Multi-Core Cache Hierarchy Modeling for Host-Compiled Performance Simulation

Parisa Razaghi and Andreas Gerstlauer
Electrical and Computer Engineering, The University of Texas at Austin
Email: parisa.r@utexas.edu, gerstl@ece.utexas.edu

***Abstract*—The need for early software evaluation has increased interest in host-compiled or source-level simulation techniques. For accurate real-time performance evaluation, dynamic cache effects have to be considered in this process. However, in the context of coarse-grained simulation, fast yet accurate modeling of complex multi-core cache hierarchies poses several challenges. In this paper, we present a novel generic multi-core cache modeling approach that incorporates accurate reordering in the presence of coarse-grained temporal decoupling. Our results show that our reordering approach is as accurate as a fine-grained simulation while maintaining almost the full performance benefits of a temporally decoupled simulation.**

I. Introduction

The increasing complexity of embedded systems has led a large portion of applications to be implemented in software to reduce development time and risk. In such systems, software is coupled with hardware to tackle real-time performance bottlenecks. In order to keep the development time short, software needs to be evaluated at early design stages. With traditional ISS-based approaches being either slow or inaccurate, so-called host-compiled or source-level simulators have recently emerged as a solution for rapid evaluation of complete system. In such approaches, a faster simulation is achieved by abstracting execution behavior at increased simulation granularity.

However, existing source-level simulators often only focus on application behavior running on a single core while neglecting speed and accuracy effects of system-wide, multi-processor and multi-core HW/SW interactions. This is especially the case in modern multi-core and multi-processor system on chips (MCPSoCs) in which cores and processors interact through shared last level caches (LLC). Due to the significant effects of the system memory hierarchy on real-time performance, fast and accurate modeling of the cache hierarchy is essential for accurate design space explorations.

A large benefit of host-compiled simulators comes from application of temporal decoupling technology. A temporally decoupled simulation method allows threads go ahead of the simulation kernel time by only incrementing a local time. Instead, a global simulated time is updated whenever the actual time is advanced by the simulation kernel. This method thereby reduces the simulation context-switch overhead and provides a faster simulation environment by coarse-grained execution.

However in the presence of temporal decoupling, accurate multi-core cache hierarchy modeling is challenging. The

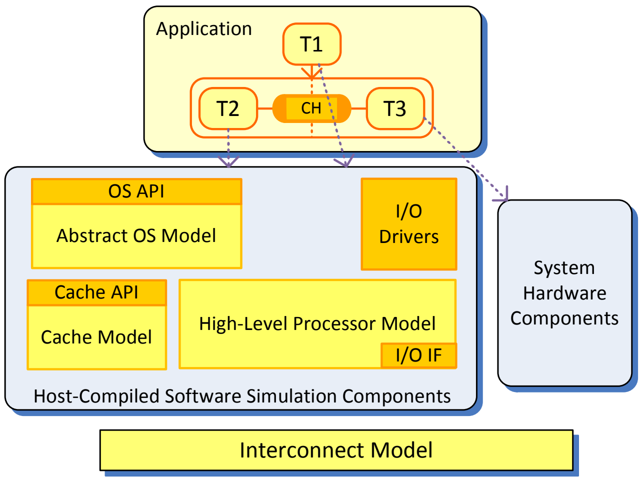


Fig. 1. Host-compile simulation platform.

reason is that memory references can be globally committed out-of-order, which leads to inaccurate cache behavior. In this paper, we introduce a multi-core out-of-order cache modeling approach, which incorporates a delayed reordering of aggregated requests to provide an accurate cache hierarchy simulation in the presence of temporal decoupling. Thus, we propose a multi-core cache model that accurately maintain its internal state at the speed of a fully decoupled simulation.

A. Host-Compiled Software Simulation

In Figure 1, the ingredients of a typical host-compiled software simulator are shown. A host-compiled simulator is composed out of high-level models of the software execution environment. An abstract operating system (OS) at the core of the simulator replicates a standard scheduler to manage the execution order of concurrent tasks on a target processor model. Similarly, a high-level processor model emulates HW/SW interactions and supports necessary interfaces and facilities for connecting software to the rest of the system. In order to consider cache effects on real-time behavior, a high-level cache model maintains the state of cache hierarchies. Abstract models are integrated into a standard system level design language (SLDL), which provides the required concurrency and event handling infrastructure. Finally, a user-application is captured as hierarchical, high-level processes which are integrated into the simulator via a provided API.

Altogether, a fast functional simulation is achieved by capturing system behavior at the source level and natively compiling and executing on a host machine. Additionally, timing-accurate results are obtained by instrumenting application source codes with target-specific execution delays.

The remainder of this paper is organized as follows: the next section gives an overview of existing full-system and high-level cache simulators. Section II elaborates our multi-core cache hierarchy modeling approach. Section III provides an experimental evaluation of our simulator. Finally, Section IV concludes and gives a summary of our work.

B. Related Work

Existing cache simulation techniques can be categorized into two common approaches: trace-driven or execution-driven simulations. Trace-driven cache simulators use a collected stream of memory accesses of applications to replicate the cache behavior [1]. This approach can be fast, but the simulator needs to deal with large trace files and data. In contrast, execution-driven simulators combine an executable cache model and simulated application instructions to capture memory accesses on-the-fly [2]. With the advent of MCPSoCs, cache simulators have also focused on simulating cache coherency and cache hierarchies [3], [4].

In order to support wide range of studies, modern full-system simulators support various architectures with different processor models, flexible system components, and memory and cache models at varying levels of detail and abstraction. Simics [5] and gem5 [6] are well-know simulators, which allow to model virtual prototypes of a complete system. Although both simulators offer multi-core support including processor models ranging from instruction-accurate models to fully cycle-accurate micro-architectural ones, the need to simulate cross-compiled applications running on top of the complete binary of an operating system kernel makes these simulators inefficient for fast and early integration and evaluation of complete systems.

On the contrary, host-compiled simulators have received widespread attention as a solution for fast and accurate system evaluation at early design stages [7]. In such approaches, the binary code of OS kernels and detailed processor models are replaced by an abstract model of the OS [8] and a high-level full-system software execution environment [9], [10], respectively.

There have been several attempts for cache modeling in host-compiled or source-level simulation. Pedram *et al.* [11] present a high-level, search-based cache model integrated into a TLM processor simulation platform. Pasadas *et al.* [12] propose a faster approach by introducing a lookup table-based data cache model. Both include approaches for instrumenting application source code to update the cache state and adjust the back-annotated delays. Stattelmann *et al.* [13] introduce a hybrid source-level cache simulator, which uses application binary codes to annotate memory accesses. However, all these approaches suffer from a lack of multi-core cache hierarchy modeling with associated speed and accuracy challenges.

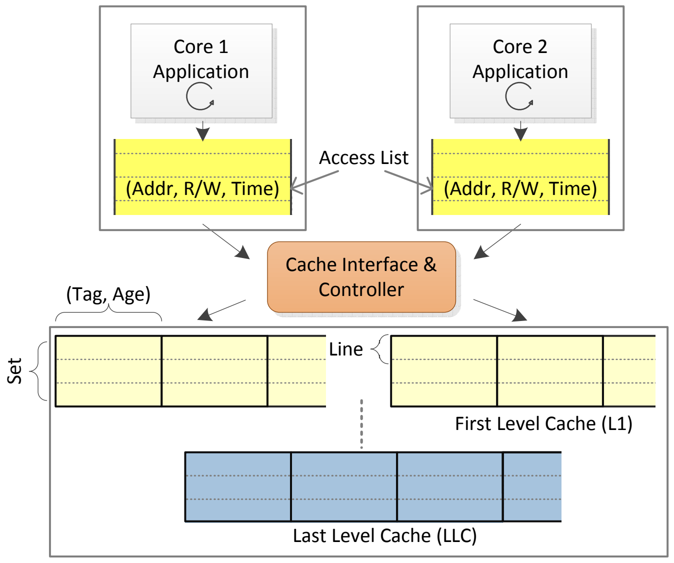


Fig. 2. High-level cache hierarchy model.

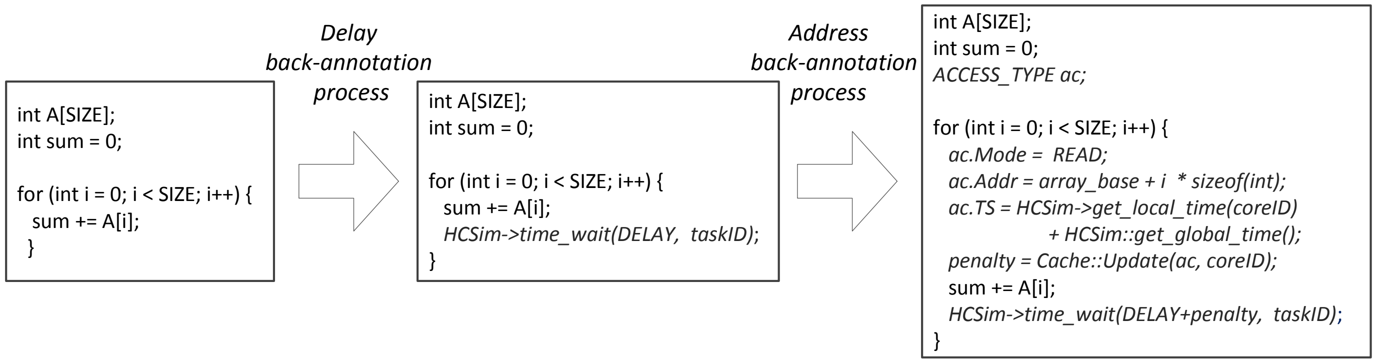
II. Multi-Core Cache Modeling

In this paper, we present a novel high-level, multi-core cache hierarchy modeling approach, which accurately models cache behavior of MCPSoCs, simulated by a temporally decoupled, host-compiled simulator. In the following, we first present an overview of our generic cache model and its integration process into a host-compiled simulator. Next, we demonstrate a novel memory access reordering technique, which is designed for accurate cache behavior simulation in a temporally decoupled execution context.

All models presented in this paper are implmented as SystemC [14] modules or channels, and the host-compiled simulator runs on top of the SystemC simulation kernel.

A. Base Approach

For accurate performance evaluation, we need to consider performance penalties due to cache misses and update static back-annotated delays during simulation. For this purpose, we developed a high-level model of a cache channel that emulates the system memory behavior by updating its internal states on every memory access. Note that we only need to model hit/miss behavior of the cache, i.e. we are not concerned about the data that is stored in the cache. Instead, the simulation host takes care of maintaining coherent data values. In our cache model, each cache line is composed out of an address tag, an age counter to implement a replacement policy, and a coherency flag to store the current state of each line compared to other cores' caches (Figure 2). Associated with each core, an access list stores locally ordered memory references reported by the application running on that core. Each location in this list contains a memory address, access mode (i.e Read or Write), and an access time. Using this information, a cache controller is able to manage the cache state updating process and to report back total miss cycles. Accordingly, the simulator



Delay back-annotation process
Address back-annotation process

```
int A[SIZE];
int sum = 0;

for (int i = 0; i < SIZE; i++) {
  sum += A[i];
 }
```

```
int A[SIZE];
int sum = 0;

for (int i = 0; i < SIZE; i++) {
  sum += A[i];
  HCSim->time_wait(DELAY,  taskID);
}
```

```
int A[SIZE];
int sum = 0;
ACCESS_TYPE ac;

for (int i = 0; i < SIZE; i++) {
  ac.Mode =  READ;
  ac.Addr = array_base + i  * sizeof(int);
  ac.TS = HCSim->get_local_time(coreID)
              + HCSim::get_global_time();
  penalty = Cache::Update(ac, coreID);
  sum += A[i];
  HCSim->time_wait(DELAY+penalty,  taskID);
}
```

Fig. 3. Source code back-annotation example for cache simulation.

can internally adjust back-annotated task delays by adding delay for extra memory cycles.

1) Source-code back-annotation: As mentioned before, a user application is integrated into the simulator at the source level. As such, low-level information including task execution delays and memory references need to be back-annotated into the source code. In this paper, we assume that designers use existing techniques to obtain task execution delays and accessed memory addresses based on a selected target platform [12], [15], [16]. Figure 3 shows source code instrumentation steps to consider cache effects during host-compiled simulation. The original application is a simple loop over an integer array. The execution delay is given to the simulator by calling the *time_wait()* method of the simulator API (shown as *HCSim*). In this way, the simulator internally advances simulated time to model task and core execution times.

In the address back-annotation step, every memory access is reported to the simulator by simply committing the access information including a 3-tuple of (address, mode, time stamp) into the cache. The time stamp is an absolute time, which is determined as the sum of the global simulated time and the core's local time. Note that absolute time stamps are required for temporal decoupling simulation and are used in our reordering technique (see Section II-B). At the end, the cache returns any miss penalties incurred during the access, which are in turn added to the task's execution delay.

2) Cache modeling: We have designed our cache model such that a designer can easily explore a wide variety of cache architectures and evaluate their effects on system performance. Table I lists all configurable options that our cache model offers. Primarily, the number of packages and cores per package need to be defined to generate the overall structure of the cache hierarchy. In addition, the number of levels of the cache hierarchy and interconnections across these levels are configurable. For example, the L2 can be defined a shared cache between all cores on a package or as a core-private cache. Furthermore, the structure of each cache including total and set size, replacement and write policies are parametrizable. By configuring the miss latency of all levels, the cache channel can provide the total miss penalty on every cache update. Finally, to keep the cache hierarchy coherent, a standard MSI-based cache coherency protocol is implemented.

TABLE I
CACHE PARAMETERS

Parameters	**Description**
Levels of hierarchy	Core-private L1 Core-private L2, or shared L2 within a package Shared L3 within a package
Cache structure	Cache size, Line size Associativity Write back, Write through (write allocation)
Replacement policy	LRU: default policy
Miss latency	Number of cycles for miss penalty of each level

B. Decoupling

Temporal decoupling is a widely-adopted mechanism to improve the simulation speed by increasing the granularity of simulation. In such an approach, which is only applicable to parallel systems, each thread keeps a local time that defines how far this thread has advanced in its execution related to the global simulated time. In other words, a thread can go ahead of the simulation kernel time without advancing global time.

A conventional host-compiled simulator advances the simulated time at every back-annotate delay using the underlying SLDL primitives, which causes a simulation kernel context switch. As a result, the simulation speed is defined by the granularity of back-annotated delays. By contrast, a temporally decoupled simulator only advances time whenever the simulation quantum expires or a local task switch or a global synchronization is required. To follow this approach, a local counter is associated with each core, which accumulates back-annotated application delays. This counter defines the core's local time in relation to the global simulated time.

However, decoupling techniques may decrease the timing accuracy due to the coarse-grained synchronization of parallel threads. Especially in multi-core cache simulation, different cores may commit their memory accesses to the cache globally out-of-order. In the following, we introduce a delayed reordering technique that provides an accurate temporally decoupled

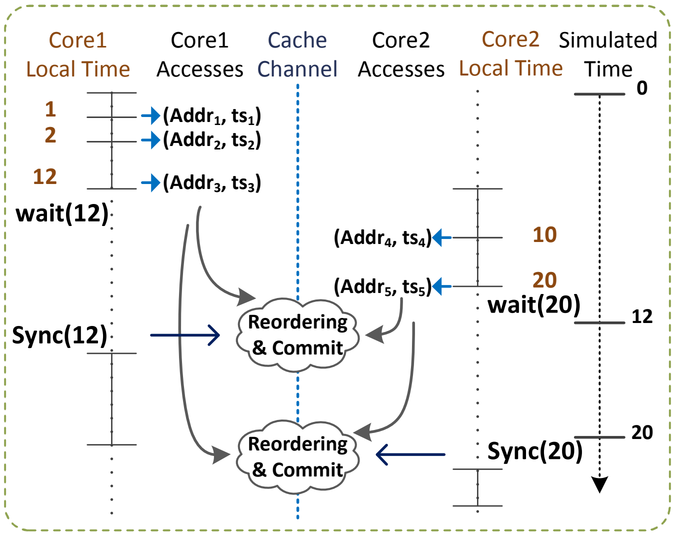


Fig. 4. Cache reordering example.

```
Function: time_wait(time_t nsec)

1 cur_core.local_time += nsec
2 sync_time = simulation_quantum
3 while cur_core.local_time ≥ sync_time do
4   SLDL::wait(sync_time)
5   cur_core.local_time -= sync_time
6   cache_time = Cache::Sync(sync_time, cur_core)
7   cur_core.local_time += cache_time
8 endwhile
```

Fig. 5. Simulation timing model.

```
Function: Sync(time_t sync_time, int cur_core)

1 while true
2   for all cores do
3     min_core = ArgMin_i(access_list[i].first.ts)
4     if (access_list[min_core].first.ts > sync_time) break
5     a = access_list[min_core].pop()
6     min_core.local_delay += Cache::Update(a, min_core)
7   endfor
8 endwhile
9 return cur_core.local_delay
```

Fig. 6. Cache synchronization and reordering algorithm.

cache hierarchy simulation.

To illustrate the general concept behind the reordering technique, we show the execution sequence of a dual-core platform in Figure 4. At the beginning of the simulation, $core_1$ starts the execution of its application code and reports three memory accesses at local times $1(ts_1)$, $2(ts_2)$, and $12(ts_3)$. At local time 12, $core_1$ notified the simulator to consume the accumulated delay. Accordingly, the host-compiled simulator internally advances the simulated time by calling the underlying SLDL *wait(12)* method. As such, $core_1$ is suspended and $core_2$ gets a chance to run its application code, which reports its two memory accesses at local times $10(ts_4)$ and $20(ts_5)$. As can be seen, ts_4 is reported after ts_3, while to be committed before ts_3. When $core_2$ request to consume the accumulated delay in a similar way, the simulator internally advances the global simulated time by 12 units and returns to $core_1$. By this time, all cores have already collected their memory accesses. Hence, $core_1$ calls a *Sync(12)* method to commit the out-of-ordered accesses with their correct sequence into the cache channel.

1) Cache Simulation: To enable delayed reordering, instead of directly committing accesses to the cache as they occur, each core keeps all referred memory addresses in an ordered list. Although each access has a time stamp, which allows the cache to detect the global sequence of all accesses, the simulator needs to invoke the cache synchronization and reordering method when all cores' accesses have been collected and it can be determined that they are safe to commit.

An efficient safe point for committing collected memory accesses is after advancing the simulation time. In this way, the underlying SLDL simulation kernel lets other cores run their tasks and collect all memory accesses up to that point in simulation time. Figure 5 shows the function that manages the simulated time. Task delays are back-annotated into the code via a call to this *time_wait()* method. The current core first updates its local time. If the accumulated delay is greater than the simulation quantum, the global simulated time is advanced by the underlying SLDL *wait* primitive (line 4). After advancing the simulation time, the *Sync()* function in the cache is called (line 6). Finally, the local time is updated by possible extra delays caused by cache miss penalties of committed accesses (line 7).

2) Reordering mechanism: The pseudo code of our cache synchronization and reordering technique is shown in Figure 6. The reordering algorithm is divided into three steps: In the first step, the core containing the access with the smallest time to commit is determined by exploring all core access lists (line 3). In the second step, any corresponding memory access with a time stamp smaller than or equal to the safe to commit time (synchronization point) is used to update the cache state (line 4 and 5). Finally, based on the cache behavior, the core's local delay is updated to record extra delays related to miss penalties (line 6). This loop continues until all accesses from all cores with a time stamp smaller than the end time are committed to the cache model.

III. Experimental Results

We evaluated our cache modeling approach by simulating parallel matrix multiplication tasks running on a dual-core 1.6 GHz Atom platform with a core-private 24K, 6-way set associative L1 and a shared last level 512K, 8-way set associative L2 cache. We executed our experiments under three different simulation modes: a conventional simulation updates the cache state at the fine granularity given by back-annotated execution times, while the temporally decoupled (TD) approaches with and without reordering (RO) method commit accumulated accesses only at simulation quantum boundaries.

TABLE II
CACHE ACCURACY RESULTS FOR MATRIX MULTIPLICATION SIMULATION ON A SINGLE-CORE PLATFORM

	Naïve Algorithm							Cache-Aware Algorithm						
Matrix	Total L1	L1 Miss Rate			L2 Miss Rate			Total L1	L1 Miss Rate			L2 Miss Rate		
Size	Accesses	Board	Sim.	Error	Board	Sim.	Error	Accesses	Board	Sim.	Error	Board	Sim.	Error
16	9,306	-	-	-	-	-	-	9,818	-	-	-	-	-	-
32	69,802	-	-	-	-	-	-	71,850	-	-	-	-	-	-
64	539,918	0.20%	0.15%	(-27.5%)	-	-	-	633,996	0.23%	0.27%	(16.9%)	-	-	-
96	1,750,378	3.12%	3.17%	(1.6%)	-	-	-	2,140,519	0.19%	0.22%	(17.3%)	-	-	-
128	2,127,242	50.1%	50.6%	(1.1%)	-	-	-	5,071,702	0.27%	0.25%	(-6.2%)	-	-	-
192	7,067,470	50.6%	50.9%	(0.7%)	-	-	-	17,123,704	0.22%	0.26%	(17.0%)	-	-	-
256	16,898,250	50.0%	50.3%	(0.6%)	0.05%	0.05%	(-0.22%)	40,572,930	43.0%	50.2%	(16.8%)	0.06%	0.06%	(0.02%)
384	56,578,186	50.3%	50.5%	(0.4%)	6.21%	6.21%	(0.03%)	136,988,302	0.26%	0.28%	(9.4%)	36.3%	38.9%	(7.25%)

TABLE III
MEASURED AND SIMULATED EXECUTION TIME FOR MATRIX MULTIPLICATION ON SINGLE-CORE AND DUAL-CORE PLATFORMS

	Single-Core Platform						Dual-Core Platform					
	Naïve Algorithm			Cache-Aware Algorithm			Naïve Algorithm			Cache-Aware Algorithm		
Matrix	Exe. Time	Simulation Error		Exe. Time	Simulation Error		Exe. Time	Simulation Error		Exe. Time	Simulation Error	
Size	Board	w/ cache	w/o cache	Board	w/ cache	w/o cache	Board	w/ cache	w/o cache	Board	w/ cache	w/o cache
16	25μs	-6.2%	-6.2%	24μs	-5.0%	-5.0%	25μs	-7.5%	-7.5%	25μs	-7.5%	-7.5%
32	180μs	2.6%	2.6%	179μs	3.3%	3.3%	183μs	1.4%	1.4%	183μs	1.4%	1.4%
64	1,472μs	1.0%	0.6%	1,563μs	-4.5%	-5.2%	1,492μs	-0.3%	-0.7%	1,569μs	-4.8%	-5.6%
96	5,484μs	-0.5%	-8.8%	5,200μs	-3.2%	-3.9%	5,329μs	2.4%	-6.2%	5,179μs	-2.8%	-3.5%
128	29,631μs	-1.6%	-60.0%	12,335μs	-3.2%	-3.9%	29,638μs	-1.6%	-60.0%	12,320μs	-3.1%	-3.8%
192	101,630μs	-2.9%	-60.7%	41,156μs	-2.1%	-2.8%	101,597μs	-2.4%	-60.6%	41,082μs	-0.5%	-2.7%
256	247,855μs	-6.0%	-61.8%	259,449μs	-8.7%	-63.5%	248,944μs	8.1%	-61.9%	259,352μs	-7.5%	-63.5%
384	1,235,815μs	-10.9%	-94.6%	419,768μs	-20.4%	-23.8%	1,270,051μs	-8.5%	-74.8%	419,905μs	-20.4%	-23.8%

TABLE IV
L2 MISS RATE AND ERRORS FOR CONVENTIONAL, RO, TD SIMULATION OF MATRIX MULTIPLICATION ON THE DUAL-CORE PLATFORM

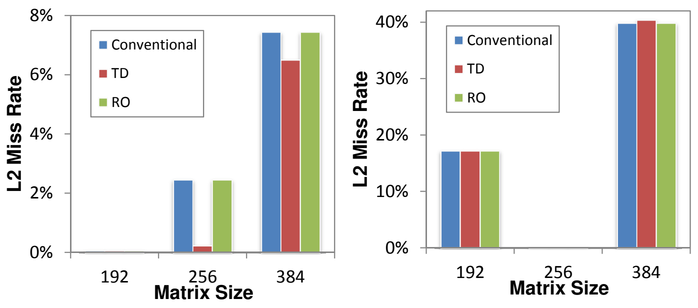
	Naïve Algorithm						Cache-Aware Algorithm					
Matrix	Total L2	Miss Rate	Miss Rate Error				Total L2	Miss Rate	Miss Rate Error			
Size	Accesses	Conv.	TD(1μs)	TD(1ms)	RO(1μs)	RO(1ms)	Accesses	Conv.	TD(1μs)	TD(1ms)	RO(1μs)	RO(1ms)
16	0	-	-	-	-	-	-	-	-	-	-	-
32	0	-	-	-	-	-	-	-	-	-	-	-
64	1,536	-	-	-	-	-	2,870	-	-	-	-	-
96	112,896	-	-	-	-	-	8,064	-	-	-	-	-
128	4,262,398	-	-	-	-	-	21,504	-	-	-	-	-
192	14,454,982	0.08%	0.00%	-3.06%	0.00%	0.00%	76,200	17.3%	0.00%	-0.34%	0.00%	0.00%
256	33,830,910	2.46%	-73.3%	-90.4%	0.00%	0.00%	34,765,822	0.25%	0.00%	1.54%	0.00%	0.00%
384	114,482,686	7.43%	-16.0%	-12.6%	0.00%	0.00%	663,552	39.8%	0.00%	1.36%	0.00%	0.00%

To analyze the effect of course-grain temporal decoupling on cache simulation accuracy, we ran our experiments under two different simulation quanta: $1\mu s$ and $1ms$.

In order to evaluate the cache behavior under different memory accesses patterns, we simulated two algorithms, a typical naïve matrix multiplication algorithm and a cache-aware, blocked algorithm with fixed 32x32 blocks. For each algorithm, we simulated a variety of matrix sizes ranging from small size matrices that would fit entirely in the L1 cache to large matrices that exceed the L2 size.

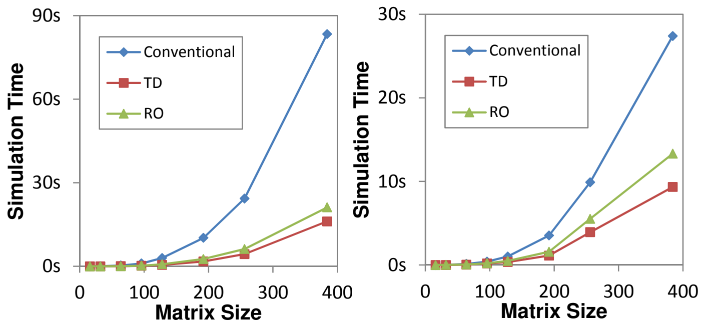
We first verified the accuracy of our cache hierarchy model on a single-core execution. For this purpose, we compared cache miss rates obtained from simulation with the actual execution on a single-core Atom board. We used Valgrind [17] to monitor the Atom cache behavior. Table II shows the total number of accesses for the L1 cache and the L1 and L2 miss rates on the Atom board and compares these with simulation results. Residual errors are caused by the back-annotation process, which does not instrument all memory accesses. Overall, the results depict that our cache simulator follows the reference cache behavior. Furthermore, low miss rates for large matrices confirm that a cache-aware implementation achieves a better performance.

We further compare the simulated execution times with the reference execution on the board. In this experiment, we used a cache-memory calibration tool [18] to measure L1 and L2 miss latency cycles and annotated our cache model accordingly in order to accurately reflect the cache effects in the host-compiled simulation. The measured and simulated execution times for both the single-core and the dual-core platforms are shown in Table III. Results show a significant improvement of the cache model on the execution time accuracy for matrices with high L1 or L2 miss rates. The average execution error in the presence of cache modeling was -3.8%, while neglecting



(a) Naïve algorithm. (b) Cache-aware algorithm.

Fig. 7. L2 miss rate for Conv, RO, and TD simulations, quantum(1ms).



(a) Naïve algorithm. (b) Cache-aware algorithm.

Fig. 8. Simulation time for Conv, RO, and TD simulations, quantum(1ms).

the impact of cache behavior can generate up to 95% timing error. Note that as shown in Table III, for a matrix size of 384 cache conflicts in the cache-aware algorithm lead to a high L2 miss rate which in turn results in a relatively high error, even with our cache model. This is due to lack of accurate models of system busses beyond the L2 in our setup.

Finally, to demonstrate the efficiency of our reordering approach in a temporally decoupled simulation, we compare L2 miss rates using different modeling setups. Table IV shows the miss rates for TD and RO simulations. As expected, for the RO approach, miss rates were constant under different simulation quanta and were identical to the conventional simulation results. By contrast, the TD approach can exhibit large deviations due to out-of-ordered cache updates. Note that for the cache-aware algorithm, there is very little L2 interference between cores. As such, the ordering of accesses does not play a significant role and a naïvely decoupled simulation already provides good results. However, such behavior is hard to predict. By contrast, reordering approach provides consistently good results.

To summarize, Figure 7 compares L2 miss rates reported by different host-compiled simulation approaches. As can be seen, temporally decoupled simulation without reordering the memory accesses can result in large errors for some configurations. Figure 8 plots the simulation time for the same experiments. Altogether, the integrated RO model provides a guaranteed accurate result while maintaining almost the full performance benefits of a temporally decoupled simulation.

IV. Summary and Conclusions

In this paper, we presented a novel cache hierarchy simulation technique, which provides an accurate multi-core cache simulation for efficient system-level evaluation and exploration. Our approach introduces a multi-core, out-of-order cache model, which incorporates a delayed reordering of aggregated requests to provide an accurate cache simulation in the presence of temporal decoupling. We evaluated the accuracy of our models on a set of benchmarks. Our results show that the highest possible accuracy is obtained by using our reordering technique, while increased simulation performance benefits from temporal decoupling.

Acknowledgment

This work has been partially supported by SRC Task 2317. We also would like to thank Xinnian Zheng for developing the initial code of the high-level cache model.

References

[1] R. Hassan, A. Harris, N. Topham, A. Efthymiou, Synthetic Trace-Driven Simulation of Cache Memory. *AINAW*, May 2007.

[2] Y. Chen, J. Cong, G. Reinman, HC-Sim: A fast and exact L1 cache simulator with scratchpad memory co-simulation support. *CODES+ISSS*, Oct. 2011.

[3] A. Jaleel, R. S. Cohn, C. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. *MoBS*, 2008

[4] M. S. Haque, R. G. Ragel, J. A. Ambrose, S. Radhakrishnan, S. Parameswaran, DIMSim: a rapid two-level cache simulation approach for deadline-based MPSoCs. *CODES+ISSS*, 2012.

[5] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer 35*,February 2002.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit.*, August 2011.

[7] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, H. Meyr. A high-level virtual platform for early MPSoC software development. *CODES+ISSS*, September 2009.

[8] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *DAES*, December 2005.

[9] A. Bouchhima, I. Bacivarov, W. Youssef, M. Bonaciu, A. Jerraya. Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration. *ASPDAC*, January 2005.

[10] G. Schirner, A. Gerstlauer, R. Dömer. Fast and Accurate Processor Models for Efficient MPSoC Design. *TODAES*, February 2010

[11] A. Pedram, D. Craven, A. Gerstlauer. Modeling cache effects at the transaction level. *IESS*, Sep. 2009.

[12] H. Posadas, L. Díaz, E. Villar. Fast data-cache modeling for native co-simulation. *ASP-DAC*, Jan. 2011.

[13] S. Stattelmann, G. Gebhard, C. Cullmann, O. Bringmann, W. Rosenstiel. Hybrid source-level simulation of data caches using abstract cache models. *DATE*, March 2012.

[14] F. Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Aplications for Embedded Systems.* Springer, 2005.

[15] Z. Wang, J.Henkel "Accurate source-level simulation of embedded software with respect to compiler optimizations." *DATE*, 2012.

[16] A. Bouchhima, P. Gerin, F. Ptrot "Automatic instrumentation of embedded software for high level hardware/software co-simulation." *ASP-DAC*, 2009.

[17] N. Nethercote, J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.

[18] A Cache-Memory and TLB Calibration Tool. Available online: *http://homepages.cwi.nl/~manegold/Calibrator/.*