On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels

ALEJANDRO RICO, FELIPE CABARCAS, CARLOS VILLAVIEJA, MILAN PAVLOVIC, AUGUSTO VEGA, YOAV ETSION, ALEX RAMIREZ, and MATEO VALERO,
Barcelona Supercomputing Center and Universitat Politècnica de Catalunya

Simulation is a key tool for computer architecture research. In particular, cycle-accurate simulators are extremely important for microarchitecture exploration and detailed design decisions, but they are slow and, so, not suitable for simulating large-scale architectures, nor are they meant for this. Moreover, microarchitecture design decisions are irrelevant, or even misleading, for early processor design stages and high-level explorations. This allows one to raise the abstraction level of the simulated architecture, and also the application abstraction level, as it does not necessarily have to be represented as an instruction stream.

In this paper we introduce a definition of different application abstraction levels, and how these are employed in TaskSim, a multi-core architecture simulator, to provide several architecture modeling abstractions, and simulate large-scale architectures with hundreds of cores. We compare the simulation speed of these abstraction levels to the ones in existing simulation tools, and also evaluate their utility and accuracy. Our simulations show that a very high-level abstraction, which may be even faster than native execution, is useful for scalability studies on parallel applications; and that just simulating explicit memory transfers, we achieve accurate simulations for architectures using non-coherent scratchpad memories, with just a 25x slowdown compared to native execution. Furthermore, we revisit trace memory simulation techniques, that are more abstract than instruction-by-instruction simulations and provide an 18x simulation speedup.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*modeling techniques*; I.6.1 [**Simulation and Modeling**]: Simulation Theory—*systems theory*; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems—*environments*

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Multi-core, simulation, abstraction levels

ACM Reference Format:
Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramirez, A., and Valero, M. 2012. On the simulation of large-scale architectures using multiple application abstraction levels. ACM Trans. Architec. Code Optim. 8, 4, Article 36 (January 2012), 20 pages.
DOI = 10.1145/2086696.2086715 http://doi.acm.org/10.1145/2086696.2086715

This research was supported by the Consolider program (TIN2007-60625) from the Ministry of Science and Innovation of Spain, The ENCORE project (ICT-FP7-248647), and the European Network of Excellence HIPEAC-2 (ICT-FP7-217068). F. Cabarcas was supported in part by Programme Alßan, the European Union Program of High Level Scholarships for Latin America (scholarship No. E05D058240CO). Y. Etsion was supported by a Juan de la Cierva Fellowship from the Ministry of Science and Innovation of Spain.
F. Cabarcas is a professor at the Universidad de Antioquia, Medellín, Colombia.
A. Vega is now at the IBM TJ Watson Research Center.
C. Villavieja is currently a postdoctoral researcher at the University of Texas at Austin.
Y. Etsion is now at Technion - Israel University of Technology.
Contact author's address: A. Rico, Barcelona Supercomputing Center, Jordi Girona 1-3, K2M 102, 08034, Barcelona, Spain, email: alejandro.rico@bsc.es.

DOI 10.1145/2086696.2086715 http://doi.acm.org/10.1145/2086696.2086715

1. INTRODUCTION

Computer architecture research and design are mainly driven by computer simulation, as the execution of a workload on a complex processor microarchitecture can hardly be modeled analytically, and prototyping every design point is prohibitively expensive. However, computer architecture simulation is a slow task: the simulation of a reference benchmark on a detailed model may take several weeks or even months. This fact is worsened with the switch to multi-core processor designs. The overall throughput of the host chip (i.e., the machine on which the simulator runs) is increased with every new generation, but single-thread performance, which is key for simulation tools, does not dramatically improve due to power constraints. Also, the *target-* (i.e., simulated-) architecture models are increasingly complex, as they have to include a growing number of cores, and the performance improvement of the host cores does not keep up with that complexity progression.

Researchers are conscious of this problem and have been proposing techniques to reduce simulation time. As an example, previous works have used sampling [Perelman et al. 2003; Wunderlich et al. 2003] to statistically choose a representative portion of the application, and provide the same insights as a full application simulation but in a much shorter time. Other works [Miller et al. 2009; Chen et al. 2009; Mukherjee et al. 2000] opt for parallelizing simulation to reduce a single-simulation time by running it on multiple threads. This is convenient for design stages requiring to quickly evaluate a single or few design points. Contrarily, when many parameters must be explored, running many serial simulations in parallel provides better simulation throughput.

Such techniques to reduce simulation time are useful for cycle-accurate simulators [Martin et al. 2005; Wenisch et al. 2005]. These simulators are fundamental tools for computer architecture research in order to understand the workload stress on microarchitectural structures, but their time-consuming operation limits their usability to exploring multi-cores with a few tens of cores. In general, cycle-accurate simulators are not suitable for simulating large-scale architectures, nor are they actually meant to be used for this.

Early high-level design decisions, or evaluations of the cache hierarchy, off-chip memory, or interconnect may not need such cycle-level understanding of the architecture, as microarchitecture decisions in those cases may be irrelevant or even misleading. As it is widely suggested by the research community [Yi et al. 2006; Bose 2011], this fact opens the door to raising the level of abstraction of the core model, as is done in previous works [Genbrugge et al. 2010]. Furthermore, it allows to raise the level of abstraction of the application, as it does not necessarily have to be represented as an instruction stream. Most existing simulation tools are tied to this level of application representation: execution-driven simulators need to work with the application instruction stream for functional emulation, as well as trace-driven simulators that model the core pipeline details.

The idea of variable-grain simulation detail is already present in the literature and, in fact, many existing simulators nowadays offer several levels of abstraction. Most of these range from microarchitectural pipeline modeling, e.g., in-order and out-of-order cores; to emulation, or functional simulation. In this paper, we categorize simulation levels of detail and introduce a precise definition of different application abstraction levels. In this effort, we focus on the more "abstract" end of the spectrum, which is necessary for simulating large-scale architectures and where, to date, comparatively little work has been done. Based on the proposed abstraction level categories, we introduce two very high-level simulation modes that are faster than emulation, and that actually provide more insight. These are complemented with two more detailed simulation modes that provide as low-level modeling as instruction-by-instruction

simulation. To benefit from having all four abstraction modes in the same tool, we have implemented all of them in TaskSim, a large-scale many-core simulation framework.

First, to raise the abstraction level beyond the instruction stream representation, we leveraged the representation of parallel applications employed in high-level parallel programming models such as OpenMP, Cilk [Blumofe et al. 1995], and new-generation task-based programming models such as OmpSs [Duran et al. 2011], Intel TBB [Reinders 2007] or IBM X10 [Charles et al. 2005]. In such models, the complexity of parallelism management operations, such as scheduling and synchronization, are hidden from the programmer and, thus, completely decoupled from the application's intrinsic functionality. The resulting abstraction level considers computation periods as single atomic operations of a certain duration, and employs synchronization events and other parallelism related operations to carry out accurate simulations for application scalability analysis. To add more detail to this abstraction level, we extended it with explicit memory transfers. This simple extension allows accurate simulation of multi-core architectures using scratchpad memories, based on the fact that the computation on processing elements working on such local memories is independent of external events. Additionally, an intermediate level of abstraction incorporates traditional techniques from trace memory simulation [Uhlig and Mudge 1997; Lee et al. 2010] to provide fast and accurate simulation of the memory system.

The different TaskSim abstraction levels are compared to the ones in existing computer architecture simulation tools in terms of simulation speed and level of detail. Also, we evaluate their utility and accuracy, and analyze their trade-offs between simulation speed and model detail. The purpose is not to promote any simulator, since they complement each other, as it happens in multi-stage simulation methodologies [Gonzalez et al. 2011]; but to show the advantages of using very high levels of abstraction above the instruction level, and to show the benefits of having them together with low levels of detail in the same tool.

The contributions of this paper are fourfold.

—We present a definition of different application abstraction levels and a discussion on their usefulness for the simulation of multi-core architectures. Section 2 introduces the concepts involved in the different application abstraction levels based on the representation of parallel applications.
—We describe the different architecture model abstractions implemented in TaskSim based on the previous definition of application abstraction levels. Section 3 explains the different architecture model abstractions in TaskSim and provides details on their implementation and use.
—We characterize the different abstraction levels available in popular computer architecture research simulators in terms of simulation speed and simulation detail. Section 4 describes the different abstraction levels in Simplescalar [Austin et al. 2002], Simics [Magnusson et al. 2002] (along with its timing module GEMS [Martin et al. 2005]), and gem5 (which is the merge of M5 [Binkert et al. 2006] and GEMS); and presents an evaluation of their trade-offs between simulation speed and simulation detail along with the abstraction levels presented in this paper.
—We evaluate the utility and accuracy of the different architecture model abstractions in TaskSim. Several experiments are shown in Section 6 to evaluate the different TaskSim model abstractions.

2. APPLICATION ABSTRACTION LEVELS

Historically, various high-level simulators [Mukherjee et al. 2000; Badia et al. 2003; Tikir et al. 2009] have used event-driven simulation to achieve early design decisions

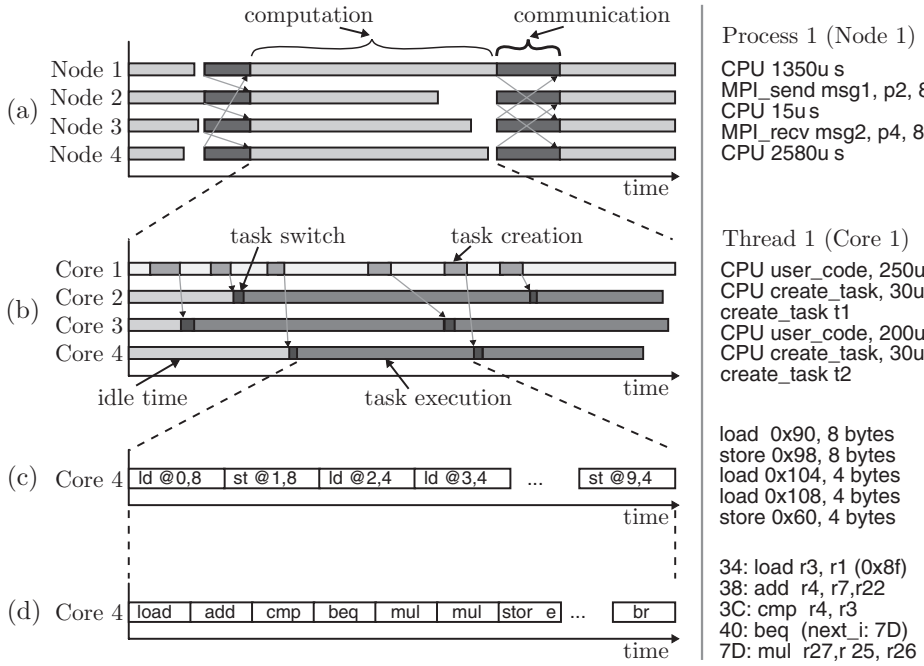


Fig. 1. Different application abstraction levels: (a) computation + MPI calls, (b) computation + synchronizations, (c) memory access list, and (d) instruction list.

and high-level insights for architectures which cycle-accurate simulation was unfeasible. An example of this kind of high-level computer architecture simulators is Dimemas [Badia et al. 2003]. It models distributed-memory clusters with up to thousands of cores, and its main target is parallel application analysis. Dimemas simulations mainly focus on reflecting the impact of different cluster node configurations and interconnection qualities on MPI communications. As different processes in an MPI application execute on different nodes that do not interfere among each other, only MPI communication has to be modeled for a proper network contention simulation. Thus, Dimemas abstracts the application computation on a processor core, as computation periods of a certain duration, and then accounts for all MPI calls for the modeling of communications, synchronization and data transfers.

Figure 1(a) shows the timeline of an MPI application. The application is divided into computation and communication periods, both referred to as *bursts*. On the right of the timeline, a sample of a Dimemas trace is shown. The contents of a Dimemas trace follow the aforementioned application structure: computation bursts are expressed as `CPU` events, and there is a specific event for each MPI call (e.g., `MPI_send`, `MPI_recv`).

For the simulation of smaller systems like a multi-core chip or a shared-memory node, the Dimemas approach is not appropriate. Different threads in a shared-memory application make use of common resources to access the shared address space, although they execute on separate cores. In that case, the level of abstraction must be lowered to be capable of observing the internals of the computation in shared-memory applications, or between inter-process communications in MPI applications.

In order to represent computation at that shared-memory level, applications can be expressed as in high-level programming models such as OpenMP, Cilk [Blumofe et al. 1995], and more recent task-level programming models such as OmpSs [Duran et al. 2011], Intel TBB [Reinders 2007] or IBM X10 [Charles et al. 2005]. In these models,

the internals of the parallelism-related functionalities, including scheduling and synchronization, are hidden and just exposed to the programmer or compiler through an API that will be invoked at runtime. Therefore, the user just has to focus on coding the intrinsic application behavior and, in some cases, annotating the resulting sequential code, or, in others, calling the necessary routines for parallelism-related operations. This strategy ends up completely decoupling the intrinsic application behavior, which is independent of the sequentiality or parallelism of the actual implementation, from the parallelism-management code, which is, in fact, encapsulated in the associated runtime system software.

Figure 1(b) shows the timeline of a task-based parallel application, which could also be part of an MPI application, as it occurs in the Roadrunner supercomputer [Barker et al. 2008] and other systems featuring GPGPUs. As for Dimemas, computation bursts are expressed as single `CPU` events, and are interleaved with synchronization and other parallelism-management operations, and, additionally, explicit memory transfer operations (for distributed-memory multi-cores), which all determine the execution interleaving on different threads. Using this approach, different tasks can be assigned to different cores during simulation, as long as they maintain the synchronization semantics expressed in the trace. For instance, a new task cannot start executing on an available core until it has been created and scheduled. Similarly, a task cannot start until its input data set is available for access. In the example in Figure 1(b), *Core 1* creates and schedules tasks (gray phases), and cores from 2 to 4 execute those tasks (phases between dark gray) as they are being created by *Core 1*.

This level of abstraction is appropriate for quickly evaluating application scalability: whether an application can make use of an increasing number of cores; and memory bandwidth requirements: whether data is transferred fast enough to cores to avoid slowdowns. It is worth highlighting that, for accelerator-based systems where computation cores work on non-coherent scratchpad memories (e.g., GPUs, embedded), this abstraction level includes all information required for an accurate evaluation of the memory system. Since such accelerators only access data in their scratchpad memories, the computation is not affected by any other components in the system. This means that, as long as the accelerator core and scratchpad memory features remain unchanged, modifications to the rest of the memory system and the interconnection network will not affect the accelerator computation delay. Thus, modeling data transfers between off-chip memory, or last-level cache (if any), and scratchpad memories is sufficient for an accurate simulation.

However, for cache-based architectures this approach has, as is intended, some limitations in favor of simulation speed. The level of abstraction must be lowered to simulate cache-based architectures accurately, so the memory system is stressed with the memory accesses performed by the application. Figure 1(c) shows a list of memory accesses corresponding to a specific chunk of computation. This kind of representation was widely used by trace memory simulation methods [Black et al. 1996; Uhlig and Mudge 1997] that benefited from trace reduction techniques (e.g., trace stripping [Puzak 1985; Wang and Baer 1990]) to speed up simulation. More recently, some works [Lee et al. 2009; Lee et al. 2010] have proposed simulation techniques to accurately simulate the performance of in-order and out-of-order cores using memory access traces at this level of application abstraction.

Additionally, when it is necessary to explore microarchitectural issues concerning the core execution pipeline, or a more detailed understanding of the application execution is required, the simulated application needs to be represented at the instruction level. Figure 1(d) shows a list of instructions pertaining to a computation burst. This is the lowest level of representation of an application, and it is the one used in execution-driven simulators, such as Simics or gem5, and in many trace-driven simulators, such

as IBM's Turandot [Moudgill et al. 1999]. As previously mentioned, this representation level allows the understanding of detailed activity on the microarchitecture, but prevents the exploration of large-scale multi-cores due to its time-consuming operation.

3. ARCHITECTURE MODEL ABSTRACTIONS

The application abstraction levels shown in Figures 1(b), 1(c) and 1(d) are employed to implement several architecture model abstractions in TaskSim. To work with such application abstraction levels, we found appropriate the use of event-driven simulation and the employment of traces for all levels, so functional emulation is not required at simulation time, and the application can be actually abstracted beyond the instruction stream level. This has some limitations, as having to manage potentially large trace files. Also, traces cannot capture timing-dependent behavior (e.g., varied I/O timing when modeling I/O components), and cannot capture wrong-path instructions for precisely simulating the effects of speculation on branch mispredictions.

Another drawback of trace-driven simulation is its limitation for appropriately simulating multithreaded applications, as the execution interleaving on different threads may vary across different architecture configurations, which is not possible to reproduce at simulation time if the application behavior was statically captured in traces. We employ the approach presented in a previous paper [Rico et al. 2011] to overcome this issue and be able to simulate the execution of multithreaded applications on the many-core architectures of interest (see Section 3.1 for more details on this method).

The use of traces also has some advantages over execution-driven simulation. For instance, trace-driven simulation provides more flexibility, due to supporting different ISAs without the need of recompilation. Also, it has much lower memory requirements because it does not need to manage application data; and allows working with available application traces, for which sources may not be accessible due to licensing issues. However, the main advantage for this work is the possibility to simulate at different application abstraction levels and also the capability of improving the already high simulation speed using event-driven simulation optimizations.

The different application abstraction levels allow the establishment of four architecture model abstractions as follows. The application abstraction level in Figure 1(b) gives the basis for a mode that only accounts for computation bursts and synchronizations, referred to as *burst*; and to a second mode that extends *burst* with explicit memory transfers, referred to as *inout*. The abstraction level shown in Figure 1(c) is employed in the *mem* mode of TaskSim for trace memory simulation; and, lastly, the application abstraction in Figure 1(d) is used to provide the *instr* mode for instruction-level simulation.

The *burst* mode operates on a set of cores that just execute the computation bursts assigned to the associated threads, and carry out their synchronization through the parallelism-management operations in the trace. Therefore, simulation of the memory system and core pipeline is not necessary, so they are not even present on simulations in *burst* mode. However, for all other modes (*inout*, *mem* and *instr*), the cache hierarchy, network-on-chip and off-chip memory are simulated for a proper timing of memory accesses. More details on the implementation of these four architecture abstractions are given in the following sections.

3.1. Burst Mode

The *burst* mode implements a fairly simple event-driven core model. There is one trace for the *main* task, corresponding to the starting thread in the program, and then one trace for every other task executed in the application.[1] A task trace includes

[1]This mode also operates with loop-level parallelism, but we limit the discussion to task-based applications for simplicity. For loop-based parallel regions, separate traces are captured for different iterations of the application.

all computation bursts as `CPU` events, and the required synchronization events for a correct parallel execution. Since different tasks have separate traces, they can be individually scheduled to available cores. The parallel work scheduling can be done either *fixed* or *dynamic*. The difference is that fixed scheduling follows the same order as in the original run, while dynamic scheduling employs an external runtime system to re-execute scheduling for the simulated architecture, as is explained later.

For fixed scheduling, as soon as a task is created and scheduled, any available core can start its execution. This mechanism is implemented using a *semaphore*-like strategy, i.e., *signal* and *wait* operations. After a task creation burst, a *signal* event unblocks the execution of the corresponding task, which is assigned to a core as soon as it becomes free.

The same signal-wait mechanism allows the simulation of thread synchronizations such as *task wait* (waiting for all previously-created tasks), *wait on data* (waiting for a producer task to generate a piece of data) and *barriers*.

The trace format for the *burst* mode using fixed scheduling includes the following event types.

—`CPU` indicates the beginning of a computation burst. It includes a computation type id and its duration in nanoseconds.
—`Signal/Wait` is used for thread synchronization. It includes a *semaphore* id.

The core model using this trace format operates as follows. One of the simulated cores starts simulation by reading the first event in the *main task* trace. Computation-burst events are then processed by adding their duration to the current cycle value in the target architecture (a ratio can be applied to burst durations to simulate different core performance levels). The rest of the cores in the system start idle, and wait for tasks to be created and scheduled. Whenever a signal event for task creation is found in the *main task*, any idle core can start the corresponding task execution. The core taking the new task becomes active, and starts processing its events.

For simulations using dynamic scheduling, TaskSim adopts the approach presented in a previous work [Rico et al. 2011]. The simulator employs an interface to a runtime system that executes the parallelism-management operations for the application "running" on the simulated machine. In this case, the *Signal* and *Wait* event types are not necessary, and the trace includes, instead, the calls to the appropriate runtime system operations. When runtime call events are found in the trace, the runtime system is invoked to execute them, and perform the corresponding scheduling and synchronization operations. These operations are executed based on the state of the simulated machine, and task execution is simulated according to the decisions made by the integrated runtime system software. More information about this method can be found in the associated publication [Rico et al. 2011].

3.2. Inout Mode

The *inout* mode builds on top of the *burst* mode to provide precise simulation of multicore architectures using scratchpad memories (non-coherent distributed shared memory systems). As previously mentioned, the execution of a core accessing a scratchpad memory is not affected by the features of other elements in the system. The access to a scratchpad memory is deterministic, and the core will only be delayed on eventual synchronizations with explicit memory transfers between the scratchpad memory and the cache hierarchy levels, or off-chip memory.

The trace format of the *burst* mode is enriched with specific events that indicate the initiation of explicit memory transfers (e.g., DMA), and the synchronization with their completion before bursts reading that data.

—DMA is used to initiate an explicit memory transfer (includes the data memory address, size and a tag id).
—DMA_wait is used to synchronize with previously initiated memory transfers (includes tag id).

The application simulation in *inout* mode works as in the *burst* mode but, when a DMA event is found, that core sends the proper commands to program the memory transfer in the corresponding DMA engine. The simulation of that memory transfer takes place in parallel with the processing of computation bursts. Then, on a DMA_wait event, the core stalls until it receives acknowledgement of the completion of the corresponding memory transfer.

The core model in both *burst* and *inout* modes is intentionally simplistic. To avoid the slow detailed simulation of pipeline structures, the target core is assumed to feature the same characteristics as the underlying machine on the traced execution (or a relative performance using ratios for computation bursts). Despite this simplicity, the isolation of computation on cores with scratchpad memories results in accurate memory system and interconnection network evaluations that, for large target accelerator-based architectures, complete in a few minutes, as is shown in Section 6.2.

3.3. Mem Mode

For appropriately stressing the cache hierarchy elements and off-chip memory on cache-based systems, the *mem* mode considers the memory accesses performed by the application. The *burst* mode is extended to add trace memory simulation to the computation bursts in the target application. For each computation burst, a trace of all corresponding memory accesses is captured and a link[2] to the resulting memory access trace is included in the associated CPU event. Then, at simulation time, when a CPU event is processed, the computation burst duration in the trace is ignored, and the corresponding memory access trace is simulated instead.

The trace formats of trace memory simulation works in the 80s and early 90s, included just the memory accesses, as the timing of the computation core was not considered, and only the cache and memory system was simulated. However, in order to model the core performance, the memory access trace also includes the number of non-memory instructions between memory accesses, thus leading to a trace format analogous to the one presented in a previous paper [Lee et al. 2010]. The records in the trace include the following information related to each memory access:

—access type: Load or Store,
—virtual memory address,
—access size in bytes,
—number of instructions since the previous memory instruction.

This information is employed in the core model to simulate the performance of an out-of-order core by modeling the re-order buffer (ROB) structure; a technique introduced in a previous paper [Lee et al. 2009], and referred to as ROB Occupancy Analysis. Using this method, the simulation runs as follows. When the simulated core reaches a CPU event, it starts processing the associated memory access trace referenced by the event record. It reads the first memory access in the trace, increases the cycle value by the number of previous instructions factored by a given CPI value, and generates the memory access that is sent to the first-level cache. Simulation keeps processing the following memory accesses in the trace in the same manner, allowing several concurrent memory accesses in flight. However, to avoid performance overestimations

[2]Currently, we employ the trace file name as a reference to the memory access trace of a computation burst.

and account for resource hazards, simulation considers the limitation imposed by the ROB size. For every memory access, the associated number of previous instructions and the memory instruction itself, are stored in the ROB. Then, when the ROB is full, no more trace records are processed, and only the memory accesses in the ROB are allowed to be concurrently accessing the memory system. Then, when a memory access has completed, the corresponding ROB entries are freed, and if there is enough space, the next memory access in the trace is processed.

This technique assumes that all memory references are independent. However, for many applications, data dependencies may be the primary factor limiting memory-level parallelism. Memory access traces can then be enriched with data dependencies and only allow independent accesses to be sent in parallel. This extension would provide a closer approximation to instruction-level simulation, and, although it is not yet included in the results shown in this paper, it is expected to add some complexity to the tracing procedure, but hardly affect simulation time. The tracing tool must tag memory references and track the dependency chains while, during simulation, it just requires to check if the tag of the preceding dependency has completed.

This method is, in any case, faster than working at the instruction level, but we further speed up simulation by applying trace stripping [Puzak 1985; Wang and Baer 1990] to TaskSim memory access traces. This technique consists of simulating the memory access trace on a small direct-mapped filter cache, and only storing misses and the first store to every cache line (only required for the simulation of write-back caches). The resulting traces experience reductions in size of up to 98%, depending on the application, and will perform the same number of misses as the original trace on simulations of caches with a number of sets equal or larger to the number of sets of the filter cache. A limitation of this technique is that the cache line size of the simulated architecture must be the same that was used in the filtering process for the filter cache.

The results in later sections show that this architecture model abstraction provides a high simulation speed that is 18x faster than simulating at the instruction level.

3.4. Instr Mode

In order to simulate core activity more accurately on systems with caches, the *instr* mode extends the *burst* operation mode to allow the employment of instruction traces for computation bursts, similarly to the extension applied for the *mem* mode. The instruction flow is captured at tracing time using PIN [Luk et al. 2005], a dynamic binary instrumentation tool, and an instruction trace file is generated for every computation burst. Then, as for the *mem* mode, the `CPU` event format is extended to include an extra field for storing a reference to the corresponding instruction trace.

The core model in instruction mode operates exactly as in *burst* mode, but the duration of `CPU` events is ignored, and the corresponding instructions are simulated instead. There are many ways to encode an instruction trace but, in general, they require the inclusion of the following information for every instruction:

—operation code, which can be more or less generic depending on the target ISA and the detail of the core model (e.g., *arithmetic* rather than the specific instruction opcode);
—input and output register lists;
—memory address (and data size): for memory instructions;
—next instruction: for branch instructions.

This mode allows a detailed core simulation. Obviously, the more accurate the core model, the longer the simulation takes to complete. In TaskSim, different instruction-level core models can be used, and even switched from one to another at simulation time, as described in the next section.

Table I. Summary of TaskSim Simulation Modes. This includes their applicability on computer architecture evaluations, and their features also comparing to state-of-the-art (SOA) simulators

Abstraction level	Applicability	Features
Burst	Application scalability Runtime system evaluation	Potentially faster than native execution 100x faster than SOA functional simulators
Inout	Memory system Interconnection network	100x-1000x faster than SOA instruction-level sims Accurate for scratchpad-based CMPs
Mem	Cache hierarchy	10-100x faster than SOA instruction-level sims
Instr	Core microarchitecture	–

3.5. Summary

Table I shows a summary of the presented simulation modes. This includes their applicability to different architecture types or software domains and their usability for computer architecture studies. The *burst* mode allows to evaluate application scalability and, when it works integrated with a runtime system (as described in Section 3.1), it also provides a framework to perform runtime system evaluations, e.g., scheduling policies. On top of these features, the *inout* mode allows to evaluate the memory system and interconnection network designs, and it is accurate for scratchpad-based architectures. The *mem* mode provides support for the evaluation of the memory system and interconnection, as well as the *inout* mode, and also allows to analyze the design of the cache hierarchy components. On top of this, the *instr* mode additionally provides support for evaluating the microarchitecture of the core pipeline structures. Table I also shows an estimate of the simulation modes speed to give an idea of the simulation speed – detail trade-offs. Nevertheless, these trade-offs are more thoroughly evaluated in Section 4.

It is worth mentioning that TaskSim allows to switch between different levels of abstraction for different computation bursts along a single simulation. This is very useful to quickly traverse initial phases of an application in *burst/inout* mode (similarly to *fast forwarding* in some sampling-based simulation methodologies), and then execute the computation bursts of interest (e.g., SimPoints [Perelman et al. 2003]) in *mem* or *instr* mode.

4. SIMULATION SPEED-DETAIL TRADE-OFF

Computer architecture simulators often provide several levels of abstraction with different simulation speeds and levels of detail. Having different abstraction levels is beneficial, not only due to having a faster (usually higher level) or more accurate simulation (usually lower level), but because each of them allows the researcher to reason about certain aspects of the design.

We have chosen three of the most popular computer architecture research simulation tools to analyze their trade-offs between simulation speed and simulation detail for their different levels of abstraction. The first is Simplescalar [Austin et al. 2002]. It became very popular during the early 2000s due to its execution-driven nature, and still nowadays is used for single-processor simulations, many of them being part of reliability studies. The second is Simics [Magnusson et al. 2002], and its timing-module GEMS [Martin et al. 2005]. Simics has been, and still is, largely used in academia for multi-core research even though its primary target is software development. It performs full-system simulation to run unmodified operating systems at a functional level, and provides an interface to incorporate modules for time accounting. One of the most popular timing modules is GEMS, which incorporates Ruby - a component to simulate multi-core and multi-chip memory systems including cache hierarchy,

Table II. Comparison of Abstraction Levels in Existing Simulators in Terms of Simulation Speed and Model Detail

Abstr. level	Speed (KIPS)	Ratio to fastest	Ratio to native	Model detail
		SIMPLESCALAR		
sim-fast	20,425	1	288.07	functional only
sim-cache	5,946	3.43	999.29	sim-fast + cache hier. model
sim-outorder	1,062	19.23	5,669.55	sim-cache + OOO pipeline model
		SIMICS + GEMS		
standalone	32,517	1	128.23	functional only
Ruby	129	251.54	31,827.10	adds cache hier. and mem. model
Ruby + Opal	14	2,237.40	260,017.16	Ruby + OOO pipeline model
		GEM5		
func	984	1	4,381.74	functional only
cache	401	2,45	10,734.61	adds cache hier. and mem. model
inorder	47	20,92	98,335.57	cache + in-order pipeline model
ooo	85	11,80	46,984.21	cache + OOO pipeline model
		TASKSIM		
burst	4,175,762	1	0.80	computation bursts and sync.
inout	132,573	31.50	25.70	burst + explicit mem. transfers
mem	2,067	2,020.32	1,561.27	burst + trace memory simulation
instr	113	36,919.01	31,173.21	burst + core pipeline model

memory controllers, off-chip memory modules and interconnection network. GEMS also includes a detailed out-of-order processor model, referred to as Opal. The third studied simulator is gem5, which is the merge of M5 [Binkert et al. 2006] and GEMS. The gem5 simulator performs full-system simulation and is gaining popularity due to its open source nature (contrarily to the commercial licensing of Simics). It can also work in system-call emulation mode which, for simplicity, is the mode analyzed in this section.

All three simulators are execution-driven, and thus rely on being provided with the executable binary files of the application. This leads them to work at the instruction level of abstraction (Figure 1(d)), but they still provide several levels of model abstraction, ranging from the highest level being functional emulation, and, the lowest, the simulation of the core pipeline structures. All abstraction levels in Simplescalar, Simics+GEMS and gem5 are compared to the ones presented in this paper in terms of simulation speed and modeling detail. As a first observation, the *instr* level in TaskSim considers the instruction stream of the application as well, but *mem*, *inout* and *burst* benefit from a higher-level abstraction of the application representation to provide higher simulation speeds while still being insightful for different purposes.

Table II includes the different abstraction levels and performance of the simulators considered in this study. The results were extracted from executions on an Intel Xeon running at 2.8GHz, with 512MB of L2 cache, 2GB of DRAM at 1333MT/s and running a 32-bit Linux distribution. For each application, simulations were repeated four times, and the fastest was taken for all abstraction levels in order to avoid interference due to operating system activity. All simulators were configured to simulate a single processor, which is the fastest configuration in all cases, since the addition of simulated cores usually leads to a superlinear degradation in simulated cycles per second. A set of scientific applications, listed in Table IV, were used for these experiments, all compiled with `gcc`, `-O3` optimization level and cross-compiled for PISA (Simplescalar), SPARC (Simics), Alpha (gem5) and x86 (TaskSim). The x86 binaries were executed on the aforementioned system to generate traces for the different abstraction levels of TaskSim.

The different abstraction levels of Simplescalar in this study are *sim-fast*, which just performs functional simulation supported by system-call emulation; *sim-cache*,

which just simulates the first- and second-level caches with immediate update and a fixed latency for L2 cache misses (off-chip memory accesses); and *sim-outorder*, which adds to sim-cache the simulation of an out-of-order processor pipeline. The three levels of abstraction correspond to the different binaries generated when compiling the Simplescalar distribution sources.

The highest abstraction level of Simics is its standalone execution (no timing modules, `-fast` option) for functional simulation. The second level is the incorporation of the GEMS Ruby module, set up for a `MOESI_CMP_token` configuration; and the lowest level is the addition of the Opal module. We could have also included an intermediate simulation setup including Opal but not Ruby. However, detailed modeling of processors' pipelines without a model for memory hierarchy does not, in our opinion, provide any additional insight because other simulators do not contain any similar level of abstraction.

As previously mentioned, gem5 is configured to run in system-call emulation mode, and to support the Alpha ISA. The different abstraction levels provide functional simulation (func: no flags are specified); the simulation of the cache system (cache: `--caches --l2cache --timing`); the addition to *cache* of an in-order processor pipeline model (inorder: `--caches --l2cache --inorder`); and the addition to *cache* of an out-of-order-processor pipeline model (ooo: `--caches --l2cache --detailed`). The TaskSim architecture model abstractions were discussed in depth in Section 3.

The second column in Table II shows the simulation speed of the different abstraction levels in kilo-instructions per second, and the third column shows the ratio of the simulation speed versus the fastest mode in the same simulator. The values shown are the average of all applications. As expected, as the level of abstraction is lowered, simulation gets slower. However, simulation speeds significantly vary between simulators due to their different simulation approaches: execution- vs. trace-driven, full-system vs. system-call emulation; and their different levels of modeling detail, e.g., Ruby, gem5 and TaskSim model the off-chip DRAM memory operation in detail, while Simplescalar applies just a fixed latency to L2 cache misses. The differences between the various functional modes of the execution-driven simulators are quite significant. The speed of functional emulation in Simics is noticeably boosted with respect to other levels of abstraction due to its software development target. Even though Simics simulates full-system, it is actually faster than sim-fast, whose model is much simpler. Contrarily, gem5 functional mode is much slower even in system-call emulation mode as, in this case, the target is architecture simulation.

The abstraction levels modeling the core pipeline details and memory system are the slowest in all simulators and their speed ranges from 14 KIPS for Ruby+Opal, which implements a very detailed model, to 1,062 KIPS for Simplescalar which, apart from being not so complex, also benefits from sitting on a simple memory system model. It is remarkable that the inorder model of gem5 is actually slower than the out-of-order model which, intuitively, should not be the case, since an out-of-order core model is more complex. This fact has been consistently found across different applications and different environment setups of gem5.

The forth column shows the ratio to the execution of the application on the host machine. It must be noticed that the comparison of simulation speed between different simulators does not match the comparison of their ratio to native execution. This is because different simulators employed binaries compiled for different architectures that required different numbers of executed instructions to complete.

It is worth highlighting that the TaskSim *burst* mode can be faster than native execution, as was found for the applications in this study, and is 128x faster than the fastest functional mode. This makes sense, as the speed of the *burst* mode depends on the number of events captured in the trace. In this case, the applications were compiled

for task-parallel execution and simulated on a single processor, which is useful for the sake of comparison, but the *burst* mode is more interesting when simulating parallel systems, as is shown in the experiments in Section 6.1. The rest of the modes in TaskSim simulate the memory system, which slows down simulation in favor of simulation detail. The *inout* mode benefits from the aforementioned isolation of execution on accelerators to get a high simulation speed, only 25x slower than native execution, as only explicit memory transfers have to be simulated. Finally, the *mem* mode provides an 18x speedup compared to the *instr* mode. Among the rest of abstraction levels that only simulate the memory system, only sim-cache is faster, which, as previously mentioned, uses a simple model for caches and does not simulate the off-chip memory.

5. SIMULATION SETUP

In this section, we provide details about the simulation setup for the experiments that evaluate the levels of abstraction of TaskSim in Section 6, and review some of the features of the TaskSim simulation environment to better understand the results.

One of the aspects that ease the implementation of different architecture abstraction levels in TaskSim is modularity. TaskSim employs an in-house simulation infrastructure that allows the description of the target architecture as a set of modules and its connections. This functionality eases the independent modeling of different architecture components thanks to their strict modular encapsulation.

As previously mentioned, the core microarchitecture model in TaskSim is intentionally simplistic (as described in Section 3). Instead, the modeling effort is devoted to the memory system –cache hierarchy, scratchpad memories, and off-chip memory– and the interconnection network. To provide a sense of the level of modeling detail in the memory system and interconnection network, a list of the main configuration parameters is shown in Table III. The first level caches are core-private, but shared caches in higher levels of the hierarchy can be split into several banks that can be configured with two different data placement policies [Vega et al. 2010]: *data replication*, and *data interleaving*. The interconnection network topology depends on the target architecture model. The current TaskSim default network module models a customizable all-to-all interconnection with configurable link latency and bandwidth, and maximum number of outstanding transfers. Memory controllers and off-chip memory modules are modeled in a high level of detail to mimic the features of DDR2 and DDR3 DRAM DIMMs.

As it is common in simulators using discrete-event simulation, only the cycles where there is any kind of activity are simulated [Jefferson and Sowrizal 1982]. Thus, in TaskSim, *idle* cycles are skipped, and in cycles with activity, only the modules involved in that activity are actually executed.

Having this infrastructure, TaskSim allows to compose different architectures reusing a common set of modules. Figure 2 shows three examples of target architectures depicted in terms of modules and connections, which are employed in the experiments in Section 6. The first case, in Figure 2, is an SMP configuration with three levels of cache, with a configurable number of cores, L3 cache banks, memory controllers and memory modules. This configuration is employed in Sections 6.1 and 6.3.

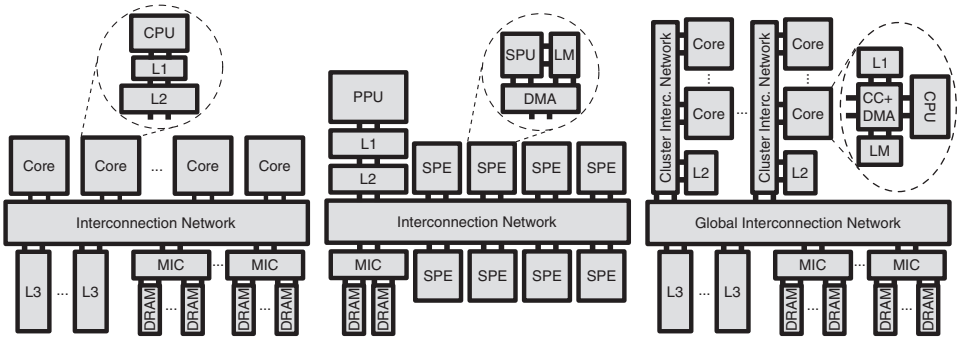
The example in Figure 2 is a model of the STI Cell/B.E. processor [Kahle et al. 2005]. In this configuration, the L1 and L2 caches are configured to mimic the ones available for the Cell/B.E. PPU, the interconnection network module was adapted to provide the same bandwidth and latency as the Element Interconnect Bus, the scratchpad memory (LM) modules were configured as Local Stores, and the DMA engine module was modified to work as the Memory Flow Controller.

The third example, in Figure 2, is the SARC project architecture [Ramirez et al. 2010]. A two-level hierarchical network connects the computation cores in the system to a three-level cache hierarchy and a set of memory controllers giving access to off-chip

Table III. TaskSim Main Configuration Parameters

Parameter	Description (units)	Default Value
Cache (L1/L2/L3)		
Associativity	Number of ways per set	2/4/4
Size	Total cache size (bytes)	32K/256K/4M
Data placement	Replication or interleaving	replication
MSHR entries	MSHR table size	8/64/64
Scratchpad memory		
Size	Total memory size (bytes)	64K
Latency	Access latency (cycles)	3
DMA engine		
Queue size	DMA queue size	16
Maximum transfers	Maximum concurrent transfers	16
Packet size	DMA packet size (bytes)	128
Interconnection Network		
Maximum transfers	Maximum concurrent transfers	number of links
Latency	Link latency (cycles)	1
Bandwidth	Link bandwidth (bytes/cycle)	8
Memory Controller		
Queue size	Access queue size	128
Data interleaving mask	Sets interleaving granularity	4096
Number of DIMMs	Number of DRAM DIMMs	4
DRAM DIMM		
autoprecharge	Enable/disable autoprecharge	disabled
data rate	Transfers per second (MT/s)	1600
burst	Number of bursts per access	8
^tRCD,^tRP,CL,^tRC,^tWR,^tWTR	Timing parameters	[3]

The experiments in this paper use the default values unless it is explicitly stated otherwise.



(a) SMP (b) STI Cell/B.E. (c) SARC architecture

Fig. 2. TaskSim architecture configurations used in the experiments in this paper.

memory. The several last-level cache (L3) banks are shared among all cores, and data is interleaved among them to provide maximum bandwidth. The L2 banks are distributed among different core groups (clusters) and their data placement policy can be configured to optimize either latency (data replication), or bandwidth (data interleaving). Each core has access both to a first-level cache (L1), and to a scratchpad memory (LM). To have deterministic quick access to data, applications may map a given address range to the LM, thus avoiding cache coherence issues. A mixed cache controller and DMA engine manages both memory structures (L1 and LM), and address translation. The STI Cell/B.E. and the SARC architecture configurations are employed in Section 6.2.

[3]Default values for DRAM timing parameters match the Micron DDR3-1600 specification.

Table IV. The Benchmark Applications used in this Paper, and Their Respective Task information

Name	Prog. model	Description	No. tasks	Avg. task runtime
LU	OmpSs	Block Matrix LU decomposition	54,814	464,058 ns
Cholesky	OmpSs	Block Matrix Cholesky factorization	45,760	383,005 ns
Kmeans	OmpSs	K-Means clustering	31,957	400,421 ns
FFT3D	CellSs	Fast Fourier Transform 3D	32,768	13,910 ns

For the experiments in the next sections, we use the set of scientific task-parallel applications listed in Table IV. The table includes the total number of spawned tasks, and the average task duration. The applications written in the OmpSs programming model are compiled with the Mercurium source-to-source compiler [mer 2011] to generate parallel code that employs the NANOS++ [nan 2011] runtime system as the underlying library managing parallelism. The OmpSs applications are instrumented to generate traces for the different abstraction levels of TaskSim following the methodology presented in a previous paper [Rico et al. 2011]. Simulations using these traces employ the dynamic scheduling approach described in Section 3.1.

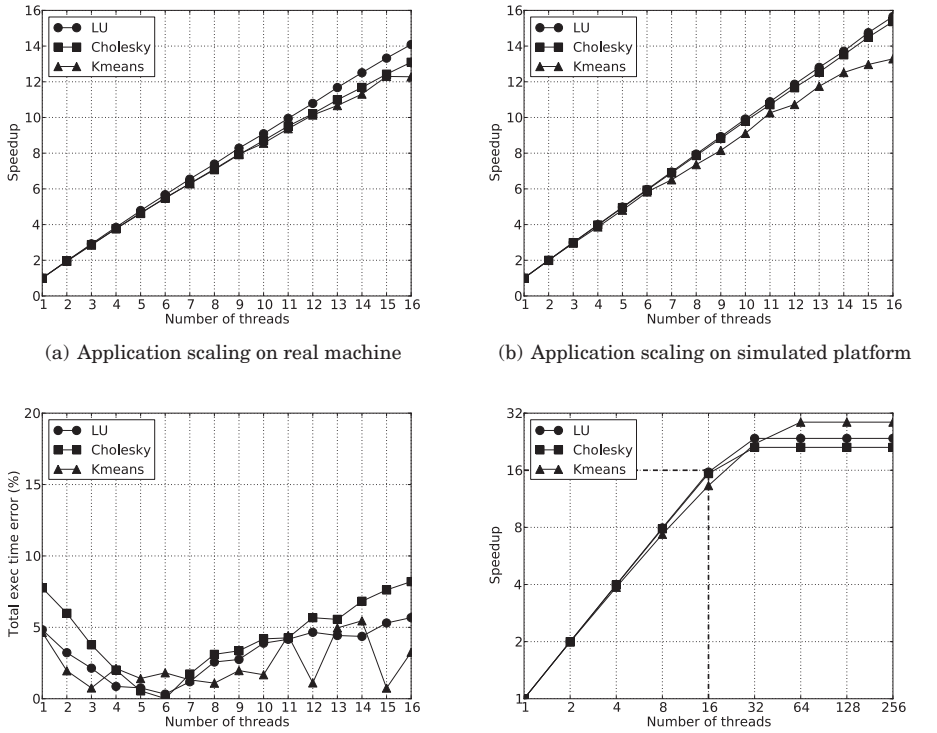
The FFT3D application is also compiled with the Mercurium compiler, but to use the CellSs runtime system [Bellens et al. 2006]. In this case, the fixed scheduling approach, also described in Section 3.1, is the one employed for simulation.

6. EVALUATION

6.1. Application Scalability Evaluation using Burst

The experiments in this section show the validation of the *burst* mode for evaluating application scalability. Simulations are compared to real executions on a 16-core SMP Linux box with four quad-core Intel Xeon E7310 chips running at 1.6GHz, and 16GB of DDR2-667 memory. The applications are first run natively on the real system for different numbers of threads ranging from 1 to 16. Execution time is measured for the computation section, avoiding the initialization part of the application. The executions are repeated four times and the shortest is taken to avoid interference due to operating system activity. Then, the applications are run with instrumentation to generate traces for the *burst* mode. Using those traces, we perform simulations also ranging from 1 to 16 threads. Figures 3 and 3 show the scalability on the real machine and on the simulated platform. As can be seen, the scalability obtained from the simulator is close to the results on the real system. Figure 3 shows the absolute percentage error of the execution time of all applications and different numbers of threads. The difference in execution time between simulation and native execution is in all cases below 8%, and part of that is due to instrumentation overheads to generate the trace, which do not occur in native executions.

Furthermore, we show the projection of scalability for larger systems. Figure 3 shows the scalability of the evaluated applications for architecture configurations with larger numbers of cores up to 256. The dotted rectangle contains the numbers already shown in Figure 3, in which the applications scale well. However, none of them scales beyond 32 processors. Simulation results show that performance is limited in those cases by the rate at which tasks are generated, a problem that has been also found in previous works [Rico et al. 2009]. Task-based applications following a master-worker scheme, have a master thread that generates tasks and a set of worker threads that execute them. If the task generation rate of the master thread is not fast enough to keep up with the workers task completion rate, workers will be idle waiting for tasks. The cores are then underutilized, and application performance does not scale. In this case, the effect starts occurring at 32 processors, and it is limiting performance for 64 and beyond. Our simulations show that, in order to scale beyond 32 and 64 cores, these applications



(a) Application scaling on real machine

(b) Application scaling on simulated platform

(c) Execution time error native vs. simulation

(d) Application scalability for larger configurations

Fig. 3. Validation of *burst* mode for scalability studies.

would require to be tuned by enlarging the task granularity or adopting a parallel task generation scheme with several master threads.

As the experiments show, the *burst* mode is useful and accurate for scalability studies and application analysis. Furthermore, it allows to perform these explorations very fast. The simulations in this section were completed in a few minutes, as the actual simulation was, in all cases, even faster than native execution.

6.2. Accelerator-Based Architecture Evaluation using Inout

In this section we introduce a set of experiments using the *inout* mode of TaskSim. Figure 4 shows several timelines of a Fast Fourier Transform (FFT) 3D application. Figure 4(a) is the real execution on a Cell/B.E. processor. Periods in gray represent computation phases, and periods in black represent time spent waiting for data transfers to complete. The application first performs an FFT on the first dimension of a 3D matrix. FFT execution is computation-bound, so it is dominated by computation time (gray). After the first FFT, it performs a transposition. The matrix transposition execution is memory-bound, so it is dominated by waiting time (black). After the transposition, it performs another FFT but on the second dimension, followed by a second transposition, and, finally, another FFT on the third dimension. The five application stages are clearly identifiable in the timeline. The second transposition takes much longer than the first one, because data is accessed following a different pattern, which does not efficiently use the available off-chip memory bandwidth.

Figure 4(b) shows the timeline of the simulation in TaskSim using the *inout* mode with a Cell/B.E. configuration (Figure 2(b)). As can be seen, the *inout* mode is able to

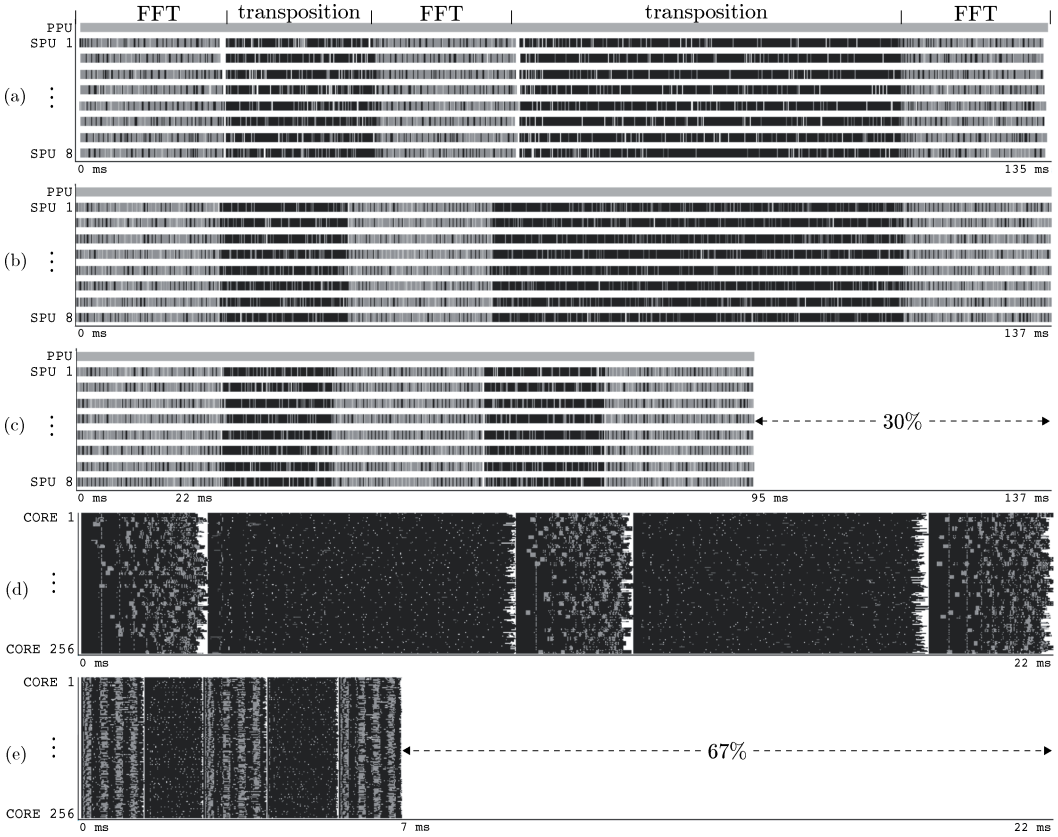
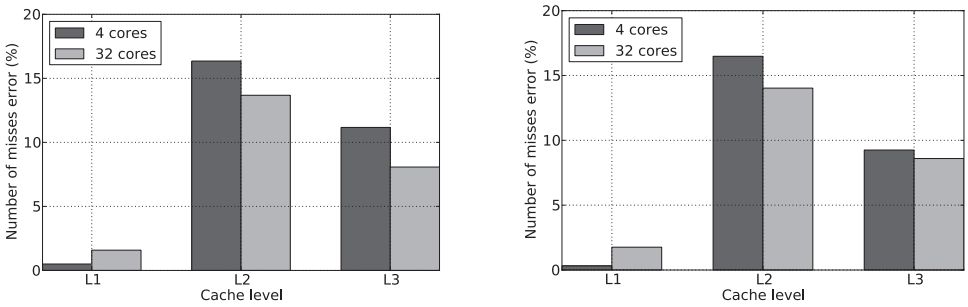


Fig. 4. Inout mode experiments on scratchpad-based architectures using an FFT 3D application: (a) execution on a real Cell/B.E., (b) simulation of a Cell/B.E. configuration, (c) simulation of a Cell/B.E. configuration using a 128-byte memory interleaving granularity, (d) simulation of a 256-core SARC architecture configuration using a 4KB interleaving granularity, and (e) simulation of a 256-core SARC architecture configuration using a 128-byte interleaving granularity. Light gray show computation periods and black shows periods waiting for data transfer completion.

closely reproduce the behavior of the real system. In Figure 4(c), the same simulation takes place but the off-chip memory data-interleaving granularity is set to 128 bytes instead of the default 4 Kbytes in the real Cell/B.E. processor. The time to complete the second transposition is then greatly shortened, resulting in a delay similar to that of the first transposition. This is because, using a 128-byte interleaving scheme, several DIMMs are always accessed in parallel in both transpositions, thus achieving close to peak bandwidth efficiency, and getting a 30% reduction in total execution time.

In Figure 4(d) and Figure 4(e), the same simulations (4KB and 128B interleaving granularities respectively) are carried out using a 256-core configuration of the SARC architecture (Figure 2) excluding the L1 caches. Computation is now spread among many more units, thus reducing the total execution time from 137ms to 22ms. Also, there is much more pressure on the memory system due to more cores concurrently accessing data. Because of this high memory congestion, there is no significant difference between the time taken by both transpositions despite the different access patterns. However, the 4KB scheme still cannot make an efficient use of memory bandwidth. The 128B scheme achieves much higher efficiency, and thus leads to a 67% reduction in total execution time.



(a) 4KB DRAM interleaving granularity (b) 128B DRAM interleaving granularity

Fig. 5. Difference in number of misses between the *mem* and *instr* modes for different simulated configurations using 4 and 32 cores, and different interleaving granularities.

This experiment shows that the problems suffered in small-scale architectures are not necessarily the same as those in large architectures. Also, the potential solutions do not have the same effect at different scales, so the detailed simulation of small architectures is not sufficient for the exploration of future large architecture designs.

The experiment also shows the potential of the *inout* mode to quickly obtain an understanding of the application behavior on different memory system and interconnection network configurations for scratchpad-memory-based architectures. It must be noticed that each 256-core simulation shown in this section required approximately three minutes to complete on an Intel Core2 T9400 running at 2.53GHz.

6.3. Memory System Evaluation using Mem

The following experiments show the error incurred by using trace memory simulation with stripped traces in the *mem* mode. We simulate the SMP architecture in Figure 2 in both *mem* and *instr* modes. The architecture is configured with 4 and 32 cores, and using two different interleaving granularities for main memory: 4KB and 128B. Figure 5 shows the percentage difference of the number of misses between the two modes using Cholesky. The number of misses is summed for all caches in the same level across the architecture. As is expected, the error in the L1 is small despite the use of stripped traces. However, for the L2 and L3 levels, the error is much higher, as the trace stripping algorithm does not account the effects on shared caches for parallel execution on multi-cores. In any case, the error is consistently below 17% for the different numbers of cores and hardware configurations.

For the sake of comparison, we also repeated the simulations in this section using non-stripped memory access traces. The maximum error in this case is 3%, thus showing the inaccuracy incurred when using trace stripping. That sets once again a trade-off between simulation speed and accuracy. On an Intel Xeon E7310 at 1.6GHz, simulations with full traces were 2x faster than instruction-level simulation, while the ones using stripped traces (with an 8KB filter cache) were around 20x faster, which is consistent with the results in Section 4.

7. CONCLUSIONS

In this paper we have analyzed the use of multiple levels of abstraction in computer architecture simulation tools as a base for simulating large-scale architectures. As part of this analysis, we have characterized the levels of abstraction in existing simulation environments. In simulators requiring target applications to be represented as an instruction stream, the highest level of abstraction is functional emulation, which has

been shown to be more than 100x slower than native execution for the highly optimized Simics platform.

Also, we presented a definition of multiple application representations that are more abstract than an instruction stream, and several simulation modes implemented in TaskSim based on these representations. The two highest-level modes in TaskSim, *burst* and *inout*, have been shown faster than native execution and 25x slower, respectively, while being more insightful than functional simulation. The *burst* mode has been proven useful and accurate for scalability studies and application analysis, with an error below 8% compared to native execution. Also, we tested the utility and accuracy of the *inout* mode for architectures employing scratchpad memories. The *inout* mode has been validated against a real Cell/B.E. chip, showing close performance behavior, and it is capable of simulating up to 256-core configurations in less than three minutes. Finally, we revisit trace memory simulation techniques that are incorporated in TaskSim to provide an 18x speedup over instruction-level simulation with a maximum error of 17% on the simulation of the cache hierarchy and memory system.

ACKNOWLEDGMENTS

The authors thank Ana Bosque for her help on getting started with Simics. We also thank Nikola Puzovic, Paul Carpenter, and the reviewers for their guidance and comments to improve the quality of this paper.

REFERENCES

2011. Mercurium Project website. https://pm.bsc.es/projects/mcxx.

2011. NANOS++ Project website. https://pm.bsc.es/projects/nanox.

Austin, T., Larson, E., and Ernst, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer 35,* 2, 59–67.

Badia, R. M., Labarta, J., Gimenez, J., and Escalé., F. 2003. DIMEMAS: Predicting MPI applications behavior in Grid environments. In *Proceedings of the Workshop on Grid Applications and Programming Tools*.

Barker, K. J., Davis, K., Hoisie, A., Kerbyson, D. J., Lang, M., Pakin, S., and Sancho, J. C. 2008. Entering the petaflop era: The architecture and performance of Roadrunner. In *Proceedings of SC '08*. 1:1–1:11.

Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. 2006. CellSs: A Programming model for the Cell BE architecture. In *Proceedings of SC '06*. 86.

Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K. 2006. The M5 simulator: Modeling networked systems. *IEEE Micro 26,* 4, 52–60.

Black, B., Huang, A. S., Lipasti, M. H., and Shen, J. P. 1996. Can trace-driven simulators accurately predict superscalar performance? In *Proceedings of ICCD '96*. 478–485.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not. 30,* 8, 207–216.

Bose, P. 2011. Integrated modeling challenges in extreme-scale computing. *Proceedings of ISPASS'11*.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA '05*. 519–538.

Chen, J., Annavaram, M., and Dubois, M. 2009. SlackSim: A platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News 37*, 20–29.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. 2011. Ompss: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parall. Proc. Lett. 21,* 2, 173–193.

Genbrugge, D., Eyerman, S., and Eeckhout, L. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of HPCA '10*. 1–12.

Gonzalez, J., Gimenez, J., Casas, M., Moreto, M., Ramirez, A., Labarta, J., and Valero, M. 2011. Simulating whole supercomputer applications. *IEEE Micro 31,* 3, 32–45.

Jefferson, D. R. and Sowrizal, H. A. 1982. Fast concurrent simulation using the Time Warp mechanism, part I: Local control. Rand Note N-1906AF, the Rand Corp.

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. 2005. Introduction to the Cell multiprocessor. *IBM J. Res. Dev. 49,* 4/5, 589–604.

LEE, H., JIN, L., LEE, K., DEMETRIADES, S., MOENG, M., AND CHO, S. 2010. Two-phase trace-driven simulation (TPTS): A fast multicore processor architecture simulation approach. *Softw. Pract. Exper. 40*, 239–258.

LEE, K., EVANS, S., AND CHO, S. 2009. Accurately approximating superscalar processor performance from traces. In *Proceedings of ISPASS'09*. 238–248.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., JANAPA, V., AND HAZELWOOD, R. K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI '05*. 190–200.

MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HÅLLBERG, G., HÖGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *IEEE Computer 35,* 2, 50–58.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News 33,* 4, 92–99.

MILLER, J. E., KASTURE, H., KURIAN, G., BECKMANN, N., III, C. G., CELIO, C., EASTEP, J., AND AGARWAL, A. 2009. Graphite: A distributed parallel simulator for multicores. Tech. rep. MIT-CSAIL-TR-2009-056, Massachusetts Institute of Technology.

MOUDGILL, M., BOSE, P., AND MORENO, J. 1999. Validation of Turandot, a fast processor model for microarchitecture exploration. In *Proceedings of IPCCC'99*. 451–457.

MUKHERJEE, S. S., REINHARDT, S. K., FALSAFI, B., LITZKOW, M., HILL, M. D., WOOD, D. A., HUSS-LEDERMAN, S., AND LARUS, J. R. 2000. Wisconsin wind tunnel II: A fast, portable parallel architecture simulator. *IEEE Concurrency 8*, 12–20.

PERELMAN, E., HAMERLY, G., VAN BIESBROUCK, M., SHERWOOD, T., AND CALDER, B. 2003. Using SimPoint for accurate and efficient simulation. In *Proceedings of SIGMETRICS '03*. 318–319.

PUZAK, T. R. 1985. Analysis of cache replacement-algorithms. Ph.D. thesis. AAI8509594.

RAMIREZ, A., CABARCAS, F., JUURLINK, B., MESA, A., SANCHEZ, F., AZEVEDO, A., MEENDERINCK, C., CIOBANU, C., ISAZA, S., AND GAYDADJIEV, G. 2010. The SARC architecture. *IEEE Micro 30,* 5, 16–29.

REINDERS, J. 2007. *Intel Threading Building Blocks*. O'Reilly.

RICO, A., DURAN, A., CABARCAS, F., ETSION, Y., RAMIREZ, A., AND VALERO, M. 2011. Trace-driven simulation of multithreaded applications. In *Proceedings of ISPASS'11*. 87–96.

RICO, A., RAMIREZ, A., AND VALERO, M. 2009. Available task-level parallelism on the Cell BE. *Scientific Program. 17,* 1-2, 59–76.

TIKIR, M. M., LAURENZANO, M. A., CARRINGTON, L., AND SNAVELY, A. 2009. PSINS: An open source event tracer and execution simulator for MPI applications. In *Proceedings of Euro-Par '09*. 135–148.

UHLIG, R. A. AND MUDGE, T. N. 1997. Trace-driven memory simulation: A survey. *ACM Comput. Surv. 29*, 128–170.

VEGA, A., RICO, A., CABARCAS, F., RAMÍREZ, A., AND VALERO, M. 2010. Comparing last-level cache designs for CMP architectures. In *Proceedings of IFMT '10*. 2:1–2:11.

WANG, W.-H. AND BAER, J.-L. 1990. Efficient trace-driven simulation method for cache performance analysis. In *Proceedings of SIGMETRICS'90*. 27–36.

WENISCH, T. F., WUNDERLICH, R. E., FALSAFI, B., AND HOE, J. C. 2005. TurboSMARTS: accurate microarchitecture simulation sampling in minutes. In *Proceedings of SIGMETRICS '05*. 408–409.

WUNDERLICH, R. E., WENISCH, T. F., FALSAFI, B., AND HOE, J. C. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of ISCA '03*. 84–97.

YI, J. J., EECKHOUT, L., LILJA, D. J., CALDER, B., JOHN, L. K., AND SMITH, J. E. 2006. The future of simulation: A field of dreams. *Computer 39*, 22–29.

Received July 2011; revised October 2011; accepted November 2011