac{N_c}{N_{total,j}})$ of the available CPU time on average during the interval j, assuming that $N_{total,j}$ loads compete for N_c CPUs. Therefore, the expected primary application slowdown under such conditions during the interval j is approximated by

$$S_j = \frac{N_{total,j}}{N_c} = \frac{N_p + N_{e,j}}{N_c}, \tag{1}$$

where $N_{e,j}$ is the number of external loads running concurrently with the primary application during the interval j. Note that this reasoning assumes that threads are computation bound. We call S_j the *proportional slowdown* during the interval j, since S_j increases proportionately to the number of interfering load processes. We assume that $N_{total,j}$ is always greater than or equal to N_c, and the interference from external loads on the primary application performance occurs if $N_{total,j} > N_c$.

The runtime of the entire application in the presence of external loads can be approximated by adding up the expected runtime across all intervals. Let T_j be the execution time required for the interval j of an FM simulation without interference. By multiplying T_j by the corresponding S_j, we obtain $T_j^{'}$, defined as the execution time required for the interval j of the simulation in the presence of external loads. Therefore,

$$T_{ideal} = \sum_{j=1}^{n} T_j^{'} = \sum_{j=1}^{n} T_j \times S_j \tag{2}$$

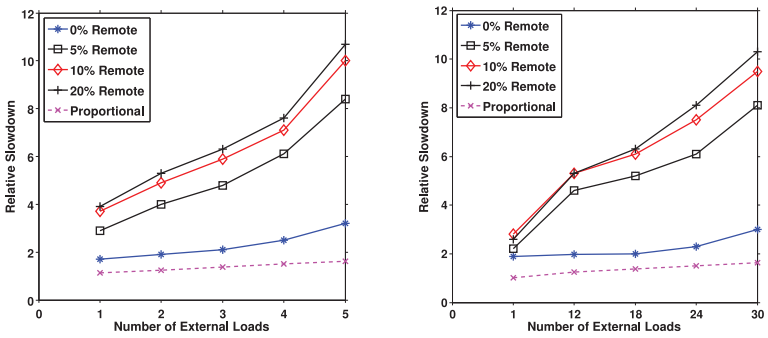
denotes the ideal runtime of the entire simulation in the presence of external loads. T_{ideal} represents a best-case scenario where the presence of interference merely reduces the amount of available resources and results in a slowdown proportional to this reduction. Such a slowdown may be experienced by embarrassingly parallel applications. In practice, the impact is significantly worse than T_{ideal} because of the dependencies between the threads belonging to one application. In the case of optimistic simulation, T_{ideal} also does not take into account the effect of rollbacks to undo the erroneous computation after a thread being interfered with is activated.

4. MEASURED IMPACT OF INTERFERENCE

In this section, we evaluate the slowdown experienced by ROSS-MT and show that it far exceeds proportional slowdown. We also explain and quantify the reasons for the slowdown.

4.1. PDES Slowdown Under Interference

For most of the experiments, we use the Phold simulation model [Fujimoto 1990b], which equally distributes a number of simulation objects among PEs. We use a controllable version of Phold that allows specifying the communication percentage between different objects on different cores. The simulation consists of eight PEs running on the Intel Core i7 platform and 48 PEs running on the AMD 48-core machine, with 1,000 objects per PE. Each PE was also mapped to a different thread: thus, all CPU



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 1. The relative slowdown of ROSS-MT caused by external loads.

Table I. Execution Time of a Four-Way Simulation on a Quad-Core Processor using WarpIV Simulator

	Optimistic	Conservative
No external load	6 sec	10 sec
1 external load takes 50% CPU of a core	12 sec	19 sec
1 external load takes 100% CPU of a core	>4.7 hr	>40 hr

resources were used by ROSS-MT threads in the absence of external loads. In addition, we selected a GVT computation interval of 128 batches on both platforms, with a batch size of 24 events. Although the results are somewhat sensitive to the GVT interval (as a small GVT interval acts as a throttle to the simulation [Tay et al. 1997]), these values are in the range where ROSS-MT is most efficient across a majority of the models.

For these experiments, we use a CPU-intensive process as the external load; the process repeatedly performs computation within a tight loop. Thus, the process when active competes continuously for CPU cycles with the ROSS-MT threads. In the experiments, the external load is started with ROSS-MT and executes for the duration of the simulation. Thus, the *proportional slowdown* factor from one external load can be calculated by Equation (1) to be $\frac{9}{8}$ for the Core i7 and $\frac{49}{48}$ for the AMD Magny-Cours.

Figure 1(a) and Figure 1(b) show the relative slowdown experienced by ROSS-MT as the number of external loads increases on the Intel Core i7 system and AMD Magny-Cours machine, respectively. The relative slowdown is calculated by dividing the execution time of simulation in the presence of interference by the one without interference. As the percentage of remote communication is increased, the dependencies among the different PEs increase. ROSS-MT with 0% remote communication performs close to proportional slowdown: since there are no dependencies between PEs, if a PE is delayed, it does not affect the progress at other PEs and, on average, all threads make progress with their computation. In contrast, the interference from external loads dramatically degrades the performance of ROSS-MT even when a small amount of remote communication exists, far beyond proportional slowdown. For example, even one external load can result in a slowdown of up to 3.9 on the Core i7 and 2.8 on the AMD Magny-Cours, whereas proportional slowdown is 1.125 and 1.02, respectively, for the two machines.

The problem is not specific to ROSS: we were able to demonstrate similar trends, and even worse slowdown, on the WarpIV PDES simulator [Steinman 2008]. Table I shows four-way optimistic and conservative simulations interfered with by one external load

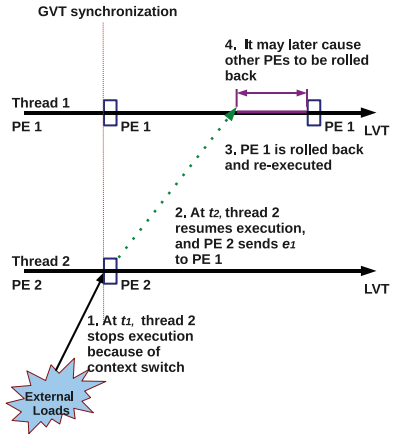


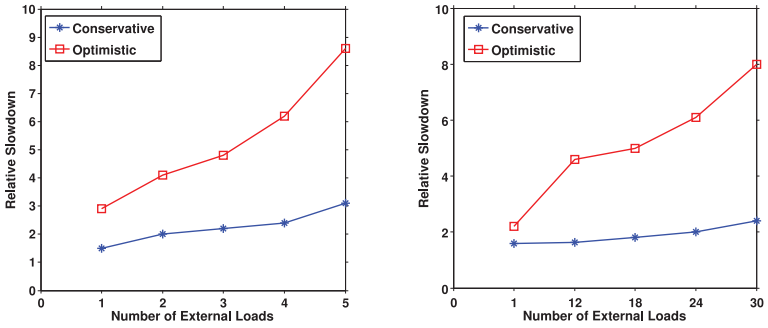
Fig. 2. A rollback caused by interferences from external loads.

on a quad-core processor; due to export control restrictions on WarpIV, we had to run this experiment on a quad-core Xeon machine. Somewhat surprisingly, the simulation almost stops when the external load takes 100% of the time on one core (a situation that occurred sometimes, a decision that the Linux scheduler makes). We believe this situation is due to the fuzzy barrier used in GVT computation in WarpIV [Gupta 1989]. At any given time, one thread is not executing and the fuzzy barrier condition is not met. However, even when the external load gets a lower scheduling priority and shares one of the CPU cores with a PDES process, the WarpIV simulation still experiences a performance slowdown factor of about 2. The situation was the same for both conservative and optimistic simulation.

4.2. Explaining the Impact of Interference

Recall that in ROSS, each thread is assigned a PE consisting of a group of simulation objects. The groups of objects assigned to each thread are selected, often via a partitioning algorithm (e.g., [Bahulkar et al. 2012]), to minimize costly communication and to load balance computation. Each thread is responsible for processing all events whose destination is an object in its PE group. Thus, the mapping of work to threads is fixed.

Consider a two-way simulation of ROSS-MT, with one LP per PE, as seen in Figure 2. PE 1 and PE 2 are executed by thread 1 and thread 2, respectively. Suppose an external load starts and interferes with thread 2 at wall clock time t_1, after a GVT computation phase (which requires barrier synchronization in ROSS). Once the interfering noise process is scheduled, thread 2 is context switched out and stops execution, while thread 1 continues. Thread 2 does not get scheduled again until the noise process exhausts its OS quantum (otherwise, a hardware context becomes available); the OS quantum is typically in the tens of milliseconds, sufficient for thread 1 to execute for several million CPU cycles. At a wall clock time t_2 ($t_2 > t_1$), thread 2 resumes execution, and PE 2 sends an event e_1 to PE 1. Due to the large pause in execution, this event is a straggler as PE 1 has executed far ahead of PE 2, limited only in the ROSS case by the GVT computation interval; in other simulators, the degree of optimism can be unbounded. Upon receiving e_1, PE 1 is rolled back to a simulation time before that of e_1 and then is re-executed. Thus, loss of efficiency occurs for two reasons: (1) inactive PE on the critical path: not only is processing time lost at PE 2 while it is context switched out, but also most of the time that is available to PE 1 is also wasted due to the dependency between the two PEs; and (2) rollback overhead: the straggler causes



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 3. The effect of rollbacks.

other PEs to be rolled back. Thus, the overhead of large rollbacks in terms of state restoration (or reverse computation), sending antimessages, and other data structure restoration exacerbates the inefficiency.

To identify the contributions to the slowdown from these two effects, we show the relative slowdown of both optimistic simulation and conservative simulation on the Core i7 and Magny-Cours platforms (Figure 3). In this experiment, we fixed the remote communication percentage at 5%. In the absence of interference, ROSS-MT performs similarly in both optimistic and conservative modes for the Phold model. Under interference, conservative simulation suffers from delays when the inactive PE slows the other PEs down as they wait for their dependencies to be satisfied (the first effect), but there is no cost for rollbacks. In contrast, optimistic simulation suffers both forms of overhead. Thus, the performance gap between optimistic and conservative simulations is a reasonable estimate of the overhead of rollbacks for this particular model. Clearly, the overhead of rollbacks is substantial and increases with the degree of interference.

5. CAN DYNAMIC MAPPING HELP?

To address the harmful behavior that occurs in the presence of interference, we pursue application-level resilience to interference. Specifically, we consider approaches for detecting the presence of interference and remapping the application to avoid contention for hardware resources. Our first attempt is *dynamic mapping* of threads to PEs. More specifically, in this scheme, we periodically remap the threads to different PEs (recall that each PE encapsulates a group of objects in the simulation). The intuition behind DM is that it allows active threads to rotate across the different PEs, avoiding having a PE lag far behind the others.

Recall that each thread in ROSS-MT executes a loop that repeatedly performs the simulation tasks such as sending and receiving events and event processing. To implement DM, we add a new step at the beginning of the loop where a thread determines which PE to associate itself with; the base implementation simply rotates threads in a round-robin fashion across the PEs. Consider the example as shown in Figure 2. After thread 2 finishes the execution of PE 2 for an iteration, it then switches to PE 1. Thus, in principle, the active thread alternates working on PE 1 and PE 2, reducing the LVT difference between them. Alternative bases for scheduling PEs to threads are possible (e.g., attempting to work on the PE with the lowest LVT).

Note that a side effect of remapping threads to PEs is a loss of data locality: FM permanently maps a hardware thread to a unit of work, and the caches for the core are populated with the data relevant to it. As DM remaps work across cores, the PE data must be brought to each new core (from shared lower-level caches or main memory).

A second, more serious, limitation of DM is its limited opportunity for assisting performance. More precisely, for correctness, two threads cannot be attached to the same PE concurrently, which prevents remapping from being able to assist if the context-switched thread happens to hold the lock on the PE. We implemented efficient synchronization using a condition variable and a spin lock for each PE. More precisely, a PE status is checked (without locking); if the status is busy, the thread moves on to the next PE. If the status is free, the thread acquires the spin lock for the PE and checks if it is still free. If it is, the thread sets the PE to busy and is admitted to work on the PE. Once the iteration is over, it sets the PE status to free and moves on again to the next PE. Thus, DM is limited if the first thread is switched out while in the middle of processing a batch since the PE will be marked as busy until the thread is scheduled again. Since this is the common case, DM cannot effectively solve the problem.

6. LOCALITY-AWARE ADAPTIVE DM

In this section, we propose an LADM scheduler that is capable of addressing limitations of DM. LADM improves DM in the following ways. The first improvement, which we call adaptive DM (ADM), adjusts the number of active threads to match the available hardware contexts: when a noise process is detected, the number of active threads is reduced to avoid competing for a core and the resulting expensive context switches. Thus, only active threads are allowed to execute and the simulation work has to be remapped to them. To support ADM, two main mechanisms are needed: one to reduce the number of active threads upon detecting the interference, and another to check if the interference is no longer there and to reactivate idle threads. The second improvement of LADM is control of the mapping of threads to work to promote high data locality. In particular, LADM uses a locality-aware scheduler to map PEs among active threads. We discuss these mechanisms in the remainder of this section.

6.1. Detecting the Presence of Interference

ADM periodically detects the presence of noise during simulation. The detection period is set to $\frac{T_{gvt}}{4}$ simulation loop iterations in our implementations, where T_{gvt} is the GVT interval. We implemented two different interference detection algorithms. In the first implementation, each active thread periodically monitors its total event processing time. The *average processing time per event* (APTE) of each active thread is calculated by dividing the total event processing time by the corresponding number of processed events. A performance anomaly is detected if the ratio of the maximum APTE to minimum APTE is beyond a defined threshold. There is a tradeoff between the responsiveness and the stability of the noise detector. If the detection threshold is set too high, then the system can become less responsive to the presence of noise. On the other hand, as the threshold is made lower, responsiveness increases but noise can be erroneously detected in the presence of natural variation in the simulation progress. We study this tradeoff in the next section.

The APTE-based interference detection algorithm assumes that the event processing time is uniform. Thus, it is vulnerable to misprediction in simulation models where event processing time varies, a condition common in realistic simulation models. Thus, we propose an alternative approach that is independent of the simulation model. In particular, to detect the presence of interference, an active thread periodically executes and times the `pthread_yield()` function to relinquish the CPU. If interference is not present, the calling thread is quickly rescheduled as there is no competition for an available CPU. In this situation, the runtime is only several microseconds. On the other hand, if interference exists, the calling thread is context switched out, resulting

in a delay of several milliseconds (a function of the the OS quantum) for the thread to get scheduled again. We call this detector the Processing Interference (PI) detector.

6.2. Deactivating PDES Threads Under Interference

After a performance anomaly is detected, the status of the thread with maximum APTE is set to "inactive." Each thread checks its status at the beginning of the simulation loop, and inactive threads remain idle. The PEs assigned to idle threads are termed orphan PEs, and the responsibility for processing their events is remapped to the remaining active threads.

ADM substantially reduces the effect of interference, and thus significantly improves the performance of the simulation in the presence of external loads. Consider a 48-way simulation interfered with by one external load on the 48-core AMD Magny-Cours machine, for example. Once a performance anomaly is successfully detected, the simulation is then executed by 47 active threads. The OS scheduler will later assign each thread to a different core, thus removing interference between PDES threads and the external load.

6.3. Reactivating Threads

As the interference from external loads may be transient, it is desirable to detect the availability of additional cores to reactivate inactive threads once resources are again available. We implemented two different reactivation mechanisms. The first implementation is to reactivate an inactive thread periodically to check if there is an available core. If noise remains present, then the deactivation logic detects that and deactivates the thread. Thus, in this implementation, the reactivation period must be significantly larger than the detection period to avoid too frequent testing (we use $10 \times T_{gvt}$, 40 times larger than the detection period). We call this approach periodic reactivation (PR).

The main disadvantage of PR is that an inactive thread may be incorrectly re-activated while noise remains present. This harms the performance of the simulation until the interference is detected again. An alternative approach is to check if the noise has disappeared before reactivating a thread. In particular, once an inactive thread is woken up by a signal sent from an active thread, it immediately relinquishes the CPU by calling the `pthread_yield()` function. The runtime of executing `pthread_yield()` function is measured to decide if the interference remains. If the runtime is only several microseconds, then the calling thread gets reactivated. Otherwise, the calling thread remains inactive. Since probing is passive, it does not interfere with the simulation when noise is present, allowing more aggressive reactivation evaluation. We call this approach reactivation after probing (RAP).

6.4. Improving the Data Locality

Similar to DM, ADM remaps threads to PEs in a round-robin fashion even when there is no interference, leading to poor data locality. To improve the data locality, we modified the ADM scheduler to increase locality: we call this implementation LADM. Similar to FM, at the initialization of simulation, each thread is assigned to a primary PE and maintains this assignment in the absence of interference to maximize locality. Once interference is detected and a thread (or more) is deactivated, the PE assigned to the inactive thread is marked as an orphan until such a time when its thread is reactivated. The remaining active threads divide their time between their primary PEs and orphan PEs.

In particular, after each event processing iteration on its primary PE, each active thread checks PEs on the orphan list in a round-robin fashion; it selects an orphan that is currently behind its primary PE in the number of processing iterations (alternatively,

LVT may be used). The status of the selected PE is then checked, and the spin lock for it is acquired if its status is free. Once the thread is admitted to work on the PE, it executes N_{batch} iterations before switching back to its primary PE. We set N_{batch} to 10, which performs well on both platforms. The thread returns to its primary PE if all the orphan PEs have caught up with it. Unlike ADM, the PEs whose primary thread is active remain exclusively processed by that thread, and only orphan PEs experience a loss of locality.

6.5. The Expected Runtime of LADM

Suppose that the simulator is configured with N_p threads at the initialization of simulation, where N_p equals the total count of hardware threads on the multicore platform. In addition, we divide the simulation into n small intervals $[X_{j-1}, X_j]$ indexed by j. Let N_j be the number of external loads running concurrently with PDES during the interval j of the simulation. Once LADM detects N_j ($N_j < N_p$) external loads, the simulation is then executed by (N_p - N_j) active threads during the interval j. The expected runtime of the entire simulation is thus approximated by

$$T_{expected} = \sum_{j=1}^{n} \frac{N_p}{N_p - N_j} \times T_j, \tag{3}$$

where T_j is the execution time required for the interval j of an FM simulation without interference. Moreover, LADM allows at least one active thread to execute the simulation if $N_j \geq N_p$.

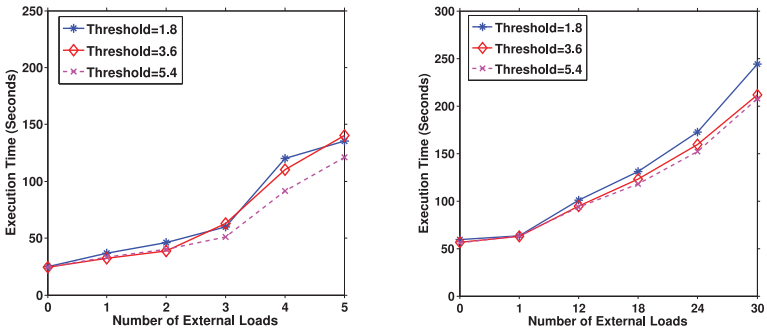
It is important to note that LADM does not achieve proportional slowdown. ADM schedulers simply give up hardware contexts that are in contention to avoid a situation where they are context switched. Because of this conservative behavior, it is possible for interference loads to crowd out the simulation threads, resulting in significant slowdown under high interference. However, the OS scheduling policy will cause inefficient operation if more threads are running than there are available hardware contexts. To approach proportional slowdown, alternative OS scheduling policies are needed.

7. EXPERIMENTAL SETUP AND SIMULATION BENCHMARKS

In most of the experiments, we use the *Phold* benchmark [Fujimoto 1990b]. In this model, simulation objects are equally distributed across PEs. During execution, each object sends a timestamped event message to a randomly selected target. Upon receiving the message, a new message may be sent to another target. *Phold* is controllable, allowing us to specify the communication percentage between different objects [Fujimoto 1990b].

We also use a personal communication system (PCS) model [Carothers et al. 1995], for some of our experiments. The PCS model simulates a cellular provider infrastructure as it manages mobile phone calls. In this model, an event represents a mobile phone call, sent from one cell phone tower to another. Each cell phone tower has a fixed number of channels. Upon receiving a call, the cell phone tower assigns an available channel to the call and later releases the allocated channel when the call completes. If all channels are busy, the call is blocked. In addition, the call is handed off to the destination cell phone tower if the call's connected mobile is leaving the area of the cell phone tower [Carothers et al. 1995]. The experimental configuration for the model on the two multicore systems is presented as follows:

(1) The first platform we use is a quad-core Intel Core i7-860 machine. The platform is running Debian 6.0.2 with Linux version 3.0.0-1. The total number of LPs was set to 8,000 in the *Phold* model and 36,864 in the *PCS* model. These LPs were equally



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 4. Impact of threshold for detecting the interference: ADM.

distributed among eight PEs. We selected a GVT interval of 128 with a batch size of 24. In addition, the simulation time was set to 30,000.

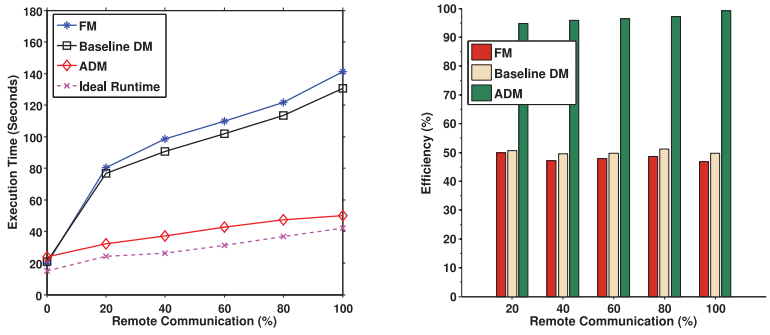
(2) We also evaluated the performance of PDES under interference on an AMD Opteron 6100 (Magny-Cours) 48-core machine with nonuniform memory access (NUMA). The platform is running Ubuntu 10.10 with Linux version 2.6.35-30-server and has 64GB memory. We fixed the total number of LPs at 48,000 in the *Phold* model and 36,864 in the *PCS* model, where LPs are equally distributed among 48 PEs. We used a GVT period of 128 with a batch size of 24 in our simulations. In addition, the simulation time was set to 10,000 for the *Phold* model and 30,000 for the *PCS* model.

8. PERFORMANCE EVALUATION

In this section, we present a performance evaluation of LADM under interference from external loads. In particular, we first evaluate the performance of ADM in comparison to both FM and DM. We follow this by evaluating the performance of the locality-aware scheduler of LADM. We then study the performance of different interference detection algorithms and thread reactivation approaches implemented in LADM to identify the most efficient implementations used in the remainder of the experiments. In addition, every experiment is repeated 10 times to bound the confidence interval; the figures plot the average of these 10 runs.

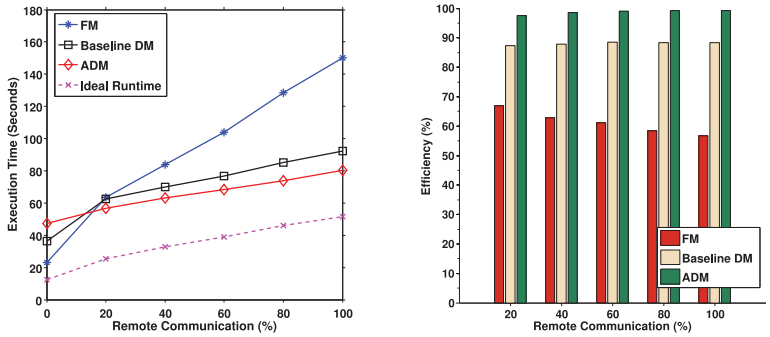
8.1. Evaluation of ADM

We first evaluate the performance of ADM without the data locality optimization. As described in Section 6, there is a tradeoff between responsiveness and stability in the design of the control mechanism that reacts to noise. If the detection threshold is set too high, ADM responsiveness is affected as we may fail to detect noise quickly. On the other hand, too small a value can cause the system to react to normal fluctuations in the application of the system, causing a thread to be incorrectly deactivated in an interference-free environment. Figure 4(a) and Figure 4(b) show the performance of ADM with different values of the threshold on the Core i7 and Magny-Cours platforms, respectively. In this experiment, we used the APTE-based interference detection algorithm. The interfering loads started with the simulation and ran for the duration. In addition, we used *Phold* model, with 40% remote communication (the communication between PEs). As shown in both Figure 4(a) and Figure 4(b), a threshold of 5.4 achieves the best performance on both platforms. Thus, we used this threshold in the rest of our experiments. In practice, this threshold can be derived empirically for important



(a) Execution Time (b) Efficiency

Fig. 5. Performance of ADM on the Intel Core i7 system (interfered with by one external load).



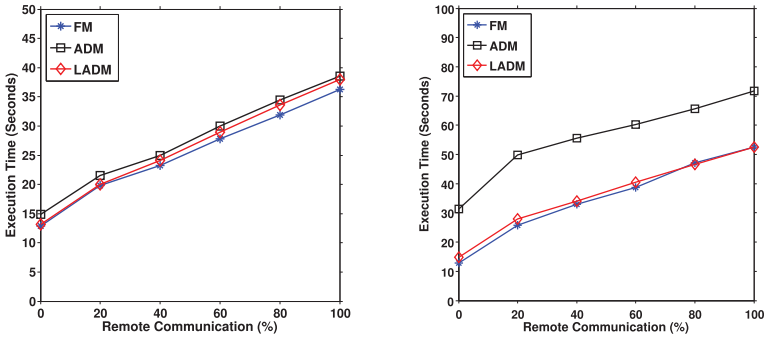
(a) Execution Time (b) Efficiency

Fig. 6. Performance of ADM on the AMD Magny-Cours system (interfered with by one external load).

applications or derived adaptively by scoring adaptation decisions and adjusting the threshold accordingly.

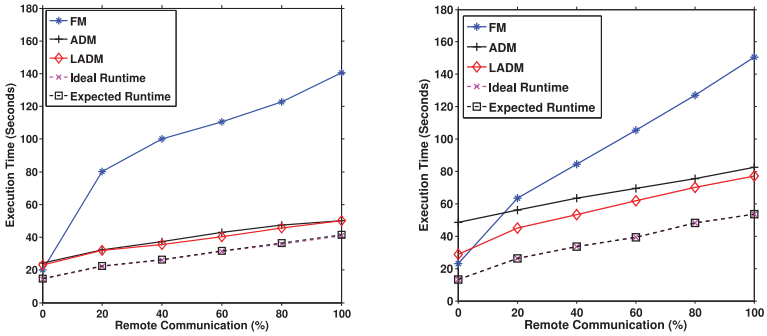
We show the performance of ADM compared to FM and baseline DM on the Core i7 (Figure 5) and Magny-Cours (Figure 6) platforms, respectively. In this experiment, the simulations were interfered with by one external load. In Figure 5(a) and Figure 6(a), we see the execution time as a function of the percentage of remote communication. ADM achieves better performance than FM on the Core i7 but only outperforms FM at high remote communication ($\geq$20%) percentages on the Magny-Cours machine. The behavior can be partially explained by the high cost of lower-level cache accesses on the Magny-Cours relative to the Core i7. The locality-aware version of ADM attempts to address this issue.

In addition, the baseline DM experiences poor efficiency as a thread or more are continuously deactivated, stalling the simulation. Efficiency is defined as the percentage of all processed events that are committed (i.e., not rolled back). ADM reduces contention by inactivating one or more threads to match the active thread count to the available hardware contexts. To evaluate this behavior, we present efficiency of corresponding simulations (Figure 5(b) and Figure 6(b)). Clearly, the baseline DM exhibits poor efficiency, similar to that of FM on the Core i7 machine, but achieves a small improvement in efficiency on the Magny-Cours machine. In contrast, ADM significantly improves efficiency to over 90% on both platforms.



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 7. Performance of locality-aware adaptive dynamic-mapping scheme (no external load).



(a) Intel Core i7 System (b) AMD Magny-Cours System

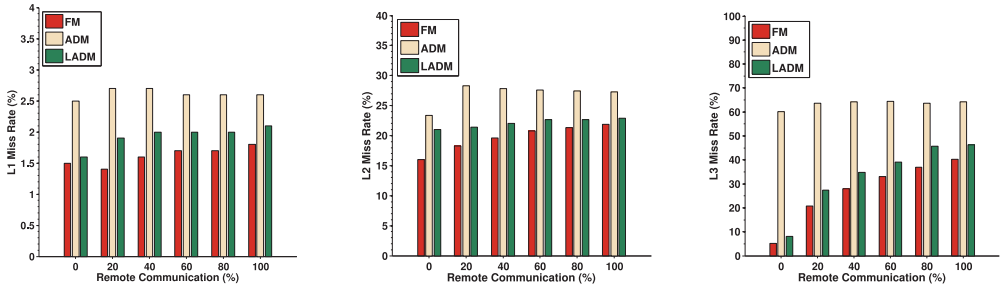
Fig. 8. Performance of locality-aware adaptive dynamic-mapping scheme (one external load).

8.2. The Impact of Data Locality

In ADM, each active thread moves to the next free PE in a round-robin fashion at the beginning of the simulation loop, even when there is no interference. Thus, ADM can lead to poor cache locality, as each thread accesses different PEs, causing their state to be interrogated between caches. LADM improves data locality by associating threads with primary PEs. Only orphan PEs (those whose primary thread is inactive) experience a loss of locality as their events are processed by the other active threads.

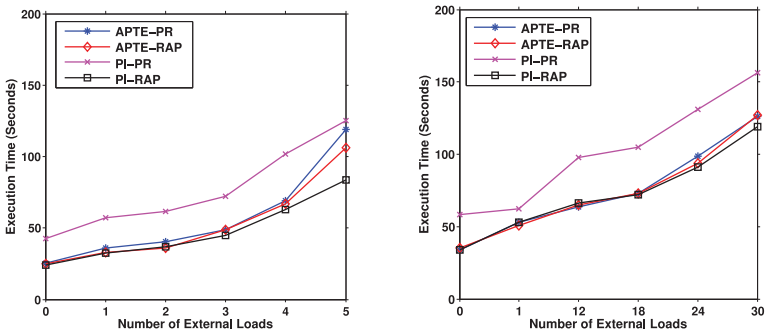
The next experiment evaluates the performance of FM, ADM, and LADM in the absence of external loads to measure the overhead of the mechanisms when they are not needed. Both ADM and LADM use the APTE-based interference detection algorithm and PR approach to reactivate PDES threads. As seen in Figure 7, LADM performs up to 11% better than ADM on the Core i7 machine and up to 53% on the Magny-Cours machine. In addition, LADM incurs small performance loss (less than 5%) relative to the FM version. The overhead is partially due to the extra checking that LADM does; however, we also noticed that rarely, LADM incorrectly detects the presence of interference.

In the next experiment, we consider a scenario with one external interfering process (Figure 8). At high remote communication ($\geq$20%), LADM outperforms the original FM by a factor of up to 2.8$\times$ on the Core i7 machine and up to 2$\times$ on the Magny-Cours machine. In addition, LADM performs up to 43% better than ADM on the Magny-Cours machine, due to the fact that LADM can achieve better data locality. Figure 9 shows



(a) L1 Cache Miss Rate (b) L2 Cache Miss Rate (c) L3 Cache Miss Rate

Fig. 9. Cache performance of 48-way simulation on the AMD Magny-Cours system.



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 10. Impact of interference detection and thread reactivation approaches.

the cache miss rates, demonstrating how LADM has substantially lower cache miss rates than ADM.

8.3. Interference Detection and Thread Reactivation Mechanisms

As described in Section 6, we implemented two different approaches for detecting the interference. The first approach monitors APTE of each active thread (threshold value of 5.4). On the other hand, the second detector (PI) detects the presence of interference by measuring the runtime of executing the `pthread_yield()` function. In addition, we developed two approaches for reactivating threads when additional resources become available. The first approach uses PR to periodically wake up a thread to check if additional resources are available. The second approach, RAP, probes the system to check the availability of resources and reactivates a thread only if it determines there are available resources. In this experiment, we compare the performance of four combinations: APTE-PR, APTE-RAP, PI-PR, and PI-RAP. Figure 10(a) and Figure 10(b) show the performance of these four versions on the Core i7 and Magny-Cours platforms, respectively. Clearly, PI-RAP outperforms the other three implementations on both platforms.

To explain why PI-RAP performs better, we show the convergence behavior of PI-RAP on both the Core i7 (Figure 11) and Magny-Cours platforms (Figure 12). In this experiment, the number of active threads of the simulation is periodically recorded as the simulation progresses. In addition, the simulations were interfered by five external loads on the Core i7 machine and 30 external loads on the Magny-Cours system. Thus, the optimal number of active threads is three on the Core i7 and 18 on the Magny-Cours.

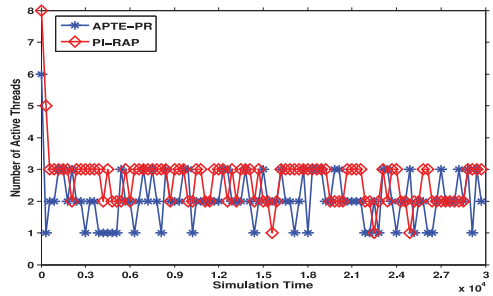


Fig. 11. Convergence on the Intel Core i7 system (five external loads).

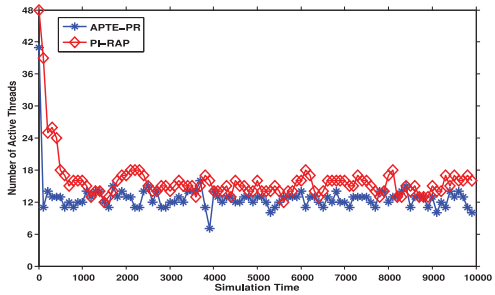
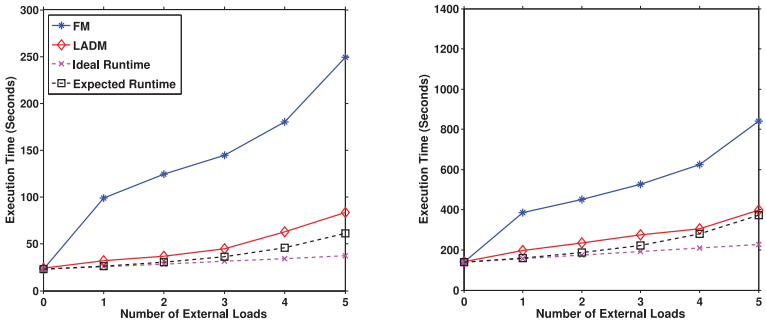


Fig. 12. Convergence on the AMD Magny-Cours system (30 external loads).



(a) EPG=0 (b) EPG=500

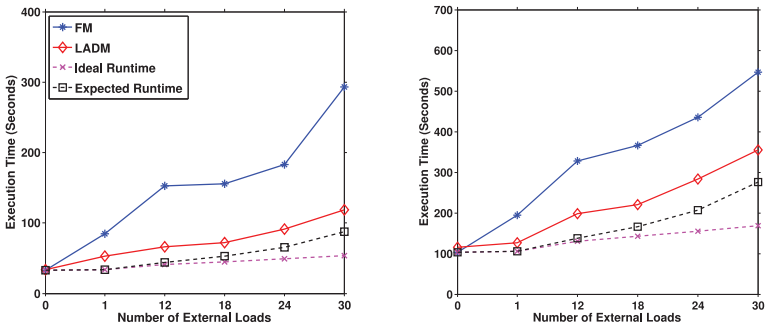
Fig. 13. Impact of event processing granularity on the Intel Core i7 machine.

As shown in Figure 11 and Figure 12, PI-RAP can track the optimal configuration more than APTE-PR on both platforms. We use PI-RAP in LADM for the remainder of the experiments.

8.4. Impact of Event Processing Granularity

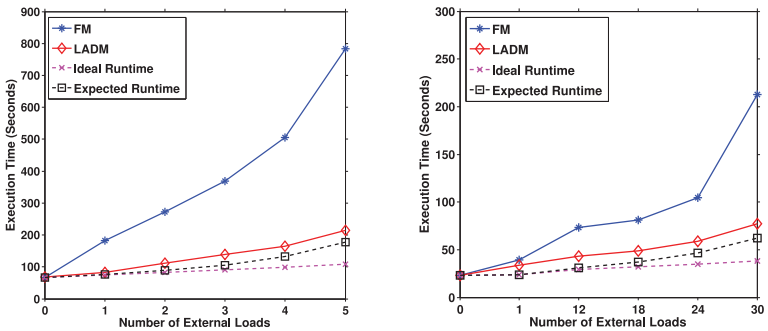
In the next experiment, we modify the Phold model to increase the granularity of event processing time. In particular, a new parameter, called event processing granularity (*EPG*), is defined to control the amount of computation for each event processing in Phold. A higher value of EPG indicates more computation per event, increasing the ratio of computation to communication.

We evaluate the performance of FM and LADM as the number of external loads is increased for both the Intel (Figure 13) and AMD Magny-Cours (Figure 14) platforms



(a) EPG=0 (b) EPG=500

Fig. 14. Impact of event processing granularity on the AMD Magny-Cours system.



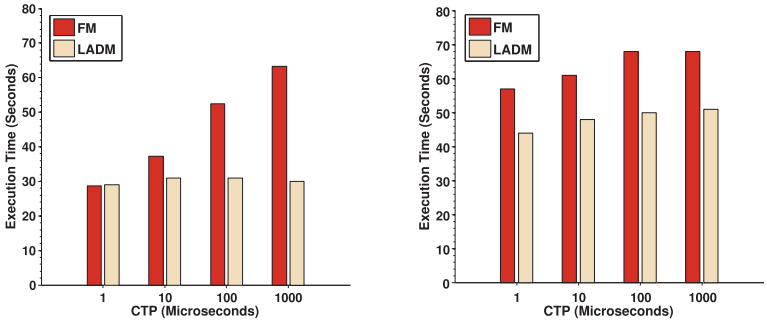
(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 15. Performance of PCS model.

at a remote communication percentage of 40%. As seen in Figure 13 and Figure 14, LADM performs better than FM on both platforms when the simulation is interfered with by external loads. In addition, the gap with the ideal performance is decreased as EPG increases. Another interesting observation is that FM performs closer to LADM as EPG increases. We discover that FM is capable of achieving relatively better efficiency at higher EPGs. As each event requires more time for processing in the case of higher EPG, the advance rate of each PE in FM is more balanced than that in the case of lower EPG.

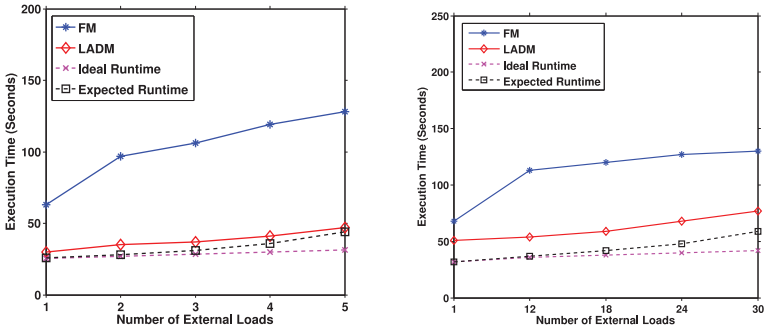
8.5. Performance Evaluation of PCS Model

In this experiment, we study a model of a PCS system [Carothers et al. 1995]. The PCS simulation consists of 36,864 cells (LPs) distributed among eight PEs on the Intel Core i7 machine and 48 PEs on the AMD Magny-Cours machine. Moreover, we fixed the number of channels per cell phone tower at 10. Figure 15(a) and Figure 15(b) show the performance of the PCS model in the presence of external loads on the Core i7 machine and the AMD Magny-Cours machine, respectively. Clearly, LADM performs better than FM on both platforms. In the case of five external loads, for example, the performance of LADM exceeds that of FM by a factor of 3.7× on the Core i7 machine. In addition, LADM outperforms FM by a factor of about 2.8× in the case of 30 external loads on the Magny-Cours machine.



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 16. Impact of on-off interference with different CTPs (one external load).



(a) Intel Core i7 System (b) AMD Magny-Cours System

Fig. 17. Impact of on-off interference as number of external loads is increased (CTP = 1,000).

8.6. Impact of Time-Varying Interference on PDES Performance

In our previous experiments, the entire simulation was interfered with by a fixed level of external loads. In order to evaluate if LADM can adapt quickly in an environment with variable interference, we use an external load model with an on-off pattern. A parameter in the external load, called *computation time per period* (CTP), is used to control the on period of the external load in microseconds every time it is started. We then use the same period to determine when the load will restart. Thus, if CTP is 1 microsecond, we have a recurring external process that runs for 1 microsecond and then sleeps for 1 microsecond.

In this experiment, we used the Phold model, with 40% remote communication. In addition, we performed the experiment under various CTPs. Figure 16(a) and Figure 16(b) show the performance of FM and LADM when the simulation is partially interfered with by an on-off external load on the Core i7 machine and the Magny-Cours machine, respectively. LADM outperforms FM by a factor of up to 2.1× on the Core i7 machine and up to 1.4× on the Magny-Cours machine.

In the next experiment, we fixed CTP at 1,000 and increased the number of on-off external loads. In this experiment, we started external loads at the same time. As shown in Figure 17, LADM outperforms FM by a factor of 2.7× in the case of five external loads on the Core i7 machine and 1.7× in the case of 30 external loads on the Magny-Cours machine.

8.7. Comparison to Gang Scheduling

Thus far, experiments have used the Linux Completely Fair Scheduler (CFS) with SCHED_NORMAL scheduling policy. Under this scheduling policy, the threads belonging to the same application are treated as independent scheduling units by the scheduler, leading to performance problems under interference. Gang scheduling is a widely used solution to control interference in parallel processing environments. It operates by coscheduling threads belonging to the same application together [Feitelson and Rudolph 1992; Wiseman and Feitelson 2003]. As a result, all the threads are running concurrently and performance problems do not arise.

Gang scheduling may be thought of as creating separation between applications in time such that different applications do not run concurrently unless there are sufficient resources to support them. In contrast, application interference resilience (AIR) does not rely on the OS, but rather attempts to create separation in space, reducing the number of active threads until contention for cores is eliminated. Given these different philosophies for managing contention, we expect the following behavior:

—Gang scheduling can lead to poor efficiency when the interfering load is sparse. For example, if one interfering process is running with an application, the resources available to the application may be cut in half as the scheduler alternates scheduling quanta between the application and the interfering process. That is, the slowdown factor may be significantly worse than proportional slowdown.
—Gang scheduling does not allow applications to share cores even when it is possible to do so. For example, an I/O-bound application may be able to gracefully share a core with a computation-bound one.
—On the other hand, application-level resilience is a passive approach and will lead to unfairness when one application reduces its number of active threads while another does not. In this case, interference is eliminated, but at the expense of the adaptive application, which relinquishes resources. The nonadaptive application gains from the additional resources, but the performance of the adaptive application suffers because it has to do with a possibly much smaller number of cores. This behavior is inherent to AIR since it has no control over the interfering application. In this case, assistance from the OS in creating separation is needed for a fair resolution of the problem. We believe that hybrid policies combining application resilience with OS coscheduling are a promising area of future research.

In this subsection, we attempt to illustrate these characteristics by comparing gang scheduling to AIR under a number of scenarios.

We use the Simple Linux Utility Resource Management (SLURM) [Jette et al. 2002] for gang-scheduling implementation. In addition, external loads were selected from PARSEC 3.0 [Bienia 2011], a benchmark suite that contains several programs from different application domains. We evaluated the performance of FM with gang scheduling in comparison to LADM with SCHED_NORMAL scheduling policy on the Core i7 machine (Figure 18). In this experiment, each external load ran one thread, with large input sets. In addition, each external load started concurrently with the PDES simulation. As shown in Figure 18, LADM with SCHED_NORMAL scheduling policy can achieve better performance than FM with gang scheduling. This is because gang scheduling keeps the other seven CPUs idle when the external load gets scheduled, thus leading to inefficiency if the degree of interference is low.

With gang scheduling, the application can be isolated in time from external interference, making the application runtime somewhat resilient to the level of experienced interference. To demonstrate this effect, we evaluated the performance of FM with gang scheduling and LADM with SCHED_NORMAL scheduling when the external

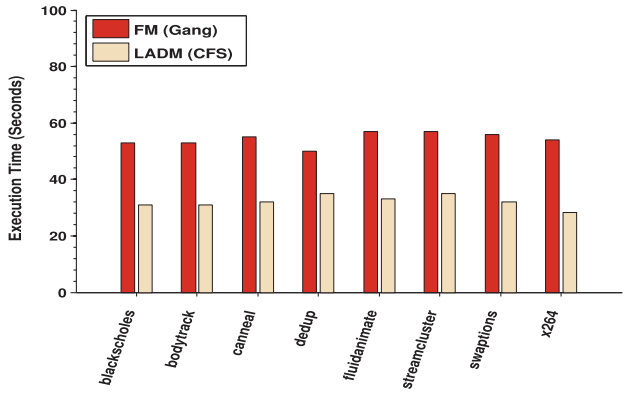
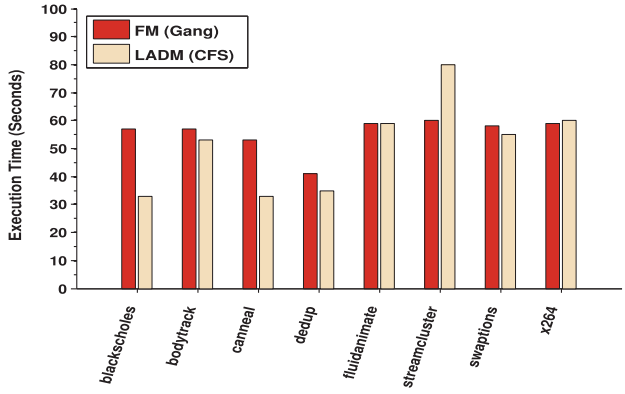
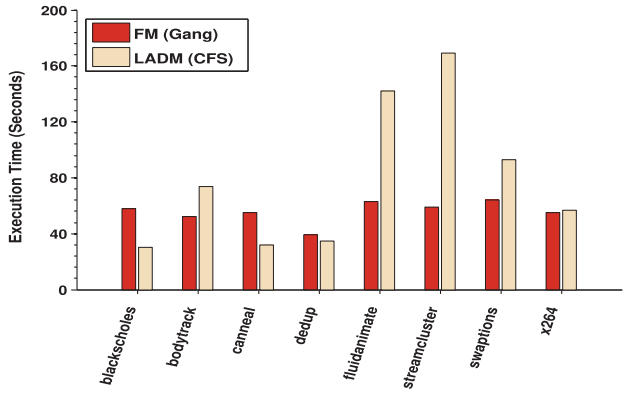


Fig. 18. Impact of OS scheduling policies on PDES performance interfered with by PARSEC benchmarks (running one thread of each external load).



(a) Running Four Threads of Each External Load



(b) Running Eight Threads of Each External Load

Fig. 19. Performance of PDES when degree of interference is increased.

load ran four threads and eight threads, respectively. As shown in Figure 19(a) and Figure 19(b), FM with gang scheduling performs worse when the simulation is interfered with by I/O-bound programs (e.g., blackscholes, canneal, dedup). This is because gang scheduling simply creates separation between PDES and the external load and

Table II. Relative Slowdown of PARSEC Benchmarks and FM with Gang Scheduling

	PARSEC Benchmark (Interfered with by FM)	FM (Interfered with by PARSEC Benchmark)
blackscholes	1.3	2.38
bodytrack	1.35	2.38
canneal	1.24	2.21
dedup	1.75	1.71
fluidanimate	1.23	2.46
streamcluster	1.25	2.5
swaptions	1.35	2.42
x264	1.82	2.46

Table III. Relative Slowdown of PARSEC Benchmarks and LADM with SCHED_NORMAL Scheduling

	PARSEC Benchmark (Interfered with by LADM)	LADM (Interfered with by PARSEC Benchmark)
blackscholes	1.15	1.38
bodytrack	1.23	2.21
canneal	1.12	1.38
dedup	1.38	1.46
fluidanimate	1.16	2.46
streamcluster	1.24	3.33
swaptions	1.22	2.29
x264	1.42	2.5

thus cannot schedule PDES threads until the time slice of the external load expires, even when the external load enters the I/O phase, leading to loss of efficiency. On the other hand, when the external load is CPU intensive (e.g., streamcluster), the performance of LADM with SCHED_NORMAL scheduling becomes worse as the number of threads from the external load increases. This is because LADM suffers as it uses a much smaller number of cores.

In our next experiment, we evaluated the corresponding performance of each PARSEC benchmark as a realistic external load experienced from another user of a computational cluster. Table II and Table III show the relative slowdown of each PARSEC benchmark and PDES with gang scheduling and SCHED_NORMAL scheduling, respectively. Recall that the relative slowdown is calculated by dividing the execution time of application under interference by the one without interference. In addition, each PARSEC benchmark ran four threads, while PDES ran eight threads initially. Clearly, each benchmark in the presence of LADM with SCHED_NORMAL scheduling can achieve better performance than the one interfered with by FM with gang scheduling. Thus, we believe that it is not sufficient to purely rely on gang scheduling to handle the interference problem. Instead, some solutions need to be built inside the PDES simulators (and parallel applications in general).

9. RELATED WORK

LADM was originally presented in a previous paper [Wang et al. 2013]. This article significantly expands on that work by introducing more efficient interference detection and thread reactivation algorithms, characterizing the overhead of interference and evaluating the performance of LADM under gang scheduling. In this section, we first review some prior works in the context of PDES. We follow this by describing the interference problem in the general parallel processing community.

9.1. Dynamic Load-Balancing Approaches

Dynamic load-balancing approaches rely on a monitoring scheme to detect load imbalance and make dynamic adjustments to improve the performance of simulation. These approaches differ in metrics of detecting load imbalance and balancing schemes.

Vitali et al. [2012] presented a load-sharing scheme developed for a symmetric multithreaded optimistic PDES simulator. Each PE is executed by multiple worker threads in order to improve parallelism of the simulation. The approach works by allowing a PE that is lagging behind to acquire additional threads to assist with its computation. Thus, this approach is on the face of it similar to our approach in that threads can be redirected to work on lagging PEs. The approach can effectively foster load-balanced simulation but cannot effectively solve the interference problem, as other threads cannot assist when threads keep getting context switched in the middle of event processing. Child and Wilsey [2012] proposed a different approach to support runtime core frequency adjustment on many-core systems, with the goal of accelerating the critical path of execution of the Time Warp simulation. To balance workloads of LPs, the cores containing LPs with larger rollbacks are underclocked, while the cores having LPs with smaller rollbacks are overclocked. Though this approach may reduce rollbacks caused by external loads, the performance issue caused by the interference still exists as LPs can't advance if their executing thread is switched out.

Carothers and Fujimoto [2000] designed a scheme to support background execution of Time Warp. A background central process periodically monitors the workload of each processor and dynamically determines the set of processors to be used for the Time Warp simulation. LPs are then distributed across these processors by using object migration. Zheng [2005] designed an application-independent load-balancing framework called *Charm++*. In a Charm++ application, the applications were divided into a large number of objects, where multiple objects can be assigned to a single processor. Once a load imbalance was detected, object migration was applied to move some objects from overloaded processors to underloaded ones. Object migration cannot solve the interference problem as well unless all objects are migrated away from a context-switched thread.

9.2. Other Approaches to Reduce the Effect of Interference on PDES

Malik et al. [2009] observed the same behavior present in the cloud environment. To reduce excessive rollbacks caused by interference, they developed a protocol called *TW-SMIP*, with the goal of identifying straggler messages early and thus avoiding frequent rollbacks. Yoginath and Perumalla [2013] proposed an LVT-based hypervisor scheduler to reduce the effect of interference in the cloud environment. In this approach, LP with a lower LVT is given a higher scheduling priority. Replication is another approach that is capable of reducing the effect of interference. As presented in Shum [1998], multiple copies of PDES simulation are executed simultaneously on a heterogeneous workstation cluster. It allows the runtime reconfiguration in terms of runtime resource availability, and thus this approach can adapt to interferences from external loads.

9.3. Interference in General Parallel Processing

Similar to PDES, most parallel applications have dependencies between executing threads. Thus, when the interference occurs, active threads have to wait for context-switched ones before continuing to execute, and the pace of the execution is determined by the slowest thread. As a result, the performance of these applications can be substantially harmed [Tsafrir et al. 2005; Zhuravlev et al. 2010]. Two approaches are widely used to balance workloads of threads at runtime: *work sharing* and *work stealing*. In work sharing, when a thread completes its task, it grabs a new one from a central work

pool shared across all threads [Andrews 1999]. In contrast, in a work-stealing scheme such as *Cilk*, once a thread finishes its tasks, it steals other threads' tasks [Frigo et al. 1998]. Turner [1998] concluded that the work-stealing scheme is not efficient for large PDES simulation, as it increases the critical path length of the simulation. To the best of our knowledge, both work sharing and work stealing are load-balancing approaches, and neither approach can solve the interference problem unless a context-switched thread does not hold any tasks.

Gang scheduling [Feitelson and Rudolph 1992] can mitigate the effect of interference by coscheduling threads belonging to an application together. Conventional gang scheduling [Feitelson and Rudolph 1992] separates applications in time to eliminate interference; however, this approach reduces system throughput. To increase system throughput, Wiseman and Feitelson [2003] proposed a paired gang-scheduling approach where the threads of two jobs can be scheduled in the same gang. However, this approach was implemented on the ParPar cluster [Anat et al. 1999] and thus may not be supported on other systems (e.g., Linux). Xian et al. [2008] proposed a lock-contention-aware scheduler to reduce lock contention in multithreaded applications by first assigning a priority to each thread associated with the number of locks the thread holds. To prevent a thread being preempted in a critical section, the scheduler assigned more time slice to a thread with higher priority. Although this approach may mitigate the effect of interference by preventing a thread from being preempted in a critical section, it may lead to unfair scheduling across threads.

10. CONCLUSIONS

In this article, we demonstrated the sometimes dramatic slowdown that can result in the presence of external interference. We presented a new metric, called *proportional slowdown*, to measure the idealized slowdown of PDES in the presence of interference and showed that in practice, the observed slowdowns far exceed it. We proposed to use dynamic mapping to allow active threads to work on the PEs in a fair way, allowing the simulation to continue to proceed even if one or more threads are context switched. We then proposed a locality-aware dynamic-mapping scheme that improves the locality of the proposed adaptive scheme by attempting to keep PEs assigned to their primary threads. Moreover, we studied the tradeoff between different interference detection and thread reactivation approaches. In addition, we developed a new external load model with an on-off pattern to evaluate PDES performance. Our experimental results showed that LADM is significantly better able to tolerate interference than a fixed-mapping implementation, thus reducing the gap with proportional slowdown. Finally, we studied the effect of gang scheduling on PDES performance and showed that gang scheduling cannot solve the interference problem effectively.

In our future work, we plan to improve the accuracy of the interference detection algorithm in LADM. Moreover, we plan to modify the OS scheduler to make it more friendly to applications. In particular, the OS informs a thread to reach a safe state without holding any task before the thread is context-switched out. Then the tasks of this thread can be safely executed by other active threads.

REFERENCES

D. F. Anat, D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-el, Y. Etsion, A. Kavas, T. Klainer, and M. A. Volovic. 1999. The ParPar System: A Software MPP. *High Performance Cluster Computing* 1 (1999), 754–770.

G. R. Andrews. 1999. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (April 2010), 50–58.

R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. 1995. The interaction of parallel and sequential workloads on a network of workstations. *SIGMETRICS Perform. Eval. Rev.* 23, 1 (May 1995), 267–278.

K. Bahulkar, J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. 2012. Partitioning on dynamic behavior for parallel discrete event simulation. In *Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 221–230.

C. Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

R. D. Blumofe and C. E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

C. Carothers, D. Bauer, and S. Pearce. 2000. ROSS: A high-performance, low memory, modular time warp system. In *Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 53–60.

C. Carothers, K. Perumalla, and R. Fujimoto. 1999. Efficient optimistic parallel simulations using reverse computation. *ACM TOMACS* (1999).

C. D. Carothers and R. M. Fujimoto. 2000. Efficient execution of time warp programs on heterogeneous, NOW platforms. *IEEE Trans. Parallel Distrib. Syst.* 11 (2000), 299–317.

C. D. Carothers, R. M. Fujimoto, and Y.-B. Lin. 1995. A case study in simulating PCS networks using time warp. In *Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 87–94.

R. Child and P. Wilsey. 2012. Dynamically adjusting core frequencies to accelerate time warp simulations in many-core processors. In *Proc. ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 35–43.

P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro* 30, 2 (2010), 16–29.

D. G. Feitelson and L. Rudolph. 1992. Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel Distrib. Comput.* 16 (1992), 306–318.

M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. 212–223.

R. Fujimoto. 1990a. Parallel discrete event simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53.

R. Fujimoto. 1990b. Performance of time warp under synthetic workloads. *Proc. SCS Multiconference on Distributed Simulation* 22, 1 (1990), 23–28.

R. Fujimoto. 2000. *Parallel and Distributed Simulation Systems*. Wiley Interscience.

R. Gupta. 1989. The fuzzy barrier: A mechanism for high speed synchronization of processors. In *Proc. ASPLOS*. 54–63.

D. Jagtap, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh. 2012a. Characterizing and understanding PDES behavior on Tilera architecture. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*.

D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. 2012b. Optimization of parallel discrete event simulator for multi-core systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'12)*. IEEE, 520–531.

D. Jefferson. 1985. Virtual Time. *ACM Tran. Program. Lang. Syst.* 7, 3 (July 1985), 405–425.

M. A. Jette, A. B. Yoo, and M. Grondona. 2002. SLURM: Simple Linux utility for resource management. In *Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP'03)*. Lecture Notes in Computer Science, Springer-Verlag, 44–60.

M. T. Jones. 2009. Inside the Linux 2.6 Completely Fair Scheduler: Providing Fair Access to CPUs since 2.6.23. Retrieved from http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

R. Koo and S. Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Software Eng.* SE-13 (Jan. 1987), 23–31.

A. W. Malik, A. J. Park, and R. M. Fujimoto. 2009. Optimistic synchronization of parallel simulations in cloud computing environments. In *Proceedings of the International Conference on Cloud Computing*. 49–56.

A. Palaniswamy and P. A. Wilsey. 1993. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS'93)*. Society for Computer Simulation, 127–134.

K. H. Shum. 1998. Replicating parallel simulation on heterogeneous clusters. *J. Syst. Architecture* 44 (1998), 273–292.

J. Steinman. 2008. The WarpIV Parallel Simulation Kernel version 1.5.2. Retrieved from http://www.warpiv.com/.

S. C. Tay, Y. M. Teo, and S. T. Kong. 1997. Speculative parallel simulation with an adaptive throttle scheme. In *Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 116–123.

D. Tsafrir, Y. Etsion, D. Feitelson, and S. Kirkpatrick. 2005. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM, 303–312.

S. J. Turner. 1998. Models of computation for parallel discrete event simulation. *J. Syst. Architecture* (March 1998), 395–409.

R. Vitali, A. Pellegrini, and F. Quaglia. 2012. Towards symmetric multi-threaded optimistic simulation kernels. In *Principles of Advanced and Distributed Simulation (PADS'12)*. IEEE, 211–220.

J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. 2013. Interference resilient PDES on multi-core systems: towards proportional slowdown. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'13)*. 115–126.

J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. 2014. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1574–1584.

J. Wang, D. Ponomarev, and N. Abu-Ghazaleh. 2013. Can PDES scale in environments with heterogeneous delays? In *Proceedings of the SIGSIM-PADS Conference*.

Y. Wiseman and D. G. Feitelson. 2003. Paired gang scheduling. *IEEE Trans. Parallel Distrib. Syst.* 14, 6 (2003), 581–592. DOI: http://dx.doi.org/10.1109/TPDS.2003.1206505

F. Xian, W. Srisa-an, and H. Jiang. 2008. Contention-aware Scheduler: Unlocking Execution Parallelism in Multithreaded Java Programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. 163–180.

Srikanth B. Yoginath and Kalyan S. Perumalla. 2013. Optimized Hypervisor Scheduler for Parallel Discrete Event Simulations on Virtual Machine Platforms. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques (SimuTools'13)*. 1–9.

G. Zheng. 2005. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. Ph.D. Dissertation. Champaign, IL. Advisor(s) Kale, Laxmikant V. AAI3202198.

S. Zhuravlev, S. Blagodurov, and A. Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of ASPLOS*. ACM, 129–142.

Received February 2014; revised June 2014; accepted December 2014