Scalable Simulation of Decoupled Accelerator Architectures

Alejandro Rico[1], Felipe Cabarcas[1,3], Antonio Quesada[1], Milan Pavlovic[1], Augusto Vega[1,2], Carlos Villavieja[2], Yoav Etsion[1], and Alex Ramirez[1,2]

[1]Barcelona Supercomputing Center, Barcelona, Spain
[2]Universitat Politècnica de Catalunya, Barcelona, Spain
[3]Universidad de Antioquia, Medellín, Colombia
[*]HiPEAC Network of Excellence

June 3, 2010

Abstract

In this paper we introduce TaskSim, a simulator targeting a new generation of accelerator-based architectures built from a few master processors that off-load computation to specialized processing elements. The task off-load programming model essentially loads data on the accelerator scratch-pad, performs the computation on the local data, and spills out the generated results back to shared memory. We observe that during the computation phase, the accelerator does not interact at all with the rest of the system: interaction is limited to the data transfer before and after the computation itself.

Based on that isolation, TaskSim abstracts CPU phases as a single delay burst instead of simulating the computation in detail. Such abstraction does not introduce any error in the simulation, since the data transfer phases are still modeled in cycle-accurate detail. However, the CPU abstraction greatly reduces the computation demands of the simulator, and allows us to simulate very large systems (with hundreds of accelerators) at a very high detail in a reasonable time.

1 Introduction

Architecture simulation tools are extremely useful not only to predict the performance of future system designs, but also to analyze and improve the performance of software running on well known architectures. However, since power and complexity issues stopped the progress of single-thread performance, simulation speed no longer scales with technology: systems get larger and faster, but simulators do not get any faster.

Simulation speed determines the size of the evaluated architectures and the level of detail at which they can be simulated in a reasonable time. Therefore, existing methodologies performing cycle-accurate simulation of the core microarchitecture, make the simulation of large-scale systems not doable. All in all, alternative methodologies need to be employed to explore the design space of next-generation large-scale multi-core architectures.

Scalability issues, related to the design of a multi-core system, need to be evaluated at their full size: the memory consistency model, the cache coherency protocol, the interconnection network, and the bandwidth of the different units need to be exercised by all the components in the full size architecture. Otherwise, simulation of downsized versions will lead to incorrect scalability assumptions.

In this paper, we introduce TaskSim, an event-driven simulator targeting large-scale accelerator-based architectures whose performance scales to hundreds of accelerators. The key for its scalability is the use of a task-level abstraction for the simulation of accelerators, which leverages their isolation for task execution. Accelerators have their task data loaded in the local memory and operate locally, so external events do not interfere with task execution. Therefore, for evaluations not requiring accelerator microarchitectural modeling, cycle-accurate simulations can be achieved

by the detailed simulation of data transfers and inter-accelerator synchronizations on the shared resources in the architecture (caches, memory and interconnection), and only account for the task execution runtime for a proper timing of those events.

On top of the task-level abstraction, TaskSim is built around an event-driven simulation framework that avoids unnecessary simulation of inactive hardware components and idle time. This is accomplished by skipping *empty* cycles (cycles with no activity) and selectively executing only the hardware components with scheduled activity or receiving external requests in a given cycle. This allows TaskSim to simulate hundreds of accelerators in less than an hour without loss of accuracy for architecture scalability studies.

2 What is new in TaskSim

TaskSim exploits the isolation property of accelerator-based architectures to decouple microarchitecture simulation of the accelerator pipelines from multiprocessor scalability studies. Since task execution in a decoupled accelerator is independent of the rest of the system, we can focus our simulation effort on the rest of the multiprocessor components: the data transfer engines (DMA controllers), the network interfaces, the interconnection network, the shared cache hierarchy (if present), the off-chip memory controllers, and the DRAM modules.

This allows TaskSim to model issues like inter-task synchronization, overlapping of data transfer with task computation, cache sizes, coherency protocols and data migration issues, interconnection bandwidth and topology, and memory latencies. Also, simulating large scale systems, TaskSim can be used to detect and anticipate software issues before many-cores become readily available.

TaskSim is built around the principles of Discrete Event Simulation. We maintain a list of active components with scheduled activity in the future, and jump over idle periods of time, simulating only those components with scheduled activity, or receiving external events. The use of the task abstraction increases dramatically the length of such idle periods, and allows TaskSim to simulate much larger multicore systems than other alternatives simulating accelerators at the instruction level.

In TaskSim, we define a *task* as the computational burst happening between two synchronization events. Among synchronization events, we count inter-task dependency checking (semaphores, locks, messages) and data transfer primitives (DMA get, put, and DMA wait). That is, if an accelerator performs a DMA transfer in the middle of a task, we consider two separate CPU bursts to be abstracted. The counterpart to the CPU burst abstraction is that their execution time must be obtained from a different source.

We consider three different sources where the CPU burst timing could be obtained:

1. From native execution on a real system: the application is instrumented to dump computation burst durations to a trace file on a real platform. We use this approach when modeling upscaled versions of the Cell architecture, or when modeling the use of Intel cores as accelerators.

2. From off-line simulation of the accelerator core: microacritecture analysis of the accelerators can be decoupled from multicore scalability analysis. Computation burst duration can be obtained from a separate cycle-accurate model of the accelerator pipeline. This simulator only needs to model one processor, not all of them.

3. From analytical models: execution time on a projected accelerator could be derived from analytical modeling based on current accelerators for quick approximations based on frequency scaling, superscalar issue width, etc.

All three methods above need to be combined with an application-level trace describing the inter-task synchronizations and data transfers to/from shared memory so that TaskSim can rebuild the execution on the projected muticore, and model the memory and interconnection systems in cycle-accurate detail.

Rebuilding the execution of the application is far more complex than what a simple multicore analytical model would achieve. TaskSim models conflicts on the access to shared resources like caches, buses, and memory controllers. We also model cache and DRAM bank conflicts, cache locality issues, and memory synchronizations

The biggest benefit of abstract CPU burst simulation is that much of the simulated time - which

consist of computational phases local to the accelerators - does not consume any computational resources on the host machine. As a result, most of the simulated cycles are skipped.

For example, simulating a matrix multiplication of 4Kx4K matrices running on a target Cell B.E.-like target incurs only a 400x slowdown compared to native execution. Moreover, simulating a target platform with 256 accelerators, 32 cache banks, 4 MICs, and 8 DRAMs (32x more processors than the Cell B.E.), is only 1.5x slower. Simulation time is almost independent of the target architecture size.

In conclusion, at a time when system scalability concerns become ever more important, TaskSim is a fast simulation platform for systems with hundreds of accelerators.

3 Related Work

There are many multi-core architecture simulators, like M5 [4], Simics [10], PLTSim [16], SimFlex [7] or GEMS [11]. These simulators work at the instruction level, and their popularity is mainly due to their execution-driven nature and their simulation detail. However, simulating the architecture in such level of detail increases the development time of new features and the simulation time and, although they are convenient for the simulation of architectures with a few cores, the simulation time for architectures with more than 16 cores becomes unreasonable.

The first optimization used by some approaches is *sampling*. Tools like SimPoint [13], SMARTS [15] or COTSon [1] use statistical analysis to select representative subsets of a benchmark. Then, only the representative subsets need to be simulated in detail to get results close to the ones from simulating the whole benchmark. That reduces simulation time, but it still requires to run the non-sample parts for a proper warm-up of caches, branch predictors, etc. That increases the range of architectures that can be simulated, but still the number of simulated cores in a reasonable time is limited to a few tens of cores.

Another simulator optimization is to parallelize simulator execution, and let different simulated threads run decoupled in time from each other. Current parallel simulators evolve from existing single thread simulators, such as Parallel Turandot CMP (PTCMP) [5], or from new developments, such as Graphite [12]. This technique works well for decoupled architectures, i.e. architectures with little synchronization between simulated threads. As an example, Graphite runs different nodes (each one including a CPU core, a private cache and a network controller) in parallel, and it does not even synchronize them on accesses to shared resources (global interconnection and memory), but only on explicit core-to-core synchronizations (e.g. barriers). COTSon also parallelizes simulation of different shared memory nodes, synchronizing on MPI messages.

A different approach to deal with simulation speed is to actually implement a prototype on hardware. Existing works like the RAMP project [14] and the SARC project [9], aim at the implementation of the target architecture (or a model of the architecture) on FPGAs. That results in very fast runs of the target evaluations, good for software development, but it involves high economic costs and requires long developments to build the prototypes.

All the previous mentioned works consider modeling at the instruction level. At early design phases, or for some specific architectures, the level of abstraction can be raised to speed up simulation. For example, Dimemas [2] abstracts the shared memory nodes in cluster architectures running MPI applications. This way, only the shared resources among different nodes need to be simulated for different cluster configurations, therefore, scaling up to tens of thousands of nodes. Another example is the work by Genbrugge et al. in [6], in which the design of the core microarchitecture is replaced by an analytical model based on the dispatch width and miss events, e.g. branch mispredictions and cache misses.

In TaskSim, the simulation of cores is abstracted without losing accuracy thanks to the independent nature of the core computation. Since each core works on its local memory, external events do not affect its timing. Therefore, only synchronization events, transfers to other local memories or to main memory, and the associated accesses to shared resources such as caches and interconnection, are modeled for a precise simulation of their interoperability and contention.

The abstraction of the accelerator core also allows TaskSim to skip empty simulation cycles in an event-driven simulation environment. Our event-driven framework differs from the original Time Warp [8] mechanism employed in parallel

discrete-event simulation, in that TaskSim synchronizes all simulated components every cycle, not allowing accelerators to run ahead in time, and thus avoiding the need to checkpoint state and rollback their execution, or sending anti-messages.

4 TaskSim

TaskSim is a trace-driven simulator for accelerator-based multi-core architectures. It targets the simulation of parallel applications coded in a master-worker task offload computational model. As an example, Figure 1 shows a graphical representation of a chunk of a CellSs [3] application execution. Each one of the horizontal bars represents the state of an application thread along time. The different colors represent the computation phase type, such as task generation for the *master* thread (pink), or task execution task for workers (brown). In addition, the trace contains information about the inter-task dependencies, shown as black lines between different threads. That information allows TaskSim to reconstruct the dependencies at simulation time and avoid tasks from starting before they are scheduled or scheduling them before their dependencies are satisfied.

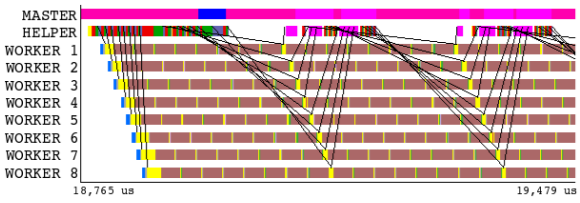

Figure 1: Abstract view of the application in terms of CPU bursts, communications and synchronizations, as simulated by TaskSim.

As previously mentioned, the computational CPU phases (bursts), such as task execution, are not simulated in detail. The burst duration is obtained from the trace file, and is simulated as a single instruction with the same runtime as the whole burst. Contrarily, the trace time for phases involving access to shared resources in the architecture, such as waiting for DMA transfers, are discarded, and their timing is simulated in a cycle-accurate way by means of detailed simulation of DMA controllers, caches, interconnection, memory controllers, and DRAM DIMMs.

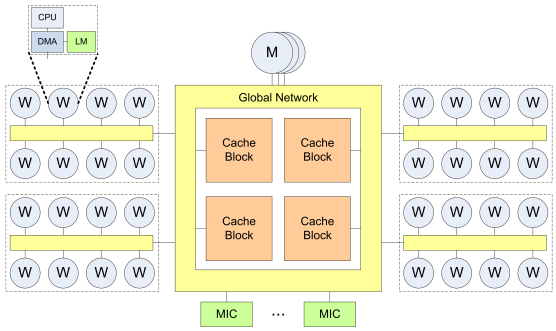

Figure 2: Sample architecture simulated in TaskSim.

Figure 2 shows an example of an architecture simulated using TaskSim. The architecture includes two types of cores: masters (M) and workers (W). Workers are grouped in four clusters, which are interconnected using a hierarchical network to the shared distributed last-level cache blocks, and the memory controllers (MIC). Master processors are also connected to the global network.

In the following sections we present *CycleSim*, the simulation infrastructure below TaskSim; *Time Warp*, the event-driven methodology used to jump over empty cycles; some validations results comparing TaskSim to and 8-SPE Cell; and finally, some simulation speed results.

4.1 CycleSim

CycleSim is the modular simulation infrastructure underneath TaskSim. It is developed in C++, with the aim of achieving maximum cycle-level simulation speed, introducing minimum overheads.

A CycleSim-based architecture simulator is described as a set of modules that connect to each other through ports. All modules are called twice every cycle to perform their the functionality, which is split across the *start* of the cycle, and the *end* of the cycle (Figure 3). The ports interconnecting modules provide time isolation: data written to a port is only readable at the other end after the next half-cycle. Then, the order in which modules are executed does not alter the simulation results.

CycleSim provides two types of ports: *data* ports and *control* ports. Modules communicate data related to memory transfers and synchronizations through data ports. In addition, a control

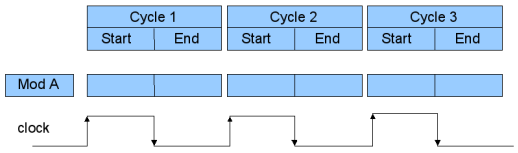

Figure 3: Functionality of modules is split among Start and End of the cycle.

port is usually associated to a data port. They provide boolean values used to implement back-pressure to sender modules, so they can check the receiver status before sending through the associated data port; or to implement reject signals, usually employed by interconnection network modules to reject data coming through a data port, depending on the network state and its routing protocol. Both types of ports are only available to be read by the receiving module in the next half cycle after they were written by the sending module.

CycleSim operates as the main simulation loop that functions as a cycle counter. For every cycle, the infrastructure runs the Start method of all modules, and then the End method of all modules, until simulation completes.

4.2 Event-driven simulation

The main simulation loop runs all the modules in every cycle. However, thanks to the abstraction of the CPU modules, there are many cycles in which none of the modules has anything useful to do: it will not generate any output (because it has nothing to do, or because its pending operation is not ready yet), and it has not received any input.

Figure 4 shows the fraction of empty cycles as a function of the number of simulated processors for a set of test applications. The percentage of empty cycles varies greatly depending on the application. It directly depends on how the application uses the memory system. Applications with lots of computation and little data transfer will have more empty time (eg. CheckLU). Applications with lots of data transfer and small computation bursts have less empty time (eg. TaskFFT).

Given that applications can overlap data transfer and computation through the DMA controllers, high memory traffic and low computation voids the benefits of the CPU abstraction, since most of the activity is on the memory system.

In addition to empty cycles, the use of the CPU abstraction and event-driven simulation also

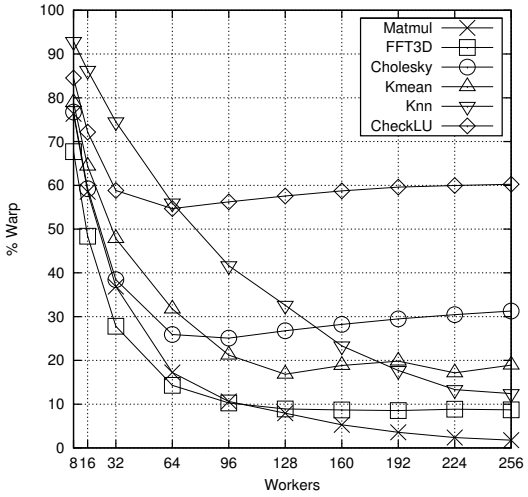

Figure 4: Percentage of empty cycles

makes many of the simulator components idle during most of the simulated time. Figure 5 shows the average number of modules that actually had to be executed in the non-empty cycles. That is, once we have skipped the full empty cycles, we count how many modules were necessary, versus the total number of modules.

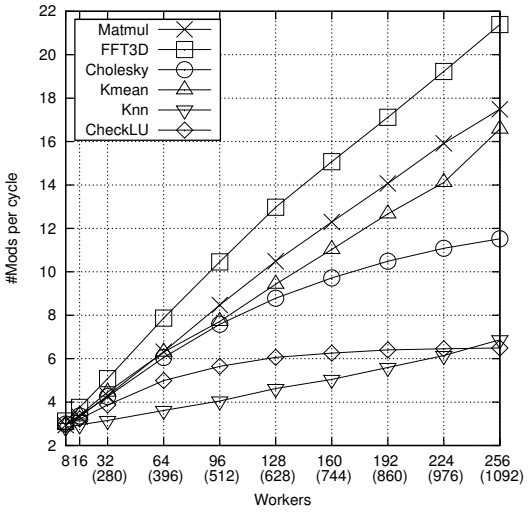

Figure 5: Number of active modules in the non-empty cycles.

Our results show that the number of modules that run in a given cycle is very small compared to the total number of modules. Furthermore, increasing the size of the simulated architecture does not always translate to an equal increase in the number of active modules. The impact of the

CPU abstraction is far more visible in terms of idle modules than in terms of full-empty cycles.

For most applications, the number of active modules is linear with the size of the simulator. For example, TaskFFT (the most bandwidth demanding application) runs 15 modules with 128 processors, and 21 modules with 256 processors. Knn (low bandwidth demand) runs 4 modules with 128 processors, and 7 with 256 processors. The higher the ratio of data transfer to computation, the more active modules.

For the rest of applications, it seems the number of active modules stalls at some point. As we will see in Section 4.5, those are the applications that have limited parallel scalability. Consequently, increasing the number of processors, only increases the number of idle modules. In any case, the number of active modules is very small compared to the total number of modules (20 / 1000, 2%), resulting in a great improvement in simulation time.

For the larger simulator configurations, the total number of modules becomes very high compared to the number of active modules. Even if we don't actually run those modules, we still have to loop through the entire list of modules to check if they need to be executed in a given cycle. Even worse, we have to go through the list twice: at the start, and the end of the cycle.

As a result, the overhead of iterating over the entire list of modules can have a detrimental effect on simulator performance. Given that the overhead of a single iteration is approximately 5 host instructions, simply iterating over a list of 1000 modules requires a 5000 instruction overhead. Executing a module only requires 200 instructions on average. That means that the overhead of the loop is equivalent to executing 25 modules — far more than the typical number of modules that need to be executed in any given cycle. Consequently, we spend more time checking if modules need to be executed than actually executing them.

Considering the overhead of the full loop traversal, we replace the array of modules for a list of *active* modules. when a module receives inputs, it is added to the active list. When a module wants to wakeup, it is added to the corresponding future active list. Maintaining such an active list is more complex than simply iterating over all the modules, but the total overhead is significantly reduced.

4.3 Simulator validation

In this section we present some experiments where we attempt to validate the results of TaskSim comparing it to the Cell B.E., a real accelerator-based architecture. For this experiment, we configure TaskSim to model a single-chip Cell system with 2 master processors (our processor module is single threaded) and 8 accelerators. We disable the TaskSim on-chip cache, and configure the interconnection and memory bandwidth to mimic that of a PowerXCell 8i.

Figure 6 shows a comparison of the real execution on the Cell and the predicted execution time from TaskSim for our set of test applications. Time has been normalized to the real execution time. The set of applications chosen for the simulator evaluation are the same six parallel scientific kernels used throughout the paper.

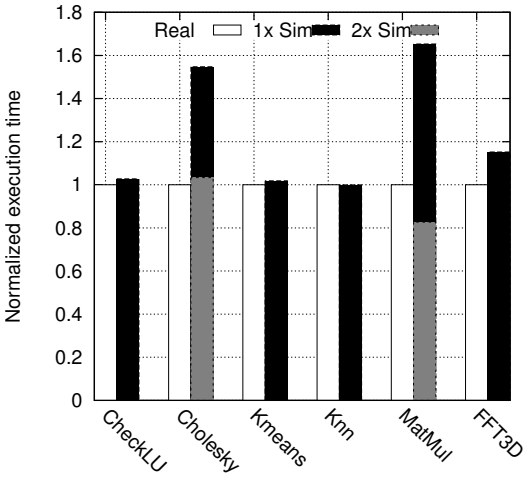

Figure 6: Comparison of real Cell B.E. execution time with the TaskSim prediction.

Our results show a very high simulator accuracy for most applications. As mentioned before, the CPU abstraction does not introduce errors. On the contrary, it eliminates the main source of simulation error: since the CPU timing is obtained from the trace file, the simulator achieves cycle-accurate modeling of the processor pipeline.

However, for MatMul and Cholesky we observe significant simulator deviation from reality. The problem with these applications is that they are dominated by sequences of very short CPU bursts in the Master thread. The other application with significant simulation error is 3D FFT, which is dominated by very short CPU bursts in the worker

processors during the cube transpose phases.

In those cases, a significant part of the prediction error is attributed to the execution overhead introduced by our application tracing mechanism. That is, by instrumenting the application to obtain the traces required to feed TaskSim, we are altering its execution time. This is specially significant in the smaller CPU bursts, since the instrumentation overhead is almost constant. If we make a simulation with an altered trace, we will of course obtain altered results.

For both MatMul and Cholesky we can instruct the simulator to apply a *correction* factor to the timing of a particular class of CPU bursts. In particular, we also show simulation results for MatMul and Cholesky where the short CPU bursts showing higher instrumentation overhead in the Master thread have been simulated as 2x smaller than stated in the trace file.

The results show that this correction factor allows TaskSim to approximate the real system performance. Once we have obtained the correct factor comparing to the real hardware, we can continue using the same factor for the simulation of larger configurations.

4.4 Simulation speed

In this section we discuss the simulation speed of TaskSim, and its scalability in terms of the simulated architecture. Figure 7 shows the simulator slowdown of TaskSim on an Intel Core2 T9400 at 2.53GHz compared to an 8-SPE PowerXCell 8i at 3.2GHz.

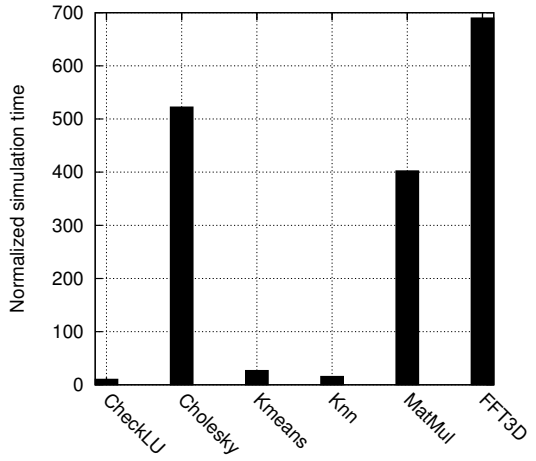

Figure 7: TaskSim simulation time vs. real execution on the Cell B.E.

Our results show that slowdown can go from a mere 5-10x, to 400-600x. This exposes the great impact of the CPU abstraction on those applications that have long CPU bursts and little data transfer.

Table 4.4 shows some relevant information about the applications used throughout the paper: the number of executed tasks, the average task runtime (abstracted by TaskSim), the total problem size, and the estimated average bandwidth for each task. It is interesting to note how the applications with a higher bandwidth demand are those that execute more modules per cycle in Figure 5, and the ones with higher simulator slowdown.

Kernel	**# Tasks**	**Task runtime (μs)**	**Problem size (MB)**	**BW per task (GB/s)**
Check-LU	54.814	45.7	256	1.11
Cholesky	357.760	28.0	512	1.68
FFT-3D	32.768	13.9	128	3.27
K-means	335.872	30.7	195	1.56
K-NN	800.768	7.9	36	0.49
MatMul	262.144	25.8	192	1.42

Table 1: Set of test applications simulated with TaskSim.

Finally, we have also evaluated the simulator slowdown as we increase the size of the simulated architecture. Figure 8 shows the increase in simulation time as a function of the number of simulated worker prcoessors.

Our results show that for most applications, the simulator does not become slower as we increase the size of the simulated target. This is a direct consequence of the event-driven simulation model: the number of operations to perform depends on the application, not on the architecture. If a number of data transfers must happen, it takes the same time to simulate them on one DMA controller, than spread across 16 controllers.

Most applications only experience a slowdown of 1.5x when going from 8 to 256 processors. The worst case is FFT, which gets 4x slower. This time overhead is due to congestion on shared resources, and operations that must be re-issued after being rejected by a busy component.

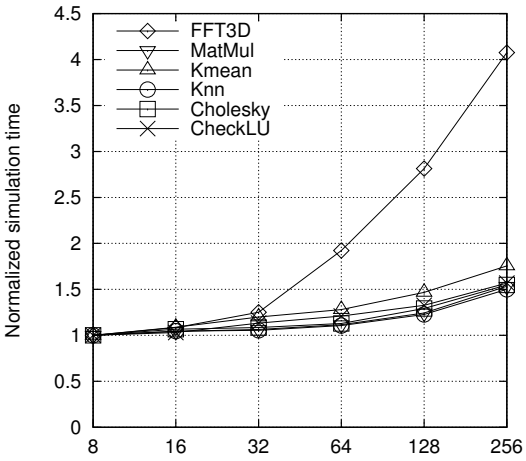

Figure 8: TaskSim slowdown as a function of the number of simulated processors.

Our results show that, thanks to the CPU abstraction we exploit in accelerator-based architectures, it is possible to simulate a CMP with hundreds of processors in cycle-accurate detail in a very reasonable time, compared to the simulation cost of cycle-accurate out-of-order processor models.

As simulation is completely CPU bound, a faster CPU immediately translates to faster simulation speed. Using TaskSim we can simulate applications that take 2-3s of real execution time on 8 processors, and simulate 256 processors in less than an hour, allowing us to quickly evaluate different metrics and application behavior such as scheduling, memory management and contention, resource usage, etc.

What is more important, is that we can simulate the entire application, as it works through the full working set. for example, running Cholesky on a lower number of processors would not expose the lack of parallelism in the code. And using a smaller application that lacks parallelism would not expose the low cache locality of the large working set.

4.5 Sample TaskSim simulations

TaskSim uses CycleSim modular flexibility to create a highly configurable simulator. It allows users to configure a large number of parameters of the architecture shown in Figure 2 through a small number of configuration files. As an example of the capabilities of TaskSim, in this section we present a simple study of our accelerator-based architecture scalability. Due to space restrictions, we limit the study to speedup results, however at the end of simulations every module prints its own statistics (like hits on the cache, bank misses on the DRAM, etc.), furthermore, there are configuration parameters to generate statistics traces, which helps us understand transitory behaviors of the applications or the architecture.

Figure 9 shows the applications scalability from 8 to 256 processors under ideal conditions: 32 MICs that provide a 819.2 GB/s bandwidth, master processors that are 128 times faster than the original, and no cache. This configuration allows us to test the applications, eliminating most bottlenecks like memory bandwidth, task generation speed. The figure shows that while four of the applications scale well to 256 Processors, the other two (CheckLU and Cholesky) only scale to 128 and 64 processors respectively. These two applications have inter-task dependency problems which can be observed through processors execution traces like the one shown in Figure 1.

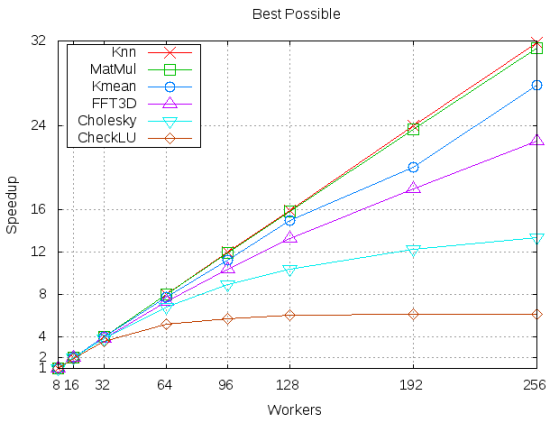

Figure 9: Applications scalability.

Having verified that there are four scalable applications, we tested the applications bandwidth requirement for 256 processor configuration. For this purpose we configure the simulator with 4, 8, 16, and 32 MICs. Given that each MIC contain 2 DDR3-1600 DRAMs with 12.8 GB/s peack bandwidth, the experiments were performed for 102.4, 204,8, 409.6 and 819.2 GB/s. As can be seen in Figure 10 Knn, and CheckLU do not require more than 409.6 GB/s, while the other applications benefit from the 819.6 GB/s. This results showed us that this architecture without a redesign of the

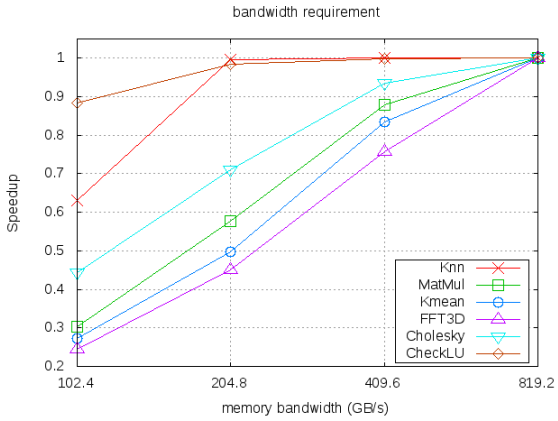

Figure 10: Applications' bandwidth sensitivity.

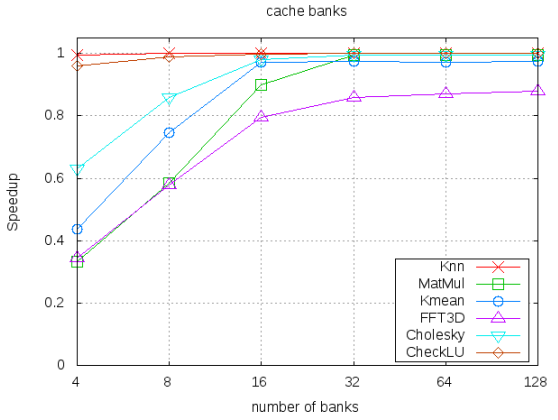

Figure 11: Cache banks for a total size 512MB.

memory system would not be useful for 256 processors; since it would not be possible, because chips pin restrictions, to install 64 DRAM modules to feed the processors with data fast enough.

As a possible solution, for the bandwidth problem, a cache could be added to the system. Figure 11 shows the result for a distributed cache with 8 to 128 banks and a fix total 512MB capacity, with 4 MICs. The speedup is measured against the 32 MICs experiment and no cache. It can be seen that all applications, except for the FFT3D, achieved almost the same performance than the ideal case. It can also be seen that it is not necessary to have more than 32 banks, because the applications do not require that much bandwidth.

5 Advantages and limitations

The results so far clearly point out the advantages of the TaskSim CPU abstraction model: very few modules need to be executed in a given cycle, and only those modules that have useful work to do are executed.

This leads to a very fast and scalable simulator. Since most of the application execution is abstracted away, large scale data processing workloads can be simulated at full scale. Since the simulation time is independent of the target architecture size, large scale systems can be evaluated for scalability evaluations.

The CPU abstraction is implemented through trace-driven simulation. As discussed in Section 2, the duration of CPU bursts is obtained from real execution or off-line CPU simulation / modeling and then recorded to be used by TaskSim during the system simulation. The need for this separate tracing / modeling stage can be seen as a limitation of the approach. Every time the application changes, it must be traced / modeled again.

Also, throughout this paper we have discussed TaskSim in the context of a task off-load programming model. However, TaskSim is not limited to it. TaskSim can also work with loop partitioning codes with barrier synchronizations (like OpenMP). The actual limitation of the model is that accelerators must be decoupled from the system. That is, we require processors to compute exclusively from their local memory, and use explicit communications to interact with the rest of the system (DMA for data transfer, explicit synchronization messages).

Finally, TaskSim is not mean for CPU microarchitecture studies. As discussed before, cycle-accurate modeling of the processor is expected to happen off-line. In such studies, it should be enough to model a single processor, and the abstraction / simplification usually falls on the memory system side. TaskSim abstracts the CPU, but is very detailed in the modeling of the DMA controllers, the interconnect, and the memory system.

TaskSim was not built for shared memory systems with cahe coherency, or CPU microarchitecture. TaskSim is built for design space exploration of a multicore system built from decoupled components with access to a shared memory, like most embedded Systems on a Chip. It allows dimensioning of the shared memory structures, interconnection topology and bandwidth, inter-module communication needs, and the number and performance of the compute elements.

6 Conclusions

In this paper we have described the TaskSim simulator. We have shown that the CPU abstraction means that most of the simulator components are idle, and do not need to be simulated in most of the cycles.

Using an event-driven methodology, TaskSim is capable of simulating very large architectures with hundreds of accelerators in a very reasonable time. We have simulated a 5-second Cholesky decomposition with hundreds of thousands of tasks, and a 512MB working set, on 256 processors in less than 45 minutes. This allows us to draw solid conclusions on the scalability of the applications and the architecture, that could only be hinted by simulation of downsized working sets or smaller architectures.

The CPU abstraction and event-driven approach not only improves simulation speed, it also leads to a scalable simulator: simulation time depends on the number of operations to be performed, not on the size of the simulated target. Meaning that very large configurations can be modeled in a reasonable time.

We are currently exploring further performance improvements by parallelizing the CycleSim infrastructure itself, and enabling shared memory and cache coherency exploration by allowing instruction level simulation in the CPU.

Acknowledgements

This research was supported by the SARC project (FP6-FET-27648), the FPI research grant (BES-2008-004599) and the Consolider contract TIN2007-60625 from the Ministry of Science and Innovation of Spain, and the IBM-BSC MareIncognito project. Cabarcas was supported in part by the Program AlBan, the European Union Program of High Level Scholarships for Latin America (scholarship No. E05D058240CO).

References

[1] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[2] R. M. Badia, J. Labarta, J. Gimenez, and F. Escal´e. Dimemas: Predicting mpi applications behavior in grid environments. In *GGF '03: Workshop on Grid Applications and Programming Tools*, June 2003.

[3] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 86, 2006.

[4] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[5] James Donald and Margaret Martonosi. An efficient, practical parallelization methodology for multicore architecture simulation. *IEEE Comput. Archit. Lett.*, 5(2):14, 2006.

[6] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA '10: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, January 2010.

[7] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Simflex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, 2004.

[8] D. R. Jefferson and H. A. Sowizral. Fast concurrent simulation using the time warp mechanism, part i: Local control. Rand Note N-1906AF, the Rand Corp., Santa Monica, CA, December 1982.

[9] George Kalokerinos, Vassilis Papaefstathiou, George Nikiforos, Stamatis Kavadias, Manolis Katevenis, Dionisios Pnevmatikatos, and Xiaojun Yang. Fpga implementation of a configurable cache/scratchpad memory with virtualized user-level rdma capability. In *IC-SAMOS'09: Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 149–156, July 2009.

[10] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[11] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[12] Jason E. Miller, Harshad Kasture, George Kurian, Nathan Beckmann, Charles Gruenwald III, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. Technical Report MIT-CSAIL-TR-2009-056, Massachusetts Institute of Technology, November 2009.

[13] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 318–319, New York, NY, USA, 2003. ACM.

[14] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanovic. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, 2007.

[15] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 84–97, New York, NY, USA, 2003. ACM.

[16] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *in ISPASS '07*, 2007.