



How to Simulate 1000 Cores

Matteo Monchiero, Jung Ho Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi
HP Laboratories
HPL-2008-190

Keyword(s):

simulation, multicore, manycore, chip multiprocessor, application scaling

Abstract:

This paper proposes a novel methodology to efficiently simulate shared-memory multiprocessors composed of hundreds of cores. The basic idea is to use thread-level parallelism in the software system and translate it into corelevel parallelism in the simulated world. To achieve this, we first augment an existing full-system simulator to identify and separate the instruction streams belonging to the different software threads. Then, the simulator dynamically maps each instruction flow to the corresponding core of the target multi-core architecture, taking into account the inherent thread synchronization of the running applications. Our simulator allows a user to execute any multithreaded application in a conventional full-system simulator and evaluate the performance of the application on a many-core hardware. We carried out extensive simulations on the SPLASH-2 benchmark suite and demonstrated the scalability up to 1024 cores with limited simulation speed degradation vs. the single-core case on a fixed workload. The results also show that the proposed technique captures the intrinsic behavior of the SPLASH-2 suite, even when we scale up the number of shared-memory cores beyond the thousand-core limit.

External Posting Date: November 6, 2008 [Fulltext]
Internal Posting Date: November 6, 2008 [Fulltext]

Approved for External Publication



To be presented at dasCMP 2008, como, Italy, November 9 2008.

© Copyright dasCMP 2008

How to Simulate 1000 Cores

Matteo Monchiero, Jung Ho Ahn, Ayose Falcón, Daniel Ortega, and Paolo Faraboschi
Hewlett-Packard Laboratories

{matteo.monchiero, jung-ho.ahn, ayose.falcon, daniel.ortega, paolo.faraboschi}@hp.com

Abstract

This paper proposes a novel methodology to efficiently simulate shared-memory multiprocessors composed of hundreds of cores. The basic idea is to use thread-level parallelism in the software system and translate it into core-level parallelism in the simulated world. To achieve this, we first augment an existing full-system simulator to identify and separate the instruction streams belonging to the different software threads. Then, the simulator dynamically maps each instruction flow to the corresponding core of the target multi-core architecture, taking into account the inherent thread synchronization of the running applications. Our simulator allows a user to execute any multithreaded application in a conventional full-system simulator and evaluate the performance of the application on a many-core hardware. We carried out extensive simulations on the SPLASH-2 benchmark suite and demonstrated the scalability up to 1024 cores with limited simulation speed degradation vs. the single-core case on a fixed workload. The results also show that the proposed technique captures the intrinsic behavior of the SPLASH-2 suite, even when we scale up the number of shared-memory cores beyond the thousand-core limit.

1 Introduction

Multi-core processors are here to stay [8, 12, 25, 26]. Today, you can find general purpose components with 2 to 16 cores and more dedicated parts with up to hundreds of cores [1, 5]. The architectural variability is rather broad: many lightweight cores versus few heavyweight cores, shared memory versus private memory, etc. These are only some of the architectural decisions that designers have to take and solve depending on their design goals and constraints. It is clear that researchers and designers of the next generation processors will have to extensively evaluate the tradeoffs related to this huge design space.

The increase of the core count as projected by several industrial and academic roadmaps [4] makes this process even

more difficult. Some *many*-core systems have already arrived [5, 9], and by simply applying Moore’s law it is likely that hundreds or thousands of cores on a single die will become a commodity in the upcoming years.

Nevertheless, simulating such a large Chip Multiprocessor (CMP) is an open problem for the computer architecture community. Some existing simulators are able to simulate moderately-sized multi-core processors, although their low simulation speed and scalability limit them to tens of cores.

This paper describes a methodology to efficiently simulate a CMP of hundreds or thousands of cores using a full-system simulator. Our framework extracts threads from a multi-threaded application running in the full-system simulator and feeds them to a timing simulator of a many-core architecture. It correctly generates instruction streams for each core, while respecting the timing constraints enforced by the synchronization of the application, such as locks and barriers. We show that we can predict the performance implications that are independent of detailed memory and interconnect models.

1.1 Background

Although detailed simulation studies of 1000-core processors have not been reported yet, the problem of simulating shared memory processors is well known and previous research has already explored various techniques to address it. Our approach has its roots in *direct-execution* simulators like the Wisconsin Wind Tunnel (WWT) [21] and the Stanford Tango-lite [11]. These simulators rely on the execution of the majority of an application on native hardware, with the simulator paying special attention only to those events that do not match the target architecture. For example, the WWT runs applications on a Thinking Machine CM-5, a large message passing parallel computer, and traps on each cache miss, which is simulated in a *shared virtual memory*. Tango-lite executes the applications on a single-processor host and reschedules the threads to ensure that events are simulated in the correct order.

We borrow many concepts from these early tools, but we apply them in a different context: we run the appli-

cations on a full-system simulator that provides isolation and hardware independence. Since we want to simulate a number of cores much larger than that available in the host machine, we designed techniques to reorder the threads, as generated by the full-system simulator, similar to Tango-lite. Our framework detects the synchronization and implements the corresponding semantics by properly delaying the threads that must wait. The synchronization is abstracted as suggested by Goldschmidt and Hennessy [11] to minimize time-dependent simulation outcomes.

1.2 Contributions

The key contribution of this paper lies in how we translate thread-level parallelism from an application running in a full-system simulator into core-level parallelism to be exploited by a many-core simulator back-end. These techniques make it feasible to simulate thousands of cores on a commodity hardware host. Because this approach can have a severe impact on the memory footprint of the simulator itself as we describe later in Section 4, we also have implemented an instruction stream compression scheme and added scheduling feedback to keep memory requirements under control. With these techniques, we can model large-scale future many-core systems on currently available hosts.

To achieve this goal, we retain some features of full-system simulation, but we give up some others. Our tool is capable of almost seamless execution of a multithreaded application, given that the source code is minimally annotated. It efficiently decouples functional and timing simulation, so that functional and timing modes can be interleaved. Nevertheless, we do not currently exploit I/O device emulation and we discard parts of OS activity. This latter limitation is in part because our simulator is based on an intrinsic mismatch between the architecture emulated by the full-system simulator and the target many-core architecture. We thus have to discard those parts of the execution that are not meaningful for the target many-core architecture. On the other hand, we can faithfully mimic the behavior of multithreaded applications in a many-core architecture with synchronization, since we actually re-implement the timing implications of the synchronization in our simulator.

The rest of the paper is organized as follows. The next section discusses alternative simulation approaches researched in the community. Section 3 describes the proposed methodology and Section 4 shows the experimental results. Finally, Section 5 concludes the paper.

2 Related Work

Simulating many cores is an active research topic in the computer architecture community. Conventional application-driven simulators can scale up to few tens of

cores (e.g., SESC [22] is reported to scale up to 64 cores). Nothing theoretically prevents these simulators from scaling up to thousands of cores. Nevertheless, application-driven simulators have other limitations. Since they typically do not simulate any OS, they require special libraries to fake a subset of OS functionalities to support a minimal threading system. Our solution is based on full-system simulation, relies on native OS libraries, and only requires a simple annotation of the synchronization.

Trace-driven simulation is the base of many modern simulators for multiprocessors [14, 22]. These simulators perform functional emulation first and then feed instructions to a timing engine to model the performance. In many cases, a binary instrumentor is used as trace generator. Binary instrumentation tools like ATOM [24] or Pin [16] allow for code profiling, microarchitectural simulation, and trace generation. Some recent simulators based on Pin [14, 20] have been used primarily for cache hierarchy studies, but few details on how these tools are designed have been disclosed.

Our approach is also based on trace-driven simulation. We use a full-system simulator based on virtualization technology to generate instruction traces. Kolding et al. [15] explore the validity of trace-driven simulation for multiprocessors, concluding that correct instruction interleaving is crucial to maximize simulation accuracy.

Recent studies on many-core systems (range of hundreds) introduce a methodology based on replicating traces [13, 30]. Traces are collected from a real system—composed of a limited number of cores—and then tweaked to approximately reflect actual data sharing. Unfortunately, this does not accurately simulate realistic data sharing [30].

Several research efforts [7, 27, 28] are directed at exploiting intrinsic hardware parallelism. The common idea is to map the target multi-core architecture onto one or more FPGAs. Some approaches are more oriented towards selectively accelerating parts of a software simulation [7], or towards the actual prototyping of the desired design [28]. Even if we believe this is an interesting promise, several issues remain unsolved, especially regarding the flexibility of the models which may be limited by the available hardware library.

Recently, full-system simulation has gained further interest compared to application-driven approach within a research community. Full-system simulators [6, 17, 18] were successfully used to investigate small to medium scale CMPs. Since full-system simulators emulate the whole software stack, scaling up to thousands of cores is not as simple as changing a parameter. Both the OS and the BIOS need to understand how to manage those cores efficiently, which may not be trivial to implement. Our approach is independent of the actual configuration in terms of the number of cores of the full-system simulator.

3 The Framework

This section describes our methodology for simulating a many-core system. We start by presenting the big picture of the framework. We then illustrate the communication interface used to make the full-system simulator and the guest OS communicate with the timing simulator. We also introduce the main principles and algorithms of the framework, describing how we extract threads from the running application and map them to CPUs, and how we convey the semantics of the synchronization from the application into the simulator. Finally, we present techniques to control the memory usage of the framework.

3.1 Overview

Our simulation infrastructure uses a functional-first simulation approach [19], in which a full-system functional simulator dynamically generates a trace of events that is consumed by a timing simulator.

An overview of the proposed framework is shown in Figure 1. It is composed of three main parts: a full-system functional simulator, an *interleaver* module (framed between the dashed lines), and a timing simulator. The functional simulator and the interleaver act as a front-end for the timing simulator, which consists of a collection of timing models for CPUs, caches, interconnect, and memories. We describe each of these parts in detail later in this section.

The functional simulator streams all executed instructions, including OS activity to the interleaver. We chose to discard those OS instructions that do not meaningfully match the target architecture. For example, we filter the instructions of the kernel scheduler, but we simulate all system calls except for I/O events. At the interleaver level, we separate instructions according to the thread they belong to, and dispatch them to the appropriate CPU model in the back-end. Apart from detecting context switches, we also detect synchronization points such as barriers and spin locks.

Any generic multithreaded application that functionally executes on our full-system simulator can be analyzed and decomposed into CPU cores with the technique presented in this document. Moreover, nothing prevents this technique to be used for independent applications, as long as there are as many tasks as cores to simulate.

3.2 From Threads to Cores

The first step in the process of mapping application threads to simulating cores is to identify the different threads running in the functional simulator. In the Linux kernel (which is the OS we run as a guest in the functional simulator), both processes and threads are treated as

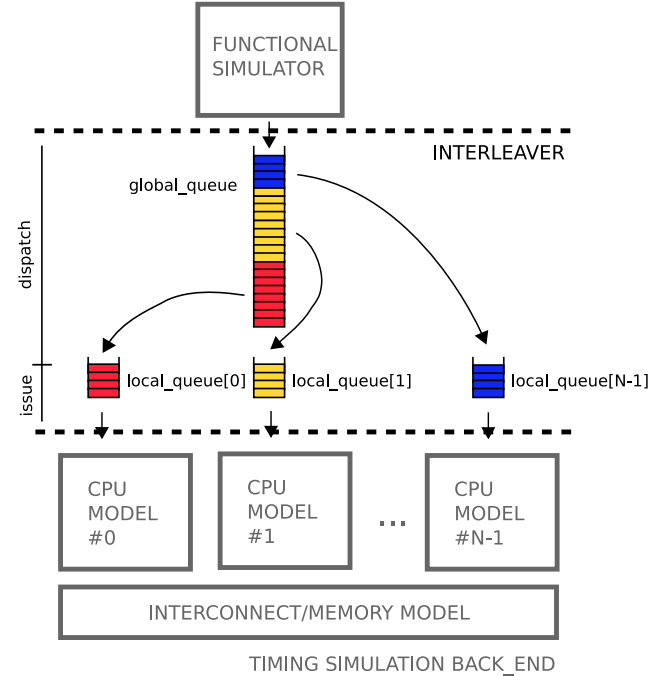


Figure 1: High level view of the simulator. The diagram shows the main data structures of the interleaver, i.e., the global queue and the local queues. The arrows represent the dispatch and issue phase of the interleaver. Instructions of different colors belong to different threads.

tasks. The OS scheduler is responsible for deciding which task takes the CPU at any given time, according to the OS scheduling policy and task priority. When a task is assigned to a CPU, only instructions from this task can be executed, until the time slice finishes or an interrupt occurs. In any case, the OS scheduler is then called to decide which task executes next.

In order to identify the different threads, we modify the context switch mechanism of the Linux kernel (version 2.6.23). We insert a *special instruction* in the OS scheduler code (`_switch_to()` function in the kernel code), telling our simulator the processID (PID) and threadID (TID) of the next task. Other simulators like Simics [17] use similar hooks to allow communication from the emulated world to the simulation infrastructure. The interleaver uses the PID and TID to dispatch instructions to the appropriate queues.

This methodology is independent of the specific functional simulator. It may emulate an arbitrary number of CPUs, but since we rely on the thread abstraction we do not care where threads are actually mapped in the functional simulator. Our methodology abstracts all mapping and scheduling done by the OS in the functional simulator and exposes threads as instruction streams to the different

Algorithm 1 Instruction dispatch

```
while global_queue.size() > 0 do
  cpu = to_cpu(global_queue.front().threadID)
  local_queue[cpu].push(global_queue.front())
  global_queue.pop()
end while
```

Algorithm 2 Instruction issue

```
if every local_queue has at least one instruction then
  for all non stalled local_queues do
    execute(local_queue.front()) on cpu(local_queue)
    local_queue.pop()
  end for
  cycles = cycles + 1
end if
```

CPU timers that model each core.

The *interleaver* is the module that glues the functional simulator and the timing models. This module separates the instructions belonging to different threads—we call this phase *dispatch*—and schedules the instructions for execution in the CPU models with the correct ordering—this phase is called *issue*. This whole phase (dispatch and issue) is the key of our methodology. It properly decouples the functional simulation and the timing simulation.

Algorithm 1 shows the basic operations of the dispatch phase: instructions are moved from the `global_queue` to the `local_queues` of their assigned CPUs. This mapping is done via the `to_cpu()` function. In our case studies, which involve high-performance computing applications, this mapping is straightforward since these applications tend to spawn the same number of threads as hardware contexts available. We use two special instructions to guide the allocation and de-allocation of threads to CPUs, namely `THREAD_BEGIN` and `THREAD_END`.

Algorithm 2 illustrates the issue phase. The condition of the `if` statement checks for availability of instructions for all CPUs. The `forall` loop does the actual scheduling of instructions to the local queues. In order to guarantee that instructions obey the execution flow, we buffer instructions until every local queue has at least one instruction. Only the `local_queues` whose CPUs are not stalled can execute the instructions in the given cycle. Synchronization slightly changes this picture as we will see in the next section.

Figure 2 shows an example of a program spawning a thread that is executed concurrently with the master thread. The beginning and end of each thread are marked with the special instructions `THREAD_BEGIN` and `THREAD_END`. Figure 3 explains how this code is executed by our simulation framework. The `THREAD_BEGIN` calls activate the mechanism. After that, the `local_queue[0]` starts buffering instructions from the thread `tid1`.

```
work(...)
{
  THREAD_BEGIN;
  /* do some work */
  THREAD_END;
}
...
tid[0] = thread_create(&work,0);

work(1);

thread_join(tid[0]);
```

Figure 2: Fragment of code containing thread creation (one thread is spawned other than the master thread) and join

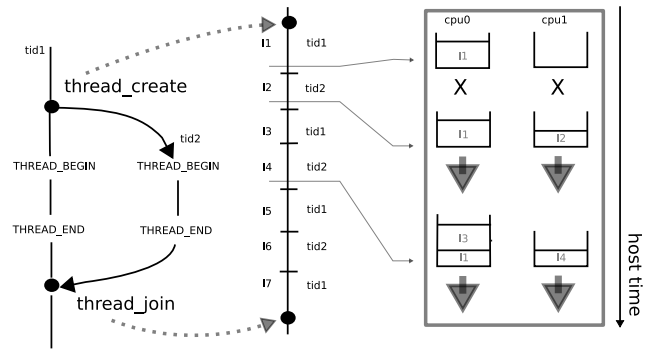


Figure 3: The basic execution flow of the framework (relative to the program in Figure 2): Two threads are executed concurrently. From left to right we show a pictorial view of the program execution, the instruction stream produced by the functional simulator, i.e., the contents of the global queue, and some snapshots of the contents of the local queues. The thin gray arrows indicate the last instruction of the global instruction stream that is considered in the corresponding snapshot. The big arrows and the X indicate whether a local queue can issue instructions or not.

Since no instructions from `tid2` are present in the `local_queue[1]`, none of the local queues can issue instructions (represented by the two Xs). As soon as instructions from `tid2` are dispatched to its local queue, the interleaver starts issuing instructions to both CPUs, and simulation time advances.

In the next subsection, we explain how we augment our mechanism to deal with programs that contain locks and barriers.

Table 1: Special instructions used for locks and barriers, and the behavior of the simulator

Special instruction	Dispatch	Issue
<i>Barriers</i>		
BARRIER_BEGIN	Dispatch and skip	wait until the head of each local queue is BARRIER_BEGIN
BARRIER_END	Dispatch	
<i>Locks</i>		
SLOCK_BEGIN	Dispatch and skip	Wait until the lock is released
SLOCK_END	Dispatch	
UNLOCK	Dispatch	Release the lock

3.3 Synchronization

Simulating synchronization between threads involves several steps which change our dispatch and issue mechanisms: (1) the synchronization event must be delivered to the *interleaver*; (2) the simulator must abstract from the time-dependent patterns that may depend on the machine emulated by the functional simulator; (3) the *interleaver* must respect the order imposed by the synchronization when issuing the instructions for execution.

To deliver a synchronization event, we annotate the synchronization points in the source code. The *special instructions* we chose to mark locks and barriers are listed in Table 1. By having these special instructions at the beginning and end of any barrier and lock, the simulator can isolate the effects of time-dependent patterns (spinning) which are used in the implementation of most threading libraries. The interleaver uses them to enforce the correct instruction ordering as we will see shortly in this section. The simulation back-end will simulate the synchronization based on the micro-architectural model of the particular experiment.

Figure 4(a) shows an example of how special instructions are applied to a barrier. When a local_queue has a BARRIER_BEGIN as the head element, it cannot issue instructions—other queues can go on consuming instructions until they also reach a BARRIER_BEGIN. The BARRIER_BEGIN also instructs the simulator to skip all instructions until a BARRIER_END is reached. Once all threads reach the barrier, normal execution resumes. This way, we abstract the barrier so that the polling of the barrier variable is just discarded and can be potentially rebuilt by the timing simulator. The skip is implemented in the *dispatch* (Algorithm 1), while the semantics of the BARRIER_BEGIN are implemented in a modified **while** condition of the *issue* (Algorithm 2).

Locks are implemented in a similar way. Figure 4(b) shows an example of a thread contending a lock that protects a critical section. SLOCK_BEGIN and SLOCK_END delimit a spin lock, while UNLOCK is placed soon after the actual unlock. All instructions between the SLOCK_BEGIN

```

work(...)
{
  ...
  BARRIER_BEGIN;
  barrier();
  BARRIER_END;
  ...
}

work(...)
{
  ...
  BEGIN_SLOCK(1);
  lock(1);
  END_SLOCK(1);
  /*critical section*/
  unlock(1);
  UNLOCK(1);
  ...
}

```

Figure 4: Fragment of code containing (a) a barrier; and (b) a spin lock protecting a critical section

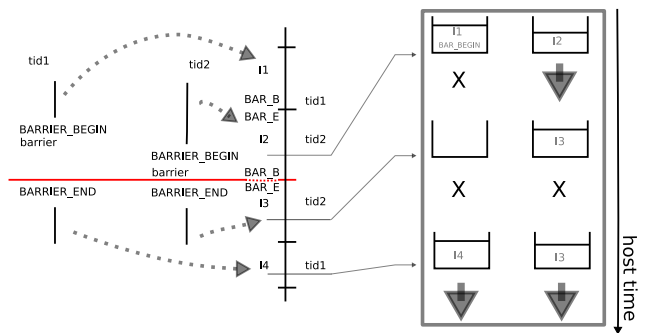


Figure 5: Diagram illustrating the execution of a barrier (code in Figure 4(a))

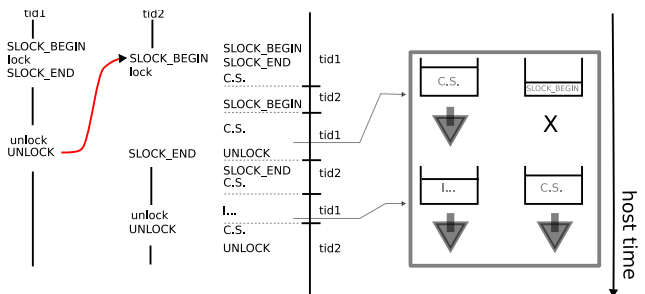


Figure 6: Diagram illustrating the execution of a lock (code in Figure 4(b))

and SLOCK_END are skipped (spinning). A thread acquires the lock at the SLOCK_BEGIN and releases it at the UNLOCK. Other threads trying to acquire the lock by performing another SLOCK_BEGIN are stopped and cannot issue instructions till the lock is released.

Similarly to Figure 3, Figures 5 and 6 illustrate the execution of a barrier and a spin lock. The diagrams of the local queues (right part of the figures) show when the local queues are stopped because a BARRIER_BEGIN has been reached or a lock has not been released yet.

3.4 Taming Memory Usage

The buffering of instructions in the local queues can easily consume enormous amounts of memory. This is especially evident for applications with heavy synchronization and unbalanced threads. To keep memory usage under control we have devised an instruction compression technique in the local queues and a scheduling feedback mechanism.

The compression mechanism is implemented in the *dispatch* phase. Since our memory back-end is cache-line addressable, we perform an early branch prediction and encode only possible misses. We perform address compression techniques both on instruction and data memory references by storing only the necessary bits to address cache lines. Having a simple pipeline in-order model allows us to enhance compression by substituting the full opcode by a broad type identifier and not tracking register values.

The Linux scheduler in the guest OS may cause an unbalanced execution of threads, and this can be exacerbated if the threads that run longer have already generated a large number of instructions. All these instructions must be stored until we have instructions from all the threads ready for being issued (recall Figure 1). In some cases, this may cause our memory usage to explode beyond reasonable limits (tens of GBs). To solve this, we have implemented a feedback mechanism to increase the priority of threads that are in waiting state in the guest OS. When the interleaver detects that one of the local queues has ran out of instructions, it instructs the guest OS to increase the priority of the thread associated to this local queue. The kernel scheduler then selects that thread for execution on the next time slice, and the empty local queue is filled with new instructions. Note that this does not affect measured performance, since the full-system emulator only acts as an instruction flow generator. Performance estimations are generated by the timing models placed in the simulator back-end.

For example, Figure 2 also illustrates a typical case where the feedback path is useful to balance thread execution. Threads are created by calling the `thread_create()` function, before the master thread is created by a subsequent call to `work()`. Since the guest OS has no visibility of the target architecture, we observed that it tends to first execute all child threads and, subsequently, the master thread. Nevertheless, no instruction can leave the local queues until the master thread has started executing too. Our feedback path is able to raise the priority of the master thread anticipating the execution and causing a more balanced usage of the local queues.

4 Experimental Results

We applied our methodology to a full-system simulation environment based on HP Labs’ COTsOn [3] and AMD’s

SimNow simulator [2]. The guest system runs Linux with kernel 2.6.23. We designed and implemented a modular *interleaver* that interfaces the functional simulator and the timing models, as described in the previous sections. We used simple in-order cores and an ideal memory system for the back-end timing simulator. Note that modeling more complex cores and memory systems is beyond the scope of this paper since our focus is on the fundamental characteristics of the simulation framework, and not on the specific timing models.

We used the SPLASH-2 benchmark suite [29] to validate our approach. Table 2 reports the programs we use and the corresponding datasets. We compiled each application with `gcc-4.2`, options `-O3 -m32 -msse2 -mfpmath=sse`. We use two datasets: DS1, the default dataset for SPLASH-2 [29] and DS2, a scaled up dataset. For DS2, we tried to scale up the dataset by 16 times.¹ This was possible for FFT, LU, Ocean, and Radix. We had to scale the dataset of Water-spatial by 64 times to provide large enough data structure for 1024 threads, while we were only able to scale the datasets of Barnes and FMM by 4 times to bound simulation time tractable. We could not fine-tune the datasets of other applications easily, so we either could find a suitably larger dataset (Cholesky, Raytrace, and Radiosity) available in the original benchmark suite, or we had to take them out of our experiments. Table 2 reports the maximum number of CPUs we simulate for each benchmark.

We chose to keep the SPLASH-2 code largely unmodified: understanding the algorithmic implications in the context of a 1000-core processor would be an interesting study by itself [23], but is outside the scope of this paper. We adapted Radix by replacing implicit synchronization via semaphore variables with explicit synchronization constructs. We carefully analyzed each benchmark to identify any possible time-dependent pattern, other than lock and barrier spinning. We found several polling loops (i.e., implicit synchronization implemented by polling a shared variable) that affected the simulation substantially and we chose to skip these parts. The last column in Table 2 indicates the benchmarks we modified.

Table 2 also reports the number of instructions across all configurations (1–1024 cores). The variation in number of instructions is quite large for some benchmarks, especially for DS1, but drops for the larger DS2—except for LU. Given that we used the same dataset for different core counts, the way the dataset is divided among the cores may differ. Some applications have to process more and larger overlapping regions when the core count increases, thus featuring an increase of the number of instruction.

Due to the mismatch of the simulated architecture and

¹Our target is 1024 cores, while the original SPLASH-2 paper used 64 cores [29]. Hence, $1024/64 = 16\times$.

Table 2: Benchmarks

	DS1 (default)			DS2			Polling Loops
	#instrs. ($\times 10^9$)	max scaling		#instrs. ($\times 10^9$)	max scaling		
Barnes	16 K particles	1.7-2.1	1024	64 K	9.5	1024	yes
Cholesky	tk15.O	0.4-0.7	1024	tk29.O	1-1.9	1024	yes
FFT	64 K points	0.03-0.04	256	1 M	0.6-0.7	1024	
FMM	16 K particles	2.9	128	64 K	10	1024	yes
LU	512 \times 512	0.35-0.7	1024	2048 \times 2048	22-43	1024	
Ocean	258 \times 258	0.45-0.7	1024	1026 \times 1026	6.3-6.8	1024	
Radiosity	room	1.5-1.6	1024	largeroom	4.0-4.7	512	yes
Radix	1 M integers	0.07-0.1	1024	16 M	1.1-1.2	1024	
Raytrace	car	0.7	1024	balls4	6.5	1024	
Volrend	head	1-1.9	1024	-	-	-	yes
Water-spatial	512 molecules	0.6	64	32 K	37	1024	

Table 3: Comparison with the original SPLASH-2 results ((SPLASH-2 – our results)/our results)

	Number of cores		
	16	32	64
Barnes	-	-	-15%
Cholesky	-	-	20%
FFT	-	-	-
FMM	-	-	-20%
LU	-	-	-
Ocean	-	-	-7%
Radix	10%	10%	10%
Radiosity	-30%	-30%	-30%
Raytrace	-50%	-50%	-55%
Water-spatial	-	-	10%

the architecture emulated by the functional simulator, those benchmarks that feature dynamic job scheduling might incur higher accuracy loss with our approach. This same problem was observed by Goldschmidt and Hennessy [11] for the Tango-lite simulator. Nevertheless, they report limited inaccuracies for the applications they used (the SPLASH benchmarks). Similar conclusions were obtained by Eggers and Kats [10] for trace-driven simulation.

Figure 7 replicates the results reported in the original SPLASH-2 paper [29]. Table 3 shows the differences between the two sets of results. Since we visually estimated the numbers, we only report those differences greater than 5%. For less than 16 cores, our results almost perfectly match those in the SPLASH-2 paper, so we do not show the numbers in the table.

As the simulation infrastructures used in both cases are different, we obviously cannot expect a perfect match in the simulation results, but ultimately the trend should be very similar. For example, the host platform, the OS, and the ISA are completely different. While the SPLASH-2 paper uses MIPS, our simulator is based on x86_64.

By looking at the original graphs, the main differences are for Raytrace and Radiosity, while for the other benchmarks we generally predict just a slightly worse scaling (differences bound to 20%). The mismatch is mainly because

we differently account for the initialization of these benchmarks.

Figure 8(a) shows the amount of synchronization cycles out of the execution time for each benchmark and each configuration. Our predictions (32 cores) also match those made in the SPLASH-2 paper, with few exceptions.

In Figure 9(a), we report the results for the scaling up to 1024 threads. Note that most benchmarks do not scale due to their small dataset, except for Barnes, which shows a moderate speedup (280 \times at 1024 cores). Figure 9(b) shows the results we obtained by increasing the dataset. Some benchmarks (FFT, Water-Spatial, Ocean, and Radix) scale meaningfully (600 \times at 1024 cores); we could indeed scale up the dataset of these benchmarks by at least 16 \times . LU is an exception, since it is limited by the synchronization—it scales up to 380 \times at 1024 cores. Other benchmarks scale less clearly since dataset, synchronization, and algorithm properties may interact. For example, Cholesky gains very little with respect to Figure 9(a), but besides the possibly small dataset, this benchmark is limited by fine-grain lock synchronization. Raytrace has a clear speedup by increasing the dataset (3 \times), but due to the fine-grain synchronization (similar to Cholesky), it does not scale with the number of cores. These trends are confirmed by Figure 8 that shows how the synchronization fraction decreases for the benchmarks that scale up.

Figure 10(a) shows the simulation speed for all configurations. We run our experiments on a simulation farm composed of AMD Dual-Core Opteron at 2.8GHz and 8GB of RAM. The average simulation speed is around 1M instructions per second for 1 core. For 1024, simulation speed decreases to 0.7M instructions per second (a 30% drop). Most of the additional slowdown is due to the synchronization. For FMM and Cholesky, simulation speed drops by 50%. These indeed have the highest synchronization component as shown in Figure 8.

Memory usage (reported in Figure 10(b)) is a critical aspect of our simulation flow. The upper bound for all applications and configurations is 8GB, corresponding to the mem-

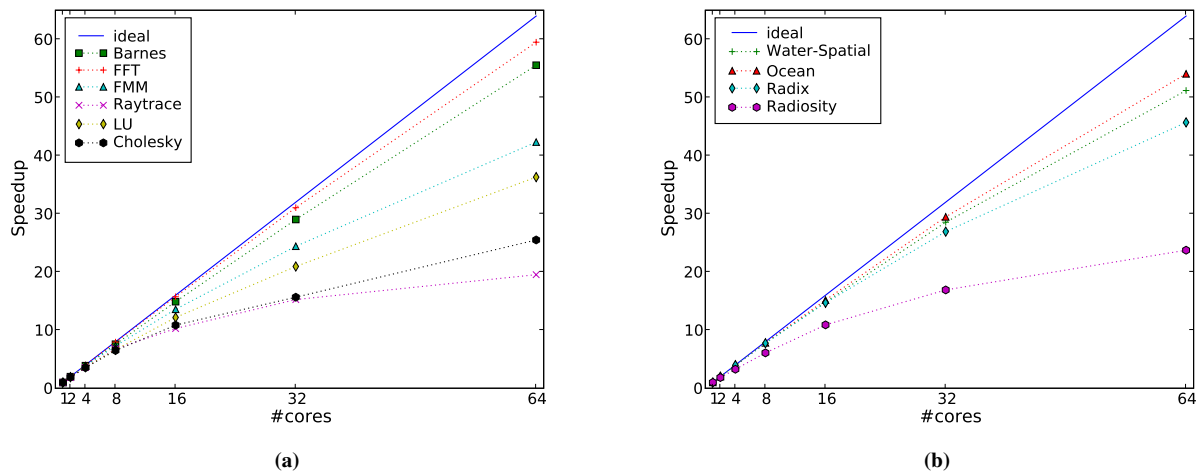


Figure 7: SPLASH-2 scaling on the default dataset (DS1) up to 64 cores

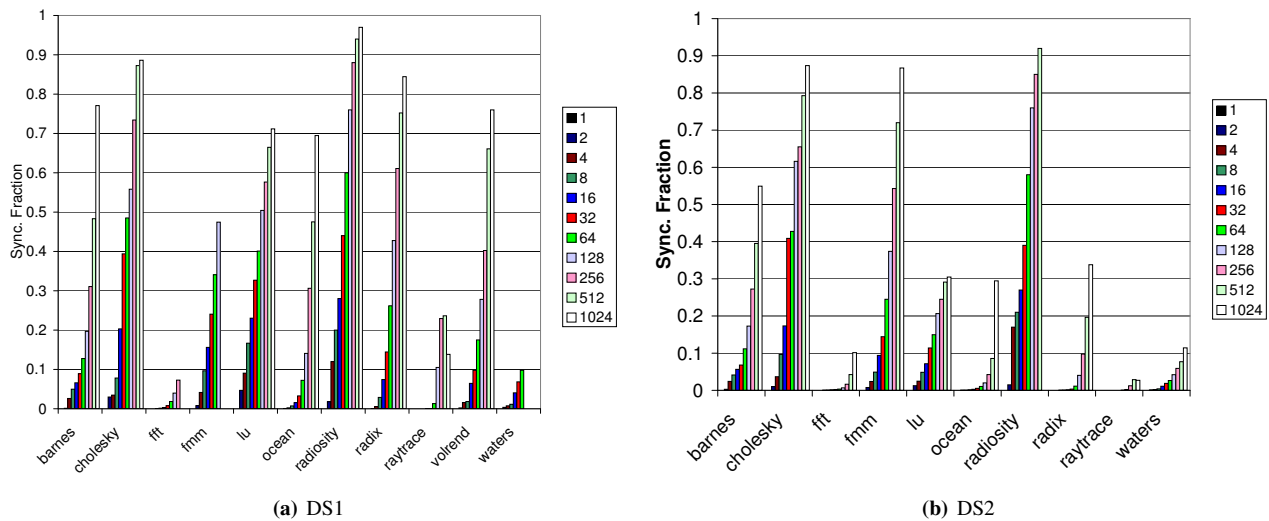


Figure 8: Fraction of the execution time spent in synchronization

ory installed in the simulation host used for the experiments. Approximately ~ 2 GB of memory are used by the baseline full-system simulator (HP Lab’s COTSon + AMD’s SimNow) itself. On top of that, many applications remain in the 3–4GB range and only a few grow up to nearly 8GB.

5 Conclusions

Virtually all next-generation high-performance computing systems will use multi-core parts. Looking at the trajectory of many-core roadmaps, we can easily envision a future with hundreds or thousands of shared-memory cores in a single CPU. One of the challenges in designing these

systems is projecting their performance, but today no full-system simulator is capable of simulating such a large-scale multi-core processor.

In this paper, we present an approach that represents an important first step towards the simulation of chip multiprocessors of an arbitrary number of cores. By converting time-multiplexed threads into space-multiplexed threads, we create a virtual chip multiprocessor in which software threads that were previously mapped to hardware threads are now mapped to CPU models, each of them simulating one of the CMP cores.

Our approach is based on full-system simulation technology which is used to run a many-thread application, whose threads are dynamically mapped into independent

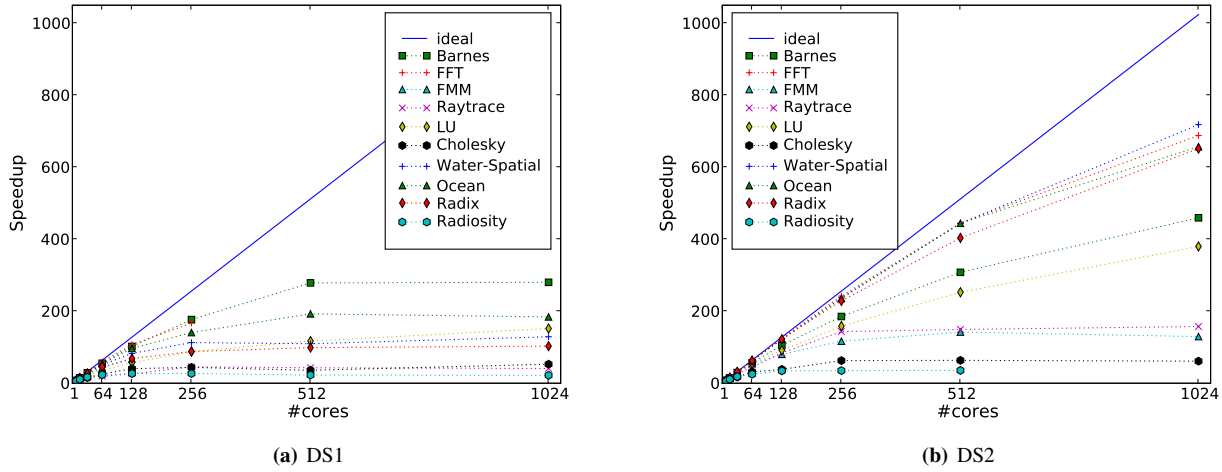


Figure 9: SPLASH-2 scaling up to 1024 cores

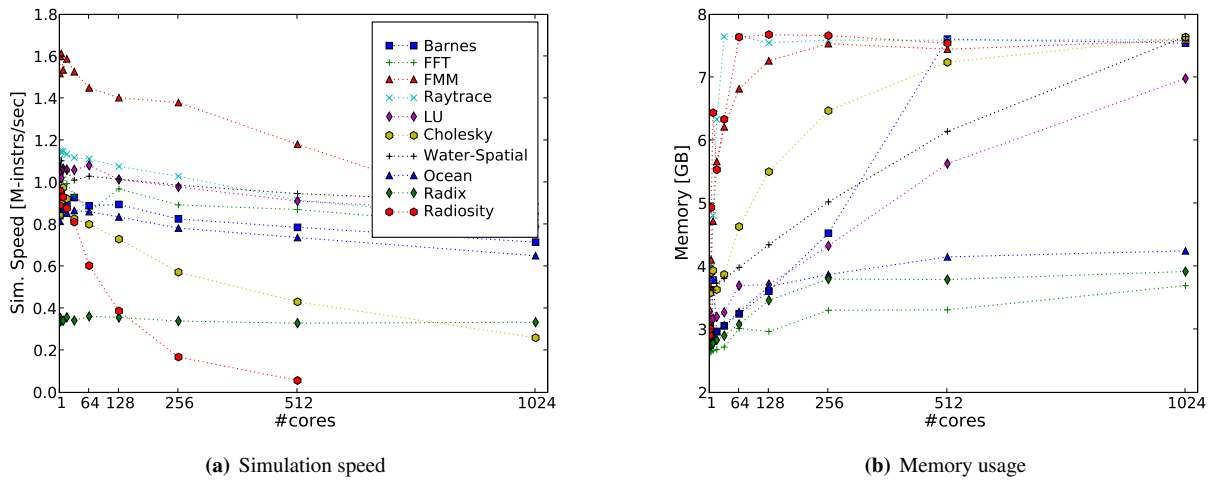


Figure 10: Simulation speed and Memory usage (DS2)

cores. Our experiments show that the simulator can scale up to 1024 cores with an average simulation speed overhead of only 30% with respect to the single-core simulation. We successfully used our approach on the SPLASH-2 benchmark suite, giving meaningful insights on the intrinsic scaling behavior of these applications. We also showed that the original dataset of the SPLASH-2 limits the scalability of these applications for the large number of cores we are targeting. By scaling up the dataset, we achieved a speedup in the range of 600–700 \times for a subset of 4 out of 10 benchmarks, while other applications are limited by either the synchronization or the poor scaling of the dataset itself.

This work highlights important directions in building

a comprehensive tool to simulate many-core architectures that might be very helpful for the future research in computer architecture. If we look beyond simple SPLASH-2-like benchmarks, the biggest challenge we envision is addressing the simulation of applications whose behavior is heavily timing-dependent, especially those relying on implicit synchronization.

Finally, putting together full-system simulation and timing simulation opens up new opportunities for system-level analysis. We plan to investigate the problems relative to the faithful simulation of these aspects within our framework to eventually being able to evaluate the performance implication of the OS and I/O subsystem in the context of many-core architectures.

References

- [1] Ambric. Massively Parallel Processor Array technology. <http://www.ambric.com>.
- [2] AMD Developer Central. AMD SimNow simulator. <http://developer.amd.com/simnow.aspx>.
- [3] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: Infrastructure for full system simulation. *SIGOPS Operating Systems Review*, Jan. 2009.
- [4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, and M. Reif. TILE64 processor: A 64-core SoC with mesh interconnect. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 2008)*, Feb. 2008.
- [6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.
- [7] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. In *Proceedings of the 16th International Symposium on Field Programmable Gate Arrays*, pages 77–86, Feb. 2008.
- [8] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 2007)*, Feb. 2007.
- [9] W. Eatherton. Keynote address: The push of network processing to the top of the pyramid. In *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS)*, Oct. 2005.
- [10] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 373–382, 1988.
- [11] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. *SIGMETRICS Perform. Eval. Rev.*, 21(1):146–157, 1993.
- [12] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell’s multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.
- [13] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. *Comput. Archit. News*, 33(4):24–33, 2005.
- [14] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP\$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MOBS’08)*, 2008.
- [15] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 244–253, 1991.
- [16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.
- [17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002.
- [18] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset. *Comput. Archit. News*, 33(4):92–99, 2005.
- [19] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. *SIGMETRICS Perform. Eval. Rev.*, 30(1):108–116, 2002.
- [20] C. McCurdy and C. Fischer. Using Pin as a memory reference generator for multiprocessor simulation. *Comput. Archit. News*, 33(5):39–44, 2005.
- [21] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60, 1993.
- [22] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. <http://sesc.sourceforge.net>.
- [23] J. Singh, J. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: methodology and examples. *Computer*, 26(7):42–50, Jul 1993.
- [24] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1994.
- [25] B. Stackhouse. A 65nm 2-billion-transistor quad-core Itanium processor. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 2008)*, Feb. 2008.
- [26] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 2008)*, Feb. 2008.
- [27] J. Wawrzyniek, D. Patterson, M. Oskin, S.-L. Lu, C. E. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, 2007.
- [28] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical FPGA-based framework for novel CMP research. In *Proceedings of the 15th International Symposium on Field Programmable Gate Arrays*, pages 116–125, 2007.
- [29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.
- [30] L. Zhao, R. Iyer, J. Moses, R. Illikkal, S. Makineni, and D. Newell. Exploring large-scale CMP architectures using ManySim. *IEEE Micro*, 27(4):21–33, 2007.