Workshop on Design, Architecture and Simulation of Chip Multi-Processors

dasCMP 2006

Sunday December 10, 2006, Orlando

Message from the Workshop Organizers

Chip multiprocessor architectures are becoming increasingly attractive as an option to provide high instruction throughput while keeping power and complexity under control. Such architectures have also been shown to have scalability and productivity advantages. Multi-core processors are fast becoming mainstream.

However, putting multiple cores on a die throws open several interesting research and design issues. From choosing the number of cores on the die to choosing the complexity of these cores, from constructing a chip multiprocessor out of off-the-shelf cores to creating a customized ``multi-core aware" multi-core design, there are several difficult questions that need to be addressed. Connecting the cores, determining the right memory subsystem, ensuring coherence and consistency of data, and trying to limit the area and power budgets, all require a deep understanding of issues and innovative application of ideas. Even simulating and evaluating a chip multiprocessor represents a significant challenge. There are equally interesting issues regarding programming and compilation for chip multiprocessors. All these questions and issues become more difficult and complicated for architectures with more than two cores.

The goal of this workshop is to bring together researchers, computer architects, and engineers working on a broad spectrum of topics pertaining to the architecture, simulation, and design of chip multiprocessors. The workshop will provide a forum for presenting and exchanging new ideas and experiences in this area and to discuss and explore hardware/software techniques and tools for efficient multi-core computation.

A successful workshop requires the help of many individuals. We would like to thank each of our program committee members for their quality reviews of workshop submissions. We would also like to thank Hsien-Hsin Lee, the Micro-39 Workshops Chair, for his help in organizing the workshop. And we would like to thank our keynote speaker and our panel members for agreeing to participate.

We hope you find the papers of this workshop to be intellectually stimulating, and that you will be inspired to contribute your ideas and efforts to future workshops!

Dean, Rakesh, and Norm

dasCMP06 Program Committee

- Luiz Barroso, Google
- Antonio Gonzalez, Intel and UPC
- Michael Gschwind, IBM
- Norman P. Jouppi, HP
- Stephen Keckler, UT Austin
- Rakesh Kumar, UIUC
- Jim Laudon, Sun
- Chuck Moore, AMD
- Kunle Olukotun, Stanford
- Mark Oskin, UWash
- Ronny Ronen, Intel
- Per Stenstrom, Chalmers
- Dean Tullsen, UCSD
- TN Vijaykumar, Purdue
- David Wood, Wisconsin

Workshop Program

8:30-8:45AM Welcome Remarks etc.

8:45-9:30AM Keynote Presentation

“Opportunities and Challenges of the 1000-thread CMP", Jim Laudon, Sun

9.30-10.00AM

"Providing Hardware Support for Software Controlled Multithreading", Aqeel Mahesri (UIUC), Nicholas J. Wang (UIUC), Sanjay J. Patel (UIUC)

10.00-10.30AM Break

10:30-12:00PM

Session Title: Cache and Bandwidth Issues

"CMP Cache Performance Projection: Accessibility vs. Capacity ", Xudong Shi (UFl), Feiqi Su (UFl), Jih-kwon Peir (UFl), Ye Xia (UFl), Zhen Yang (UFl)

"From Chaos to QoS: Case Studies in CMP Resource Management ", Hari Kannan (Stanford), Fei Guo (NCSU), Li Zhao (Intel), Ramesh Illikkal (Intel), Ravi Iyer (Intel), Don Newell (Intel), Yan Solihin (NCSU), Christos Kozyrakis (Stanford)

“Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS", Masaaki Kondo (Univ of Tokyo), Hiroshi Sasaki (Univ of Tokyo), Hiroshi Nakamura (Univ of Tokyo)

12:00-1:30PM Lunch

1:30-3:00PM
Session Title: Exploiting and Expressing Parallelism

"Starvation-Free Commit Arbitration Policies for Transactional Memory Systems ", M.M. Waliullah (Chalmers) and Per Stenstrom (Chalmers)

"An hardware/software framework for supporting Transactional Memory in a MPSoC environment ", Cesare Ferri1 (Brown), Tali Moreshet (Swarthmore), R.Iris Bahar (Brown), Luca Benini (Bologna), and Maurice Herlihy (Brown)

"Function Level Parallelism Driven by Data Dependencies", Sean Rul (Ghent), Hans Vandierendonck (Ghent), Koen De Bosschere (Ghent)

3:00-3:30PM Break

3:30-5:00PM
Panel Discussion: Top CMP Research Problems: An Industrial Perspective

Hardware Support for Software Controlled Multithreading

Aqeel Mahesri Nicholas J. Wang Sanjay J. Patel
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{mahesri,nwang,sjp}@crhc.uiuc.edu

Abstract

Chip multi-processors have emerged as one of the most effective uses of the huge number of transistors available today and in the future, but questions remain as to the best way to leverage CMPs to accelerate single threaded applications. Previous approaches rely on significant speculation to accomplish this goal. Our proposal, NXA, is less speculative than previous proposals, relying heavily on software to guarantee thread correctness, though still allowing parallelism in the presence of ambiguous dependences. It divides a single thread of execution into multiple using the master-worker paradigm where some set of master threads execute code that spawns tasks for other, worker theads. The master threads generally consist of performance critical instructions that can prefetch data, compute critical control descisions, or compute performance critical dataflow slices. This prevents non-critical instructions from competing with critical instructions for processor resources, allowing the critical thread (and thus the workload) to complete faster. Empirical results from performance simulation show a 20% improvement in performance on a 2-way CMP machine, demonstrating that software controlled multithreading can indeed provide a benefit in the presence of hardware support.

1. INTRODUCTION

As superscalar processors approach practical limits to their complexity, the industry is shifting toward chip multiprocessors (CMPs) to maintain efficient use of available transistors. CMPs efficiently increase throughput for multiple, concurrent applications or multi-threaded applications. Unfortunately, they do not provide the performance speedups for single-threaded applications that the computing community has grown accustomed to.

It is possible to convert single-threaded applications to a multi-threaded target by hand or by compiler. However, the manual process is tedious and compiler techniques are ineffective on general code. Moreover, even when code can be parallelized, a CMP cannot take advantage of any but the coarsest grained parallelism, limiting the performance potential.

To alleviate these limitations, we propose NXA, a software/architecture approach which utilizes static, compile-time analyses to partition workloads into master and worker threads. Master threads generally consist of performance critical instructions and are given the full resources of a single processor. Worker threads are spawned off from master threads onto other processors. This prevents worker thread instructions from competing with critical master instructions for processor resources, allowing the master threads (and thus the workload) to execute faster. Further details on high level NXA concepts are provided in Section 2.

We present an implementation of NXA to support master-worker threading all the way down to a fine granularity. This implementation of NXA builds on a conventional CMP by adding hardware mechanisms for spawning out worker threads from one processor core to the others, as well as mechanisms to maintain a consistent memory and register state across the cores. This implementation of NXA is described in detail in Section 3.

To create master threads from a workload, we propose and evaluate algorithms to identify critical portions of the workload. Since different workloads present different critical paths to the hardware, we develop three slicing methods to find the critical portions. In the first scheme, control decoupling, the main thread computes the global control flow through the program while the work threads perform data computation. In the second scheme, memory decoupling, the main thread advances miss-prone loads while the work threads defer computation dependent on those loads. In the third scheme, critical path decoupling, the main thread computes values along the dataflow critical path while the work threads defer non-critical computation. Each of these decoupling schemes is described in Section 4.

We evaluate the performance improvement offered by NXA and observe an average speedup of 20%. As expected, we find that each decoupling system works best on certain classes of workloads. The performance of NXA and the decoupling schemes are examined in Section 5.

Section 6 compares and constrasts prior work in decoupled architecture. Finally, Section 7 provides concluding remarks.

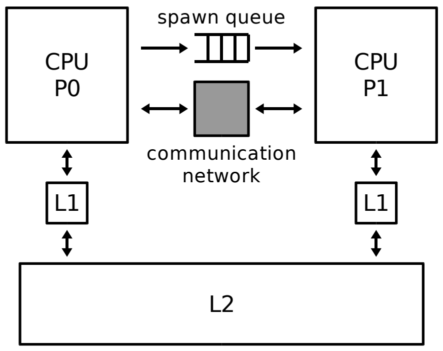


Figure 1. NXA conceptual model.

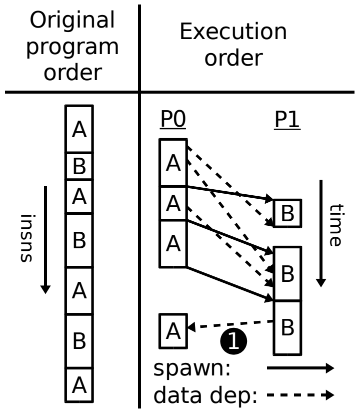


Figure 2. NXA decoupling model example.

2. PROGRAMMING MODEL

In this section we introduce the NXA programming model. We describe the NXA architecture at a conceptual level as a chip multiprocessor with a mechanism to queue thread spawns between cores and a communication network to maintain a consistent register and memory state. We introduce the decoupling model used to break up single-threaded programs into threads to run on NXA. Finally, we define NXA as a compiler target by examining the interface between the compiler and the NXA hardware.

2.1. Conceptual Architecture

A conceptual model of NXA is shown in Figure 2.1. The architecture contains two processor cores, P0 and P1. Connecting the processor cores is a spawn queue, which allows P0 to offload work to the other processor core. When the code running on P0 comes upon a spawn instruction, the target of the spawn is pushed onto the spawn queue. P1 then dequeues the spawn from the queue and executes beginning at the target location. Hence, the main thread running on P0 can run work threads on P1 without the high overhead of forking a full-blown thread.

Also connecting the execution cores is a communication network. The purpose of the communication network is to ensure that instructions on P0 and P1 both see a well defined execution state. In particular, data that is generated in the main thread before a spawn is enqueued needs to be visible in the work thread. Hence, the communication network must be able to grab data from P0 and transmit it to P1 on request. Moreover, the main thread may issue a spawn without knowing for certain that it doesn't have a data dependence on the spawned work thread. In this case, the P0 must signal the communication network that it needs to see the data generated by the work thread on P1. The communication network can stall the main thread, forward any data that may have been incorrectly read by the main thread, or allow the main thread to move forward once it has proved that there are no dependence violations. All this communication is transparent to the software, which only sees a continuous flow of architectural state that remains the same as it would if the program were all run on a uniprocessor.

Finally, the cores are connected via the memory system. In NXA, the caches for the two cores are connected by a cache coherence system, in particular cache coherence with an update protocol, which is well suited to the producer-consumer relationship between the main and worker threads.

Compare NXA with a conventional CMP. In a conventional CMP, threads are initiated via OS support, by a system call that creates a new process control block, explicitly copies over the initiating thread's state, and assigns to a new processor via the OS kernel's thread scheduling system. Such a heavyweight system for thread creation makes it impossible to gain performance from any but the coarsest grained parallelism, a restriction which excludes much of the program's intrinsic parallelism. Moreover, the conventional CMP has no communication network directly linking the processing cores. All data communication must flow through memory. This severely limits the compiler's ability to extract thread-level parallelism, as it must carefully synchronize inter-thread communication via shared memory and using synchronization calls provided by the OS. Experience has shown that conventional CMPs are a difficult target for automatic parallelization for most workloads, and NXA aims to alleviate the limitations.

2.2. Decoupling Model

NXA offers a platform for any arbitrary decoupling scheme. However, in this work we evaluate NXA as a platform for implementing a slicing approach to parallelization. The basic idea of the slicing approach is to find parallel threads that can be interleaved in the original program; each thread is a chain of dependent instructions that constitutes a "slice" of the program. Figure 2.2 provides a graphical reference for the ideas discussed in this section.

Slicing parallelization is implemented on NXA by decomposing the original program into a main thread and worker threads (labeled A and B respectively in the example). Some slice of the program is extracted from the original code and used to form the main thread. This main slice consists of the portion of the code that is critical in determining the program's execution time. For instance, this critical set may be computing the control flow through the program, or it may be servicing cache misses. In any case, the code within this slice must run as quickly as possible. Code outside of this slice is placed into work threads. This code should be non-critical to the performance. It should consist of instructions or tasks that can be deferred without penalizing the execution time.

On NXA, the main thread runs on P0. With the program's non-critical instructions deferred to work threads, we reduce the resource constraints impeding the performance of the critical instructions in the main thread. When the main thread arrives at a point in the original program where a work thread task needs to be run, it issues a spawn instruction (solid arrows in Figure 2.2) instructing P1 to begin running that task. The communication network provided by the NXA hardware transparently forwards any data needed by the work thread from P0 to P1.

In all slicing approaches to parallelization, the flow of data values (dotted arrows in the figure) and control decisions is predominantly from one slice to another, with little or no flow in the reverse direction. On NXA, the main thread is the slice of the program running out in front, while the work thread is lagging behind. Hence, the bulk of the communication between the main and work threads must be from the main thread to the work thread. Any communication in the reverse direction risks a data dependence violation (data dependence "1" Figure 2.2). Although the communication network is capable of fixing these violations, they are costly for performance. The main thread must be mostly data independent of the work thread for good performance. A similar limitation applies to control dependences.

2.3. Compiler Target

Our goal is to provide a target for automatic, static parallelization of code by a binary analyzer or compiler. To this end, NXA exposes three new instruction to the software for spawning work threads and ending execution on a thread. In our approach, the binary analyzer or compiler analyzes the program, perhaps with the aid of profile information, and determines the slice of the program that forms the main thread and what the dependences between the main and work threads are. The compiler then inserts explicit spawn instructions into the program as needed.

The first instruction, the unchecked spawn, *pbrnc* (non-checked parallel branch), spawns off a work thread but does not check to see whether data generated by the work thread is read back in by the main thread. The *pbrnc* instruction itself consists of the opcode and the spawn target. When a processor core (P0) encounters a *pbrnc*, it sends its architectural state to P1 (details in the next section). When P1 dequeues the spawn, it begins execution at the spawn target. P0 continues executing the instruction following the *pbrnc*. When a work thread is spawned by a *pbrnc*, it is expected that the output of the spawned task will not be accessed by the main thread. If the P0 does access a register or memory location written by the spawned task, then it is ambiguous whether it will see the old value or the new one.

The ambiguity is alleviated by the second instruction, the checked spawn, *pbr* (checked parallel branch). Like *pbrnc*, this instruction spawns off a work thread, but it also guarantees that the main thread will see any output generated by the spawned instructions. A processing core will not retire instructions following a *pbr* until it can guarantee that those instructions see correct data (although it may execute them before the spawned task finishes).

The instruction to mark the end of execution on a work thread is the endblock, *pjn* (parallel join). When the work thread processor sees a *pjn*, it halts execution on the current path and sends back to the main thread processor a map of what architectural state it updated. At this point, the work thread processor checks to see if its spawn queue has any pending spawns, and if so it dequeues and begins executing the next spawn. If there are no pending spawns, the processor stalls.

3. ARCHITECTURE

In this Section we examine an example implementation of the NXA model that is suitable for fine-grained decoupling. The microarchitecture of this proposed implementation, shown in Figure 3.1 is based on a dual-core CMP. Each core is a deeply-pipelined, out-of-order superscalar processor. Added to this CMP are the spawn queues plus a data communication network. These added blocks are shown in dark gray.

3.1. Spawn Queuing

The NXA main spawn queue is implemented as a FIFO connecting the rename stage of P0 with the fetch state of P1. When P0 decodes a *pbr* or *pbrnc* instruction, it sends the target of the spawn to the FIFO. At the fetch stage, P1 initiates fetch from the target of the spawn. If P1 encounters a *pjn*, it initiates fetch from the next spawn in the FIFO, or stalls if no spawns are enqueued.

3.2. Register Communication

Register communication between P0 and P1 is performed lazily, with bitmasks used to identify which registers need to be communicated. At the rename stage, both processors maintain a register update bitmask recording the set of architectural registers that are being updated by the current group of instructions. When P0 encounters a *pbr* or *pbrnc* instruction, it sends the bitmask to P1 in the spawn request along with the target PC, and resets the bitmask. Likewise, P1, when it encounters a *pjn* instruction, sends and resets its register update bitmask back to the ROB in P0. In our Alpha-based implementation, the register update mask is 76 bits wide, with a bit for each integer, floating point, and special register.

Register communication from P0 to P1 is simple. When P1 dequeues a spawn, it reads the bitmask to determine the

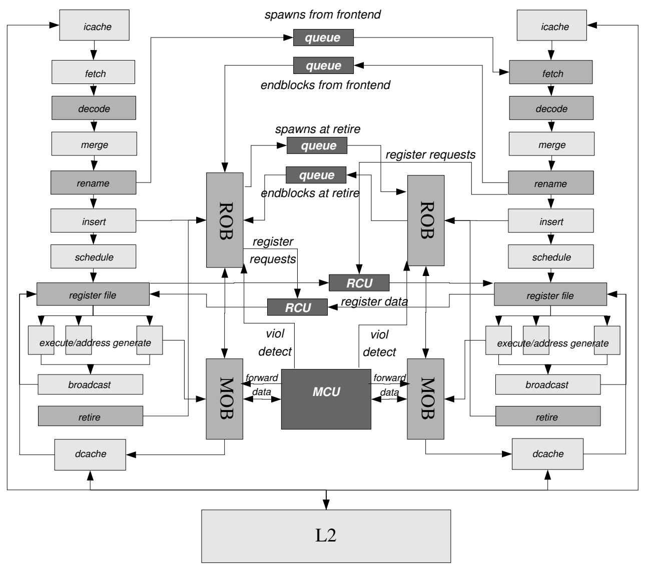


Figure 3. An NXA implementation.

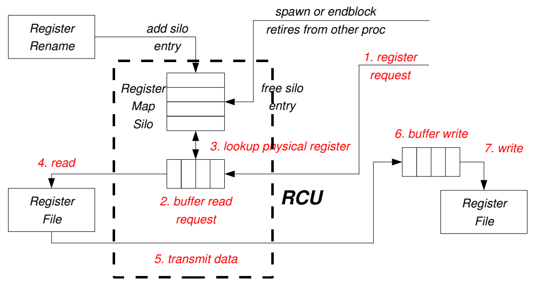


Figure 4. Register communication unit for one of the processor cores. Both cores have their own RCU.

set of registers written by P0. This set is recorded as the stale register mask. When an instruction in P1 passes the decode stage, its inputs are compared to the stale register mask to determine which must be read from P0.

Register communication from P1 back to P0 is more complicated. As stated, when P1 encounters a *pjn* instruction, it sends its register update mask to P0, along with an identifier for the originating spawn. P0 records this as its stale register mask. If the *pjn* corresponds to a checked (*pbr*) spawn, P0 checks before the retire stage to see whether instructions incorrectly read any stale registers; if so, it enqueues a request for these registers and replays any dependent instructions.

The actual register communication is carried out by the structure shown in Figure 3.2. The mechanism contains, on each processor's side, a register map silo (containing mappings from the architectural register to the physical register at the point of each *pbr/pbrnc* or *pjn* instruction), a buffer of pending register requests originating on that processor, and a buffer of pending reads from that processor's register file.

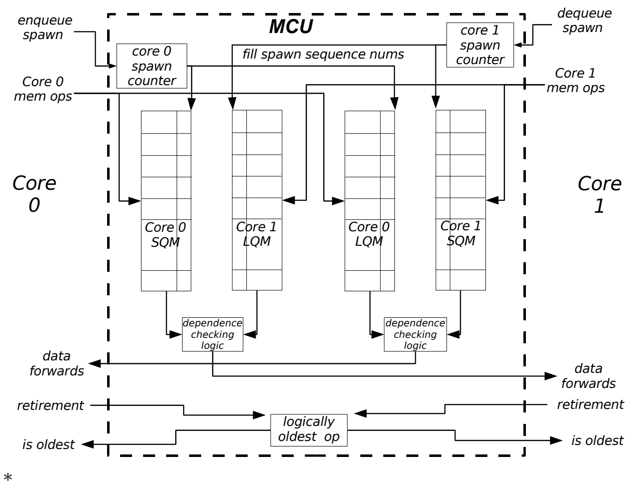


Figure 5. Memory communication unit.

3.3. Memory Communication

Memory communication between P0 and P1 is handled by the memory communication unit (MCU) shown in Figure 3.3. The basic idea of the MCU is to mirror the memory order buffer from each processor, and use these mirrors to detect and data dependence violations and perform interprocessor store forwarding. The structure itself consists of a load queue mirror (LQM) and a store queue mirror (SQM) for each processor. The LQM for processor core 0 is coupled to the SQM for processor core 1, and vice versa. The MCU also contains a spawn sequence counter for each core, as well as a register keeping track of the logically oldest memory operation that has not yet been committed. Finally, the MCU contains logic to detect memory dependences and perform the store forwarding.

When one processor executes a load or a store, it forwards the operation and its address to the MCU. The MCU then uses the LQM and SQM to determine whether there is a matching load or store from the opposite core. If so, it forwards the data value. The MCU also informs each core whether it contains the oldest current memory operation. In the case of a *pbr*, P0 will not commit memory operations until they are the oldest, so that the MCU can guarantee that there are no more dependence violations. This restriction does not apply with *pbrnc* instructions.

4. DECOUPLING IMPLEMENTATION

The NXA architecture provides a target for decoupled programming. In this Section, we examine how to implement our approach to parallelization in a compiler or static binary analyzer. Each of the decoupling schemes we present can be performed statically by data and control flow analysis on the program.

Our first method of decoupling the program is to separate the portions of the program that compute control flow from the portions that compute data values. Our second method is to separate the memory accessing portions of the program from the consumers of those memory accesses. Our third method is to separate the dataflow critical path of the program from the non-critical portion. All three methods take advantage of the same iterative algorithm for thread selection.

4.1. Parallelization Algorithm

The decomposition of the program into main and work threads is carried out in several steps. We first choose an initial set of instructions for the main thread. Then, we compute the control dependence and control flow graphs. Next, we use iterative dataflow analysis [14] to compute the dataflow and control dependence closure of the initial set, which becomes the main thread. We separate out the main thread and the work thread, and finally we insert spawn and endblock instructions to couple the threads.

4.1.1. Iterative main thread selection.

Given an initial seed for the main thread, we can compute the remainder of the thread using iterative dataflow analysis. After computing the control dependence and control flow graphs, we compute the remainder of the main thread (MT) by computing the set of variables that are live in the MT at each point in the program. We define a variable as MT_live at a point p in the program if for any possible execution of the program starting at point p, the variable is used by a MT instruction before it is overwritten. Any instruction that writes an MT_live variable is part of the MT.

The computation of MT_live sets is similar to live variable analysis. Like live variables, MT_live variables can be computed by iterative solution of dataflow equations. The dataflow equations for MT_live variables are shown below. Note that these equations are virtually identical to those for live variable analysis, except for the presence of $MT\_use(B)$ in place of $use(B)$.

$$\begin{aligned} MT\_live_{in}(B) &= MT\_use(B) \cup (MT\_live_{out}(B) - kill(B)) \\ MT\_live_{out}(B) &= \bigcup_{S \in Succ(B)} MT\_live_{in}(S) \end{aligned}$$

The actual computation of the MT_live sets is carried out by iterating over the CFG in reverse order. At the bottom of each basic block, we apply the equations to determine MT_live_{out}. Then, we iterate through the instructions in the block in reverse, keeping track of the MT_live set. If we encounter an instruction that writes a MT_live variable, we add it to the MT_inst_set. We also remove its outputs from and add its inputs to the MT_live set. If we encounter an instruction already in the MT_inst_set, we similarly remove its outputs from and add its inputs to the MT_live set, and in addition we add any instruction it is control dependent on to the MT_inst_set. At the top of the basic block we union MT_live_{in} with the MT_live set from within the block. We do this for every block in the CFG, and then repeat until there are no more additions to either the MT_inst_set or any of the MT_live sets.

4.1.2. Profile-directed optimizations.

In many cases, dataflow dependences that are present in the static code are not significant in the dynamic execution of the code. This is because many of the code paths in the program are rarely or never taken, and so dependencies across those paths never occur. We can identify which program paths are significant through profiling, and use the information to reduce the size of the main thread and hence improve its performance.

One such optimization is infrequent path pruning (IPP). Using profile information to identify infrequently taken paths, we remove infrequent edges from the CFG while computing MT liveness. If the profile information is accurate, IPP will remove unnecessary instructions from the MT more often than it introduces dependence violations.

A second profile directed optimization is biased branch exclusion (BBE). To apply BBE, we first determine from the profile which branches are highly biased, and add these branches to a table. Then, during the MT_liveness analysis, we propagate the MT_liveness of a variable only if its reader is not one of the biased branches. Finally, we mark each biased branch for delayed resolution.

4.1.3. Spawn Coalescing.

After the MT selection has taken place, we wind up with code that often interleaves MT and WT instructions. As each spawn instruction requires dependence checking, this can be wasteful. Spawn coalescing reorders contiguous blocks of MT and WT instructions to minimize the number of transitions between the two, and hence reduce the number of spawn (and endblock) instructions.

4.1.4. Checked spawns.

Profile-directed optimizations introduce backwards data dependences, where code in the main thread may depend on data values produced by the work thread. Though these dependences should only be realized across infrequently executed paths, we still need to check for violations with *pbr* checked spawns. To determine whether each spawn should be a *pbrnc* or *pbr*, we unprune the CFG and perform the MT_liveness analysis again. Wherever an instruction previously identified as being in the WT writes a value that is now MT_live, we must use a *pbr*.

4.2. Control Decoupling

Studies such as [11] have shown that control flow is a major bottleneck limiting the available instruction-level parallelism. A strategy to alleviate this bottleneck is to accelerate the computation of control decisions. On NXA, we can perform this by putting the control flow computation in the main thread, as a control thread, and deferring data computation to the work thread. By doing so, we alleviate resource constraints limiting the performance of the control-flow computing portion of the program.

To compute the control thread, we use the program's control instructions (i.e. branches, jumps, subroutine calls) as the seed for the iterative thread selection.

We can improve control decoupling with Local Control Exclusion (LCE), an optimization where we move "local" control flow to the work thread while reserving "global" control flow for the control thread. Because these local control flow constructs are not part of the control flow bottleneck, LCE can reduce the complexity of the control thread without increasing the control flow bottleneck. This can benefit performance significantly when, as is usually the case, the control thread would otherwise contain a very large fraction of the total program.

We perform LCE over three specific "local" control flow constructs: if-then-else constructs (ITE), tight loops consisting of a single basic block (TL), and loops containing an if-then-else (ITEL). This is done by excluding anything within these constructs from the iterative selection seed.

4.3. Memory Decoupling

As noted, control flow is a major bottleneck constraining the available parallelism. Another bottleneck is memory access. When a memory access suffers a long latency miss, all its dependent instructions are delayed. Because of resource constraints in real superscalar machines, this often delays surrounding instructions as well.

There are two solutions to this problem. One solution, seen in work on memory helper threads [3] [12] [4], is to perform miss-prone loads in advance of the main program execution, to alleviate the latency of such misses. Another solution, which we propose here, is to defer instructions not depended on by miss-prone loads, to alleviate the resource constraints that can delay the main execution flow. In this case the main thread is the memory thread, and the deferred instructions are the work thread.

We perform memory decoupling on the NXA architecture with a 2 step process. The first step is to examine the program and determine the miss-prone loads. These miss-prone loads are used as the seed for computing the memory thread.

To identify miss-prone loads, we ran each benchmark through a cache simulator, which provided the frequency with which each memory operation missed in the cache and also the cumulative latency incurred by those memory operations. The memory thread seed can be selected from this information by setting a threshold for the cumulative latency (i.e. an instruction is in the initial memory thread if it incurred a cumulative latency above a certain threshold) or by miss count (i.e. an instruction is in the initial memory thread if the miss count was above a certain threshold).

Several of the optimizations presented for control decoupling work the same way for memory decoupling. In particular, we can use IPP, BBE, and spawn coalescing unchanged from control decoupling. LCE, however, is not applicable since our initial main thread does not consist of control instructions.

4.4. Critical Path Decoupling

Finally, dataflow dependences themselves are a major constraint on achievable parallelism. Thus, just as we used work threads to defer non-control instructions and non-memory instructions, we can use them to defer non-critical path instructions.

The key to selecting the critical path thread is determining the critical path through the execution. There are several classes of "critical path" that can be obtained from a dynamic execution of the program. One is the longest spine of the dataflow tree height. Another is the longest spine of the dataflow tree height augmented with control dependences, in a manner similar to Lam and Wilson [11]. A third is dataflow+CD+memory latencies. Fourth, we can use dataflow+memory latencies, without control dependences. Beyond this, we can even use a critical path predictor similar to the one in [5]. In this paper we focus on dataflow+memory as experimentation showed it to provide the best performance.

Once we compute the critical path we want, we use the static instructions corresponding to that path as the seed for computing the critical path thread, using the same iterative selection algorithm as for control and memory thread selection.

As before, we can apply infrequent path pruning and BBE to critical path thread selection. In addition, we can reduce the size of the seed by noting that some static instructions occur in the dynamic critical path just a few times or even only once. In such cases, the penalty of having that instruction and its dependences in the critical path thread during all the non-critical executions might be greater than the benefit of having the instruction in the critical path thread during the few critical executions. Hence, we can eliminate from the seed instructions that are critical only once (the crit2 optimization).

5. PERFORMANCE EVALUATION

In this section, we use microarchitectural simulation to evaluate the performance of NXA with the three decoupling techniques.

5.1. Methodology

We simulate the performance of NXA using Joshua, a detailed microarchitectural simulator we have configured to execute Alpha binaries. Joshua simulates an array of aggressive out-of-order superscalar processors in a chip multiprocessor. Here, Joshua has been configured to simulate the NXA implementation described in Section 3. The parameters for the simulated baseline machine are given in Table 1. Additional parameters for the NXA machine are given in Table 5.1.

Table 1. Simulated processor parameters (per processor core)

processor parameter	value
Issue/retire width	4
In-flight instructions	256
Integer units	8: 3 simple int, 1 complex int, 1 multiply, 2 load, 1 store
FP units	3: 2 simple FP, 1 complex FP
Pipeline latency	minimum 15 cycles
L0 Cache	8KB/1 cycle latency instruction and 16KB/1 cycle latency data
L1 Cache	128KB/8 cycle latency unified
L2 Cache	4MB, 20 cycle latency
Memory	200 cycle latency
Branch prediction	2K entry BTB plus 1Mbit hybrid predictor

Table 2. Simulated NXA parameters

NXA parameter	value
Spawn queue size	256 spawns
Register communication queue	128 pending reads per processor
Register communication latency	2 cycles minimum
Register communication bandwidth	2 communications per cycle
Memory communication unit	320 total entries
Memory bypass latency	5 cycles minimum
Memory bypass bandwidth	1 bypass per cycle
Cache coherence	update protocol

As workloads, we used 7 benchmarks each from SPECint2000, SPECfp2000, and MediaBench. We used reduced MinneSPEC [9] input sets for the SPEC benchmarks, and the standard input data set for MediaBench. We use binaries compiled with the Compaq Alpha C V5.9 and C++ V6.5 compilers with optimization level -O4. All SPEC benchmarks were run for 200M instructions, while MediaBench benchmarks were run through to completion. Profile information was gathered from the first half of the performance run.

We perform the control, memory, and critical path decoupling via static binary analysis. The binary analyzer takes as input the program binary, decoded into an internal format, (optionally) the branch bias information from profiling, and finally the main thread seed generated by the memory or critical path analysis.

5.2. Performance results

Figure 5.2 shows the performance results by benchmark. The three foreground bars show, for each benchmark, the speedup obtained by control, memory, and critical path decoupling. Table 5.4 shows the combination of optimizations that provided this speedup for each benchmark. The gray background bars show the speedup attained by the best of the three decoupling models. The results show an average speedup of 1.16 for control decoupling, 1.14 for memory decoupling, and 1.15 for critical path decoupling. When

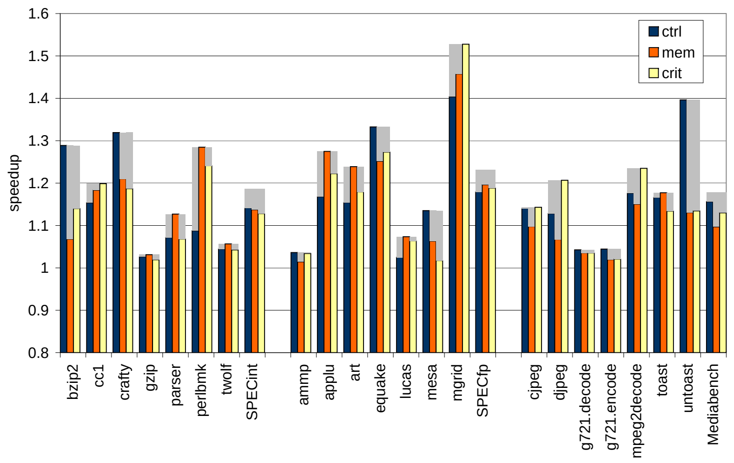


Figure 6. Performance of best decoupling scheme for each benchmark.

the best decoupling method (shown in the last column of Table 5.4) is applied to each benchmark, the average speedup becomes 1.20.

5.3. Effect of decoupling schemes

From Figure 5.2, we observe that different benchmarks perform best with different decoupling schemes. In general, we expect benchmarks that are control-flow bound to perform best with control decoupling, benchmarks that are memory bound to perform best with memory decoupling, and benchmarks that are bound by dependent chains of instructions to perform best with critical path decoupling.

By "control-flow bound", we mean those programs where being able to accelerate the computation of control flow decisions exposes significant additional parallelism. Performing well with control decoupling are bzip2, crafty, equake, and mesa, as well as untoast from Mediabench. Bzip2 and equake consist of tight loops with a relatively small amount of code. The control flow within these loops can be computed in parallel with the data generation along at least the more frequent paths. Most of mesa runs in nested loops where the inner loop is a tight loop that can be spawned off to the work thread. With untoast, control decoupling with LCE just manages to distribute the workload very evenly across the two processing cores, and control dependences are less of a factor.

The benchmarks that perform best with memory decoupling are gzip, parser, perlbmk, twolf, applu, art, lucas, and toast. Of these gzip, twolf, and lucas perform weakly on all the decoupling schemes. However, in parser, perlbmk, and art, upwards 30% of instructions are memory operations, and they are indeed memory bound. Applu has fewer memory operations, but it has a low hit rate in the dcache of just 86%. Toast on the other hand is not memory bound; in fact, toast is ALU bond and memory decoupling simply did a better job of distributing the code from toast across the

ALUs of the two processing cores than the other decoupling schemes.

The benchmarks performing best with critical path decoupling are gcc (cc1), mgrid, and jpeg/mpeg. Gcc has a large number of dependence chains that flow through memory operations that miss in the caches; when we compute the critical path using the dataflow+memory model, these chains appear in the critical path. Hence, critical path decoupling helps gcc performance by moving chains of dependences through miss-prone loads to the main thread and deferring chains through non-miss-prone loads to the work thread, reducing pressure on the load/store system during the critical sections of the code. On the other hand, mgrid is bound by the number of FP ALUs, and (similar to toast) critical path decoupling just does the best jobs of distributing the code across the ALUs. The reason is similar for the jpeg/mpeg benchmarks.

5.4. Effect of optimizations

Figure 5.4 shows the speedups obtained by specific decoupling methods with specific optimizations applied to all the benchmarks. From the graph, several trends can be observed.

Infrequent path pruning tends to benefit SPECint a lot, have mixed results on MediaBench, and hurt performance on SPECfp. The integer benchmarks tend to have a lot of control instructions and hence a lot of control dependence edges. Pruning helps cut down on the number of instructions that get included in the main thread due to those dependences. In the FP benchmarks, there are relatively few control instructions; hence, decreasing the size of the main thread by pruning hurts load balance. In MediaBench, some benchmarks observe the integer trend, while others observe the FP trend.

BBE has a mixed impact. On many benchmarks (especially FP), the reverse dependences introduced by the optimization impede performance much more than the reduction in the main thread's complexity. However, on some integer benchmarks such as crafty and gcc, there are highly biased, easily predicted branches whose inputs have a long chain of dependences. In this case, there is a substantial gain from being able to move this whole chain over to the work thread.

Overall, the results of Figure 5.4 illustrate the potential usefulness of heuristics to determine automatically which optimizations to apply.

5.5. Sensitivity studies

To further evaluate the NXA architecture design space, several hardware parameters were varied to determine their impact on performance. Two classes of parameters were investigated: inter-core communication latency and bandwidth. The best performing partitioning scheme for each benchmark was used in this study.

None of the communication parameter changes had an appreciable effect ($<1\%$) on performance. Register communication latency was increased up to 8 cycles while bandwidth was varied between 1 and 4 communications per cycle. Memory communication latency was increased up to 15 cycles while (independently) bandwidth was upped to 4 communications per cycle. The result gives evidence that our partitioning schemes work well to keep inter-thread communication flowing in only one direction.

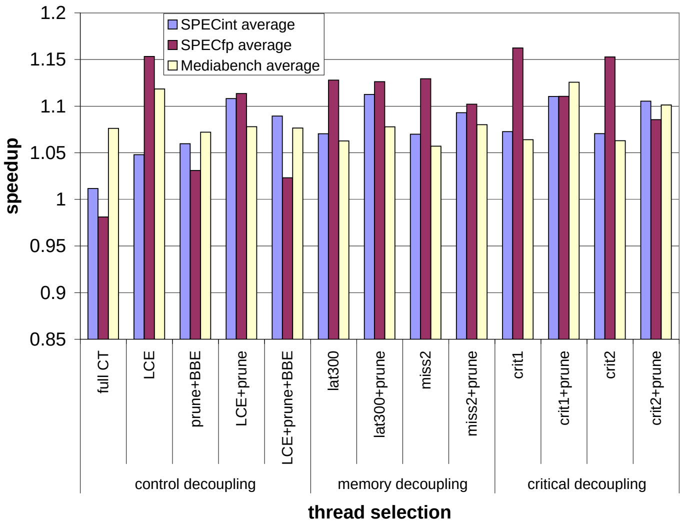


Figure 7. Speedup with different decoupling methods and optimizations.

6. RELATED WORK

This work extends a large body of work on decoupled architectures. Previous work in this area can be roughly classified into three categories: the early work based on premodern processor (non-superscalar or short pipeline) architectures; speculative multithreading, which attempts to find relatively course-grained, thread-level parallelism; and the slicing approach, where threads consist of "slices" of dependent instructions. Our hardware design borrows features from speculative multithreading, but differs greatly in the granularity of parallelism that we can exploit. Moreover, our work uses the slicing technique for generating the threads, but executes those threads in a fundamentally different manner, hence constituting a fourth category.

6.1. Early work

One of the first proposals for decoupled architectures was the Decoupled Access/Execute Architecture [17], where the memory address stream was decoupled from the execution stream. This enabled memory references to slip ahead of execution and provide a prefetching effect. The PIPE [7] work provides a more thorough evaluation of DAE. The ACRI work [1] was another early proposal for decoupled architecture. Bird et al. introduced the term "control decoupling", proposing to not only decouple the memory address

Table 3. Best combination of optimizations for each benchmark.

benchmark	ctrl	mem	crit	overall
bzip2	all LCE+prune	lat300+prune	crit2+prune	ctrl
cc1	all opts	lat300+prune	crit2+prune	crit
crafty	all opts	lat300+prune	crit2+prune	ctrl
gzip	ITEE+TLE	lat300+prune	crit2+prune	mem
parser	all LCE+prune	lat300+prune	crit2+prune	mem
perlbmk	all LCE	lat300	crit2	mem
twolf	all opts	miss2+prune	crit2+prune	mem
ammp	prune+BBE	miss2+prune	crit2+prune	ctrl
applu	all LCE	miss2	crit1	mem
art	ITEE	lat300+prune	crit1	mem
equake	all LCE+prune	lat300+prune	crit1+prune	ctrl
lucas	all LCE	miss2	crit2	mem
mesa	ITEE+TLE	lat300+prune	crit1	ctrl
mgrid	all LCE	miss2	crit1	crit
cjpeg	all opts	lat300+prune	crit2+prune	crit
djpeg	all LCE+prune	miss2+prune	crit1+prune	crit
g721.dec	all opts	miss2+prune	crit2+prune	ctrl
g721.enc	all LCE	miss2+prune	crit1+prune	ctrl
mpeg2dec	ITEE	lat300	crit1+prune	crit
toast	all opts	lat300+prune	crit2+prune	mem
untoast	ITEE+TLE	miss2+prune	crit1+prune	ctrl

stream, but also the control stream. Our work generalizes these techniques by evaluating several different methods of program partitioning rather than focusing specifically on the memory access or control stream.

6.2. Speculative multithreading

More recently, a variety of speculative multithreading architectures have been proposed. Speculative multithreading attempts to get around the limitations of traditional parallelizing compilers, which can only parallelize codes when they can guarantee correctness. The idea of speculative multithreading is to go ahead and parallelize even when the compiler is not sure, and then use hardware to squash the parallelism if it turns out incorrect.

An early example of a speculative multithreaded architecture is Multiscalar [6] [18]. Whereas Multiscalar required static analysis, SM processors proposed in [13] attempted to achieve parallelism using only control speculation in hardware. This technique achieved strong performance on FP codes but fell flat in parallelizing integer codes. More recent proposals, including [16], [10] and [19], propose mechanisms that require less hardware overhead. In [16], Olukotun et al. describe how to support fine-grain speculative parallelism on the Hydra CMP. In [10], a CMP design is proposed that adds a modest set of structures for speculative multithreading. This design differs from Multiscalar in loosening the appearance of a single register file, using a software tool to find where registers are shared. The architecture is further developed in [2]. In [19] and [20], Steffan et al. describe an approach that imposes even less hardware overhead, by extending cache coherence mechanisms to detect dependence violations.

The main difference between NXA and speculative multithreading is the method of thread selection and the source of the performance improvement. Instead of partitioning the dynamic instruction stream into relatively large contiguous regions based on control independence, we finely slice the instruction stream, isolating critical portions of control and data flow. By separating the worker threads from the main thread, we prevent non-critical work from competing for resources and slowing overall execution. Because we deal with finer grained parallelism, we are also able to reduce the degree of speculation by eliminating speculative spawns and hence simplify the hardware.

6.3. Slicing approaches

Still more recently, research in decoupled architecture has focused on slicing approaches. These approaches are exemplified by the various work on prefetching helper threads [3] [12] [4]. In these approaches, a helper thread is constructed from the program slice used to compute memory addresses rather than control-flow. The helper thread runs concurrently with the main thread and reduces the cache miss rate of the main thread. The branch-decoupled architecture [22] [15] attempts a similar feat on branches, where the branch outcomes from the branch thread serve as branch predictions for the main thread.

The Slipstream processor [21] [8] is another example of the slicing approach. In Slipstream, one processor executes a filtered program called the *advanced stream* (A-stream), using it to generate intermediate results that are fed to another processor running the original program (called the R-stream).

A final example of the slicing approach is master-slave speculative parallelization [23] [24]. In MSSP, there is one main thread that operates on speculative data and one or more lagging threads that run in parallel and verify the data. The main thread thus focuses on data computation while the slave threads are responsible for verifying much of the control flow.

Each of these slicing approaches makes the observation that either control flow, memory accesses, or error checking code can be pulled out of the program and run in parallel with it. With NXA we exploit all three observations with variations on a common parallelization algorithm and a single architecture.

At an implementation level, all of the previous slicing approaches share the same basic approach. Some slice/subset of the program is taken out and executed as a speculative thread. Then, another thread or multiple threads comes along and executes the entire program non-speculatively, using the speculative slice to speed its execution. Hence, all the approaches use a speculative thread/verification thread(s) paradigm. NXA does not. With NXA the chosen slice is not a prediction thread but rather the main thread of execution. The NXA approach has significant advantages. NXA manages to exploit the same underlying parallelism, but it does so without redundant instructions or speculative threads, reducing resource constraints during execution and simplifying the hardware implementation.

7. Conclusion

Automatic parallelization of sequential code is difficult on conventional chip multiprocessors. In this paper, we offer three contributions to alleviate the limitations:

1. NXA [†], a novel multi-core architecture based on fine-grained parallelism within a single thread of control. Previous approaches lacked the ability to exploit fine grained threads.

2. A unified paralellism model that generalizes slice-based approaches. We use a single architecture to exploit observations made by separate prior work that control flow, memory accesses, or error checking code could be pulled out of the program and executed in parallel with it.

3. Effective algorithms for parallelizing a non-parallel application using a fine-grained architecture.

We evaluate the performance of control, memory, and critical path decoupling on the NXA architecture through microarchitectural simulation and demonstrate that the combination does indeed offer performance benefits. By choosing threads with the best combination of optimizations for each benchmark, we obtained speedups of 1.19 on SPECint2000, 1.23 on SPECfp2000, and 1.18 on MediaBench when compared against an aggressive 4-wide superscalar processor.

[†]New eXciting Architecture

References

[1] P. L. Bird, A. Rawsthorne, and N. P. Topham. The effectiveness of decoupling. In *Proceedings of the 7th International Conference on Supercomputing*, pages 47–56, 1993.

[2] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 13–24, Vancouver, Canada, June 2000.

[3] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. fong Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, 2001.

[4] P. W. et. al. Helper threads via virtual multithreading on an experimental itanium 2 processor-based platform. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[5] B. A. Fields, S. Rubin, and R. Bodik. Focusing processor policies via Critical-Path prediction. pages 74–85.

[6] M. Franklin. The multiscalar architecture. Technical Report 1196, Computer Sciences Department, University of Wisconsin - Madison, Nov. 1993.

[7] J. R. Goodman, J. tu Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. Pipe: A vlsi decoupled architecture. In *ISCA*, pages 20–27, 1985.

[8] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 179–190, 2003.

[9] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research, 2002.

[10] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, Sept. 1999.

[11] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 248–259, 1992.

[12] C. Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, 2001.

[13] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, pages 77–84, 1998.

[14] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[15] A. S. Nadkarni and A. Tyagi. A trace based evaluation of speculative branch decoupling. In *IEEE International Conference on Computer Design ICCD'2000*, pages 300–, 2000.

[16] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *International Conference on Supercomputing*, pages 21–30, 1999.

[17] J. E. Smith. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, Nov. 1984.

[18] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[19] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[20] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, pages 65–, 2002.

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[22] A. Tyagi, H.-C. Ng, and P. Mohapatra. Dynamic branch decoupled architecture. In *IEEE International Conference on Computer Design ICCD'1999*, pages 442–450, 1999.

[23] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, 2001.

[24] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002.

CMP Cache Performance Projection: Accessibility vs. Capacity

Xudong Shi, Feiqi Su, Jih-kwon Peir, Ye Xia, Zhen Yang
Dept. of Computer and Information Science and Engineering
University of Florida
Gainesville, FL, USA
{xushi, fsu, peir, yx1, zhyang}@cise.ufl.edu

***Abstract*—Efficient utilizing on-chip storage space on Chip-Multiprocessors (CMPs) has become an important research topic. Tradeoffs between data accessibility and effective on-chip capacity have been studied extensively. It requires costly simulations to understand a wide-spectrum of the design space. In this paper, we first develop an abstract model for understanding the performance impact with respect to data replication. To overcome the lack of real-time interactions among multiple cores in the abstract model, we propose a global stack simulation strategy to study the performance of a variety of cache organizations on CMPs. The global stack logically incorporates a shared stack and per-core private stacks to collect shared/private reuse (stack) distances for every memory reference in a single simulation pass. With the collected reuse distances, performance in terms of hits/misses and average memory access times can be calculated for multiple cache organizations. We verify the stack results against individual execution-driven simulations that consider realistic cache parameters and delays using a set of commercial multithreaded workloads. Our results show that stack simulations can accurately model the performance of various cache organizations with 2-9% error margins. The single-pass stack simulation results demonstrate that the effectiveness of various techniques for optimizing the CMP on-chip storage is closely related to the working sets of the workloads as well as to the total cache sizes.**

Keywords: Performance Modeling and Projection, Stack Simulation, CMP Caches, Data replication.

I. Introduction

Organizing on-chip storage space on Chip-Multiprocessors (CMPs) has become an important research topic. Balancing between data accessibility due to wiring delay and the effective on-chip storage capacity due to data replication has been studied extensively [2, 17, 9, 22, 29, 8, 3, 24, 14, 13, 16]. The goal is to dynamically allocate data blocks closely to the requesting core for fast data access without adversely increasing expensive off-chip memory traffic. A shared cache organization provides the maximum cache capacity that leads to the least off-chip traffic. However, in such a design, data blocks are usually allocated across multiple banks, resulting in an increase in the cache access delay due to a high percentage of remote bank accesses. A private cache organization, on the other hand, reduces the cache access delay by allocating the recently-accessed blocks in the local cache. Multiple copies of a single block may exist in multiple caches to reduce remote accesses. However, multiple copies decreases the effective cache capacity and increases off-chip memory traffic.

A CMP memory hierarchy typically includes small, private instruction/data L1 caches for fast accesses. The interesting issue is the organization of the on-die L2 cache. Recently, there have been several research works [2, 17, 9, 22, 29, 8, 3] proposing various combined private/shared L2-cache organizations following two general directions. The first is to organize the L2 as a shared cache for maximizing the capacity. To shorten the access time, a portion of the L2 can be set aside for replication [29]. The second is to organize the L2 as private caches for minimizing the access time. To achieve higher effective capacity, data replications among multiple L2s are constrained and different private L2s can steal each other's capacity by block migration [22, 3, 8]. These studies must examine a wide-spectrum of the design space. Due to the lack of an efficient methodology, near-sighted conclusions can potentially be drawn as a result of missing comprehensive views from all essential design parameters.

Analytical models provide quick performance estimations [1, 12]. However, they usually depend on statistical assumptions, and generally cannot accurately model systems with complex real-time interactions among multiple processors [7]. For fast simulations, the stack simulation technique proposed in [20] simulates multiple cache sizes in a single pass under the LRU replacement policy. Several extensions and enhancement have been made to improve the speed of the single-pass stack simulation or the coverage to variable set-associativities [20, 4, 25, 12, 15, 23]. However, these stack simulation methods target only uniprocessor caches where no cache coherence complexity is involved and the memory access delay hardly affects the order of memory requests. Extensions of the stack simulation to multiprocessors are reported in [27, 28]. Those works focus on solving the problem of multiprocessor cache invalidations in stack simulations for general set-associative caches. However, the remote cache hit, an important measure on CMP, is not included. Furthermore, the accuracy of using traces to simulate different multiprocessor cache organizations is not evaluated.

In this paper, we present a general framework for fast projection of CMP cache performance. Four L2 cache organizations, Shared, private, shared with data replication, and private without data replication are studied. We focus on understanding the performance tradeoff between data accessibility and the cache capacity loss due to data replication. The outline and contributions of our approach are as follows.

1) Modeling Data Replication: We first develop an analytical model to assess general performance behavior with respect to data replications in CMP caches. The model injects replicas (replicated data blocks) into a generic cache. Based on the block reuse-distance histogram obtained from a real application, a precise equation is derived to evaluate the impact of the replicas. The results demonstrate that whether data replication helps or hurts L2 cache performance is a function of the total L2 size and the working set of the application. Existing CMP cache studies may have overlooked this general replication behavior because they failed to examine the entire design space.

2) Single-Pass Stack Simulation: To overcome the limitations of modeling, we developed a single-pass stack simulation technique to handle shared and private cache organizations with the invalidation-based coherence protocol. The stack algorithm can handle complex interactions among multiple private caches. For fast simulations, we adopt fully-associative stacks based on hashing. We also partition the stack into fixed-size groups to avoid scanning the entire blocks for each memory request [15]. This single-pass stack technique can provide local/remote hit ratios and the effective cache size for a range of physical cache capacities.

3) Performance Projection of Data Replication: We demonstrate that we can use the basic multiprocessor stack simulation results to estimate the performance of other interesting CMP cache organizations. For example, given different percentages of the L2 cache reserved for data replication, we can derive the average L2 access time under a shared L2 cache organization. Such a cache organization closely resembles the L2 cache with victim replication [29].

4) Performance Verification: Finally, we verify the projection accuracy from the stack simulation against the detailed execution-driven simulation for each individual cache configuration using three multithreaded workloads. We observe that both the modeling and the single-pass stack simulation produce consistent performance views for CMP caches with different degrees of data replication. We also show that the single-pass stack simulation produces small error margins ranging 2-9% in comparison with execution-driven simulations for all simulated cache organizations.

The paper is organized as follows. Section 2 describes the abstract analytical model established based on a multithreaded database workload (OLTP). The modeled performance results with respect to data replication are also discussed. Section 3 introduces the CMP stack simulation algorithm that handles both the private and the shared caches. Section 4 describes the simulation and validation methodology. This is followed by performance evaluations of the four target L2 organizations: shared, private, shared with data replication, and private without data replication in section 5. The results obtained from the stack simulations are verified against realistic CMP cache simulations. Section 6 has the related work. We give a brief conclusion in section 7.

II. Modeling Data Replication

In this section, we develop an abstract model independent of private/shared organizations to evaluate the tradeoff between the access time and the miss rate of CMP caches with respect to data replication. The purpose is to provide a uniform understanding on this central issue of caching in CMP that is present in most major cache organizations. This study also highlights the importance of examining a wide enough range of system parameters in the performance evaluation of any cache organization, which can be costly.

In Fig. 1, a generic histogram of block reuse distances is plotted, where the reuse distance is measured by the number of distinct blocks between two adjacent accesses to the same block. A distance of zero indicates a request to the same block as the previous request. The histogram is denoted by *f(x)*, which represents the number of block references with reuse distance x. For a cache size S, the total cache hits can be measured by $\int_0^S f(x)\,dx$, which is equal to the area under the range of the histogram curve from 0 to S. This well-known, stack distance histogram can provide hits/misses of all cache sizes with a fully-associative organization and the LRU replacement policy.

To model the performance impact of data replication, we inject replicas into the cache. Note that regardless the cache organization, replicas help to improve the local hit rate since replicas are created and moved close to the requesting cores. On the other hand, having replicas reduces the effective capacity of the cache, and hence, increases cache misses. We need to compare effect from the increase of local hits against that from the increase of cache misses.

Suppose we take a snapshot of the L2 cache and find a total of *R* replicas. As a result, only *S-R* cache blocks are distinct, effectively reducing the capacity of the cache. Note that the model does not make reference to any specific cache organization and management. For instance, it does not say where the replicas are stored, which may depend on factors such as shared or private organization. We will compare this scenario with the baseline case where all *S* blocks are distinct. First, the cache misses are increased by $\int_{S-R}^S f(x)\,dx$, since the total number of hits is now $\int_0^{S-R} f(x)\,dx$. On the other hand, the replicas help to improve the local hits. Among the $\int_0^{S-R} f(x)\,dx$ hits, a fraction *R/S* hits are targeting the replicas. Depending on the specific cache organization, not all accesses to the replicas result in local hits. A requesting core may find a replica in the local cache of another remote core, resulting in a remote hit. We assume that a fraction L accesses to replicas are actually local hits. Therefore, compared with the baseline case, the total change of memory cycles due to the creation of R replicas can be calculated by:

$$P_m \times \int_{S-R}^{S} f(x)\,dx \; - \; G_l \times \frac{R}{S} \times L \times \int_0^{S-R} f(x)\,dx \qquad (1)$$

where P_m is the penalty cycles of a cache miss; and G_l is the cycle gain from a local hit. With the total number of memory

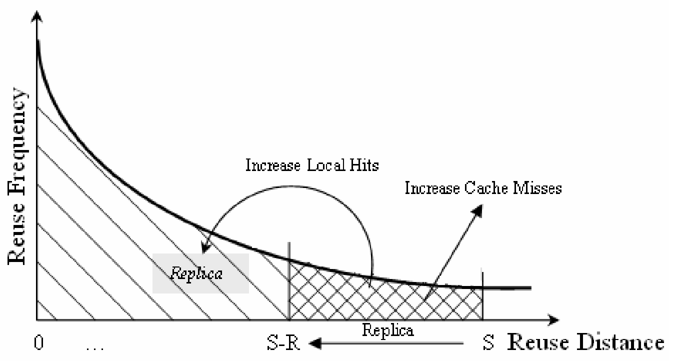


Figure 1. Cache performance impact with replica

accesses, $\int_0^\infty f(x)dx$, the average change of memory access cycles is equal to:

$$\left(P_m \times \int_{S-R}^{S} f(x)\,dx \;-\; G_l \times \frac{R}{S} \times L \times \int_0^{S-R} f(x)\,dx\right) \Big/ \int_0^\infty f(x)dx \qquad (2)$$

Now the key is to obtain the reuse distance histogram f(x). We conduct experiment using an OLTP workload [21] and collect its reuse distance histogram. With the curve-fitting tool of Matlab [19], we obtain the equation *f(x) = Aexp(-Bx)*, where $A = 6.084*10^6$ and $B = 2.658*10^{-3}$. This is shown in Fig. 2, where the cross marks represent the actual reuse frequencies from OLTP and the solid line is the fitted curve. We can now substitute *f(x)* into equation (2) to obtain the average change in memory cycles as:

$$P_m \times (\exp(-B \times (S-R)) - \exp(-BS)) \;-\; G_l \times \frac{R}{S} \times L \times (1 - \exp(-B(S-R))) \qquad (3)$$

Equation (3) provides the change in L2 access time as a function of the cache area being occupied by the replicas. In Fig. 3, we plot the change of the memory access time for three cache sizes, 2, 4, and 8 MB, as we vary the replicas' occupancy from none to the entire cache. In this figure, we assume G_l=15, P_m= 400, and L = 0.5. Note that negative values mean performance gain. We can observe that the performance of allocating L2 space for replicas for the OLTP workload varies with different cache sizes. For a 2MB L2, the results indicate no replication provides the shortest average memory access time, while for a larger 8MB L2 cache, keeping only 35% of distinct cache blocks and allocating the other 65% of cache for the replicas has the best memory performance. With a medium-sized 4MB L2 cache, allocating 40% of the cache for the replicas has the smallest access time. These results are consistent with the reuse histogram curve shown in Fig. 2. The reuse count approaches zero when the reuse distance is equal to or greater than 2MB. It increases significantly when the reuse distance is shorter than 2MB. Therefore, it is not wise to allocate space for the replicas when the cache size is 2MB or less.

The general behavior due to data replication is consistent with the detailed simulation result as will be given in Section 5. Further discrepancies can be explained as follows. Note that the fraction of replicas cannot reach 100% unless the entire cache is occupied by a single block. For a CMP with eight

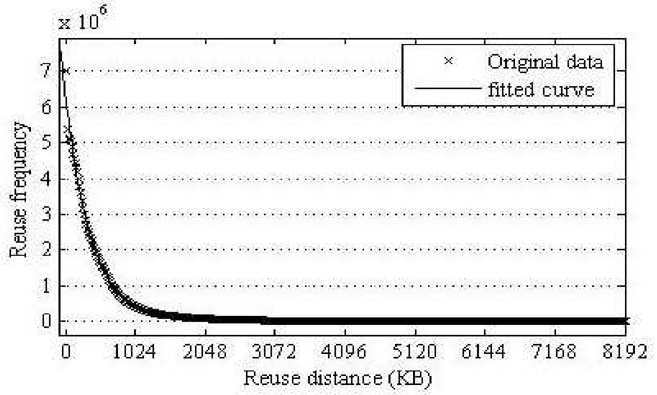


Figure 2. Curve fitting of reuse distance histogram of OLTP

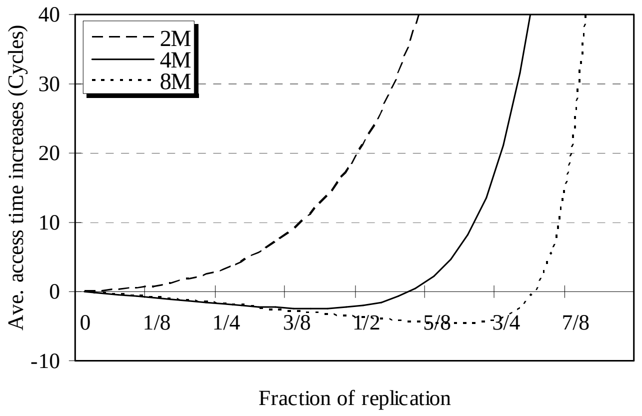


Figure 3. Performance with replicas for different cache sizes

cores, the fraction of the cache actually occupied by the replicas may approach three-fourth assuming the average degree of sharing is four. This is true even if the entire L2 is permissible for replication. Therefore, in Fig. 3, the average memory time increase is not very meaningful when the fraction of replicas is approaching to the cache size.

In addition to cache replication, we expect the cache sharing behavior also varies with different workloads and cache sizes. It is essential to study a set of representative workloads with a spectrum of cache sizes to understand the tradeoff of accessibility vs. capacity on CMP caches. A fixed replication policy may not work well for a wide-variety of workloads on different CMP caches. Although mathematical modeling can provide understanding of the general performance trend, its inability to model sufficiently detailed interactions among multiple cores makes it less useful for making accurate performance prediction. To remedy this problem, in the following section, we will describe a global-stack based simulation for studying CMP caches.

III. ORGANIZATION OF GLOBAL STACK

Fig. 4 sketches the organization of the global stack that records the memory reference history. In the CMP context, a block address and its core-id uniquely identify a reference, where the core-id indicates from which core the request is issued. Several independent linked lists are established in the

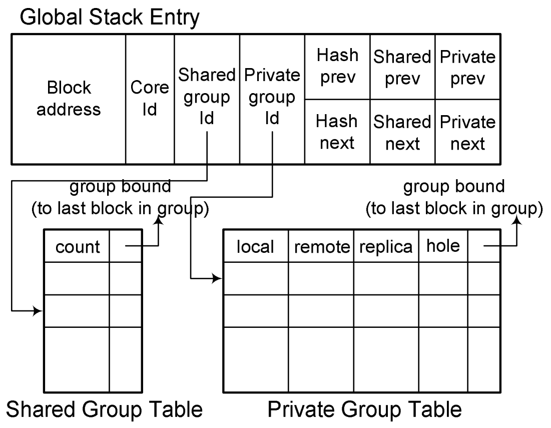


Figure 4. Global stack organization

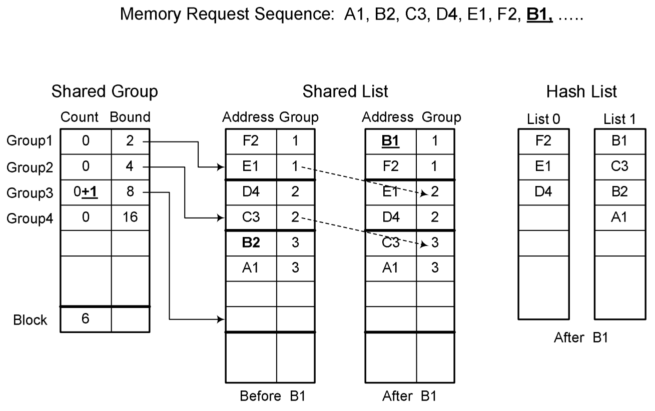


Figure 5. Shared cache example

global stack for simulating a shared and several per-core private stacks. Each stack entry appears exactly in one of the private stacks determined by the core-id, and may or may not reside in the shared stack depending on the recency of the reference. For fast search, an address-based hash list is also established in the global stack.

Since only a set of discrete cache sizes are of interest for cache studies, both the shared and the private stacks are organized as groups [15]. Each group consists of multiple entries for fast search during the stack simulation and for easy calculations of cache hits under various interesting cache sizes after the simulation. For example, assuming the cache sizes of interest are 16KB, 32KB, and 64KB. The groups can then be organized according to the stack sequence starting from the MRU entry with 256, 256, 512 entries for the first three groups, respectively, assuming the block size is 64B. Based on the stack inclusion property, the hits to a particular cache size are equal to the sum of the hits to all the groups accumulated up to that cache size. Each group maintains a reuse counter, denoted by G1, G2, and G3. After the simulation, the cache hits for the three cache sizes can be computed as G1, G1 +G2, and G1+G2+G3 respectively.

Separate shared and private group tables are maintained to record the reuse frequency count and other information for each group in the shared and private caches. A shared and a private group-id are kept in each global stack entry as a pointer to the corresponding group information in the shared and the private group table. The group bound in each entry of the group table links to the last block of the respective group in the global stack. These group bounds provide fast links for adjusting entries between adjacent groups. The associated counters are accumulated on each memory request, and will be used to deduce cache hit/miss ratios for various cache sizes after the simulation. The following subsections provide detailed stack operations.

A. *Shared Caches*

Each memory block can be recorded multiple times in the global stack, one from each core according to the order of the requests. Intuitively, only the first-appearance of a block in the global stack should be in the shared list since there is no replication in a shared cache. A first-appearance block is the one that is most recently used in the global stack among all blocks with the same address. The shared stack is formed by linking all the first-appearance blocks from MRU to LRU. Fig. 5 illustrates an example of a memory request sequence and the operations to the shared stack. Each memory request is denoted as a block address, A, B, C, …, etc., followed by a core-id. The detailed stack operations when B1 is requested are described as follows.

- Address B is searched by the hash list of the shared stack. B2 is found with the matching address. In this case, the reuse counter for the shared group where B2 resides, group 3, is incremented.
- B2 is removed from the shared list, and B1 is inserted at the top of the shared list.
- The shared group-id for B1 is set to 1. Meanwhile, the block located on the boundary of the first group, E1, is pushed to the second group. The boundary adjustment continues up to the group where B2 was previously located.
- If a requested block cannot be located through the hash list, (i.e. the very first access of the address among any cores), the stack is updated as above without incrementing any reuse counters.
- After the simulation, the total number of cache hits for a shared cache that is large enough to include exactly the first m groups is the sum of all shared reuse counters from group 1 up to group m.

B. *Private Caches*

The construction and update of the private lists are essentially the same as those of the shared list, except that we link accesses from the same core together. We collect crucial information such as the local hits, remote hits, and number of replicas, with the help of the local, remote, and replica counters in the private group table. For simplicity, we assume these counters are shared by all the cores, although per-core counters may provide more information. Fig. 6 draws the contents of the four private lists and the private group table, when we extend the previous memory sequence (Fig. 5) with three additional requests.

1) Local/Remote Reuse Counters

The local counter of a group is incremented when a request falls into the respective group in the local private stack. In this example, only the last request, A1, encounters a local hit, and in this case, the local counter of the second group is incremented. After the simulation, the sum of all local counters from group 1 to group m represents the total number of local hits for private caches with exactly m groups.

Counting the remote hits is a little tricky, since a remote hit may only happen when a reference is a local miss. For example, assume that a request is in the third group of the local stack; meanwhile, the minimum group id of all the remote groups where this address appears is the second. When the private cache size is only large enough to contain the first group, neither a local nor a remote hit happens. If the cache contains exactly two groups, the request is a remote hit. Finally, if the cache is extended to the third group or larger, it is a local hit. Formally, if an address is present in the local group L and the minimum id of the remote groups that contains it is R, the access can be a remote hit only if the cache size is within the range from group R to L-1. We increment the remote counters for groups R to L-1 (R <= L-1). Note that after the simulation, the remote counter m is the number of remote hits that a cache with exactly m groups encounters. To differentiate them from the local counters, we call them accumulated remote counters.

In the example, the first highlighted request, B1, encounters a local miss, but a remote hit to B2 in the first group. We accumulate the remote counters for all the groups. The second request, A2, is also a local miss, but a remote hit to A1 in the second group. The remote counter of the first group remains unchanged, while the counters are incremented for all the remaining groups. Similar to B1, all the remote counters are incremented for C1. Finally, the last request, A1, is a local hit in the second group and is also a remote hit to A2 in the first group. In this case, only the remote counter of the first group is incremented since A1 is considered as a local hit if the cache size extends to more than the first group.

2) Measuring Replica

The effective cache size is an important factor for shared and private cache comparisons [2, 9, 29, 8]. The single-pass stack simulation counts each block replication as a replica for calculating the effective cache size along the simulation. Similar to the remote hit case, we use accumulated replica counters. As shown in Fig. 6, the first highlighted request, B1, creates a replica in the first group, as well as any larger groups, because of the presence of B2. The second highlighted request, A2, does not create a new replica in the first group. But it does create a new replica in the second group because of A1. Meanwhile, A2 pushes B2 out of the first group, thus reduces a replica in the first group. This new replica applies to

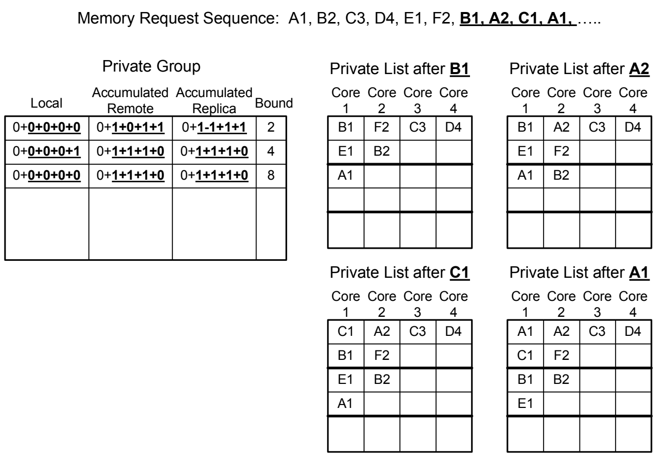


Figure 6. Private cache example

all the larger groups too. Note that the addition of B2 in the second group does not alter the replica counter for group 2, since the replica was already counted when B2 was first referenced. Similar to B1, the third highlighted request, C1, creates a replica to all the groups. Lastly, the reference, A1, extends a replica of A into the first group because of A2. The counters for the remaining groups stay the same.

3) Handling Memory Writes

In private caches, memory writes may cause invalidations to all the replicas. During the stack simulation, write invalidations create holes in the private stacks where the replicas are located. These holes will be filled later when the adjacent block is pushed down from a more-recently-used position by a new request. No block will be pushed out of a group when a hole exists in the group. To accurately maintain the reuse counters in the private group table, each group records the total number of holes for each core. The number of holes is initialized to the respective group size, and is decremented whenever a valid block joins the group. Maintaining the hole-count for each group can avoid searching for the existing holes.

IV. SIMULATION METHODOLOGY

We use the full-system Virtutech Simics 2.2 simulator [18] to simulate an 8-core CMP system with Linux 9.0 and x86 ISA. The processor module is based on the Simics Microarchitecture Interface (MAI) and models timing-directed processors in detail. Each core has its own instruction and data L1 cache. The global stack runs behind the L1 caches and simulates every L1 misses, essentially replacing the role of L2 caches. During simulations, stack distances and other related statistics are collected as described in Section 3. The results of the single-pass stack simulation are used to derive the performance of shared or private caches with various cache sizes and the sharing mechanisms for understanding the accessibility-vs.-capacity tradeoff in CMP caches.

The results from the stack simulation are verified against execution-driven timing simulations, where detailed cache models with proper access latencies are inserted. In the detailed timing simulation, we assume the shared L2 has eight banks, with one local and seven remote determined by the least-significant three bits of the block address. For the private L2, we model both local and remote accesses. The MOESI coherence protocol is implemented to maintain data coherence among the private L2s. For comparison, we use the hit/miss information and average memory access times to approximate the execution time behavior because the single-pass stack simulation cannot provide IPCs. Table 1 summarizes important simulation parameters.

We use three multithreaded commercial workloads, OLTP (Online Transaction Processing), Apache (Static web server), and SPECjbb (java server), as our benchmarks. We consider the variability of these multithreaded workloads by running multiple simulations for each configuration of each workload and inserting small random noises (perturbations) in the memory system timing for each run.

V. Evaluation and Validation

The accuracy of the CMP memory performance projection can be assessed from two different angles, the accuracy of predicting individual performance metrics, and the accuracy of predicting general cache behavior. By verifying the results against the timing simulation, we demonstrate that the stack simulation can accurately predict cache hits and misses for the four targeted L2 cache organizations, and more importantly, it can precisely project the sharing and replication behavior of the CMP caches.

One inherent weakness of stack simulation is its inability to insert accurate timing delays for variable L2 cache sizes. Since the fluctuation in memory delays may alter the sequence of memory accesses among multiple processors, we try a simple approach to insert memory delays into the stack simulation for several discrete cache sizes. In our simulation, these interesting cache sizes are 1MB, 2MB, 4MB, 8MB, to 16MB, and the corresponding simulation results are denoted as stack-1, stack-2, stack-4, stack-8, and stack-16, respectively. An off-chip cache miss latency is charged if the reuse distance is longer than each discrete cache size.

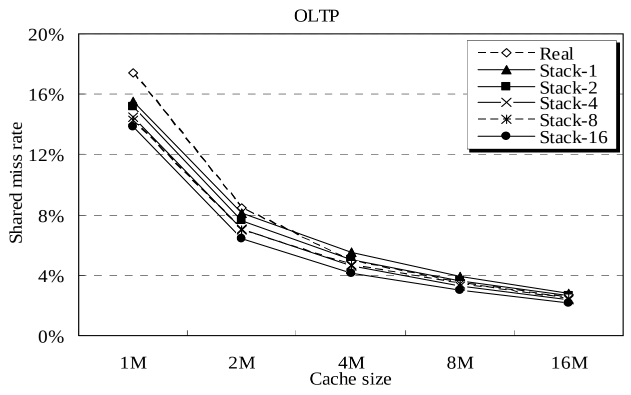
A. *Hits/Misses for Shared and Private L2 Caches*

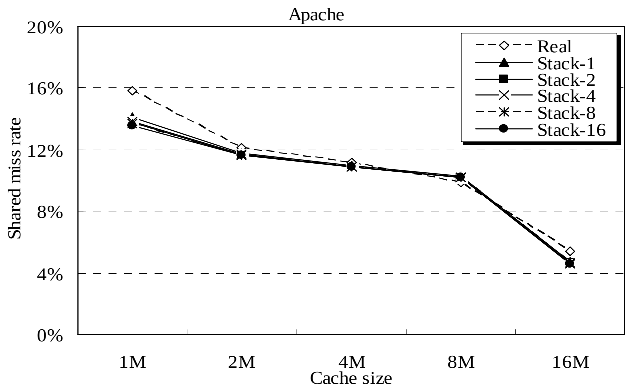
Fig. 7 shows the projected and real miss rates for shared caches, where “real” represents the results from the timing simulations. In general, the stack results follow the timing results closely. For OLTP, stack-2 matches the timing results with about 5-6% average error. For Apache and SPECjbb, the difference among different delay insertions is less apparent. The stack results predict the miss ratios with about 2-6% error, except for Apache with a small 1MB cache.

There are two major factors that affect the accuracy of the stack results. The first one is cache associativity. Since we use a fully-associative stack to simulate a 16-way cache, the stack simulation usually underestimates the real miss rates. This effect is apparent when the cache size is small, due to more conflict misses. This associativity issue can be solved with more complicated set-associative stack simulations [20, 12].

TABLE I. Simulation parameters

CMP: 8 cores, 3.2GHz, 128 entry ROB
Branch predictor: g-share, 64K, 4K BTB
Branch misprediction penalty: 10 cycles
L1-I: 32KB, 4-way, 64B line, MESI
L1-D: 32KB, 4-way, 64B line, MESI
L1-I/L1-D latency: 0/2 cycles
L2: 16-way, 64B line, MESI
Private L2 size (KB): 128/256/512/1024/2048 per core
Private L2 local/remote latency: 15/30 cycles
Shared L2: 8 banks, 1 local/7 remote
Shared L2 size (MB): 1/2/4/8/16
Shared L2 local/remote latency: 15/30 cycles
Memory latency: 400 cycles
Stack: 16KB / group, 1024 groups (16MB maximum)





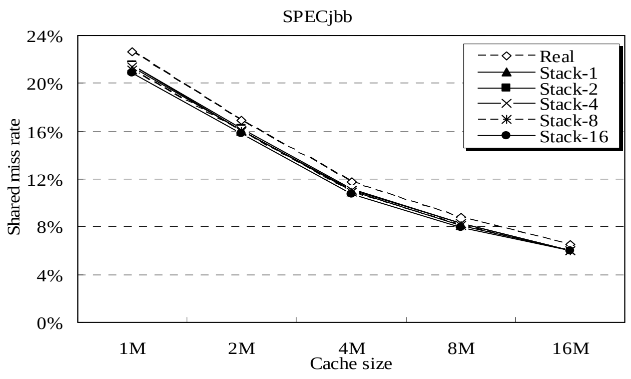


Figure 7. Miss ratios for shared caches

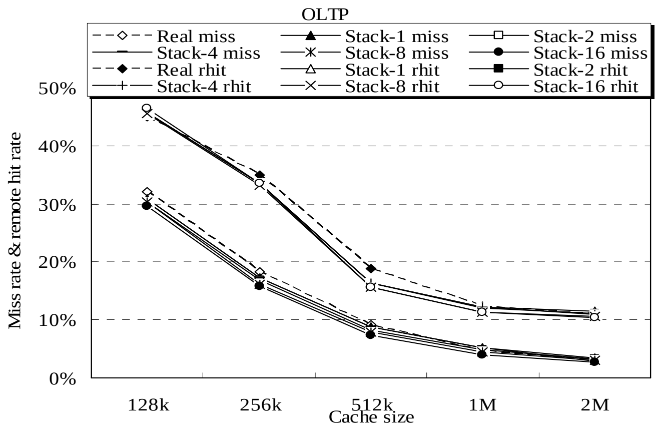
For simplicity, we keep the stack fully-associative. More sensitivity studies also need to be done to evaluate L2 with smaller set associativity. The second effect is due to inaccurate delay insertions. For example, the OLTP stack-1 simulation of a larger cache inserts cache miss latency for accesses with reuse distance falling between 1M and the actual cache size, where cache hit latency should have been applied. Such wrongly inserted extra delays for larger caches cause more OS interference and context switches that may lead to more cache misses. At 4MB cache size, the overestimate of cache misses due to the extra delay insertion exceeds the underestimate due to the full associativity. The gap becomes wider with larger caches. On the other hand, the stack-16 simulation for a smaller cache mistakenly inserts hit latency, instead of miss latency, for accesses with reuse distance from the cache boundary to 16M, causing less OS interferences, thus less misses. In this case, both the full associativity and the delay insertion lead to underestimate of the real misses, which makes the stack-16 simulation the most inaccurate.

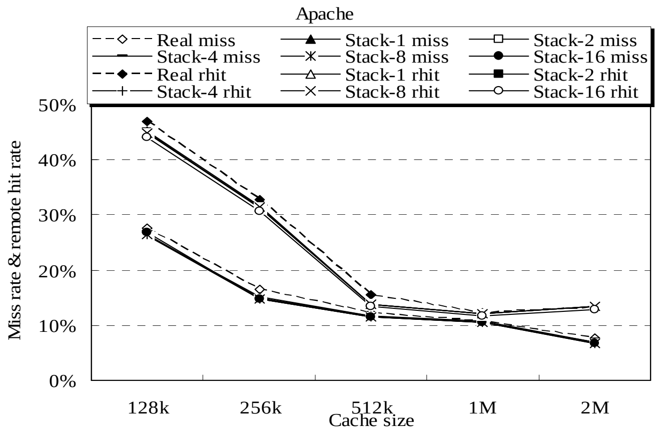
Fig. 8 shows the private cache results. The overall misses, the remote hits are compared. Note that the horizontal axis shows the size of a single core. With eight cores, the total sizes of the private caches are comparable to the shared cache sizes in Fig. 7. We can make two important observations. First, in comparison with the shared cache, the timing simulation results show that the overall L2 miss ratios are increased by 14.7%, 9.9%, 4.3%, 1.1%, and 0.5% for OLTP for the respective private cache sizes from 1MB to 16MB. For Apache and SPECjbb, the L2 miss ratios are increased by 11.8%, 4.4%, 1.1%, 1.0%, 2.2%, and 7.3%, 3.1%, 2.9%, 0.6%, 0.5%, respectively. Fig. 9 further plots the average L2 access time ratio between the private caches and the shared caches with equal capacity. When the total capacity is small, although the private-cache cases have more local hits, they also encounter more L2 misses. The private cache may have up to 50% longer average L2 access time. However, when the total capacity is sufficiently large, the private cache becomes better. With bigger caches, the difference in L2 misses diminishes, but the private L2s have much more local hits, which makes the average L2 access time shorter.

Second, the estimated miss rates and remote hit rates from the stack simulation match closely to the results from the timing simulations, with less than 10% error margin. We also simulate the effective capacity for the private-cache cases. The effective cache size is the average over the entire simulation period. In general, the private cache does reduce the cache capacity due to replicated and invalid cache entries. The effective capacity is reduced to 45-75% for the three workloads with various cache sizes. The estimated capacity based on the stack simulations is almost identical to the result from the timing simulations. Due to its higher accuracy, we use the stack-2 simulation in the following discussion.

B. Shared Caches with Replication

To balance accessibility and capacity, victim-replication [29] creates a dynamic L1 victim cache for each core in the





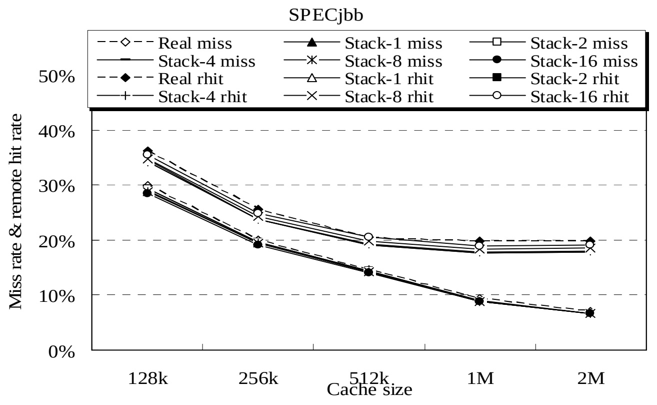


Figure 8. Miss ratio, remote hit ratio for private caches

local slice of the L2 to trade capacity for fast local access. In this section, we analyze the performance of a static victim-replication scheme. We allocate from 0% to 50% of the L2 capacity as the L1 victim caches with variable L2 capacities from 2MB to 8MB. For performance comparison, we use the average memory access time, which is calculated based on the local hits to the victim caches, the remote hits, the hits to the shared L2, and the L2 cache misses.

The average memory access time of the static victim replication can be derived directly from the results of the stack simulation described in the previous sections. Assuming the

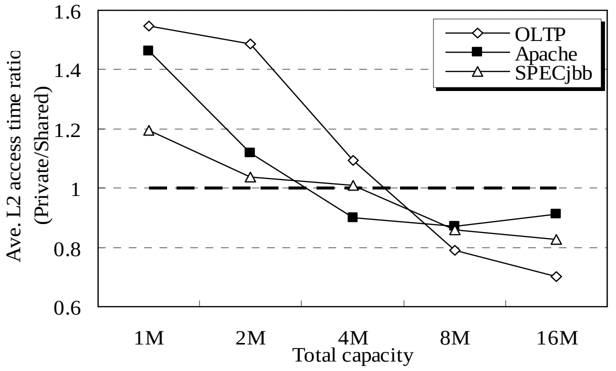
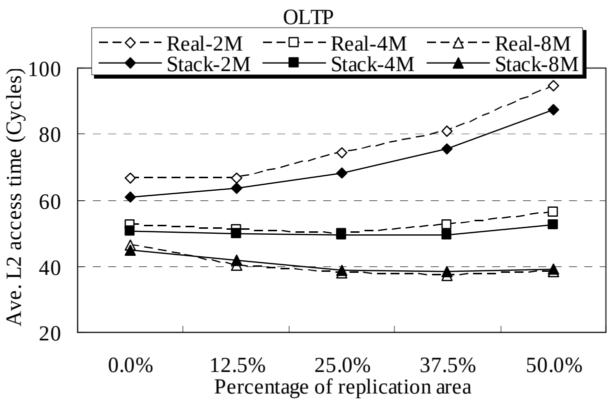
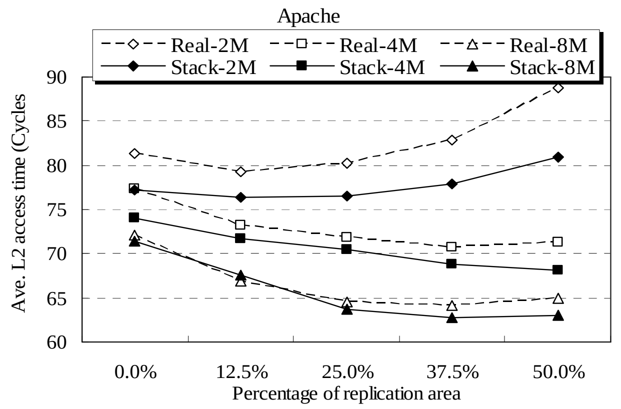


Figure 9. Ave. L2 access time ratio (private/shared)





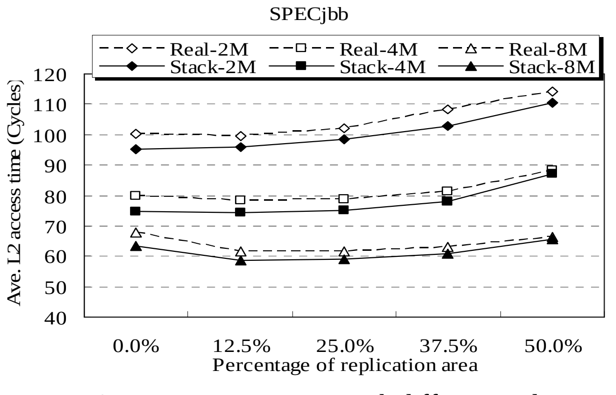


Figure 10. Ave. L2 access time with different replication

inclusion property is enforced between the shared potion of the L2 and the victim potion plus the L1. Suppose the L1 and L2 sizes are denoted by C_{L1}, and C_{L2}, r is the percentage of the L2 allocated for the victim cache, and n is the number of the cores. Then, each victim-cache size is equal to $(r*C_{L2})/n$, and the remaining shared portion is equal to $(1-r)*C_{L2}$. The average memory access time includes the following components. First, since the L1 and the victim cache are exclusive, the total hits to the victim cache can be estimated from the private stacks with the size of the L1 plus the size of the victim:$C_{L1}+(r*C_{L2})/n$. Note that this estimation may not be precise due to the lack of the L1 hit information that alters the sequence in the stack. Second, the hit to the shared portion of the L2 and the L2 misses can be calculated from the shared stack with the size $(1-r)*C_{L2}$. Finally, the hit to the remote cache can be calculated by subtracting the hits to the victim, the hits to the shared L2, and the L2 misses from the total L2 accesses.

Fig. 10 demonstrates the average L2 access time with static victim replication. Generally, large caches favor more replications. For a small 2MB L2, except that Apache has a slight performance gain at low replication levels, the average L2 access times increase with more replications. The optimal replication levels for OLTP are 12.5%, and 37.5% respectively for 4MB and 8MB L2. Such general performance behavior with respect to data replication for OLTP is similar to what we have observed from the analytical model in section 2. For SPECjbb, 12.5% replication shows the best for both 4MB and 8MB L2. The figure for Apache shows that the performance is better with replications as large as 50% for 4MB, but 37.5% for 8MB L2s. The seemly contradiction comes from the fact that L2 misses start to reduce drastically around 8M caches, as demonstrated in Fig. 7. We can also observe that the optimal replication levels match perfectly between the stack simulations and the real timing simulations. With respect to the average L2 access time, the stack results are within 2%-8% error margins.

C. Private Caches without Replication

Private caches sacrifice capacity for fast access time. It may be desirable to limit replications in the private caches. To understand the performance of the private L2 without replication, we run a stack simulation in which the creation of a replica causes the invalidation of the original copy.

Fig. 11 demonstrates the L2 access delays of the private caches without replication, shown as the ratio to those of the private caches with full replication, obtained from both the stack and timing simulations. As expected, with small 128KB and 256KB private caches per core, the average L2 access times without replication are about 5-17% lower than those with full replication for all the three workloads. This is because the benefit of the increased capacity more than compensates the loss of local accesses. With large 1MB or 2MB caches per core, the average L2 access time of the private caches without replication is 12-30% worse than the full-replication counterpart, suggesting that increasing local

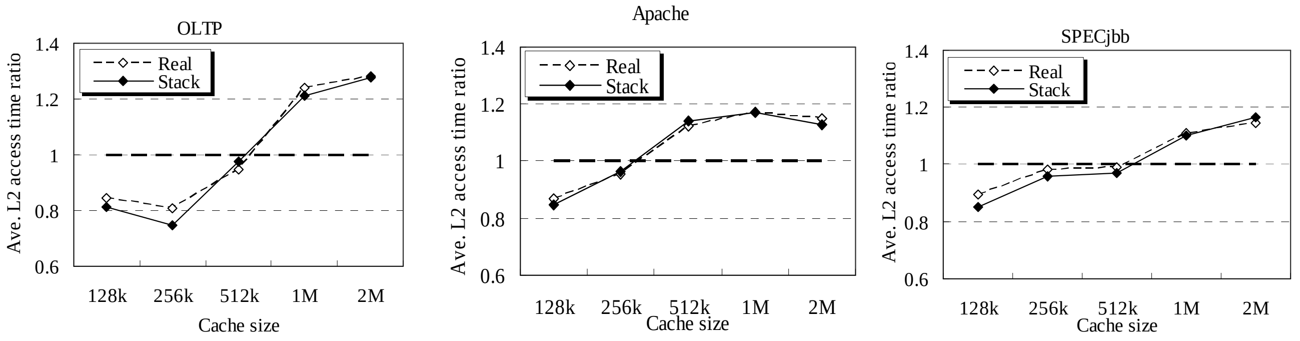


Figure 11. Average L2 access time ratio of private caches without replication

accesses is beneficial when enough L2 capacity is available. The stack simulation results follow this trend perfectly. They provide very accurate results with only 2-5% margin of error.

VI. Related Work

Optimizing on-chip storage space on CMPs has been studied extensively [2, 17, 9, 22, 29, 8, 3, 24, 14, 13, 16]. The goal is to dynamically allocate data blocks for fast access without adversely increasing off-chip traffic due to the L2 misses. These studies must examine a wide-spectrum of the design space, which requires costly simulations.

There have been several techniques for speeding up cache simulations. Mattson, et al. [20] presents a stack algorithm to measure cache misses for multiple cache sizes in a single pass. For fast search through the stack, tree-based stack algorithms [4, 26] are proposed. Kim, et al. [15] provide a much faster simulation by maintaining the reuse distance counts only to a few potential cache sizes. All-associativity simulations [7, 2] and generalized forest simulations [12, 23] allow a single-pass simulation for variable set-associativities. Meanwhile, various prediction models have been proposed to provide quick cache performance estimation [1, 11, 10, 26, 5, 6]. They apply statistical models to analyze the stack reuse distances. But, it is generally difficult to model systems with complex real-time interactions among multiple processors. StatCache [5] estimates capacity misses using sparse sampling and static statistical analysis.

All above techniques target uniprocessor systems where there is no interference between multiple threads. Several works aim at modeling multiprocessor systems [27, 28, 7, 6]. StatCacheMP [6] extends StatCache to incorporate communication misses. However, it assumes a random replacement policy for the statistical model. Chandra, et al [7] propose three analytical models based on the L2 stack distance or circular sequence profile of each thread to predict inter-thread cache contentions on the CMP for multiprogrammed workloads that do not have interference with each other. Two other works [27, 28] pay attention only to miss ratios, update ratios, and invalidate ratios for multiprocessor caches. The proposed stack simulation method aims at the L2 caches on CMPs where the remote cache hits are an important performance metric. The single-pass stack simulator simulates both shared and private L2 caches, and projects the cache performance for various CMP cache organizations with different degrees of sharing.

VII. Conclusion

In this paper, we developed an abstract model for understanding the general performance behavior of data replication in CMP caches. The model showed that data replication could degrade cache performance without a sufficiently large capacity. We then used the global stack simulation for more detailed study on the issue of balancing accessibility and capacity for on-chip storage space on CMPs. With the stack simulation, we can explore a wide-spectrum of the cache design space in a single simulation pass, which otherwise requires multiple, costly simulations. We simulated the schemes of shared caches, shared caches with replication, private caches, and private caches without replication of various cache sizes, and generated the cache hits/misses and the average memory access time of each scheme. We verified the stack simulation results with the execution-driven simulations using a set of commercial multithreaded workloads and demonstrated that the single-pass stack simulation results can characterize the CMP cache performance with high accuracy. In addition, the stack simulation can accurately estimate the trends in the average L2 access time when data replicas are injected into the shared caches or when they are limited to the private caches, under different L2 sizes. Our results proved that the effectiveness of various techniques to optimize the CMP on-chip storage is closely related to the L2 size.

References

[1] A. Agarwal, M. Horowitz and J. Hennessy, "An analytical cache model," ACM Transactions on Computer Systems, Vol. 7, No. 2, May 1989, pp. 184-215.

[2] B. Beckmann and D. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," Proc. of 37th Int'l Symp. on Microarchitecture, Dec. 2004, pp. 319-330.

[3] B. M. Beckmann, M. R. Marty, and D. A. Wood. "ASR: Adaptive Selective Replication for CMP Caches," Proc. of the 39th Int'l Symp. on Microarchitecture, Dec. 2006.

[4] B. T. Bennett and V. J. Kruskal, "LRU Stack Processing," IBM journal of R & D, July 1975, pp. 353-357.

[5] E. Berg and E. Hagersten, "StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis," Proc. of Int'l Symp. on Performance Analysis of Systems and Software, March 2004.

[6] E. Berg, H. Zeffer and E. Hagersten, "A Statistical Multiprocessor Cache Model," Proc. of Int'l Symp. on Performance Analysis of Systems and Software, March 2006.

[7] D. Chandra, F. Guo, S. Kim and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture", Proc. of 11th Int'l Symp. on HPCA, Feb. 2005, pp. 340-351.

[8] J. Chang and G. Sohi, "Cooperative Caching for Chip Multiprocessors," Proc. of 33rd Int'l Symp. on Computer Architecture, June 2006.

[9] Z. Chishti, M. D. Powell and T. N. Vijaykumar, "Optimizing Replication, Communication, and Capacity Allocation in CMPs," Proc. of 32nd Int'l Symp. on Computer Architecture, June 2005, pp. 357-368.

[10] G. Edwards, S. Devadas, and L. Rudolph, "Analytical Cache Models with Applications to Cache Partitioning," Proc.of 15th Int'l Conf. on Supercomputing, June 2001, pp. 1-12.

[11] B. Fraguela, R. Doallo, and E. Zapata, "Automatic Analytical Modeling for the Estimation of Cache Misses," Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques, Sep. 1999.

[12] M. Hill and J. Smith, "Evaluating Associativity in CPU Caches", IEEE Transactions on Computers, Dec. 1989, pp. 1612-1630.

[13] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," Proc. of 19th Int'l Conf. on Supercomputing, June, 2005.

[14] C. Kim, D. Burger, and S. Keckler, "An Adaptive Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," Proc. of 10th Int'l Conf. on, Oct. 2002.

[15] Y. H. Kim, M. D. Hill and D. A. Wood, "Implementing Stack Simulation for Highly-associative Memories," Proc. of 1991 SIGMETRICS conf. on Measurement and Modeling of Computer Systems, May 1991, pp. 212-213.

[16] R. Kumar, V. Zyuban, and D. M. Tullsen. "Interconnections in Multi-core Architectures: Understanding Mechanisms, Overhead and Scaling," Proc. of 32nd Int'l Sump. on Computer Architecture, June 2005.

[17] C. Liu, A. Sivasubramaniam and M. Kandemir, "Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs," Proc. of 10th Int'l Symp. on HPCA, Feb. 2004, pp. 176-185.

[18] P. S. Magnusson et al. "Simics: A Full System Simulation Platform," IEEE Computer, Feb. 2002, pp. 50-58.

[19] Matlab, http://www.mathworks.com/products/matlab/.

[20] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation Techniques and Storage Hierarchies," IBM Systems Journal, 9, 1970, pp. 78-117.

[21] Open source development labs database test 2. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/.

[22] E. Speight, H. Shafi, L. Zhang and R. Rajamony, "Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors," Proc. of 32nd Int'l Symp. on Computer Architecture, June 2005, pp. 346-356.

[23] R. A. Sugumar and S. G. Abraham, "Set-associative Cache Simulation using Generalized Binomial Trees," ACM Transactions on Computer Systems, Vol. 13, No. 1, Feb. 1995, pp. 32-56.

[24] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," The Journal of Supercomputing, 28(1), 2004, pp. 7-26.

[25] J. G. Thompson, "Efficient Analysis of Caching Systems," Computer Science Division Technical Report UCB/Computer Science Dept. 87/374, University of California, Berkeley, October 1987.

[26] X. Vera and J. Xue, "Let's Study Whole-Program Cache Behavior Analytically," Proc. of 8th Int'l Symp. on High Performance Computer Architecture, Feb. 2002.

[27] C. E. Wu, Y. Hsu, Y. Liu, "Efficient Stack Simulation for Shared Memory Set-Associative Multiprocessor Caches," Proc. of 1993 Int'l Conf. on Parallel Processing, Aug. 1993.

[28] Wu, Y. and Muntz, R. 1995, "Stack Evaluation of Arbitrary Set-Associative Multiprocessor Caches," IEEE Transactions on Parallel and Distributed Systems, Sep. 1995, pp. 930-942.

[29] M. Zhang, and K. Asanovic, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," Proc. of 32nd Int'l Symp. on Computer Architecture, June 2005, pp. 336-345.

From Chaos to QoS: Case Studies in CMP Resource Management

Hari Kannan[!], Fei Guo[$], Li Zhao[*], Ramesh Illikkal[*], Ravi Iyer[*], Don Newell[*], Yan Solihin[$], Christos Kozyrakis[!]
[!] Stanford University, Stanford, CA, USA
[$] North Carolina State University, Raleigh, NC, USA
[*] Intel Corporation, Hillsboro, OR, USA

Abstract

As more and more cores are enabled on the die of future CMP platforms, we expect that several diverse workloads will run simultaneously on the platform. A key example of this trend is the growth of virtualization usage models. When multiple virtual machines or applications or threads run simultaneously, the quality of service (QoS) that the platform provides to each individual thread is non-deterministic today. This occurs because the simultaneously running threads place very different demands on the shared resources (cache space, memory bandwidth, etc) in the platform and in most cases contend with each other. In this paper, we first present case studies that show how this results in non-deterministic performance. Unlike the compute resources managed through scheduling, platform resource allocation to individual threads cannot be controlled today. In order to provide better determinism and QoS, we then examine resource management mechanisms and present QoS-aware architectures and execution environments. The main contribution of this paper is the architecture feasibility analysis through prototypes that allow experimentation with QoS-Aware execution environments and architectural resources. We describe these QoS prototypes and then present preliminary case studies of multi-tasking and virtualization usage models sharing one critical CMP resource (last-level cache). We then demonstrate how proper management of the cache resource can provide service differentiation and deterministic performance behavior when running disparate workloads in future CMP platforms.

1. INTRODUCTION

As the momentum behind on-chip multiprocessor (CMP) architectures [7][12][13] continues to grow, it is expected that future client and server microprocessors will have several cores sharing the on-die and off-die resources. The success of CMP platforms depends not only on the number of cores but also heavily on the platform resources (cache, memory, etc) available and their efficient usage. Traditionally, processor and platform architectures are designed to perform well when a single parallel application is running on them. However, with the evolving software use models, CMP platforms will also be used to run multiple applications simultaneously in both client and server domains. The rapid deployment of virtualization [2][6][15][17] as a means to consolidate multiple applications on to a platform is a prime example.

When multiple applications run simultaneously on CMP architectures, the quality of service (QoS) that the platform provides to each individual application will be non-deterministic (*or chaotic*) because it depends heavily on the behavior of the other simultaneously running workloads. As expected, recent studies [3][5][9][10][14] have indicated that contention for critical platform resources (e.g. cache) is the primary cause for this lack of determinism and QoS. In this paper, we highlight this problem further and motivate the need for QoS support in CMP platforms. We focus on one of the important CMP platform resources (last-level cache space) and show case studies describing the effect of sharing this resource in multi-tasking and virtualization scenarios. Based on these observations, we investigate QoS policies and mechanisms to efficiently manage these shared resources in the presence of disparate applications (or threads).

Recent studies on (cache) resource management have either advocated the need for fair distribution [3] between threads and applications or the need for unfair distribution [5] with the purpose of improving overall system performance. In contrast, the work presented here aims to improve the performance of an individual application at the cost of the potential detriment of others with guidance from the operating environment. This is motivated by usage models such as server consolidation where service level agreements motivate the degree of performance differentiation [1][4] desired for some applications. Since the relative importance of the deployed applications is best managed by the operating software environment, we experiment with software-guided priorities (e.g. assigned by server administrators) to efficiently manage hardware resources. We compare the use of software-guided priorities (QoS-aware caches) against dedicated runs (isolated cache use), shared runs (unmanaged shared caches) as well as fair caches (equal private cache partitions per application).

In order to experiment with QoS, we have developed two QoS software prototypes (QoS-aware Linux as well as QoS-aware Xen). The primary contribution of this paper is the description of these prototypes as well as the case studies performed using them. The case studies include multi-tasking usage scenarios as well as virtualized usage scenarios and show the trade-offs between four resource management approaches (dedicated, shared, fair and QoS) on CMP platforms. We show that enabling and enforcing software-guided priorities is much more effective in providing QoS as compared to traditional approaches like static partitioning to CMP cache management.

2. Background and Challenges

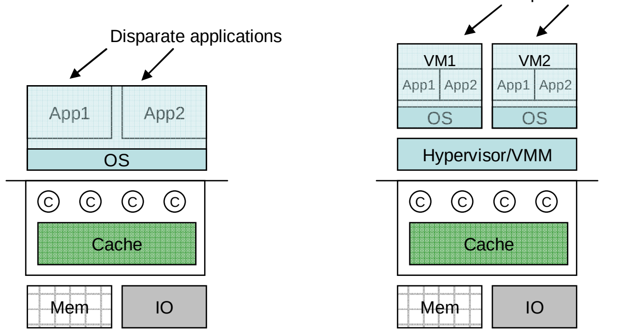
In this section, we highlight the motivation behind QoS by describing disparate threads of execution on CMP platforms and the shared resource problem.

2.1. Disparate Threads of Execution

The key trends in software that promote the use of disparate threads of execution (Figure 1) on CMP platforms are as follows:

(a) ***Multi-tasking becoming more common:*** As more threads and cores are enabled on the processor die, the compute capability is best utilized by multiple simultaneously executing tasks or applications (see Figure 1a). The behavior and platform resource usage of these simultaneous threads of execution will be quite disparate in nature (e.g. one is cache-friendly and the other is streaming in nature). At the same time, it is possible that one application is of much more importance than the other (e.g main application execution running along with a background streaming activity like network backup).

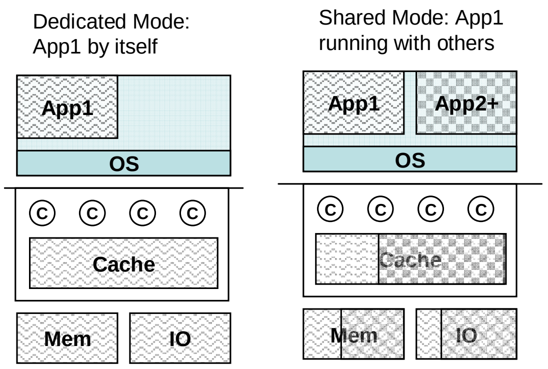
(b) ***Virtualized workloads becoming mainstream:*** While the concept of virtualization [6] has been around for a while, the recent re-emergence of virtualization as a means to consolidate workloads in the datacenter exposes the critical need to pay attention to the performance behavior of heterogeneous virtual machines running simultaneously on a server. This becomes even more important as virtualization-based usage models continue to rapidly evolve and encompass office workstations/desktops and even home PCs/laptops. In these scenarios, many disparate workloads are consolidated together and hardware performance isolation [4] becomes a desired feature for the high priority applications that can be identified by the user or system administrator.



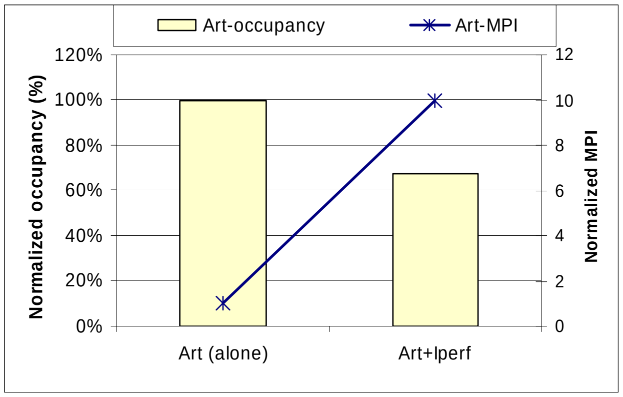
(a) Multi-Tasking (b) Virtualization/Consolidation
Figure 1. Disparate Threads on CMP Platforms

2.2. Example Resource Sharing Impact

In this section, we present an example of resource sharing impact based on trace-driven simulation of cache sharing in a CMP platform.


Dedicated Mode: App1 by itself
Shared Mode: App1 running with others
App1
App2+
OS
Cache
Mem
IO

(a) Illustrating Dedicated and Shared Usage



(b) Performance Impact of Resource Sharing
Figure 2. Implications of Resource Sharing

Figure 2 illustrates the motivation for QoS and resource management. The figures show the resource and performance implications of a high priority application running in standalone (dedicated) mode versus when it is running in shared mode with a low priority application. We chose Art - a SPEC 2000 benchmark [16], to represent a high priority application, and Iperf (a network benchmark) to represent the low priority application. These experiments were conducted in a virtualized environment with each application running in a different virtual machine. This simulation study (Figure 2b) shows how sharing the cache uniformly between applications affects the performance of the high priority application (Art). The Y axis on the left depicts the cache occupancy of Art relative to the case when it runs alone. The Y axis on the right plots the MPI (Misses per Instruction) of Art, also relative to the case when it runs alone.

Figure 2b shows that the high priority application's MPI increases by 9.9X when co-scheduled with Iperf. Iperf exhibits very poor cache locality and effectively thrashes the cache, which accounts for the observed degradation in

Art's performance. In order to minimize the effects of resource contention, the priority of the application needs to be comprehended by the platform in order for it to allocate hardware resources (in this case, cache space) accordingly. In this paper, we experiment with QoS policies and mechanisms to manage the cache resource distribution between high and low priority applications. It should be noted that the need for resource management stems both from the need for providing better QoS to high priority applications as well as the need for better determinism in the platform.

3. FROM CHAOS TO QOS

In this section, we introduce the QoS policies for resource management, and the architectural modifications necessary to support these policies.

3.1. Resource Management & QoS Policies

Most modern processors do not enforce any QoS in the cache. In these systems, the amount of cache space consumed by each thread depends upon both its memory footprint and its memory accesses patterns. This leads the cache utilization to be determined solely by the individual application's demand. This has been described as "capitalistic" in a previous study [5].

We classify QoS policies into three categories: utilitarian, fair and elitist. The utilitarian policy attempts to improve the overall throughput (the greater good) of the platform by maximizing resources for the cache-friendly application and minimizing resources for the cache-unfriendly application. The fair policy attempts to equalize the cache resources provided to each of the individual applications or provide unequal cache resources to ensure equal performance degradation for each of the applications [11]. The elitist policy considers the relative priority of the applications running simultaneously and ensures that a high priority application is provided more platform resources than the low priority applications. In other words, this policy caters to the elite application(s) at the possible expense of the non-elite classes of applications.

In this paper, we consider elitist policies for resource management and compare them to fair resource management (equal resources) as well as no resource management. This study focuses on performing resource management in the last level cache. We experiment with both static and dynamic mechanisms for achieving elitist QoS.

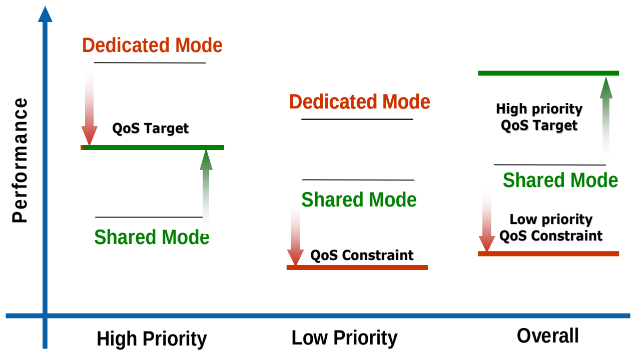


Figure 3. Platform QoS Policies

Figure 3 illustrates the impact of QoS mechanisms on both the applications, and the whole system. Static and dynamic QoS policies differ in the way the target performance and resource constraints are specified. Static policies are defined if the specified target performance does not require continuous adjustment of resources. This policy defines the QoS target and constraint in terms of the resource usage metric (e.g. cache space) provided to the high priority application and low priority applications respectively This requires mechanisms to statically distribute resources, but not dynamically alter them based on resultant performance. For the cache resource, it is intuitive that we consider space as the primary metric.

Dynamic QoS requires resource allocation to be continuously monitored based on the observed performance and the targets/constraints. The targets and resource constraints are specified in terms of process' miss rates and cache space allocation respectively. The dynamic policy we implemented involved varying both the resources provided to different processes and the targets assigned to them based on the current phase of the application. Both applications start out sharing all the available resources without any constraints. The high priority process' initial target is set to be half its observed miss rate and the low process' degradation threshold to be twice its observed miss rate. The adjustment of targets and constraints is performed every 50 million instructions. If the high priority application's miss rate is lower than the target, and the degradation threshold for the low priority application is not exceeded, then the percentage of resources assigned to the low priority application is decreased by a factor of two. If the low priority application's performance degrades beyond its target miss rate, the percentage of resources assigned to it is increased by 10%. The targets for the applications are made more aggressive (decrease the target miss rate by 10%) or conservative (increase the target miss rate by 10%) depending upon whether they are met or not.

3.2. QoS-Aware Platform Architecture

The elitist policy requires that threads or applications be classified into various priorities and architectural support be added to control the hardware resource consumption based on the classification. Figure 4 illustrates the architecture that meets these requirements. As shown in the figure, applications are assigned various priorities by the administrator. Based on the priorities, the QoS resource table (QRT) is updated with cache space limitation data for each priority level. When threads are scheduled to run, their memory requests are tagged with priority levels and cache space limitation information. The cache subsystem is modified to maintain the priority level associated with each line. There are counters that monitor cache space used by each priority level. These counters are used by the replacement policy to evict the appropriate lines within the set. For example, when the low priority resource usage limit is reached, it requires that a low priority victim be chosen from the cache set. Similarly, the high priority process should preferentially replace the low priority process' entries since we our goal is an elitist policy. We accomplished this by modifying the LRU policy to first identify low priority lines and pick the victim among these lines for replacement, in the abovementioned scenarios.

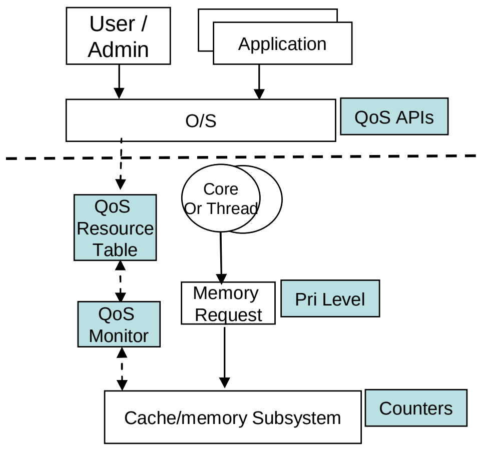


Figure 4. Architecture support for QoS

In order to enable users or administrators to specify the priorities of the application/thread/process, we require QoS support in the execution environment (OS or virtual machine monitor). In section 4, we describe in more detail the architectural support required to provide QoS, and introduce prototypes that mimic this support.

4. QoS Prototypes for Experimentation

In this section, we describe the QoS support required in systems software (OS and VMM) and present prototypes that we developed to emulate this support.

4.1. QoS-Enabled OS

The OS based platform QoS prototype is built on the Linux operating system. In order to provide QoS support, we made several modifications and additions on the 2.6.16 Linux kernel on a Fedora Core 5 host machine. These changes are illustrated in Figure 5.

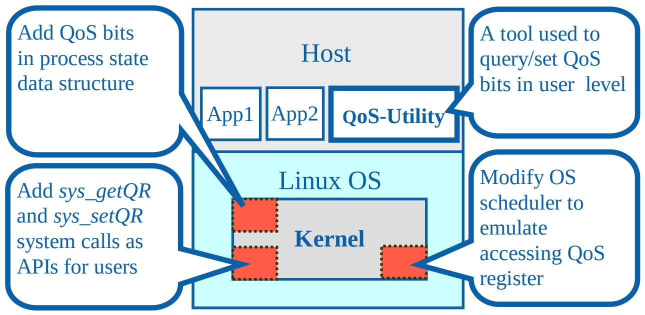

Add QoS bits in process state data structure
Host
App1
App2
QoS-Utility
A tool used to query/set QoS bits in user level
Add sys_getQR and sys_setQR system calls as APIs for users
Linux OS
Kernel
Modify OS scheduler to emulate accessing QoS register

Figure 5. Illustration of QoS-Enabled Linux

The QoS software prototype implementation comprises of these major components:

(a) ***QoS bits in process state:*** QoS bits that indicate the priority level and associated information were added to each process' state. This information is saved and restored during context switches.

(b) ***QoS register emulation:*** The Linux scheduler was modified to emulate saving and restoring the QoS bits from the process state and to commit it to processor architecture state (QoS register). This was achieved by employing a special I/O instruction during every processor context switch. More specifically, we first read the QoS bits value from the process context that was switched in. Then we issued an *out* (x86 ISA) instruction that sent this value to an unused I/O port 0x200(typically, this port was reserved for joystick). This instruction was used to communicate the process' QoS value to the hardware. Port 0x200 was registered as the "QoS" port in the kernel I/O resource registration module to guarantee that it would not be used by other I/O devices.

In addition, to allow administrators to manage QoS values associated with running processes, the Linux kernel was modified to provide:

(a) ***QoS APIs for user/administrator:*** Two extra system calls were added to the Linux kernel to provide interfaces for programs to access the QoS bits which were stored in kernel address space.

(b) ***QoS utility program:*** This tool was implemented in the host Linux machine to query and modify the QoS value of the running applications.

4.2. QoS-Enabled VMM

Our virtualization software prototype is similar to the OS prototype, except the enhancements were made to the hypervisor instead of the operating system. This involved making the following modifications to a popular hypervisor, Xen 3.0.2:

(a) ***QoS at virtual domain level***: QoS bits that indicate the priority level and associated information were added to each domain's (virtual machine's) state. This information is saved and restored during context switches.
(b) ***QoS register emulation***: The Xen scheduler was modified to emulate setting the processor architecture state (QoS register) based on the priority of the domain. We used a setup similar to that described in Section 4.1 to expose this register to the underlying hardware.

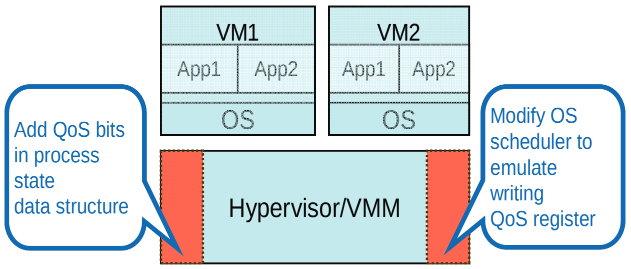

VM1
App1
App2
OS
VM2
App1
App2
OS
Add QoS bits in process state data structure
Hypervisor/VMM
Modify OS scheduler to emulate writing QoS register

Figure 6. Illustration of QoS-Enabled Xen

With this infrastructure, it is possible to make the cache subsystem cognizant of the priority of the virtual machine it is servicing.

4.3. Experimental Framework

In order to evaluate proposed OS/VMM prototypes, we setup the experimental framework shown in Figure 7.

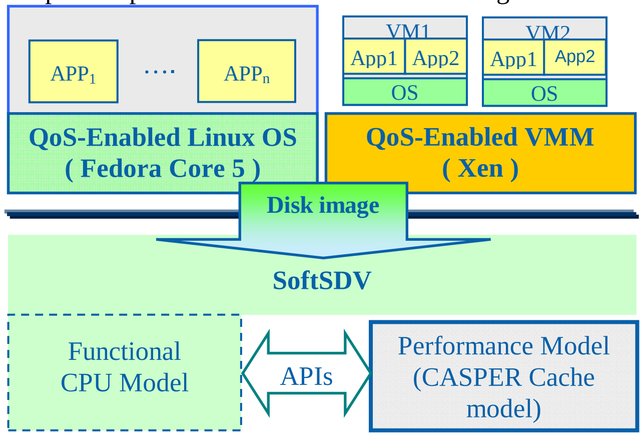


Figure 7. Simulation framework for QoS aware OS/VMM

We employed SoftSDV [18], a full system simulator that allowed us to functionally model the architecture and provided an interface for enabling performance models. We used the functional model of SoftSDV to boot a Fedora Core 5 Linux disk image for a QoS-enabled OS simulation. A disk image of Xen 3.0.2 running virtual machines with Suse 9.1 Linux was used for QoS-Enabled VMM simulation. SoftSDV provided us with APIs which we used to pass the instructions executed by the running applications or VMs from our functional model to the performance model. These instructions included the special *out* instructions mentioned in Section 4.1, which triggered the performance model to simulate the ensuing memory accesses with the specified priority. We integrated CASPER [8] - a functional cache simulator which was modified to support the cache QoS policies described in Section 3.3, into this experimental framework, for evaluating cache performance.

Parameters	**Values**
Core Number	1/2/4
L1 (Private)	Unified, non-inclusive, 32KB, 16 way, 64B Block, LRU
L2 (Shared)	Unified, 512/1024/2048/4096/8192 KB, 16 way, 64B Block

Table 1. Cache Simulation Parameters

Table 1 summarizes the parameters for our experiments. We evaluated the QoS-Enabled OS prototype using a few applications from the SPEC 2000 benchmark suite [16] - Ammp, gcc, art, applu and mcf, which show large cache sharing impact when they are co-scheduled. The standard *ref* input sets were used with these benchmarks. The QoS-Enabled VMM prototype was evaluated using Iperf (a networking benchmark) and SPEC 2000 benchmarks – art, swim, mesa and bzip2. In all experiments, we collected the cache sharing statistics for billion instructions executed by the system.

5. PRELIMINARY RESULTS AND ANALYSIS

In this section, we describe the experimental results for our QoS-Aware Linux and QoS-Aware Xen software prototypes.

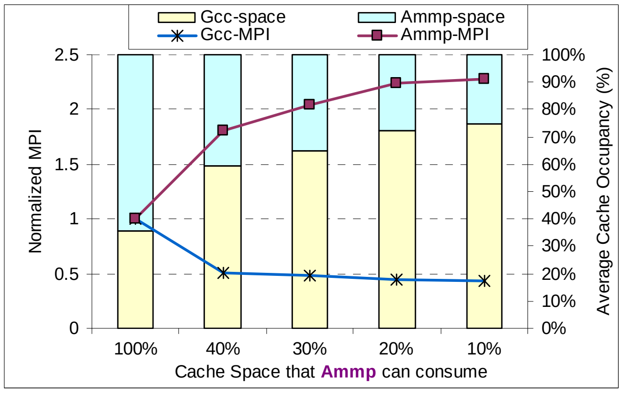
5.1. Multi-Tasking Experiments on QoS-Aware Linux

In the multi-tasking experiments, we assumed the execution platform to be two/four core CMP architecture. Two or four applications run simultaneously and share the L2 cache.

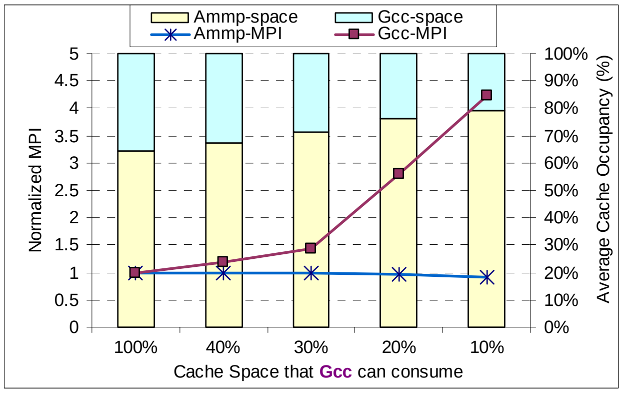
5.1.1 Two-Core CMP Scenario

Our evaluation of the static QoS policy on two-core CMP is done by varying the cache space limit for low priority applications from 10% to 40% of the cache. We compare this to the shared mode execution without QoS, which is denoted by a cache space limit of 100%. We use the resource performance metric to evaluate effectiveness in terms of the L2 cache's MPI (misses per instruction).

Figure 8 shows the impact of QoS policy when Ammp and Gcc are running simultaneously (on separate cores) and share a 512KB cache.



(a) Gcc(hi) + Ammp(lo)



(b) Ammp(hi)+Gcc(lo)

Figure 8. Impact of QoS in two-core CMP scenario

The two Y-axes represent the MPI (lines) and average cache space occupancy (bars) of the co-scheduled applications respectively. The MPI value is normalized to the case when both applications share the L2 cache without any prioritization. Figure 8a illustrates the scenario when Gcc has a higher priority than Ammp. As expected, the MPI of Gcc reduces when we reduce the cache space available for Ammp. This is accompanied by an increase in the the MPI of Ammp. The MPI reduction was seen to be as much as 57% when Ammp was constrained to occupy 10% of total cache size. A noteworthy finding was that Gcc benefited more from cache Qos than Ammp, even though Ammp occupied more cache space than Gcc in the base case with no QoS (around 65%). This conclusion was based on the fact that Ammp's MPI increased by only around 125% when it lost about 40% of total cache space (Figure 8a), while the MPI increase of Gcc was around 320% when it lost about 15% of total cache space (Figure 8b). This explains why reducing the cache space available for Gcc did not cause significant MPI reduction for Ammp (Figure 8b).

Note that although we limit the cache space for low priority application, this limitation is not a hard bound and applications can sometimes exceed the specified limits. This is because of a couple of reasons. One, sharing of data by applications, (shared libraries etc) results in processes sometimes accessing data tagged with the priorities of other processes. In our implementation, cache lines are tagged with the priority of the last application that touches the data. This accounts for the calculated low priority application occupancy potentially being larger than the prescribed space limit. Secondly, cache locality dictates that we need not always have a replacement candidate with the same priority present in every set. Since we try and constrain total cache occupancy and not the occupancy per set, such situations lead to the low priority process potentially exceeding the bounds set for it.

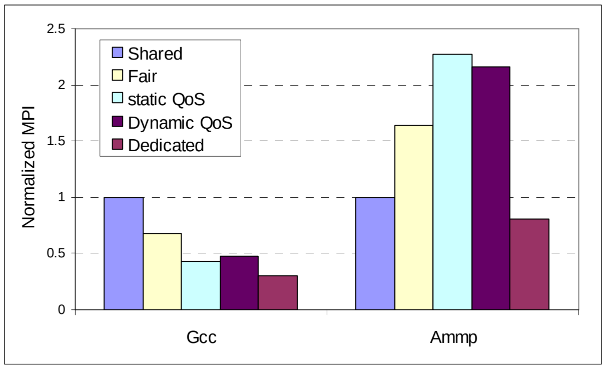


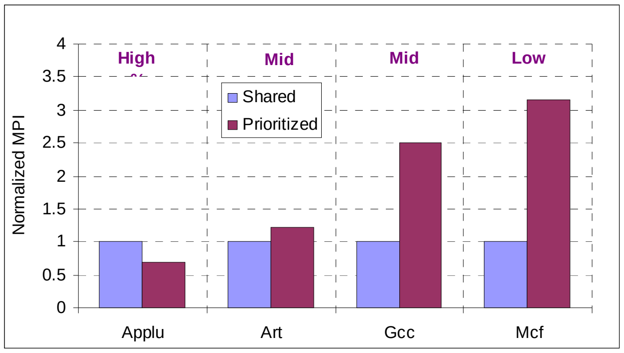
Figure 9. Performance of Gcc and Ammp running simultaneously with different execution modes

Figure 9 illustrates the MPI of Gcc and Ammp under shared mode (without prioritization), fair mode (each application occupy half of the cache), static QoS mode (ammp is low priority and is constrained to occupy 10% of the cache), dynamic QoS mode (ammp is low priority and the amount of cache it occupies is dynamically modified) and dedicated mode (applications occupy the whole cache). The MPI value of each application is normalized to the case when it runs under shared mode. Figure 9 shows that both static and dynamic QoS are more efficient policies to approach the performance improvement bound (dedicated mode) than fair QoS. For applications like Ammp that occupy large percentages of cache space in the shared mode, it is not surprising that fair QoS adversely affects performance.

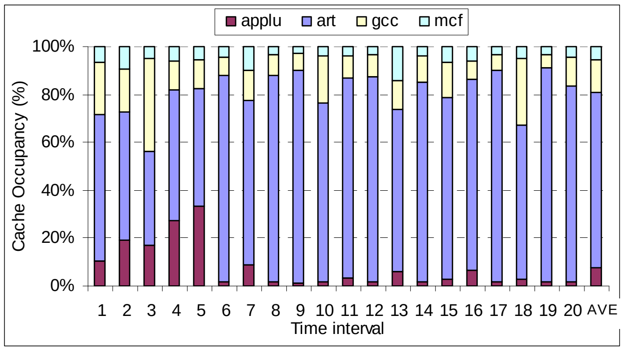
5.1.2 Four-Core CMP Scenario

In the four-core CMP scenario, we assume that one high priority application, two mid level priority applications and one low priority application run simultaneously and share a 1MB cache. Under the prioritized running mode, the middle priority application is limited to occupy 10% of total cache space and the low priority application will bypass the L2 cache (i.e. 0%).

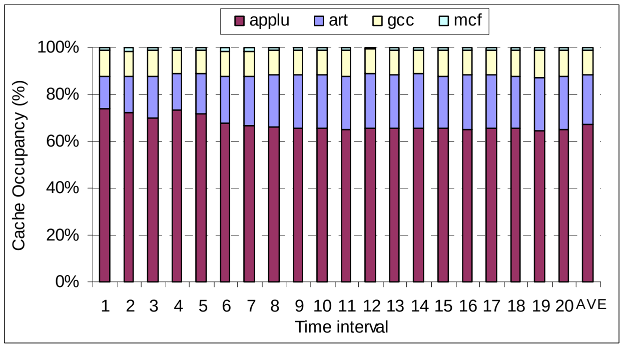
Figure 10 shows the impact of QoS when Applu (high priority), Art (mid level priority), Gcc (mid level priority) and Mcf (low priority) are co-scheduled. In Figure 10a, the MPI value of each application is normalized to the case when it shares the cache with other applications without prioritization. Figures 10b and 10c show the cache occupancies of four applications without and with cache QoS respectively. When we limit the cache space of Art and Gcc and bypass the cache accesses from Mcf, the MPI of Applu reduces by 33%. As a result, the MPI of Art, Gcc and Mcf will increase by 21%, 150% and 216% respectively. Figure 10b and Figure 10c clearly show that employing cache QoS can efficiently assign a deterministic amount of cache space to the high priority application. The cache occupancy of Applu increases from 7% to 67% on average with prioritization. Mcf's occupancy is slightly larger than 0% and Art's occupancy is around 30% of the cache, due to shared Linux libraries as explained in section 5.1.1



(a)Performance



(b) No Cache QoS



(c) With Cache QoS

Figure 10. Impact of QoS in four-core CMP scenario

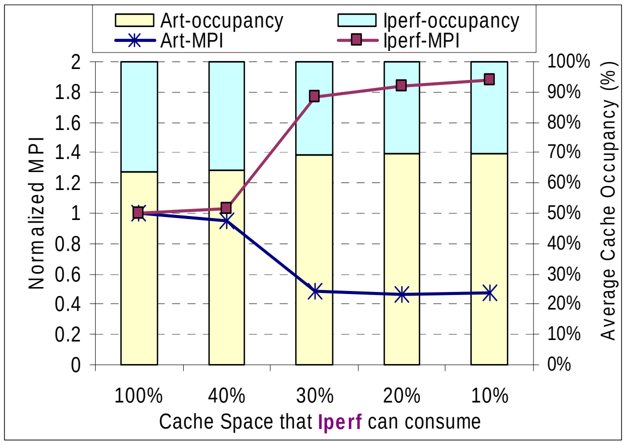
5.2. Virtualization Experiments on QoS-Aware Xen

We ran our experiments on a uni-processor with split Level-1 instruction and data caches (each 32 KB) and a unified Level-2 cache. Only the Level-2 cache was made QoS-aware. Our experiments involved running two benchmarks, in different virtual machines. One of the virtual machines was given a higher priority, allowing it to use the whole cache, while the other had a lower priority and was constrained to use only a portion of the cache. The cache size was varied between 1MB and 8MB to mimic large server workloads. We also varied the amount of cache allocated to lower priority virtual machines between 10% and 40% of the cache. We ran three sets of benchmarks which are described in this section:

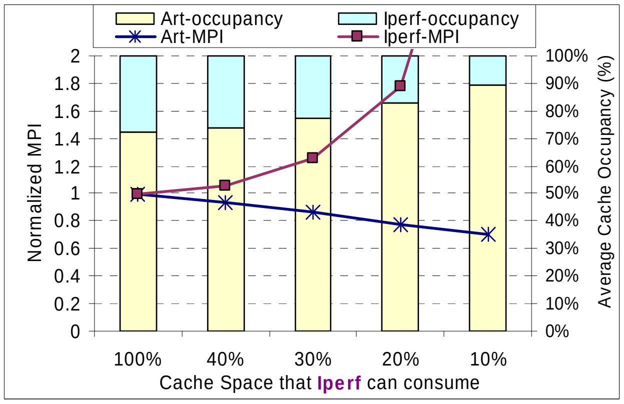
5.2.1 Co-scheduling Compute intensive and I/O intensive workloads

Compute intensive tasks typically have more uniform cache access patterns and are far more cache friendly than I/O intensive applications. Constraining the amount of cache available to the I/O application should help improve the MPI of the compute intensive process, when they are co-scheduled, since I/O applications cause an increase in the number of conflict misses, effectively thrashing the cache.

For the purpose of this study we ran Iperf and Art in a virtualized environment. We ran Iperf in Xen's Domain 0 (along with the I/O VM) and assigned it a low priority. We enabled Xen's native scheduler which allows Domain 0 to use as much as 75% of the CPU. Art was run in a different virtual machine (Domain 1) which was allowed to access the whole cache, and was scheduled for up to 25% of the CPU time. The two benchmarks were co-scheduled for a billion instructions varying both the amount of cache at Iperf's disposal and the total size of the cache.



(a) MPIs of Art and Iperf sharing a 4MB cache



(b) MPIs of Art and Iperf sharing a 2MB cache

Figure 11. Impact of QoS while co-scheduling Art and Iperf

Figure 11 shows the static QoS evaluation results from our QoS prototype experiment with an I/O application (Iperf) and a computation application (Art) sharing a last level cache. These plots also show the percentage of cache occupied by Art and Iperf at different cache sizes. When running Art (high priority) and Iperf (low priority) with a 4MB level-two cache (Figure 11a), we find that placing a limit on the cache space that the lower priority application can occupy significantly decreases the MPI of the higher priority application (reduced to around 40%). This MPI reduction is corroborated by the cache occupancy graph which shows Art's occupancy stabilizing when Iperf is limited to 30% of the 4MB cache. The reduction in Art's MPI is more than the increase in that of Iperf, indicating that resource management in this situation not only has a positive influence over the high priority application, but also helps lower the net MPI of the system (Art and Iperf running together). Another point to note from the graphs is that the occupancies of the low priority applications aren't strictly bound by the upper limits we set. As explained in section 5.1.1, this is due to sharing of VMM entries (which inherit the priorities of the processes they are invoked from) and maintaining global cache occupancy counters.

When the cache size is less than 2MB (Figure 11b), we see that the percentage occupancy of Art increases monotonically to use most of the cache. This indicates that the cache is too small to fit Art's working set completely. Constraining the amount of cache accessible by Iperf has a positive impact on Art's MPI at the cost of Iperf cache performance. The overall MPI of the system increases significantly when Iperf is constrained to run with less than 10% of the 2MB cache, indicating that resource management of small caches could adversely affect the system's performance if the lower priority application is not allocated a minimum amount of cache space.

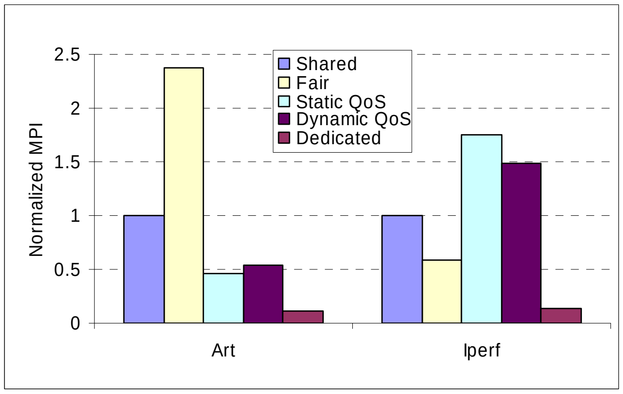


Figure 12. Performance of Art and Iperf running simultaneously with a 4MB cache with different execution modes

Figure 12 illustrates the MPI of Art and Iperf running in shared mode (without prioritization), fair mode (each application occupies half of the cache), static QoS mode (Iperf is assigned a low priority and occupies 10% of the cache), dynamic QoS mode (Iperf is assigned a low priority and the amount of cache it occupies is dynamically modified) and dedicated mode (both applications have dedicated 4MB caches). The MPI value of each application is normalized to the case when it runs in shared mode. Figure 12 shows that both static and dynamic QoS are more efficient policies to approach the maximum performance improvement point (dedicated mode) than fair QoS. In addition, dynamic QoS can be tuned to provide better performance for the platform in general and not just one application.

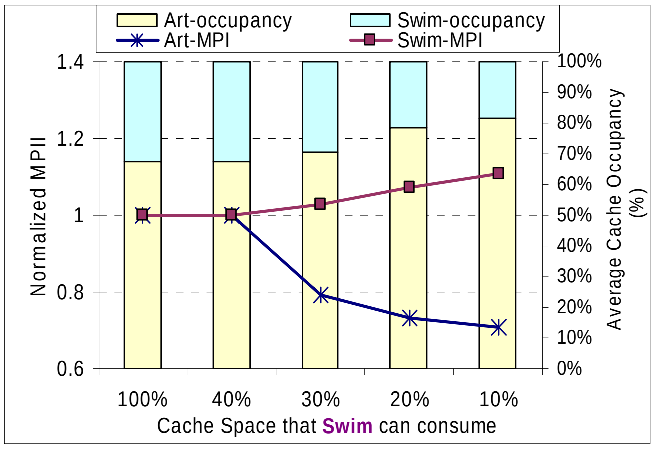
5.2.2 Co-scheduling two compute intensive workloads

In this study, we ran Swim and Art, two SPEC2000 workloads that have very different cache scaling patterns. Swim's MPI (misses per instruction) remains fairly constant as the cache size changes while Art's MPI falls rapidly as the size of the cache increases. We constrained the amount of cache the domain running Swim could use, while allowing the domain running Art access to the whole of the cache. We modified Xen to schedule both domains for equal time slices (while using the default period of the SEDF scheduler).

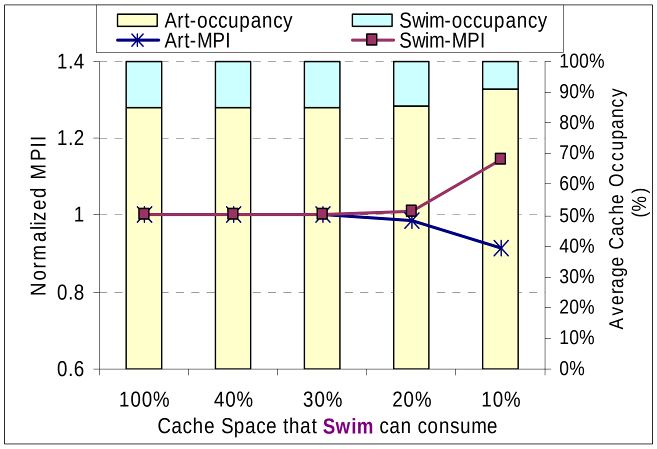
Figure 13 shows the static QoS evaluation results. These plots also show the percentage of cache occupied by Art and Swim at different cache configurations. While running Art (high priority) and Swim (low priority) with an 8MB level-two cache (Figure 13a), we find that placing a limit on the cache space that the lower priority application can occupy significantly decreases the MPI of the higher priority application (reduced to less than 70%). Cache management in this situation not only has a positive influence over the high priority application, but it also helps lower the net MPI of the system (Art and Swim running together).

At 4MB (Figure 13b), we see that Art occupies most of the cache, even in the shared case. This high and almost constant cache occupancy graph explains the lack of a radical reduction in Art's MPI at this cache configuration when Swim's cache space is changed. For cache sizes smaller than 4MB, the performance boost for Art is small due to the fact that the cache size is too small to accommodate the working sets of both Art and Swim.

These studies indicate that enabling cache-priorities at the granularity of virtual machines definitely decreases the amount of contention in the cache and allows the higher priority applications to be assured a much better quality of service. They also suggest that even overall system performance could receive a boost at certain hardware configurations, which are dependent upon the workloads in question.



(a) MPIs of Art and Swim sharing an 8MB cache



(b) MPIs of Art and Swim sharing a 4MB cache

Figure 13. Impact of QoS while co-scheduling Art and Swim

Figure 14 illustrates the MPI of Art and Swim running in shared mode (without prioritization), fair mode (each application occupies half of the cache), static QoS mode (Swim is assigned a low priority and occupies 10% of the cache), dynamic QoS mode (Swim is assigned a low priority and the amount of cache it occupies is dynamically modified) and dedicated mode (both applications have dedicated 8MB caches). The MPI value of each application is normalized to the case when it runs in shared mode. The figure shows that both static and dynamic QoS are more efficient policies to approach the maximum performance improvement point (dedicated mode) than fair QoS. This figure also illustrates the uniform behavior of Swim across the different modes, due to which static and dynamic QoS actually provide a performance boost to not only Art but the whole system.

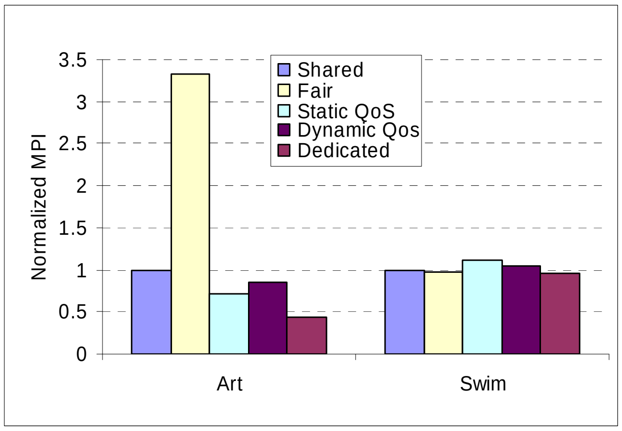


Figure 14. Performance of Art and Swim running simultaneously with a 8MB cache with different execution modes

5.2.3 Four-VM CMP Scenario

In this study, we scaled the number of virtual machines to four, with each of them running a different application, to gauge the effectiveness of our scheme in larger CMP environments. In the four-VM CMP scenario, we assume one high priority application, two mid-level priority applications and one low priority application are running simultaneously and share a 4MB cache. Under the prioritized running mode, the mid-level priority applications are limited to occupy 20% of the total cache space and the low priority application is limited to occupy 10% of the total cache space. All these applications were run with equal scheduling priority.

Figure 15 shows the impact of QoS when Art (high priority), Mesa (middle priority), Bzip2 (middle priority) and Iperf (low priority) are co-scheduled. Each of these applications run in different virtual machines which share the physical resources available to one core. In the Figure, the MPI value of each application is normalized to the case when it shares the cache with other applications without prioritization. As is evident from the figure, limiting the amount of space available to the other applications results in a 45% reduction in miss rate for Art. Mesa also surprisingly benefits from QoS which is perhaps because of the decreased interference from Iperf. The performance degradation for the other mid-level priority process - Bzip2, is minimal. Iperf's performance sees a degradation which is consistent with its getting allotted just 10% of the cache.

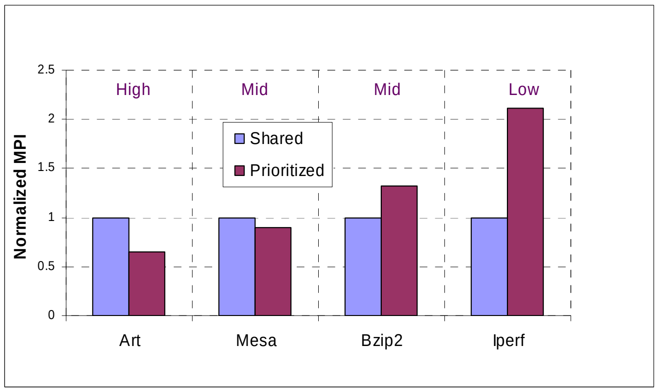


Figure 15. Impact of QoS in four-VM CMP scenario

6. SUMMARY AND FUTURE WORK

In this paper, we presented the motivation behind QoS-aware platforms and execution environments by showing case studies of CMP cache resource usage. We showed that it is important to provide better determinism in the platform especially in the context of multi-tasking and virtualization. By enabling QoS mechanisms in the hardware as well as exposing it to the systems software, we can address this requirement. We described two software prototypes (QoS-aware Linux and QoS-aware Xen) that allow experimentation with multi-tasking and virtualization usage scenarios. The preliminary case studies with these usage scenarios showed that QoS-enabled caches work better than unmanaged caches and fair caches. We also show that enabling QoS provides better determinism to the high priority application by bringing its performance closer to that of when the application is running by itself.

In future, we hope to experiment more with dynamic QoS mechanisms that improve the performance of not just one application, but the whole platform. We also plan to experiment with QoS in other CMP resources (memory, interconnect and I/O). We plan to investigate the impact of QoS for realistic applications and virtual machined deployed on future architectures. Last but not least, we found that the lack of benchmarks for virtualization and multi-tasking scenarios is a significant problem for experimenting with such usage models. This is an open problem area that is to be addressed as these usage models become widespread.

REFERENCES

[1] Azul Compute Appliance," Azul Systems, can be found at http://www.azulsystems.com/products/cpools_cappliance.html

[2] P. Barham, et al, "Xen and the Art of Virtualization", In the Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), October 2003

[3] D. Chandra, F. Guo, S. Kim and Y. Solihin, "Predicting inter-thread cache contention on a chip multiprocessor architecture", In Proc. 11th International Symposium on High Performance Computer Architecture (HPCA), Feb 2005

[4] T. Deshane, D. Dimatos, et al., "Performance Isolation of a Misbehaving Virtual Machine with Xen, VMWare and Solaris Containers," , submitted to USENIX 2006; also at http://people.clarkson.edu/~jnm/publications/isolationOfMisbehavingVMs.pdf

[5] L. Hsu, S. Reinhardt, R. Iyer and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006

[6] R. P. Goldberg, "Survey of virtual machine research," IEEE Computer, 34—45, 1974.

[7] Intel Corporation. "Intel Dual-Core Processors -- The First in the Multi-core Revolution," http://www.intel.com/technology/computing/dual-core/

[8] R. Iyer, "CASPER: Cache Architecture, Simulation and Performance Exploration using Re-streams," Intel's Design and Test Technology Conference (DTTC), 2001.

[9] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," 18th Annual International Conference on Supercomputing (ICS'04), July 2004.

[10] C. Kim, D. Burger, S. W. Keckler, "Nonuniform Cache Architectures for Wire-Delay Dominated On-Chip Caches," IEEE Micro 23(6): 99-107 (2003)

[11] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", in Proc. Of the 13th International Conference on Parallel Architectures and Complication Technique (PACT), Sep-Oct 2004

[12] K. Krewell, "Best Servers of 2004: Where Multicore is Norm," Microprocessor Report, www.mpronline.com, Jan 2005.

[13] K. Olukotun, B. A. Nayfeh , et. al., "The case for a single-chip multiprocessor," Proceedings of the 7th International Conference on Architectural support for Programming Languages and Operating Systems, October 01-04, 1996.

[14] N Rafique, et al, "Architectural Support for Operating System-Driven CMP Cache Management", International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006

[15] M. Rosenblum and T. Garfinkel : Virtual Machine Monitors : Current Technology and Future Trends. IEEE Computer 38(5) : 39-47(2005)

[16] "SPEC", http://www.spec.org/cpu2000/

[17] R. Uhlig, et al., "Intel Virtualization Technology," IEEE Computer, 2005.

[18] R. Uhlig, R. Fishtein, et. al., "SoftSDV: A Presilicon Software Development Environment for the IA-64 Architecture", Intel Technology Journal, Q4, 1999. (http://www.intel.com/technology/itj)

Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS

Masaaki Kondo, Hiroshi Sasaki, Hiroshi Nakamura
Research Center for Advanced Science and Technology,
The University of Tokyo
4-6-1 Komaba, Meguro-ku, Tokyo, Japan
{kondo,sasaki,nakamura}@hal.rcast.u-tokyo.ac.jp

Abstract

Recently, a single chip multiprocessor (CMP) is becoming an attractive architecture for improving throughput of program execution. In CMPs, multiple processor cores share several hardware resources such as cache memory and memory bus. Therefore, the resource contention significantly degrades performance of each thread and also loses fairness between threads.

In this paper, we propose a Dynamic Frequency and Voltage Scaling (DVFS) algorithm for improving total instruction throughput, fairness, and energy efficiency of CMPs. The proposed technique periodically observes the utilization ratio of shared resources and controls the frequency and the voltage of each processor core individually to balance the ratio between threads. We evaluate our technique and the evaluation results show that fairness between threads are greatly improved by the technique. Moreover, the total instruction throughput increases in many cases while reducing energy consumption.

Keywords: Chip Multiprocessor, Fairness, Resouce Contention, DVFS.

1 Introduction

A single chip multiprocessor (CMP) is an attractive architecture for future high performance processors because of its power and performance efficiency. Designing a single complex core with higher clock frequency is getting difficult due to power and thermal problems. On the contrary, a CMP with relatively simple processor cores does not suffer from these problems, and thus has a potential to achieve higher performance. Recent research has found that CMPs offer performance benefit and energy efficiency [16, 9]. Therefore, CMPs are expected to be used for wide variety of platforms even in mobile or embedded platforms. Reducing energy consumption of CMP architecture is still emerging requirement.

In CMPs, multiple cores usually share a memory system which includes caches, memory buses, and memory banks. More memory requests should be handled by the memory system. Hence, the load of the memory system gets heavier than ordinary uniprocessors. However, since performance gap between processors and main memory is still widening, memory system limits performance more in CMPs. Things get worse for memory system contention. When multiple cores generate memory requests simultaneously, some of memory accesses are delayed, and consequently significant performance degradation is expected for threads whose memory accesses are delayed. Therefore, in CMPs, performance of a thread is affected by the other threads which are co-scheduled with it.

There have been several studies to avoid resource contention for efficient execution in CMP architectures. For example, Suh et al. [20] have proposed a method which avoids the contention on a shared L2 cache and increases execution throughput. Their methods avoid inter-thread cache contention by partitioning the shared L2 cache. Although the prior work can solve the contention on caches to some extent, the contention on memory buses or memory banks has been ignored. Since bus contention significantly impacts the performance in CMPs, it should be taken into account when using CMPs.

This paper proposes a technique to mitigate memory bus and memory bank contention by controlling execution speed of each thread running on each core. Dynamic Voltage and Frequency Scaling (DVFS) is used to control the progress of the execution. The technique chooses the clock frequency and the supply voltage individually for each processor core to balance the impact of the resource contention between threads. By relaxing the contention, total execution throughput of the chip increases. Since the supply voltage is reduced when throttling a thread, energy consumption is also saved.

Our DVFS algorithm is based on the concept of *fairness* which indicates how uniformly the performance of each thread is affected by the contention. As discussed in [8], fairness is a critical issue especially for an operating system (OS) when a processor executes multiple threads. Because the task scheduler of OS assumes that fairness is satisfied when the OS assigns time-slice to threads with taking their priority into account, the effectiveness of the OS task scheduler depends on the hardware to provide fairness to all the co-scheduled threads. Therefore, improving fairness is a very important issue for CMPs. Our first goal is to optimize fairness by DVFS. Moreover, it is reported that fairness improvement leads to higher throughput in most of the applications [8]. Thus, the proposed algorithm has another advantage of improving throughput.

This paper is organized as follows. The next section describes the related work. We address the impact on memory system contention in Section 3. In Section 4, we describe the proposed technique. Section 5 describes the evaluation environment and assumptions and the evaluation results are presented in Section 6. We conclude in Section 7.

2 Related Work

Techniques for improving throughput and fairness for CMPs which share an L2 cache have been studied so far. Suh et al. [20] and Kim et al. [8] have proposed partitioning a shared L2 cache to minimize the number of cache misses or maximizing fairness. Chandra et al. [3] have proposed performance models that predict the impact of cache sharing on co-scheduled thread. The performance degradation caused by resource contention is more significant in simultaneous multi-Threading (SMT) architectures because many hardware resources are shared in SMTs. Therefore, many researchers have focused on improving fairness and throughput for SMTs [19, 11, 13].

Prior work in CMPs, however, has ignored the effect of memory bus and/or bank sharing. Although increase in cache miss rate due to the contention degrades performance, bus contention also significantly impacts the performance in CMPs. Thus, when optimizing throughput and fairness, effect of bus contention should be taken into account.

There are many research effort on power/energy reduction by *Dynamic Voltage / Frequency Scaling (DVFS)* technique which controls the supply voltage and the clock frequency during execution of a task according to computation requirement of the task. Because dynamic power consumption in a CMOS circuit scales quadratically with the supply voltage, a significant power reduction is expected by DVFS technique.

Recently, *Globally Asynchronous Locally Synchronous (GALS)* system [4], in which the chip is broken into several independent synchronous blocks and each block operates with its own local clock, is becoming attractive. Because GALS design can eliminate the need for distributing global clock signal across an entire chip, power consumption and complexity for clock distribution network is suppressed.

Applying DVFS to GALS system is a promising way to save power consumption. Because GALS systems allow voltage and clock frequency of local clock domains to be independently controlled, we can flexibly control the voltage and the clock frequency for each local domain. The previous work has explored the power reduction for GALS microprocessors with local domain DVFS [7, 12, 18, 17]. It divides a microprocessor chip into individual clock domains, front-end, integer unit, floating-point unit, and load/store unit. There is the possibility of further reduction of power consumption by this fine-grain optimization.

In CMPs, DVFS can be applied to each core individually by supplying independent clock signal and voltage line to each processor core. In the industry, real System on Chip (SoC) LSI, where the voltage and the clock frequency of some modules of the SoC is controlled independently, is already developed [5, 14]. We believe that CMP with GALS design style will soonly become a realistic design choice in general purpose microprocessors.

Liu et al. [10] have proposed a technique for optimizing power consumption of CMP. Their technique controls the clock frequency and voltage of each core individually. The techniques exploits idle times by a processor waiting for other processor because of the barrier synchronization, and clock frequency and voltage is lowered so that the idle times is eliminated.

3 Impact of Resource Sharing in CMP

3.1 Impact of Memory Bus Contention

In most of CMPs, memory buses and main memory banks are shared by multiple processor cores to maximize resource utilization. The performance of a thread is affected by other threads running on different cores due to memory bus and memory bank contention. Figure 1 presents the normalized performance (Instruction per Cycle) of *swim* benchmark program from SPEC2000 when it is executed on a dual-core CMP in which the memory bus are shared by the two cores but the L2 cache is not shared. In Figure 1, *swim-alone* indicates one of the core executes swim and the other core does not execute any program and swim+*program_name* represents one core executes swim and the other core executes *program_name* in parallel. The evaluation environment is described in Section 5.

As seen from the figure, compared with swim-alone, the performance of swim degrades when the other thread is executed simultaneously on the other core. Since both cores

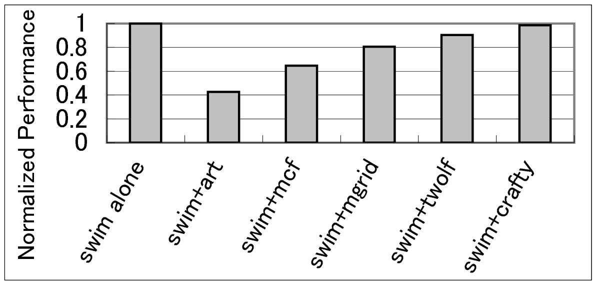


Figure 1. Performance degradation by bus contention

have their own L2 cache in this evaluation, inter-thread cache contention does not occur. Therefore, this performance degradation is caused by the contention of the memory system. It should be noted that the performance degradation depends on the co-scheduled thread. For example, in the case of *swim+crafty*, where *crafty* is co-scheduled with swim, the performance of swim is almost the same as that of swim-alone. Because the number of L2 cache misses in crafty is very small, almost no resource contention occurs. On the other hand, significant performance degradation is observed in *swim+art*. Since frequent L2 cache misses occurs in art, the memory bus is usually occupied by main memory accesses from the core executing art. The main memory access requests from swim are delayed for a long time. This is the reason why the swim is greatly slowed down when art is co-scheduled.

3.2 Definition of Throughput and Fairness

3.2.1 Total Throughput

In CMP architectures, the total chip performance, that is, the total instruction throughput for threads running on all the cores, is important metric as well as the performance of each thread. In this paper, we use Instruction Per Second (IPS) as a metric of total chip performance[1].

Let $Inst_i$ be the number of executed instructions for $thread_i$ which is executed on PU_i within *t*[second]. The performance of $thread_i$ is defined as $IPS_i = \frac{Inst_i}{t}$. For a CMP which has N cores, the total throughput IPS_{total} is defined as follows:

$$IPS_{total} = \sum_{i=0}^{N-1} IPS_i \qquad (1)$$

[1]Because the clock frequency changes by DVFS, we use IPS instead of instruction per cycle (IPC).

3.2.2 Fairness

Fairness is the metric which indicates how uniformly the performance of each co-scheduled thread is affected by the resource sharing. We follow the definition of *execution time fairness* used in [8]. Let $Tded_i$ and $Tshr_i$ be the execution time of thread i when it is executed alone and when it is executed with other threads, respectively. We assume the number of co-scheduled threads is n. Here, the ideal fairness is achieved if the following condition is satisfied:

$$\frac{Tshr_1}{Tded_1} = \frac{Tshr_2}{Tded_2} = ... = \frac{Tshr_n}{Tded_n} \qquad (2)$$

In this paper, we define the *total throughput fairness*. Let $IPSded_i$ and $IPSshr_i$ be the IPS of thread i when it is executed alone and when it is executed with other co-scheduled threads, respectively. Assuming the number of co-scheduled threads is n, the ideal fairness is achieved if the following condition is satisfied:

$$\frac{IPSshr_1}{IPSded_1} = \frac{IPSshr_2}{IPSded_2} = ... = \frac{IPSshr_n}{IPSded_n} \qquad (3)$$

As used in [8], we quantify fairness using the following equation:

$$Fair_{ij} = |X_i - X_j|, where X_i = \frac{IPSshr_i}{IPSded_i} \qquad (4)$$

The smaller the $Fair_{ij}$, the better the fairness.

Fairness is a critical aspect for an Operating System (OS) [8]. The task scheduler of the OS assumes that fairness is satisfied when the OS assigns time-slice to threads with taking their priority into account. If fairness is not satisfied, several problems such as thread starvation and priority inversion arise. Therefore, the effectiveness of the OS task scheduler depends on the hardware to provide fairness to all the co-scheduled threads. Hence, improving fairness is an important issue for CMPs.

4 Improving Fairness and Throughput by DVFS

In this section, we propose a technique which optimizes fairness and throughput and reduces energy consumption by controlling the frequency and the voltage of each core individually. Figure 2 shows our assumed CMP organization. Because Kim et al. have found that improving fairness leads to throughput improvement [8], the technique tries to improve the fairness.

4.1 overview

The bus contention occurs when main memory requests due to L2 cache misses are issued from different processor

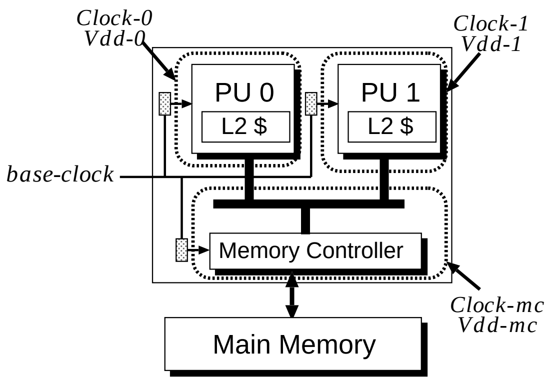


Figure 2. GALS-CMP

cores closer in time. Because memory accesses are delayed in this case, main memory access latency for a thread is lengthened. Consequently, significant performance degradation is expected.

If a thread invokes L2 cache misses very frequently, the performance of other threads degrades significantly due to the bus contention. In this situation, fairness tends to be lost. In order to mitigate the impact of the contention and improve fairness, the shared resource access ratio from all the co-scheduled threads should be balanced. For this purpose, we lower the clock frequency of the core which executes the thread with frequent memory bus accesses.

Our algorithm consists of the following two steps: (1) predicting the impact on performance due to bus contention, (2) adjusting the clock frequency and supply voltage to redress the performance balance between co-scheduled threads. As for (1), the hardware needs to monitor the bus behavior because the performance impact is affected by this behavior which depends on dynamic characteristics of the co-scheduled threads. We use an interval-based approach for the monitoring and the adjustment.

Figure 3 summarize our proposed algorithm.

4.2 Predicting the Performance Impact

We introduce a counter named $Creq_i$, which records the number of existing main memory read requests every cycle This $Creq_i$ is provided for each core (PU) and $Creq_i$ indicates the counter for PU$_i$. The memory controller increments $Creq_i$ if an L2 cache miss requests for PU$_i$ is invoked. When one of these requests is completed, the counter is decremented.

In addition to $Creq_i$, a state register named $Rreq$ is introduced. $Rreq$ holds the identifier of the PU for which currently the memory bus services the memory access request. In every bus clock cycle, $Creq_i$ and $Rreq$ are monitored. When $Creq_i$ is greater than 1 and i is not equal to $Rreq$, PU$_i$'s memory requests are delayed due to the contention.

```
$N$ = number of PUs;
$Creq_i$ = number of pending read request;
$Rreq$ = PU number currently transfered on bus;
for each bus-cycle {
  $j = Rreq$;
  $for\ (k = 0;\ k < N;\ k$++){
    $if\ (k \neq j$ && $Creq_k > 0)$
      $Cdelay_{jk}$++;
  }
  /* for every $T_{itvl}$ */
  $if\ ((BusCycleCount\ \%\ T_{itvl}) == 0)${
    $for\ (j = 0;\ j < N;\ j$++){
      $diff_j = \sum_{k=0}^{N-1}(Cdelay_{jk} - Cdelay_{kj})$
      $if\ (diff_j < 0$ && $Clock_{lev_j} != MaxClockLev)$
        $SetMaxClockLev(ClockLev_j)$;
      $elseif\ (diff_j > Th_u$ && $ClockLev_j > MinClockLev)$
        $DownClockLev(ClockLev_j)$;
      $elseif\ (diff_j < Th_l$ && $ClockLev_j < MaxClockLev)$
        $UpClockLev(ClockLev_j)$;
      $else$
        /* $UnchangingClockLevel$ */;
      $for\ (k = 0;\ k < N;\ k$++)
        $Cdelay_{jk} = 0$;
    }
  }
  $BusCycleCount$++;
}
```

Figure 3. Proposed DVFS algorithm

To keep track of this delay or postponed cycles, a set of counters named $Cdelay_{jk}$ are introduced. $Cdelay_{jk}$ indicates the number of cycles that the requests from PU$_j$ delay those from PU$_k$.

By subtracting $Cdelay_{kj}$ from $Cdelay_{jk}$, we can obtain how many cycles the thread on PU$_j$ keeps the thread on PU$_k$ waiting. We calculate the summation of the waiting cycles for each PUs ($diff_j$ in Figure 3).

We predict the performance impact on PU$_j$ by watching $diff_j$. If $diff_j$ is relatively large, the thread on PU$_j$ is likely to impede the execution of the other threads running on the other PUs due to the contention. On the other hand, if $diff_j$ is a minus value, execution of the thread on PU$_j$ might be degraded due to the contention.

4.3 Adjusting the Clock Frequency and Supply Voltage

In order to improve fairness, the clock frequency and the supply voltage of each PU is controlled so that the $diff_j$ for every PU becomes the same. Let T_{itvl} be the time interval for changing the supply voltage and clock frequency. The supply voltage and clock frequency for the next T_{itvl} period

Table 1. Configuration of each processor core

Fetch/Issue/Commit	4 instruction / cycle
Branch prediction	Combined bimodal (4K-entry) gshare (4K-entry) selector (4K-entry)
BTB	1024 sets, 4way
Branch Mis-penalty	7 cycles
RUU size	64
LSQ size	32
Functional units	Int: 6 ALU, 2-mult/div FP: 6 ALU 4 mult/div Load/Store: 2 ports
L1 I-Cache	32KB, 32B line, 2way 1 cycle latency
L1 D-Cache	32KB, 32B line, 4way 2 cycle latency
L2 Cache (for each PU)	1MB, 128B line, 8way 10 cycle latency in 1.6GHz
DRAM # of banks	4
DRAM access latency	row: 3, column: 3, precharge: 10
Bus (channel)	# of channels: 1, 8B-width each
Bus clock	200MHz

are selected based on the current *diff*$_j$ values. If *diff*$_j$ is relatively large, the clock frequency and voltage for PU$_j$ must be lowered. If *diff*$_i$ is relatively small, the clock frequency and voltage for PU$_j$ are raised. Moreover, if *diff*$_j$ is a minus value, the processor sets the maximum clock frequency and voltage level for PU$_j$.

The values of *diff* are calculated for all the cores at every ending point of T_{itvl}. We introduce two threshold values Th_u and Th_l for *diff*, which indicate the upper threshold and lower threshold, respectively. Thrashing problem is avoided by introducing two kinds of thresholds. If *diff*$_j$ exceeds Th_u, the clock frequency of PU$_j$ is lowered by one level, whereas if *diff*$_j$ is below Th_l, the clock frequency of PU$_j$ is raised by one level. The frequency is not changed if the value is between Th_u and Th_l. Note that $Cdelay_{jk}$ is reset every time interval.

5 Experimental Setup

We evaluated fairness, performance and energy consumption of our proposed technique compared with a conventional non-DVFS CMP. We used the SimpleScalar Tool Set [1] as our base simulation environment. We extended SimpleScalar to evaluate the CMP shown in Figure 2. For estimating the energy consumption, we used the Wattch [2] extension. Table 1 shows the assumptions of the processor configuration for the evaluation.

Since we are focusing on contentions in memory accesses, the memory access model is very important for the evaluation. Therefore, a DDR SDRAM model was incorporated in the simulator in order to evaluate the very accurate access timing of memory bus and memory banks. We assume the number of DRAM banks is four, and thereby the memory system can handle up to four memory access requests at a time. However, data for one of the requests can be transferred on a memory channel at a time. We also incorporated a basic memory access scheduling technique proposed in [15] into the simulator.

Table 3. The programs used in evaluation

L2 Miss	Program (L2 Miss-rate, IPC)
High	179.art (20.0%, 0.17) 181.mcf (13.8%, 0.12)
Middle	171.swim (3.17%, 0.57) 300.twolf (0.95%, 0.96)
Low	172.mgrid (0.69%, 2.4) 176.gcc (0.11%, 1.27)

The assumption of the supply voltage and the clock frequency used in the evaluation is shown in Table 2. These values are based on the Intel Pentium M processor. Because there is a lowest limit on supply voltage, we assume that the same voltage is used for clock frequency of 0.4GHz and below. According to the data-sheet of the Intel Pentium M processor[6], the core is unavailable for up to $10\mu s$ during frequency ramping. We pessimistically assume $20\mu s$ of penalty for a frequency and voltage transition.

We used several programs from the SPEC CPU2000 benchmark suite with the *ref* input set. We fast-forwarded two billion instructions and simulated until one of co-scheduled threads executes 100 million instructions. The evaluated programs are presented in Table 3. The table also shows the L2 cache misses per load/store instructions and IPC for evaluated program regions. Since the proposed method does not make any sense if bus contention rarely occurs, the programs with very low L2 cache miss-rate (less than 0.1%) are not considered. Among the SPEC CPU2000 benchmarks, seventeen programs have relatively high L2 miss-rate. We selected the nine programs out of the seventeen so as to include programs with wide variety of miss-rate and IPC. The programs were compiled by the DEC C compiler for Alpha AXP instruction set architecture (ISA).

The following values are used for the parameters of the our algorithm described in Section 4.

- T_{itvl}: 250000 bus-cycle (= 625 [μs])
- Th_u: 15000, Th_l: 10000

Table 2. Combinations of clock frequency and supply voltage

Clock [GHz]	1.6	1.2	0.8	0.4	0.2	0.1	0.05
Vdd [V]	1.484	1.276	1.036	0.956	0.956	0.956	0.956

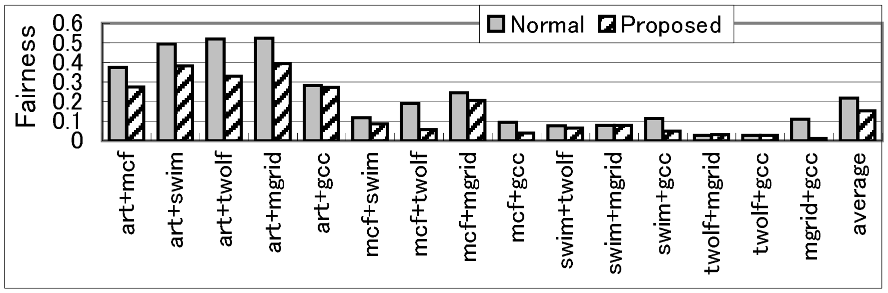


Figure 4. Fairness (2-core)

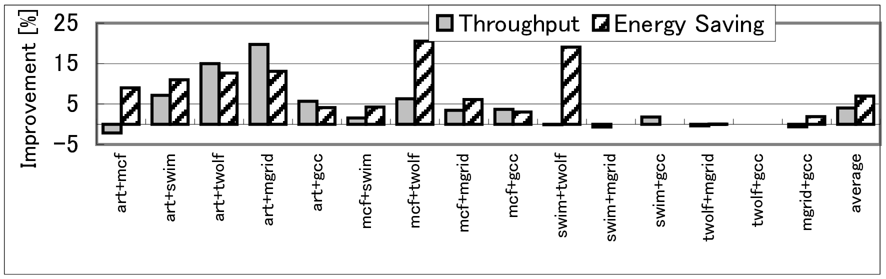


Figure 5. Throughput improvement and energy saving (2-core)

6 Evaluation Result

6.1 Results

6.1.1 Fairness

Figure 4 and Figure 6 show fairness value ($Fair_{ij}$) for a conventional non-DVFS CMP (Normal) and our proposed method (Proposed) for 2-core and 4-core CMPs, respectively. We show the results of all the combinations of selected benchmark programs. The fairness for 4-core CMP is defined as the maximum $Fair_{ij}$ in all the combinations of cores.

As seen from Figure 4 and Figure 6, the proposed method improves fairness for almost all the evaluated cases (The lower the fairness, the better it is). The improvement is significant for the pairs which include programs with relatively high cache miss rate. If L2 cache miss rate of one of the thread is high, the memory bus is frequently accessed by the thread. This leads frequent bus contention, and consequently fairness is likely to be lost in the normal CMP. Because we try to balance the impact of the resource contention between threads by controlling frequency, fairness improves with proposed method especially for programs with higher miss rate. By averaging all the programs, The proposed CMP improves fairness by 30% and 5% in 2-core and 4-core CMPs, respectively.

Compared with results for 2-core CMP, improvement in fairness is not significant for 4-core CMP. In the evaluated program pairs, the cache miss rate of one particular thread is by far higher than those of other threads, and thus one thread impedes the memory accesses from other threads. In this case, our method lowers the frequency of just one thread. Although the fairness between that thread and the other threads is improved, the fairness among remaining threads is still bad. In some cases in Figure 6, fairness gets slightly worse by the proposed method. In these program pairs, over-adjustment happens when the method controls the clock frequency in order to balance the bus contention.

6.1.2 Throughput and Energy Saving

Figure 5 and Figure 7 present total throughput (IPS_{total}) improvement of the proposed method over the normal processor for 2-core and 4-core CMPs, respectively. These figures also show the saving of energy per committed instruc-

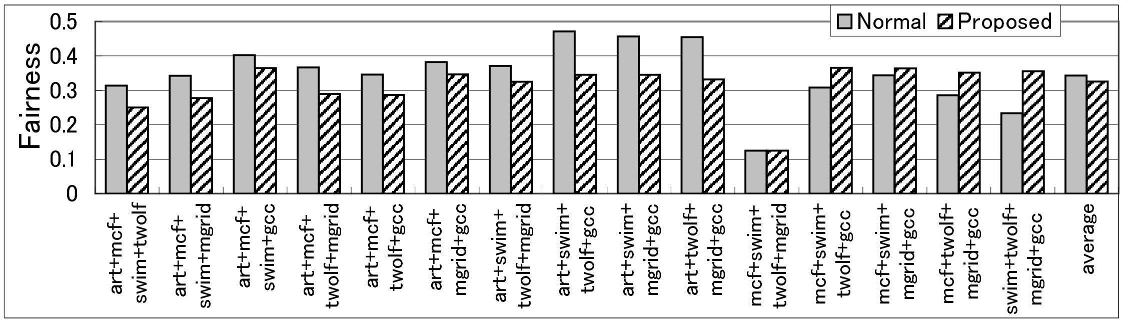


Figure 6. Fairness (4-core)

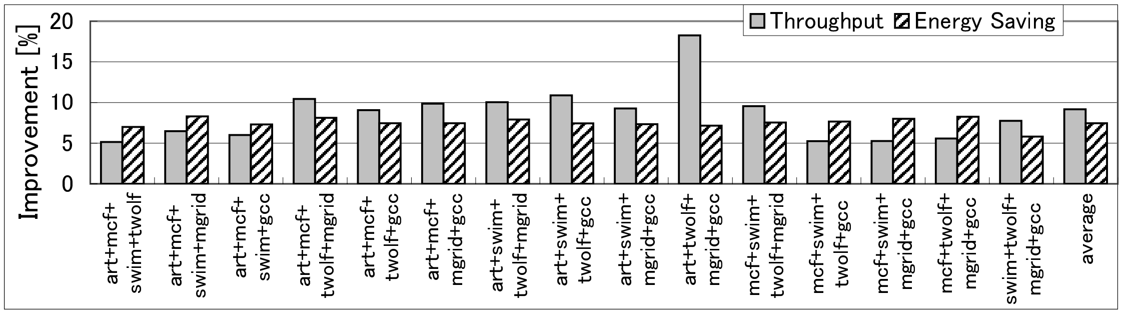


Figure 7. Throughput improvement and energy saving (4-core)

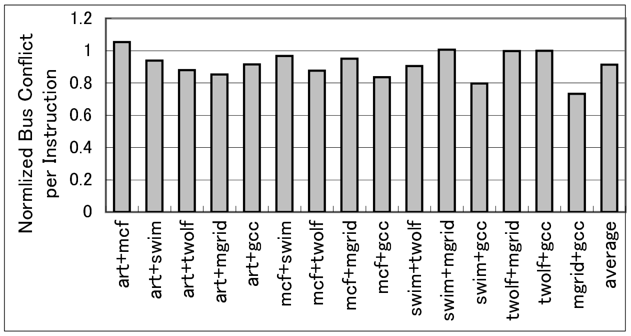


Figure 8. Relative bus conflict cycles per instructions for proposed method (2-core)

tion (EPI).

As seen from the figures, total throughput improves almost all the program pairs. This results confirms that improving fairness also improves throughput in most of the cases. Figure 8 presets relative bus conflict cycles per committed instructions for the proposed method normalized by the normal CMP. From the figure, bus contention is actually reduced by the method. This reduction of bus contention contribute to the improvement of the performance. The improvement is significant when *art* is executed on one of the cores except for the case of `art+mcf`. Because the memory buses are accesses frequently in art, the other threads co-scheduled with it are greatly slowed down due to the contention. The proposed method lowers the clock frequency of the core where art is executed, and thereby, the execution of the co-scheduled threads becomes smooth. That is the reason why the proposed method improves total throughput.

The noticeable throughput decrease is observed in `art+mcf` though the fairness is improved. This indicates that the improving fairness does not always increase total throughput. This is the consistent observation with earlier study [8].

Note that the performance overhead for clock frequency and voltage transitions is negligible. Because time interval T_{itvl} used in the evaluation is relatively long compared with the time overhead ($20 \mu s$), the overall impact on performance due to clock frequency and voltage transitions is mitigated.

In addition to increasing throughput, our proposed method also saves energy consumption by lowering supply voltage. Energy per committed instruction (EPI) in proposed CMP is reduced compared with the normal CMP. Since the opportunity to lower the supply voltage increases in the program pairs with high cache miss rate, a large amount of energy can be saved in these pairs.

With our method, both throughput and energy efficiency

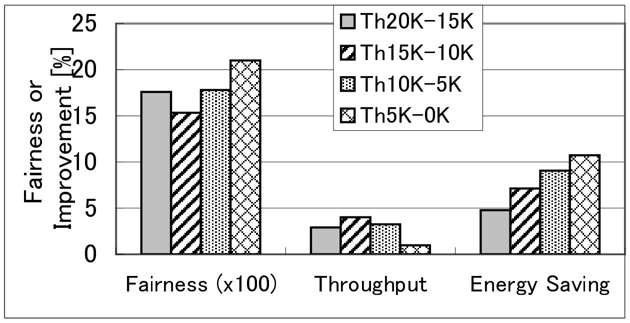


Figure 9. Evaluation result varying the threshold values

are improved in many cases. This is a great advantage of our proposed method. Therefore, we argue that the proposed technique is quite effective for CMP architectures.

6.2 Sensitivity Study

Our proposed DVFS method changes the clock frequency and supply voltage of each core according to how many cycles a thread impedes the memory accesses from other threads. We used the two threshold values, Th_u and Th_l, for frequency and voltage control decision. It seems that the results heavily depend on these threshold values. Therefore, in this section, we evaluate our proposed method varying the threshold values.

Figure 9 present fairness, throughput improvement and energy saving for our proposed method varying the threshold values in the case of 2-core CMP. The results in the figure shows the average values across all the evaluated combinations. Four kinds of upper and lower threshold pair are evaluated. The gap between upper and lower threshold is fixed to 5000. The legend in the figure indicates upper threshold and lower threshold (for example, Th15K-10K indicates the case that Th_u is 15000 and Th_l is 10000).

By lowering threshold values, our algorithm tries to lower the clock frequency and supply voltage aggressively. Therefore, energy consumption is further reduced with lowering threshold values. However, fairness and throughput become worse if the threshold is too large or too small. When threshold is too large, the method cannot mitigate the performance impact caused by the contention because frequency of the cores tends to be unchanged. On the other hand, when threshold is too large, over-adjustment occurs and fairness can be lost. Therefore, threshold values should be chosen carefully.

7 Concluding Remarks

In this paper, we proposed a novel technique to improve fairness, total instruction throughput, and energy efficiency for CMP architectures through dynamic voltage and frequency scaling (DVFS). The proposed technique controls the clock frequency and the voltage of each processor core individually to balance the performance impact by memory bus contention.

We evaluated fairness, total throughput, and energy reduction of proposed technique compared with an original non-DVFS CMP. The evaluation results revealed that the proposed technique improves fairness in almost all the program pairs, and total throughput increases in many cases. On average, our technique improves fairness by 30% in a 2-core CMP and 5% in a 4-core CMP. It increases the throughput by 4.0% and 9.2% and saves energy consumption by 7.0% and 7.5% in 2-core and 4-core CMPs respectively. These results indicate that controlling the clock speed of each core to balance the utilization ratio of memory bus between threads is very helpful for efficient execution in CMP architectures. Since our proposed method improves all of these metrics which are very important for CMPs, the proposed technique is quite effective for CMP architectures.

We are planing to develop more energy efficient algorithm and evaluate our technique using wider variety of programs.

Acknowledgment

We would like to thank anonymous reviewers for providing useful comments on this paper. This work was supported in part by STARC (Semiconductor Technology Academic Research Center).

References

[1] T. M. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.
[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th ISCA*, pages 83–94, June 2000.
[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th HPCA*, pages 340–351, Feb. 2005.
[4] D. M. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. PhD thesis, Stanford Univeristy, 1984.
[5] T. Fujiyoshi and et al. Intel pentium m processor datasheet. In *2005 ISSCC*, pages 132–133, Feb. 2005.
[6] Intel. *Intel Pentium M Processor Datasheet.*, June 2003.
[7] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *29th ISCA*, pages 158–168, May 2002.

[8] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *13th PACT*, pages 111–122, Oct. 2004.

[9] Y. Li, D. Brooks, Z. Hu, and K. Skadron. Performance, energy, and thermal considerations for smt and cmp architectures. In *11th HPCA*, pages 71–82, Feb. 2005.

[10] C. Liu, A. Sivasubramaniam, M. T. Kandemir, and M. J. Irwin. Exploiting barriers to optimize power consumption of cmps. In *2005 IPDPS*, Apr. 2005.

[11] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS2001*, pages 164–171, Nov. 2001.

[12] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *30th ISCA*, pages 14–25, June 2003.

[13] T. Moseley, D. Grunwald, J. L. Kihm, and D. A. Connors. Methods for modeling resource contention on simultaneous multithreading processors. In *23rd ICCD*, pages 373–380, Oct. 2005.

[14] K. Nose, A. Shibayama, H. Kodama, M. Mizuno, M. Edahiro, and N. Nishi. Deterministic inter-core synchronization with periodically all-in-phase clocking for low-power multi-core socs. In *2005 ISSCC*, pages 296–297, Feb. 2005.

[15] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory access scheduling. In *27th ISCA*, pages 128–138, June 2000.

[16] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of cmp vs. smt for multimedia workloads. In *18th ICS*, pages 196–206, June. 2004.

[17] G. Semeraro, D. H. Albonesi, S. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *35th MICRO*, pages 356–367, Dec. 2002.

[18] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *8th HPCA*, pages 29–42, Feb. 2002.

[19] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS IX*, pages 234–244, Nov. 2000.

[20] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *8th HPCA*, pages 117–128, Feb. 2002.

Starvation-Free Commit Arbitration Policies for Transactional Memory Systems*

M.M. Waliullah and Per Stenstrom

Department of Computer Science and Engineering,
Chalmers University of Technology, SE-412 96 Goteborg, Sweden

ABSTRACT *In transactional memory systems like TCC, unordered transactions are committed on a first-come, first-serve basis. If a transaction has read data that has been modified by the next transaction to commit, it will have to roll-back and a lot of computations can potentially be wasted. Even worse, such simple commit arbitration policies are prone to starvation; in fact, the performance of Raytrace in SPLASH-2 suffered significantly because of this.*

This paper analyzes in detail the design issues for commit arbitration policies and proposes novel policies that reduce the amount of wasted computation due to roll-back and, most importantly, avoid starvation. We analyze in detail how to incorporate them in a TCC-like transactional memory protocol. We find that our proposed schemes have no impact on the common-case performance. In addition, they add modest complexity to the baseline protocol.

1. INTRODUCTION

As multi-core architectures are becoming commonplace, the need to make parallel programming easier is becoming acute. Transactional memory (TM) [1,2,3,4,7] promises to reduce the programming effort by relieving the programmer from resolving complex, fine-grain, inter-thread dependences by classical synchronization primitives such as locks and event synchronizations. Instead, coarse program segments form transactions that will either execute atomically or not at all. If transactions run by different threads have no dependencies, they can run concurrently. On the other hand, if a data dependency or a conflict appears, one of the transactions is squashed and must re-execute. Therefore, transactional memory trades programming simplicity for wasted execution at run-time.

There are two approaches by which conflicts are detected: eager and lazy [4]. In systems with lazy conflict detections, such as TCC [2], the modifications done by a transaction are isolated until the point when the transaction commits. When the transaction commits, such transactions that have speculatively read data modified by the committing transaction will be squashed. Squashing does not only waste useful work, it can lead to starvation as the following example shows.

*This work is sponsored by the SARC project by funds from the EU FET programme. The authors are members of EU IST FP6 HiPEAC.

Suppose that a parallel program consists of two threads, where each of them executes the same transaction *N* times. Transaction *Tx1* executed by the first thread modifies variable A and transaction *Tx2* executed by the second thread reads A. If *Tx1* is sufficiently shorter than *Tx2* it will commit before *Tx2* which may then be repeatedly squashed until all *N* instances of *Tx1* have committed. Obviously, all transactions can be serialized because of starvation. We have experimentally observed that similar behavior showed up in `raytrace` in SPLASH-2. The reason is that the commit arbitration policy simply allows unordered transactions to commit on a first-come, first-serve basis. In the original paper on TCC [2], this problem is acknowledged. However, the recommendation is to insert “pseudo-barriers” – an idea that is neither elaborated on nor analyzed in detail.

This paper makes several contributions. Firstly, it analyzes in detail how to implement feasible commit arbitration schemes for TCC-like TM protocols. Secondly, and most importantly, it contributes with two novel starvation-free commit arbitration policies. Our overall approach to detect and remedy a potential starvation problem is to track how many times a certain transaction has been squashed. At the time a thread is ready to commit, it will not be allowed to do so if there is an ongoing transaction that has been squashed more times than the committing one. Then, the committing thread is stalled until that transaction has committed. Apart from avoiding devastating starvation situations, we show experimentally, using five applications from SPLASH-2, that our starvation-free policies have virtually no impact on common-case performance and that they can be implemented with modest modifications to a TCC-like TM protocol.

As for the rest of the paper, we first introduce the architectural framework and frame the problem in more detail in Section 2. Section 3 is devoted to the novel arbitration schemes and especially how they are incorporated in the architectural framework. We then move on to the experimental results in Sections 4-5 by first describing the methodology. Section 6 puts our work in context of the TM literature and we conclude in Section 7.

2. SYSTEM FRAMEWORK

In this section, we define the framework of our study including the software assumptions in Section 2.1 and the

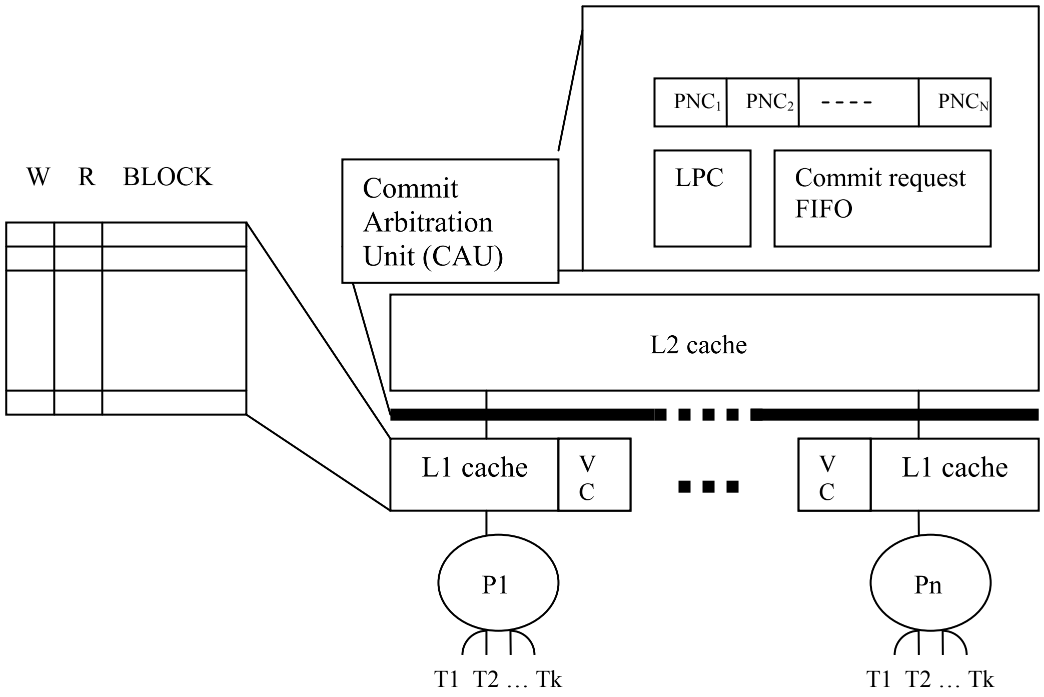


Figure 1: Baseline architectural framework.

architectural framework in Section 2.2. Finally, we frame the problem addressed by this research in detail in Section 2.3.

2.1 Programming Model

Under a transactional memory programming model, transactions are used as a construct with the following properties. It is either executed entirely as an atomic entity, or, if it has a data dependency conflict with another *committed* transaction, it is squashed and re-executed, and its partial effect on the system state is eliminated in this process.

We limit ourselves to parallel applications that adhere fully to this model. As a result, transactions are explicitly delimited in the code and when one transaction ends, another one starts. Assuming parallel applications using critical sections and barriers, transactions are formed so that the following simple rules are followed: 1) critical sections are guaranteed to be encapsulated within a transaction and 2) a transaction is terminated and a new transaction starts at a barrier. However, a transaction can be terminated and a new one can start between two barriers as long as it happens outside critical sections [6,8,9,10].

It is assumed that a *phase number* is associated with each transaction which is incremented when a barrier is passed. All transactions that are started after the dynamic invocation of a certain barrier get the same phase number. Two transactions with the *same* phase number are said to be unordered with respect to each other whereas transactions with *different* phase numbers are ordered and must commit in the ascending order of their phase numbers.

Barrier semantics suggest that no thread may execute any code beyond a barrier before all threads have arrived at the barrier. Under the transactional memory programming model, threads can execute beyond a barrier as long as they do not conflict with a thread that has not reached the barrier. This is supported using the notion of *ordered* as well as *unordered* transactions.

Let's next consider the system model that supports this software model.

2.2 Architectural Framework

We consider a multi-core system that consists of *n* nodes where each node consists of a processor core (or core for simplicity) with its private L1 cache connected to a shared L2 cache via a bus or other broadcast medium according to Figure 1. Each core can be optionally (simultaneously) multithreaded with *k* hardware threads, where *k* is typically a small number (four or less). This system framework supports transactional memory similarly to TCC.

To support TCC, each L1 cache is extended with meta data to keep track of which blocks have been speculatively read and written using a read (R) and a write (W) bit, respectively, by setting the corresponding bits. If a block that has been speculatively written is replaced, it is placed in a victim cache (denoted as VC in Figure 1) attached to each L1 cache. If the victim cache has to replace a speculatively modified block, the L1 as well as the victim caches will block until the node can gain exclusive access to the bus. Once the exclusive access to the bus is granted, the transaction will proceed, while holding the bus, until it is finished. This will prevent other nodes from violating the atomicity requirement of transactions. When a transaction is finished, it will try to commit by requesting the bus.

In the baseline system, multiple commit requests are arbitrated through a central arbitration unit (denoted as CAU in Figure 1). To adhere to the semantics of ordered transactions, it attempts to select a committing transaction among the ones with the lowest phase number. Among these unordered transactions, it selects a candidate using FIFO.

To implement the baseline arbitration policy, the CAU uses three components: phase number counters (PNC), a lowest phase counter (LPC), and a FIFO with all commit-requests. The PNCs keep track of the current phase number of each thread using $N = n \times k$ phase-number counters, given *n* nodes and *k* hardware threads per node.

There are two types of commit requests: ordered and unordered. When a thread passes a barrier, an ordered commit request is sent. When an ordered commit request is granted by the CAU, the corresponding PNC is incremented. The LPC keeps track of the lowest phase number of any thread, i.e., $\min(PNC_i)$, i=1,…,*N*. Finally, the FIFO simply keeps all pending commit requests on a first-come, first serve basis.

Given these components, the CAU uses the LPC to filter out the requests in the FIFO that can commit, i.e., the transactions having the lowest phase numbers. It then picks the first one of these in the FIFO queue.

A node with a granted commit request broadcasts its write set (the set of blocks having a W-bit set) to all L1 caches. All nodes with a block belonging to the write set and with its R-bit set will be notified to squash their ongoing transactions. Squashing a transaction involves the following steps: 1) invalidate all blocks having either the R or W-bit set; 2) gang-clear all R and W bits; and, 3) restart the transaction by reinstalling the architectural state for the starting point of the transaction.

2.3 A Starvation Scenario

A major limitation of any transactional memory system is the performance lost due to squashes. More seriously, the simple arbitration policy assumed in the original TCC proposal is actually prone to starvation as the following example clearly demonstrates.

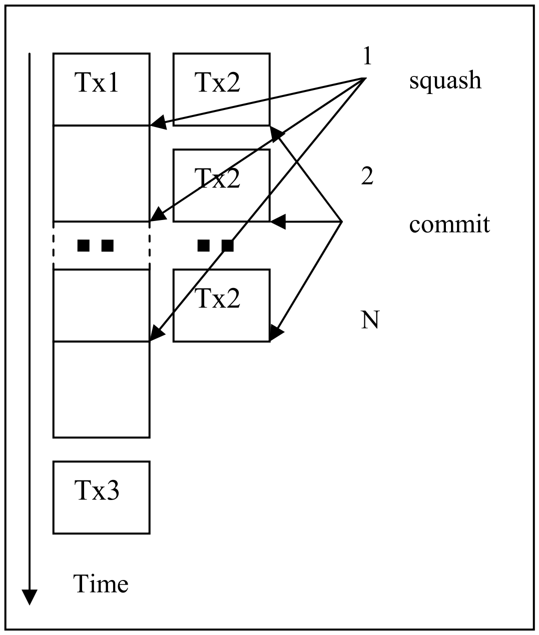
Let's consider two threads (T1 and T2) that execute the code in Figure 2A. T1 executes a transaction (Tx1) that reads variable X followed by another transaction that does not conflict with any (Tx3). On the other hand, T2 executes a transaction (Tx2) that modifies X N times. Further, the execution time of Tx1 is assumed to be longer than that of Tx2.

Now consider the execution scenario in Figure 2B in which the execution of Tx1, Tx2, and Tx3 is tracked along the vertical time axis and where the commit and squash points are marked. As Tx1 and Tx2 obviously conflict, Tx2 will successfully commit whereas Tx1 will be squashed. As T2 will invoke Tx2 again, while T1 attempts to re-execute Tx1, the same scenario may repeat N times. As a result, the execution of all transactions will be serialized.

```
T1{            T2
Tx1_begin{     for(i=1;i++;i<N){
…=X;           Tx2_begin{
   …
                  X=…;
…              }
…
               }

}
Tx3_begin{
 Unrelated();
}
}
```

A)



B)

Figure 2: A starvation scenario.

In another scenario, assuming the same code, Tx1 could have successfully committed before the first invocation of Tx2. Then, the execution of Tx3 and possibly subsequent transactions could overlap with the execution of the N-1 invocations of Tx2. Clearly, although the commit arbitration mechanism assumed in TCC meets fairness objectives at the level of commit requests between ordered and unordered transactions, it may result in starvation at the software level.

We have observed that a nearly as bad scenario showed up for `raytrace` in SPLASH-2 (to be discussed in Section 5). The key reason for the devastating scenario of Figure 2B is that the CAU is completely unaware of the history by which transactions get squashed. In the next section, we propose novel arbitration policies that address this shortcoming.

3. STARVATION-FREE POLICIES

To avoid starvation, the overall approach is to give priority to ongoing transactions that already have suffered from being squashed. We consider two schemes that differ in the way transactions are given priority to be selected and in implementation complexity. They are referred to as the *naïve* and the *elaborate* schemes.

3.1 The Naïve Scheme

Recall that the baseline scheme selects a transaction to commit among the threads with the lowest phase number on a first-come-first-serve basis. In the **naïve** scheme, we also select a thread to commit in this set. However, we put an additional constraint on which threads to commit based on how many times their ongoing transactions have been squashed. A thread can only commit if it has been squashed more or the same number of times as any other thread with the lowest phase number. Unlike in the baseline scheme, the set of threads that can commit can be the empty set, i.e. all threads attempting to commit so far can be stalled.

To keep track of the number of squashes per thread, the CAU is extended with *N* miss-speculation counters (MSC), assuming *N* threads, where each MSC is initially cleared. When a thread suffers from a squash, the corresponding MSC is incremented. Conversely, when a thread commits its transaction, the corresponding MSC is cleared. In Figure 3, we show the extra mechanisms needed by the **naïve** scheme.

The CAU obviously knows when a thread can commit its ongoing transaction but lacks knowledge about when a transaction is squashed. Hence, it needs to know when to increment the MSCs. This is accomplished by requiring that each time a node selects to squash a transaction, it notifies the CAU so that the corresponding MSC can be incremented. Hence, the commit protocol involving broadcasting the write set and matching it against the read set in each node, must be extended with a final phase to notify the CAU about the hardware threads on each node that were squashed.

One solution is to let all nodes serially report to the CAU about squashed transactions. This would obviously cause a lot of overhead so we propose the following solution. All hardware threads have a SQUASH signal that is raised when a transaction has been squashed. Further, the bus is extended with *N* dedicated lines; one for each SQUASH-signal. These lines are used as increment-enable signals to the set of MSCs maintained by the CAU as shown in Figure 3.

In summary, in terms of structures, the **naïve** scheme is extended with as many miss-speculation counters as the number of hardware threads. Additionally, it needs a special-purpose bus with as many lines as the number of hardware threads. Given the fairly limited number of nodes and hardware threads per core in a multi-core chip, this solution appears to be reasonable.

3.2 The Elaborate Scheme

In the **naïve** scheme, a committing transaction is stalled when any transaction with the lowest phase number and a higher MSC exists in the system disregarding the fact that these transactions could be data independent of each other. Intuitively, it would be unwise to stall a transaction that has no conflict with a transaction that has a higher MSC because, then, it is not the cause of a potential starvation scenario.

In the **elaborate** scheme, a committing transaction that has the lowest phase number and has no conflict with any transaction with the same phase number and a higher MSC is allowed to

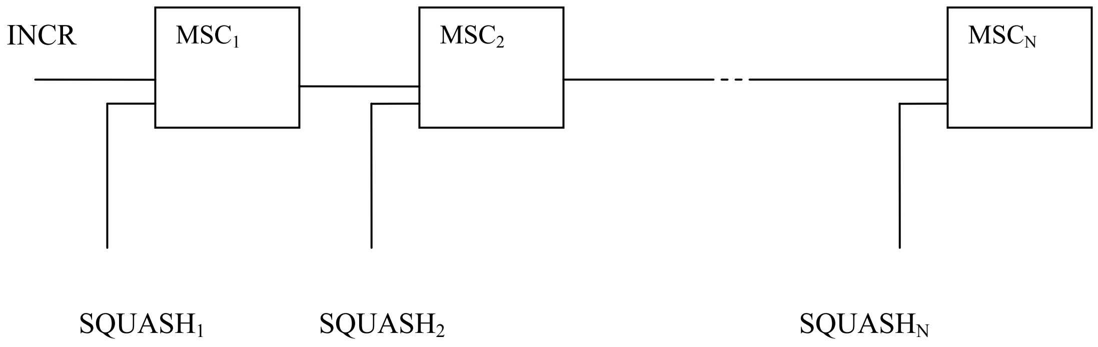


Figure 3: Mechanisms used by the naïve scheme not part of the baseline.

commit. Conversely, the CAU will stall a transaction either if the committing transaction has not the lowest phase number (as in the baseline) or it has the lowest phase number and it conflicts with another transaction that has the same phase number and a higher MSC.

Unfortunately, the CAU has no knowledge about what transactions would be squashed if it grants a commit request. Therefore, we need to extend the **naïve** scheme by breaking up a commit operation into two phases: 1) *check-for-data-races* 2) *make-commit-decision* where the first phase is not part of the **naïve** and the baseline scheme.

To check for data races, the committing node first broadcasts the write set of its transaction. Each node checks locally whether its ongoing transaction conflicts with the committing transaction. If this is the case, it activates the SQUASH signal introduced in the **naïve** scheme (see Figure 3). The CAU will check the MSCs of the transactions with their SQUASH signals activated and if any of them is higher than the MSC of the committing node and they have the same (lowest) phase number, the committing node is not allowed to commit. On the other hand, if the committing transaction will be allowed to commit, the CAU will notify the nodes that have their SQUASH signals activated to squash their ongoing transactions.

In summary, in comparison with the **naïve** scheme, the **elaborate** scheme must extend the commit protocol with a phase that checks whether the transaction that requests to commit can do so without squashing a thread that has the same phase number but a higher MSC value. Note that even if a committing transaction will stall, a bus transaction that checks for conflicts is still needed. Thus, the **elaborate** scheme causes more traffic than the **naïve** scheme.

4. EXPERIMENTAL METHODOLOGY

While the new arbitration policies will eliminate starvation, they can hurt overall performance by delaying the execution of transactions. This is particularly true for the **naïve** scheme that may delay also transactions that do not cause starvation. In addition, while the **elaborate** scheme more accurately selects transactions that can be safely committed, it may still delay unrelated transactions unnecessarily.

In order to analyze whether the new arbitration schemes result in fewer useless cycles in terms of delays or miss-speculations, we have built a simulation model of the baseline system and augmented it with the **naïve** and the **elaborate** scheme.

We model a 16-core system with a single hardware thread per core. Each node has a 128-Kbyte, 4-way private cache with a block size of 16 bytes. This will mimic the effects of a two-level private cache where the L2 cache is 128 Kbytes and the L1 cache is smaller and maintains inclusion with the L2 cache. Further, we maintain an infinite buffering space for speculatively modified blocks by assuming an infinite victim cache. As we are only concerned with the number of cycles lost due to stalling a committing thread and a thread that miss-speculates, we have opted for a simple model of the memory system and do not charge any cycles for cache misses at any level of the memory hierarchy.

The system model is driven by traces generated by the five SPLASH-2 applications [11] in Table 1. We use a trace-driven methodology to drive the system model. Each of the five benchmarks is first run on Simics [5] with Sun Sparc as the target machine. Benchmarks are run with the original synchronization primitives and we collect statistics only in the parallel phase of each benchmark.

Table 1: Benchmarks and their data sets.

Applications	**Inputs**
Barnes	2048 particles
Ocean	34x34 grid
Volrend	Head
Raytrace	teapot.env
Water-Nsquared	512 molecules

Transactions are marked by using the Simics magic instructions in-lined in the implementation of locks and barriers in the ANL-macros. There is a magic instruction at the beginning and at the end of these macros. All memory references between these magic instructions are filtered out of the trace.

A new transaction starts at a barrier and will terminate at the next barrier or after 1000 instructions which of them happens first unless that point occurs inside a critical section. If so, the transaction is continued until the corresponding unlock construct is executed. For nested locks, we flatten them out with the outermost lock, i.e., no transaction can start or terminate within the boundary of this outermost critical section.

It is well-known that a trace-driven approach that we use will not reflect the correct interleaving of events. However, as our goal is not to assess absolute performance but rather to compare the number of useless cycles using the different arbitration policies, we feel that the methodology adopted is adequate for this purpose.

5. EXPERIMENTAL RESULTS

In Section 5.1, we experimentally confirm that the starvation scenario in Section 2.3 can happen. Then, we analyze the tradeoffs between the **naïve** scheme, and the **elaborate** scheme in detail in Section 5.2.

5.1 The Case for Starvation of the Baseline Protocol

Figure 4 shows the execution time of each of the 16 threads normalized to the slowest running thread when running the `raytrace` application on three systems. For each thread, the three bars correspond to the baseline system, the **naïve** scheme, and the **elaborate** scheme, from left to right. Further, each bar is decomposed into three sections that break down the execution time into useful cycles, useless cycles, and idle cycles, from bottom to top, where the latter corresponds to the number of cycles lost due to stalls associated with commit arbitration

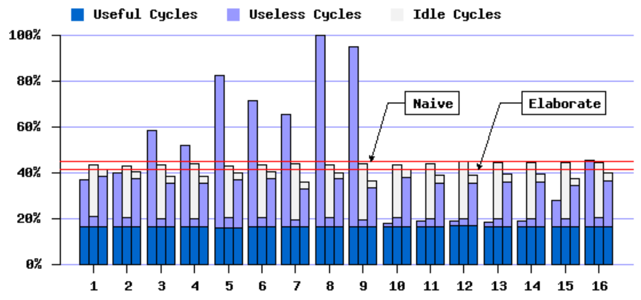


Figure 4: Execution time of individual threads for `raytrace` normalized to the thread that takes the longest time to finish. The first, second, and third bars in each cluster correspond to the baseline, the naïve, and the elaborate arbitration scheme, respectively.

As can be seen in Figure 4, there is a huge difference in execution time between different threads under the baseline arbitration scheme. For example, the baseline system execution time for thread 8 is five times longer than that of thread 10! However, the number of useful cycles across the threads is about the same so the effect is not attributed to load imbalance. The difference stems from the number of useless cycles. Trace inspection has revealed that `raytrace` suffers from a similar starvation situation as described in Section 2.3. Hence, the program becomes serialized which seriously damages the performance of the parallel program.

Considering the execution times across threads for the **naïve** and the **elaborate** schemes (the two rightmost bars in each cluster), we can see that the difference between the slowest and the fastest threads is small. This suggests that these schemes successfully eliminate starvation. In fact, the execution time, which is dictated by the slowest thread, is cut down by as much as between 55% and 59% using the **naïve** and the **elaborate** schemes respectively.

Continuing with the difference between the **naïve** and the **elaborate** schemes, it is clear from Figure 4 that the **naïve** scheme suffers a lot from stalling. However, it is interesting to note that the **elaborate** scheme also suffers from performance overheads due to useless cycles which are attributed to miss-speculations. A close inspection of Figure 4 reveals that the **naïve** scheme tends to reduce the number of cycles lost for miss-speculations at the expense of cycles lost for stalling threads. Overall, however, the **elaborate** scheme performs slightly better than the **naïve** scheme – the execution time is around 4% shorter.

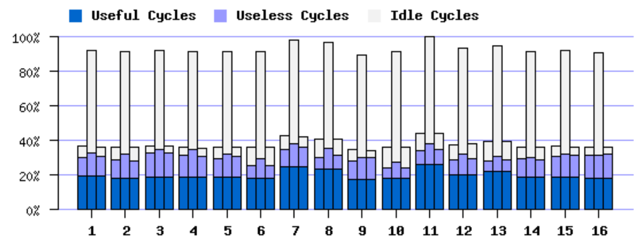


Figure 5: Execution time of individual threads for `ocean` normalized to the thread that takes the longest time to finish. The first, second, and third bars in each cluster correspond to the baseline, the naïve, and the elaborate arbitration scheme, respectively.

5.2 Tradeoffs between Naïve and Elaborate

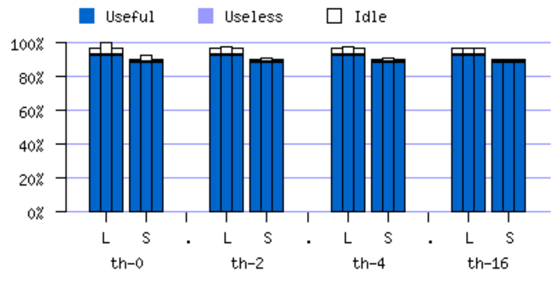
In Figure 5, we present the same results as in Figure 4 but for the `ocean` application. A striking observation is that the **naïve** scheme suffers from a huge idle time. This is caused by transactions that have to wait for unrelated transactions with a higher miss-speculation count to commit. This is unfortunate, as the waiting transactions could have been committed in the first place.

On the other hand, the **elaborate** scheme manages to keep the number of stall cycles low but at the expense of extending a commit transaction with a “check-for-data-race” cycle. It is interesting to understand whether a simple modification of the **naïve** scheme could reduce the number of idle cycles.

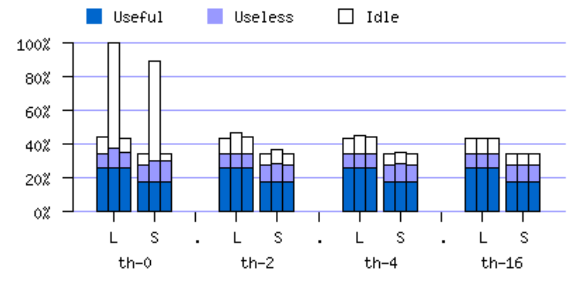
This encouraged us to pursue an approach by which we more selectively choose threads that have to stall under the **naïve** scheme. Recalling the example starvation scenario in Section 2.3, the chief observation is that a symptom of a starvation situation is that a request has been denied several times in a row. In our improved **naïve** scheme, we allow a transaction (with the lowest phase number) to commit if its MSC-count is lower than another one (with the lowest phase number) by an offset, which we refer to as a *threshold*.

Obviously, a key issue is to select an appropriate threshold. Therefore, we experimented with several threshold values. Figure 6 shows the performance of the five applications under all three schemes using threshold values of 0, 2, 4 and 16 for the **naïve** scheme which correspond to the different bar clusters (each consisting of six bars) in each diagram. In the figure, we have depicted the execution time for the longest (L) and the shortest (S) running threads for each application. Comparison between longest and shortest threads enables us to pinpoint a starvation scenario if it exists.

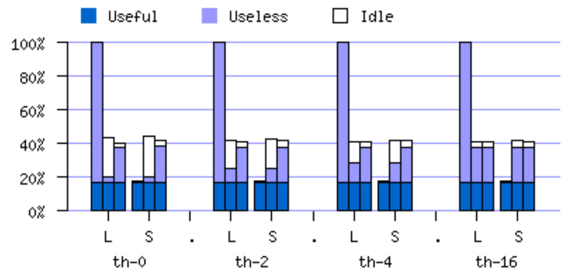
As we go from lower to higher threshold values, the idle time for the **naïve** scheme is reduced significantly. It is important to note that it becomes as low as the baseline arbitration for a threshold value of 16 for the applications which do not encounter any starvation.



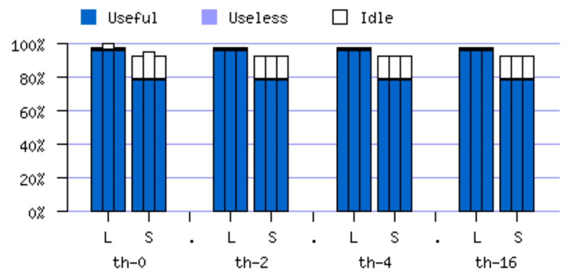
(a) Barnes



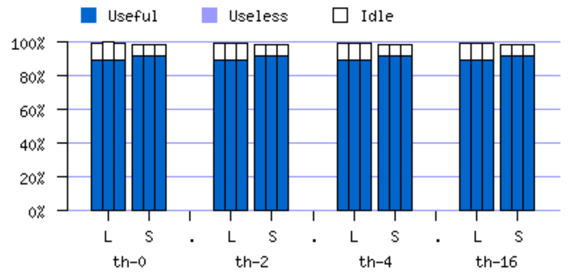
(b) Ocean



(c) Raytrace



(d) Volrend



(e) Water-Nsq

Figure 6: Relative performance of the commit arbitration schemes for the five SPLASH-2 applications using threshold values 0, 2, 4, and 16 plotted in Figure 6 through diagrams (a) to (e). The first and second clusters under each threshold represent the execution time for the longest and shortest running threads respectively. Each cluster uses three bars that represent the relative execution time for the same thread using all three arbitration schemes.

When an application uses barrier synchronizations, the performance of an application that is run under a transactional memory paradigm will suffer from idle cycles lost due to commits. This is because a transaction with a higher phase number must wait for a transaction with a lower phase number to commit. As a result, also the baseline scheme suffers from idle cycles in all applications that use barriers to synchronize as can be seen from Figure 6. One exception is `raytrace` which does not use barriers.

Finally, in Figure 7, we compare the execution times for each application under the baseline, the **naïve**, and the **elaborate** schemes. We show four bar clusters for each application that correspond to different threshold values. The execution times for each application are normalized to the slowest run of the application.

A striking observation is that as we increase the threshold, the performance difference between the **naïve** scheme and the **elaborate** scheme diminishes. Therefore, our recommendation is that the **naïve** scheme with a fairly high threshold can safely eliminate starvations.

6. RELATED WORK

We have proposed arbitration schemes for TM systems to avoid starvation. Hill et al. have classified TM systems in two categories depending on lazy versus eager conflict detection [4]. While [1,4,7] are referred to as eager, [2] does lazy conflict resolution. This paper has focused on the starvation problem for TM systems using lazy conflict resolution. However, none of the papers on neither eager nor lazy conflict resolution have carefully analyzed whether the systems are starvation-free.

TM systems that detect conflict lazily, e.g. TCC, know about the conflict at the commit point of any of the involving transactions. The straightforward first-come first-serve arbitration policy used in TCC may lead to a starvation situation as described in Section 2.3. Our proposed arbitration scheme targets a TM system that detects conflicts lazily and especially uses a central arbitration unit to select a thread to commit.

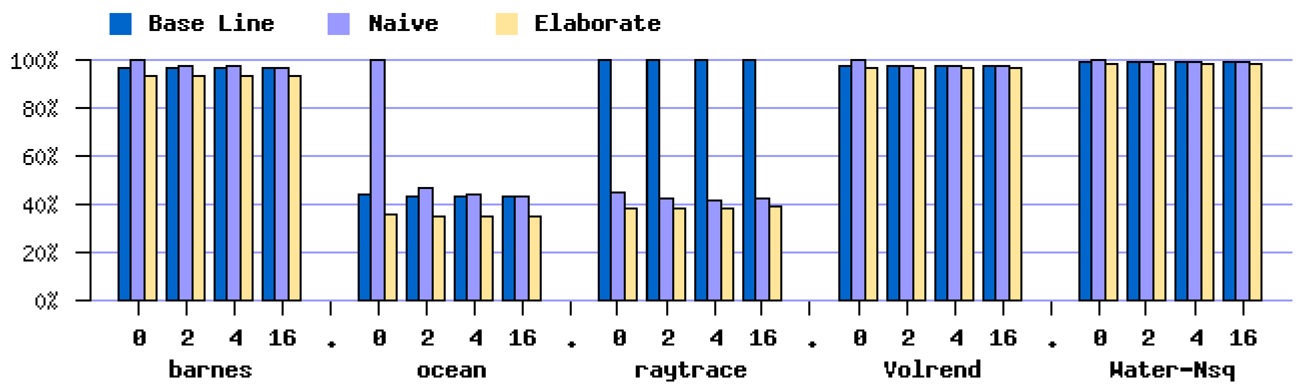


Figure 7: Relative execution times of the five SPLASH-2 applications for four different threshold values under all three arbitration schemes. Execution times for each application are normalized to the slowest run of the application.

Hammond et al. [2] proposes the use of pseudo-barriers to make forward progress for all processors. The idea is to arbitrarily insert a barrier implicitly so that a thread that is subject to starvation will get a chance to catch up. Unfortunately, the paper does not elaborate on how to make use of the idea. In fact, we have shown in this paper that it is important to monitor when a starvation scenario is about to happen. We do this by using miss-speculation counters. By arbitrarily inserting barriers, there is simply no such monitoring mechanism.

7. CONCLUDING REMARKS

In this paper, we have proposed novel commit arbitration schemes for TM systems using lazy commit resolution and present how they can be implemented in a framework based on TCC. The baseline commit arbitration scheme assumed by TM systems that lazily resolve conflicts is prone to starvation.

As a general approach to avoid starvation, we propose in this paper that the commit arbitration policy should be extended with information about how many miss-speculations that other threads have experienced.

To this end, we propose the **naïve** scheme and the **elaborate** scheme. Through a detailed implementation and performance evaluation study we found the following. All of our proposed schemes can avoid starvation but do it at the expense of lost cycles due to delaying the point at which a thread can commit.

We analyzed the design space of the simplest policy – the **naïve** policy – and found that it can eliminate most of the stall cycle by introducing a threshold value between the committing thread's and the other ongoing thread's miss-speculation count. By doing this, we have found that the simplest scheme manages to remove most of the stall cycles at the expense of some quite modest structure and protocol extensions. Overall, this paper shows that it is possible to avoid starvation at a modest implementation cost.

REFERENCES

[1] C. S. Ananian, K. Asanovi'c, B. C Kuszmaul, C. E. Leiserson, and S. Lie ``Unbounded Transactional Memory." In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05), San Francisco, CA,* pp. 316-327, February 2005.

[2] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, J. Davis, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. *Proc. of the 31st Annual International Symposium on Computer Architecture*, pp. 102-113, München, Germany, June 19-23, 2004.

[3] M. Herlihy, J. E. B. Moss. ``Transactional Memory: Architectural support for lock-free data structures." In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 289-300, 1993

[4] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood, "LogTM: Log-based Transactional Memory" In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, Austin, TX February 11-15, 2006.

[5] P.S. Magnusson, M. Christianson, J. Eskilson et al., *``Simics: A full system simulation platform"*, IEEE Computer vol.35 no.2 (Feb.2002), pp.50-58.

[6] J. Martinez and J. Torrellas. ``Speculative synchronization: Applying thread-level speculation to parallel applications." In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[7] R. Rajwar, M. Herlihy, and L. Lai. ``Virtualizing transactional memory." In *Proceedings of the 32nd International Symposium on Computer Architecture*, pp. 494-505, June 2005.

[8] R. Rajwar and J. Goodman. ``Speculative Lock Elision: enabling highly concurrent multithreaded execution." In *MICRO 34: Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pp. 294-305. IEEE Computer Society, 2001.

[9] R. Rajwar and J. Goodman. "Transactional Lock-free Execution of Lock-Based Codes." In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* October 2002.

[10] P. Rundberg and P. Stenstrom. ``Reordered speculative execution of critical sections." In *Proceedings of the 2002 International Conference on Parallel Processing*, February 2002.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. ´´The SPLASH-2 Programs: Characterization and Methodological Considerations.´´ In *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24-36, Santa Margherita Ligure, Italy, June 1995.

An Hardware/Software Framework for supporting Transactional Memory in a MPSoC Environment

Cesare Ferri[1], Tali Moreshet[2], R.Iris Bahar[1], Luca Benini[3], and Maurice Herlihy[4]

[1]Brown University, Division of Engineering, Providence, RI 02912, USA
[2]Swarthmore College, Department of Engineering, Swarthmore, PA 19081, USA
[3]Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna, 40136 Bologna, Italy
[4]Brown University, Department of Computer Science, Providence, RI 02912, USA

Abstract

Manufacturers are focusing on multiprocessor-system-on-a-chip (MPSoC) architectures in order to provide increased concurrency, rather than increased clock speed, for both large-scale as well as embedded systems. Traditionally lock-based synchronization is provided to support concurrency; however, managing locks can be very difficult and error prone. In addition, the performance and power cost of lock-based synchronization can be high. Transactional memories have been extensively investigated as an alternative to lock-based synchronization in general-purpose systems. It has been shown that transactional memory has advantages over locks in terms of ease of programming, performance and energy consumption. However, their applicability to embedded multi-core platforms has not been explored yet. In this paper, we demonstrate a complete hardware transactional memory solution for an embedded multi-core architecture, consisting of a cache-coherent ARM-based cluster, similar to ARM's MPCore. Using cycle accurate power and performance models for the transactional memory hardware, we evaluate our architectural framework over a set of different system and application settings, and show that transactional memory is a promising solution, even for resource-constrained embedded multiprocessors.

1 Introduction

Multi-core architectures have become pervasive in large-scale digital integrated systems. Manufacturers of general-purpose processors have essentially given up trying to increase clock speed, and instead are now focusing on multiprocessor-system-on-a-chip (MPSoC) architectures, in which multiple processors residing on a single chip communicate directly through shared hardware caches, providing increased concurrency instead of increased clock speed [8]. Integrated platforms for embedded applications are even more aggressively pushing core-level parallelism. SoCs with tens of cores are commonplace [23, 30, 32], and platforms with hundreds of cores have been announced [24].

In principle, multi-core architectures have the advantages of increased power-performance scalability (assuming on-chip interconnect scales well [7]) and faster design cycle time (by exploiting replication of pre-designed components). However, performance and power benefits can be obtained only if applications exploit a high level of concurrency. Indeed, one of the toughest challenges to be addressed by multi-core architects is how to help programmers expose application parallelism.

Embedded applications in multimedia, imaging, and communication, have a high degree of exposed thread-level parallelism, and this is one of the main reasons for the rapid diffusion of multi-core architectures in embedded systems [9]. However, managing concurrency is not easy. One major issue is how to synchronize concurrent accesses to memory. When multiple threads access a shared data structure, care must be taken to ensure that concurrent accesses do not interfere. Embedded multi-core architectures usually provide lock-based synchronization support for this purpose.

Lock-based synchronization relies on dedicated hardware. Atomic read-modify-write memory access is probably the most commonly deployed synchronization mechanism (e.g., it is used in the ARM MPCore architecture [5]). It requires support in the processor tile, in the communication fabric, and/or in the memory targets. For instance, MPCore relies on hardware support for locked transactions in the system interconnect. Other architectures use semaphore target devices [22] that provide direct atomic read-modify-write support.

Embedded platforms are almost invariantly deployed in tightly resource constrained contexts. Hence, the performance and power cost of synchronization is a serious concern that is further compounded by the issue of providing effective programming abstractions. In fact, managing locks is very difficult and error prone. Deadlocks are difficult to avoid, especially when systems have multiple resources. Moreover, even when the system is operating correctly, conditions like lock spinning may impose a significant overhead in power and performance, since they tend to flood the interconnect with useless transactions.

Transactional memories have been extensively investigated in general-purpose systems as an alternative to lock-based synchronization (e.g., [13, 16, 18, 10]). Transactional memory enables speculative execution of threads without acquiring locks, guaranteeing that transactions appear to execute atomically. Transactional memory has advantages over locks in terms of ease of programming, performance, and energy consumption [19, 25]. However, their applicability to embedded multi-core platforms, based on simple cores and memory system interfaces, has not been explored yet. The main objective of this paper is to demonstrate, for the first time, a complete hardware transactional memory solution for an embedded multi-core architecture, consisting of a cache-coherent ARM-based cluster, similar to ARM's MPCore. We have developed cycle accurate power and performance models for the transactional memory hardware, and we developed a lean software library that supports critical sections based on transactional memory. This hardware-software framework has been evaluated over a set of different system and application settings. Our analysis shows that, while transactional memory can provide clear performance advantages, careful consideration to hardware design is essential in order to meet the tight energy constraints of an embedded multiprocessor platform. Still, even with these tight energy constraints, our results show up to a 23% reduction in energy consumption, a 62% reduction in execution time, and a 75% improvement in energy delay product for an embedded application using transactional synchronization compared to the application using locks.

2 Background and Previous Work

Locks are the most common approach to synchronization. A lock indicates whether a shared data object is in use. A thread must acquire a lock before accessing the shared object, and release the lock after it is done. Locks may be be implemented in many different ways, including semaphores or interrupts.

Despite their widespread use, locks have serious limitations. Coarse-grained locks, which protect relatively large amounts of data, simply do not scale. Threads block one another even when they do not really interfere, and the lock itself becomes a source of contention. Fine-grained locks, which protect smaller regions of memory, may appear more scalable, but they are difficult to use effectively and correctly. In particular, they introduce substantial software engineering problems, since the conventions associating locks with objects become more complex and error-prone. Locks also cause vulnerability to thread failures and delays: if a thread holding a lock is delayed by a page fault, or context switch, other running threads may be blocked. Locks also inhibit concurrency because they must be used conservatively: a thread must acquire a lock whenever there is a possibility of a synchronization conflict, even if such a conflict is actually rare. Finally, locks have a disadvantage in terms of energy consumption [19].

Transactional memory [13] is a speculative alternative to locks. Transactional memory can be implemented in hardware [16, 21, 25, 27] or in software [10, 11, 12, 15, 17, 28], or as a hybrid hardware-software combination [3, 18, 26]. In this paper, we investigate transactional synchronization specifically for embedded architectures. We focus on hardware transactional memory, since we feel this implementation is more appropriate for the needs of embedded systems. A more detailed overview of hardware transactional memory follows.

In transactional memory, each transaction is executed speculatively by a single thread without acquiring a lock. If the transaction completes without conflicting with another transaction, it commits, and its effects become permanent. Otherwise, if conflicts were detected during execution by the native hardware cache coherence protocol, the transaction aborts, its effects are discarded, and the transaction is restarted at a later time.

Until a transaction commits, its effects are not visible outside the transaction itself. To ensure such isolation, hardware transactional memory keeps tentative updates in a thread-local cache. If the transaction commits, these modified entries may be written back to memory. Data conflicts with other transactions will be detected by the native cache coherence mechanism. If a conflict is detected, the transaction is aborted, and its effects are discarded.

Transactional memory requires a checkpointing mechanism to support roll-back and re-issue of transactions in case of a conflict. Various checkpointing mechanisms have been proposed for recovering processor state. Many of these techniques periodically create a system-wide logical checkpoint of the system, which includes the state of the

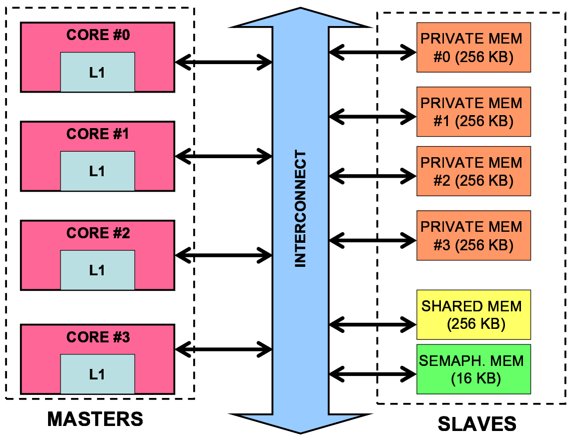


Figure 1. Example of the system configuration with 4 CPUs.

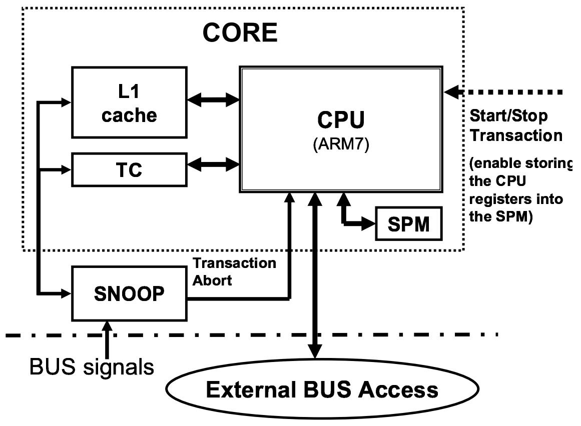


Figure 2. Architecture overview.

processor registers, memory values, and coherence permissions [29, 20]. For our purposes, it is sufficient to checkpoint the local registers of the processor that started the transaction.

Transactional memory improves ease of programming and performance of shared memory multiprocessors [13]. Hardware transactional memory provides a level of concurrency at least equivalent to the finest granularity locking, at a modest hardware cost [25]. Transactional memory also has an advantage over locks in terms of energy consumption [19], due to reduced contention and the absence of lock acquisition overhead.

In this paper, we take a first step to investigate the extent to which these advantages of transactional synchronization for general-purpose processors carry over to embedded MPSoC architectures.

3. Transactional Memory for Embedded Systems

3.1 Simulation Platform

We developed our architecture on top of the MPARM simulation framework [4, 14]. MPARM is a cycle-accurate, multi-processor simulator written in SystemC that provides high flexibility in terms of design space exploration. The designer can use the MPARM facilities to model any simple instruction set simulator with a complex memory hierarchy (supporting, for example, caches, scratchpad memories, and several types of interconnects). Finally, MPARM includes cycle-accurate power models for many of the simulated devices. The power models have been characterized by a 0.13μm technology provided by STMicroelectronics [31].

As shown in Figure 1, the adopted basic system configuration consists of a variable number of ARM7 cores (each having an 8KB direct-mapped L1 cache), a set of private memories (256KB each), one shared memory (256KB) and one bank (16KB) of memory-mapped registers which work as hardware semaphores. The interconnect is an AMBA-compliant communication architecture [1]. A Cache-coherence protocol (MESI) is also provided by snoop devices connected to the master ports. Note that while the private and shared memories are arbitrarily sized rather large (256KB) they do not significantly impact the performance or power of our system (as will be shown in Section 4).

3.2 Implementation

The basic idea behind the Transactional Memory working model is simple: each transaction is speculatively executed by the CPU and, if no conflicts with another transaction are detected, its effects become permanent (that is, the transaction *commits*). Otherwise, if conflicts are detected, its effects are discarded (that is, the transaction *aborts*), and the transaction is restarted. Hence, the support of the hardware that a Transactional Memory generally requires is *1)* a safe location for storing/modifying transactional data, and *2)* a rollback mechanism for re-executing the transaction.

We modeled our Transactional Memory after [13], and implemented a small (512B) fully-associative *Transactional Cache* (TC). The TC is accessed in parallel with the L1 cache; its main task is to manage all the read/write operations during a transaction. Preliminary experimental results suggest that most transactions are quite small, both in common benchmarks and even in the Linux kernel [2]. Therefore, in this paper, we do not consider transactions that would exceed the capacity of our TC.

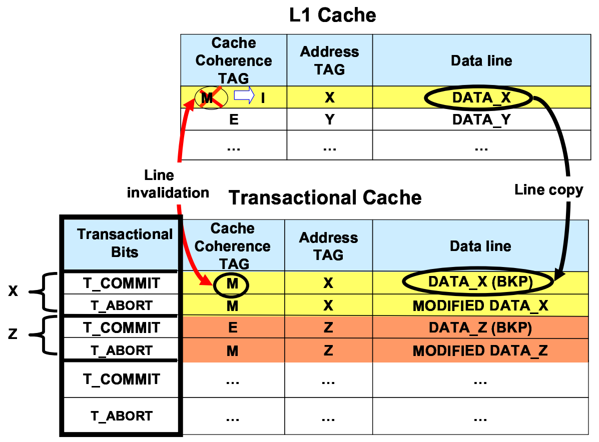


Figure 3. The TC maintains two copies for each line and is exclusive to the L1 cache.

Figure 2 provides an overview of the implemented architecture. To start a transaction, the CPU creates a local checkpoint by saving the contents of its registers into a small (128B) Scratchpad Memory (SPM) [6]. Note that during the time the CPU is writing into the SPM, the pipeline is stalled. In our case, moreover, the SPM must be carefully resized in order to contain the entire set of CPU registers.

During the execution of the transaction, two copies of accessed data will be stored in the TC, as shown in Figure 3. That is, two physical lines are maintained for each address: one line stores the backup copy of the data and the other contains the data modified during the transaction. This scheme requires adding 2 extra bits to the cache coherence tag vector. These bits will allow us to distinguish between the backup copy (marked with the *T_COMMIT* bit) and the modified data (marked with the *T_ABORT* bit). If neither bit is set, this indicates that the data in the line is not contained within a transaction and therefore can be managed in the usual manner by the snoop device.

In case of a data conflict, the snoop device notifies the CPU with the *Abort_Transaction* signal which causes all the *T_ABORT* lines to be invalidated. Once the CPU receives the *Abort_Transaction* signal, it *1)* stops the execution of the transaction, *2)* restores the registers values by reading from the SPM, and finally *3)* changes its status to "low power" mode. The CPU remains in this low power mode for some random backoff period after which it can begin re-executing the transaction. The range of the backoff period is tuned according to the conflict rate. That is, the first time a transaction conflicts it waits an initial random time period (<100 cycles) before restarting. If the transaction conflicts again, the backoff period is doubled each time until the transaction completes successfully.

If there is no conflict, and the transaction commits, the TC invalidates the *T_COMMIT* lines and resets the *T_ABORT* bits of the tag vector. This results in the lines that were marked before as *T_ABORT*, now containing useful data for the rest of the system. As a consequence, the snoop device will manage these lines according to the native cache coherence protocol.

For correct implementation, the system needs a write-back (with write-allocate) cache policy, (e.g., all the writes have to be done within the TC). In addition, if the TC requires data contained in the L1 cache, the corresponding L1 lines will be copied to the TC and then invalidated in the L1 (guaranteeing non-overlapping data-sets). It is important to emphasize that the snoop device will continue to manage all the non-transactional valid lines (*i.e.*, the lines which do not have the *T_COMMIT*, *T_ABORT*, or *INVALID* bits set) according to the standard cache-coherence protocol (the MESI scheme, in our case).

The software support for the transactions is provided by a library of special instructions that are invoked using macros. The macros produce a write into a special memory-mapped location that controls the signals *Start_Transaction* and *Stop_Transaction*, as shown in Figure 2.

4 Experimental Results

In this section, we evaluate the power and performance benefits of using hardware transactional memory with an embedded multi-core system. For this purpose, we developed a parameterizable micro-benchmark that can be used to represent a number of common shared memory access patterns in embedded applications. The micro-benchmark consists of a sequence of atomic operations executed on a shared matrix, which is logically subdivided into overlapping regions. To ensure that concurrent accesses to shared data do not produce incorrect results, processors must obtain exclusive access to each region. Using a conventional scheme, we would associate a lock with each region to ensure that processors do not see inconsistent data at the boundaries of the matrix. Such a scheme is typically used in image-processing applications for embedded systems (such as plotters, printers, digital cameras). To run the micro-benchmark with transactional memory, we replaced the locks and critical sections defined above with transactions.

Embedded system applications may include varying computational workloads within critical sections, as well as outside of critical sections. To study the effect of such variations, we include four different configurations of our micro-benchmark, marked as *C1*, *C2*, *C3*, and *C4*, in Table 1. For example, *C2* denotes a micro-benchmark configuration with a medium computational workload inside critical sections and a light computational workload outside of them, meaning that transactions compose a large portion of the micro-benchmark.

Benchmark Configuration	Inside Crit. Section	Outside Crit. Section
C1	∼60%	∼40%
C2	∼85%	∼15%
C3	∼20%	∼80%
C4	∼5%	∼95%

Table 1. Micro-benchmark workload characterization.

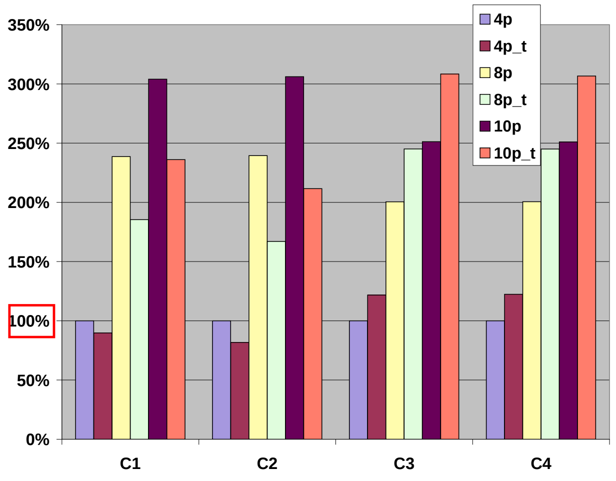


Figure 4. System energy of different micro-benchmark configurations with 4, 8, and 10 CPUs using locks (4, 8, 10p) and transactions (4, 8, 10p_t). The results within each group are normalized with respect to 4p.

Figure 4 shows the system energy of the different micro-benchmark configurations running on 4, 8, and 10 CPUs. Varying the number of CPUs allows us to explore the scalability of using transactions with increasing amounts of interconnect congestion. The system energy represents the sum of the energy of the cores, transactional caches (TC), L1 caches, scratchpad memories, RAMs and buses. Each group of bars represents a micro-benchmark configuration from Table 1 (*C1*–*C4*). The leftmost pair of bars in each group represents the system energy when running on 4 CPUs and using locks (*4p*), or transactions (*4p_t*). The second and third pairs of bars in each group have a similar representation for 8 (*8p*, *8p_t*) and 10 (*10p*, *10p_t*) CPUs, respectively. Results within each group are normalized with respect to the first bar in the group (*4p*). Figure 5 uses similar notation to show the overall execution time for the different micro-benchmark configurations relative to the *4p* configuration.

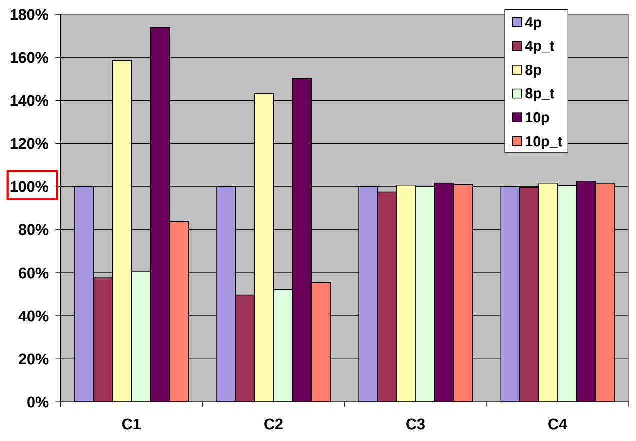


Figure 5. Execution time of different micro-benchmark configurations with 4, 8, and 10 CPUs using locks (4, 8, 10p) and transactions (4, 8, 10p_t). The results within each group are normalized with respect to 4p.

From looking at the first two groups of bars of Figures 4 and 5 (*C1* and *C2*), we see that replacing locks with transactions reduces the system energy by 10%–31% and execution time by 42%–64%, relative to a configuration with the same number of processors, but using locks instead of transactions for synchronization. By contrast, the last two groups of bars (*C3* and *C4*) show that although replacing locks with transactions has a negligible effect on the execution time, it has a noticeably negative effect on system energy. In these micro-benchmark configurations, most of the computation occurs outside the synchronized code, so transactions do not play a major role. The additional energy is consumed by the transactional cache described in Section 3. In the simple design considered here, the transactional cache is active for the duration of micro-benchmark execution since, once a transaction commits, it is possible that the only valid copy of certain data resides in this cache. It follows that every access to the L1 cache requires a parallel access to the transactional cache, even if no transaction is in progress. For micro-benchmark configurations *C1* and *C2*, this additional energy is more than balanced by eliminating the extra energy consumed by locks, but not for configurations *C3* and *C4*, which do much less synchronization. These results suggest that it would be worth investigating more complex implementations that, for example, empty and power down the transactional cache when no transaction is in progress. Our preliminary experiments using such an implementation suggest that energy consumption could be improved substantially. For example, on a system with 8 CPUs, the energy consumption for benchmark configura-

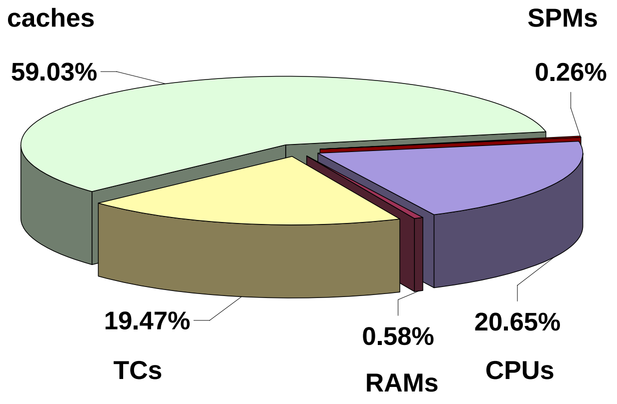


Figure 6. Energy Distribution for benchmark *C1* with transactions and 4 CPUs (4p_t).

tions *C3* and *C4* would be reduced by 16% and 18% respectively, compared to results shown in Figure 4 for the 8p_t case. In other words, the overall energy consumption of the system would remain about the same whether a lock-based or transaction-based synchronization scheme were used for the *C3* or *C4* configurations. This makes sense since the fraction of time spent executing critical sections of the code is quite small.

To get a better understanding of the overall energy consumption, Figure 6 shows the energy distribution for the *C1* micro-benchmark configuration with transactions and four CPUs.[1] As may be expected, the caches and CPUs are the most significant contributors to the system energy (the RAMs are on-chip, and consume a very small fraction of the energy). It is notable that the transactional caches (TC) and the CPUs consume comparable amounts of energy (20.65% and 19.47% respectively). The transactional cache is a fully-associative cache, which explains its high energy consumption. These results further emphasize the need to consider alternative schemes for operating the transactional cache, especially when running applications where transactions make up a relatively small portion of the total execution time. While our initial experiments suggest substantial energy benefits from being able to turn off the TC when the CPU is not executing transactional code, future work will investigate more precise hardware and software solutions for dealing with this issue.

To summarize our results, Figure 7 shows the energy delay product (EDP) of the different micro-benchmark configurations. Note from the figure that transactions introduce substantial improvements over locks for the *C1* and *C2* micro-benchmark configurations, mainly through reduction of overall execution time. For example, the *C1* configuration with transactions and 8 CPUs (8p_t) obtains a system energy reduction of 23%, a 62% shorter execution time, and a 71% better EDP over the same configuration with locks (8p). Table 2 summarizes the experimental results for *C1* and *C2*. Although increasing the time spent inside a transaction increases the probability for conflicts, transactional memory still manages to obtain higher throughput and lower energy.

[1]Energy distribution for other configurations using transactions showed similar results.

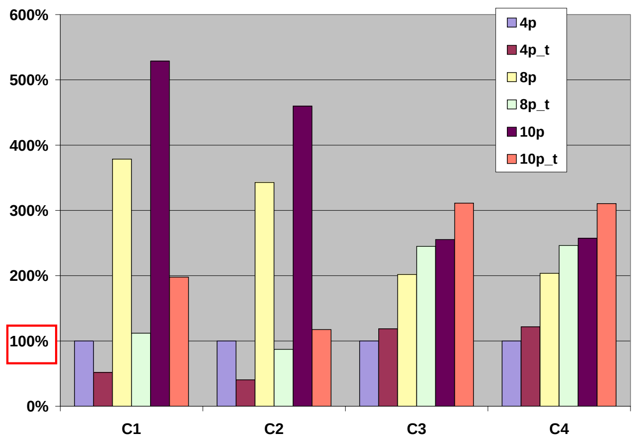


Figure 7. Energy*Delay Product (EPD), normalized with respect to 4p.

Configuration	System Energy%	Exec. Time%	EDP%
C1			
4p_t *vs.* 4p	89.8%	57.6%	51.7%
8p_t *vs.* 8p	77.7%	38.1 %	29.6%
10p_t *vs.* 10p	77.7%	48.2%	37.4%
C2			
4p_t *vs.* 4p	81.7%	49.6%	40.5%
8p_t *vs.* 8p	69.7%	36.5%	25.5%
10p_t *vs.* 10p	69.1%	36.9%	25.5%

Table 2. Summary of *C1*, *C2* results (lower values are better).

5. Conclusions

We view this work as a first step toward understanding the extent to which applications on embedded MPSoC architectures can benefit from transactional synchronization.

In particular, we are the first to present a transactional memory solution for a SoC platform technology that is modeled with cycle-accurate precision, and with accurate power models. We use frequency and power numbers and architectural assumptions that are appropriate for an embedded multiprocessor based on simple cores. Our results show that, while transactional memory can provide clear performance advantages, careful consideration to hardware design is essential in order to meet the tight energy constraints of an embedded system. This finding is substantially different from one obtained when analyzing transactional memory on a general purpose platform, since energy consumption is not so tightly constrained for these systems and the hardware to support transactional memory contributes a much smaller fraction to overall energy, as reported in [19]. Still, even with these tight energy constraints, our results show substantial improvement in terms of both energy and performance are possible using transaction-based synchronization on an embedded platform.

Significant work remains to be done to investigate other benchmarks and a wider range of architectural choices and hardware implementations. One interesting question is whether embedded applications will require new hardware mechanisms to support embedded applications. For example, little is known about how transactional synchronization interacts with patterns such as software pipelining, or with soft real-time constraints.

References

[1] ARM Ltd. The advanced microcontroller bus architecture (AMBA) homepage. www.arm.com/products/solutions/AMBAHomePage.html.

[2] C. S. Ananian, K. Asanovic, B. C. Duszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, pages 316–327, 2005.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *International Symposium on High-Performance Computer Architecture*, February 2005.

[4] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous MPSoC design space exploration. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1145–1150, 2006.

[5] MPCore multiprocessor family. www.arm.com.

[6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.

[7] L. Benini and G. D. Micheli. Networks on chip: a new SoC paradigm. *IEEE Computer*, 35(1), January 2002.

[8] S. Borkar and et al. Platform 2015: Intel processor and platform evolution for the next decade. Technical report, Intel, 2005. White Paper.

[9] G. Declerck. A look into the future of nanoelectronics. In *IEEE Symposium on VLSI Technology*, pages 6–10, 2005.

[10] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2003.

[11] T. Harris, S. Marlowe, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming (PPOPP)*, 2005.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *Symposium on Principles of Distributed Computing*, July 2003.

[13] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, May 1993.

[14] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MP-SoC environment. In *Design and Test in Europe Conference (DATE)*, pages 752–757, February 2004.

[15] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. Technical Report TR 868, Computer Science Department, University of Rochester, May 2005.

[16] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[17] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, New York, NY, USA, 1997.

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *International Symposium on High-Performance Computer Architecture*, February 2006.

[19] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In *Workshop on Memory Performance Issues*, February 2006. in conjunction with HPCA.

[20] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReVive/IO: Efficient handling of i/o in highly-available rollback-recovery servers. In *International Symposium on High-Performance Computer Architecture*, February 2006.

[21] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[22] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné, and G. Nicolescu. Parallel programming models for a multiprocessor platform applied to networking and multimedia. *IEEE Transactions on VLSI Systems*, 14(7):667–680, July 2006.

[23] Philips nexperia platform. www.semiconductors.philips.com.

[24] PC205 platform. www.picochip.com.

[25] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[26] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *International Symposium on Computer Architecture*, June 2005.

[27] P. Rundberg and P. Stenström. Speculative Lock Reordering: Optimistic Out-of-Order Execution of Critical Sections. In *International Parallel and Distributed Processing Symposium*, April 2003.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.

[29] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, May 2002.

[30] Nomadik platform. www.st.com.

[31] STMicroelectronics. www.stm.com.

[32] OMAP5910 platform. www.ti.com.

Function Level Parallelism Driven by Data Dependencies

Sean Rul Hans Vandierendonck Koen De Bosschere

Department of Electronics and Information Systems (ELIS),
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
E-mail: {srul, hvdieren, kdbosche}@elis.ugent.be

Abstract

With the rise of Chip multiprocessors (CMPs), the amount of parallel computing power will increase significantly in the near future. However, most programs are sequential in nature and have not been explicitly parallelized, so they cannot exploit these parallel resources. Automatic parallelization of sequential, non-regular codes is very hard, as illustrated by the lack of solutions after more than 30 years of research on the topic. The question remains if there is parallelism in sequential programs that can be detected automatically and if so, how much parallelism there is.

In this paper, we propose a framework for extracting potential parallelism from programs. Applying this framework to sequential programs can teach us how much parallelism is present in a program, but also tells us what the most appropriate parallel construct for a program is, e.g. a pipeline, master/slave work distribution, etc.

Our framework is profile-based, implying that it is not safe. It builds two new graph representations of the profile-data: the interprocedural data flow graph and the data sharing graph. This graphs show the data-flow between functions and the data structures facilitating this data-flow, respectively.

We apply our framework on the SPECcpu2000 bzip2 benchmark, achieving a speedup of 3.74 of the compression part and a global speedup of 2.45 on a quad processor system.

1 Introduction

Today we are at the dawn of a new era in computer architecture. While during the past decade the computer scene was mainly dominated by complex uniprocessors with deep pipelines, we now see the rise of CMPs containing several slimmer cores. This transition was fueled by several incentives. Firstly, the instruction level parallelism (ILP) has been exploited to its full extent, such that extracting more ILP becomes overly complex. Secondly, power dissipation kept increasing alarmingly, caused by implementing a single large core with long wires and clocked at high frequencies.

While this transition was necessary, we are also confronted with new issues. A big question is how we can exploit all this parallel processing power in the new processor generation? Although there is a group of programs, such as scientific and media applications, that inherently have a lot of easily exploitable thread-level parallelism (TLP), another majority of programs are inherently sequential. These programs, exemplified by the SPECcpu integer benchmark suite, have remained out of scope of research on parallelization up to the last few years.

Automatically parallelizing inherently sequential programs is hard due to the difficulty of correct static analysis of both control flow and data flow. To extract large amounts of thread-level parallelism, it is necessary to look past these limiting control flow and data flow restrictions. In this paper, we develop a framework for extracting thread-level parallelism from sequential programs that assumes perfect knowledge of these dependencies. As such, it is able to discover large amounts of TLP.

Our goal is to discover non-speculative parallelism in the first place, without being restricted by the unpredictability of control flow and data flow. Hereto, we measure the control flow and data flow exhibited during a particular run of the program. Analysis of this control flow shows opportunities for parallelization. However, by measuring dependencies during one or more particular runs of a program, the parallelism extracted by our framework may be unsafe, i.e., particular dependencies may arise depending on the input data set of the program. This situation can be handled using, e.g. thread-level speculation (TLS) systems [9, 14] that allow dependencies to be violated, but detects and corrects them with the aid of hardware support. It is also possible to use our framework as an analysis tool for a parallel programmer, who can use it to detect parallelism, but still needs to validate their correctness.

Our analysis focusses on memory dependencies, since register dependencies can be predicted [15] or precomputed [1]. We form two graph representations that form an abstraction of the profiled dependencies. These will, together with the call graph, help in detecting the chunks of code that can be parallelized.

The field of applications of our framework is very broad. Our main objective is to utilize the parallel processing power in chip multiprocessors (CMPs) for sequential programs. The same request for thread-level parallelism exists when programming the Cell processor [2]. A Cell processor contains a control processor and eight synergistic processing elements (SPEs) for performing streaming operations. Each SPE has its own local storage, so the additional problem arises of how to partition the data in a program across the SPEs. Here too, we believe our framework can help: when identifying threads, it also identifies the data structures private to a thread and shared by threads. Multiprocessor embedded systems also pose the problem of partitioning code and data across multiple cores, but here power, communication and cost constraints become very important. IMEC has developed the SPRINT [13] tool to partition code and data for embedded systems taking these constraints into account.

This paper is organized as follows: in Section 2 we describe our approach to find parallelism in a sequential program, the next section gives some results for a real life example as a proof of concept. A comparison with related work is made in Section 4. In Section 5 we summarize our contributions and look ahead on future work.

2 Method

As mentioned before, memory dependencies impede parallelism. Therefore we want to locate these dependencies in the program. Since an exact analysis is very difficult due to the required alias-analysis, we choose for a profile based approach. While a profile based technique is less general than an exact parallelization, since the information may be input dependent, it can give valuable information for parallelization and be used as a hint to the programmer.

Another issue that needs taking care of is at which level the memory dependencies are measured. This choice mainly determines the granularity of our parallelism. Since the main application field targets CMPs or multiprocessors, the granularity of the parallelism should be large enough in order to justify the use of threads, which comes with a certain overhead. We attempt to parallelize large chunks of code, with an order of magnitude of at least millions of instructions. Moreover the chunks may not be data dependent on one another. Therefore an initial choice is to look at the function level, which keeps life simple. For our profiling technique we decided to measure the memory dependencies between different functions. This approach gives a sufficiently detailed overview of the program.

Profiling distinguishes all functions, but also all data structures used by the program, such that it is possible to detect exactly what data structures communicate data from one function to another and what data structures are private to a function. This information is essential to parallelize functions in the program. To facilitate detecting parallelism between functions, based on data dependencies, we introduce two new graph representations to visualize these dependencies. These graphs identify parallelizable code, as well as data structures that need synchronization. The main focus of this paper is on the construction of the new graph representations and the ability of detecting parallel code. As this work is in its infancy, some steps in this process are not yet completely formalized.

2.1 Analysis

To build up a call graph, we record all the function calls and returns. These caller/callee relations form a first restriction on program parallelism, since the caller passes arguments and the callee may give a return value back. We keep track of how many times the function is called, the number of different functions it calls and the fraction of execution time it consumes. The latter one is taken into account for balancing the work between different threads. In Figure 1 we give an example of a call graph.

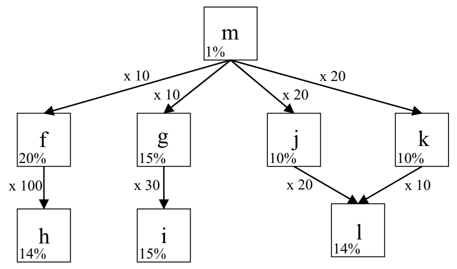


Figure 1. Example of call graph

Data dependencies are measured during a profiling phase to discover which function is reading what data from which functions. We record this information in a matrix-like data structure, one for each memory address. The matrix basically records data flow: cell (i, j) of the matrix shows how many times function i read a value that was produced by function j. Table 1 shows an example of this data structure for address `0x1000`. Each function that accesses this address has its own row. The column *Producer* shows the functions accessing this address. The column *Times produced* shows for each function how many times it has written a value to this address. The columns *Consumer* show for each row how many times it has read a value, produced by a function indicated in the subcolumn. All this information allows us to reconstruct the different producer/consumer relations between functions. To obtain this information all load and store instructions update the data structure representing their involved memory address. A function that issues a store instruction increments the value in the column *Times produced* on the row representing the current function. Furthermore the variable *Current Producer* associated with this memory address, is altered to the current function. When a function performs a load instruction, the column in *Consumer* is selected by *Current Producer* and the value in the row of the current function is incremented. Figure 2 shows a small example of memory operations in a trace, which result in the values stored in Table 1.

Address `0x1000`				
Producer	Times produced	Consumer		
		`F1`	`F2`	`F3`
`F1`	1			
`F2`	2	1	2	
`F3`			2	

Table 1. Matrix representation of memory behavior of exemplary trace

For 32-bit applications addresses are aligned on 4 byte, so a byte-operation to address `0x1000FFF2` and a word-operation to address `0x1000FFF0` are both stored in a data structure labeled with address `0x1000FF0`. A quad-operation is regarded as two 4-byte operations.

```
Function    Operation    Current producer
   F1         store             ??
   F2         load              F1
   F2         store             F1
   F2         load              F2
   F3         load              F2
   F2         store             F2
   F2         load              F2
   F3         load              F2
```

Figure 2. Exemplary trace of memory operations to address `0x1000`

When there is a load operation for which there is no previous producer, we assume that it is constant (an extra *Consumer*-subcolumn in the matrix). This situation occurs for example when a program reads data from a constant data section, or when data is produced by a system call.

Interprocedural data Flow graph For each memory address that is touched during execution we build a matrix similar to the one presented in Table 1. With this information we can determine which function reads data from other functions. This can be represented in a directed graph where the nodes are the functions and the edges show the data streams (Figure 3). We use the notation $f \xrightarrow{x} g$ to indicate that a function g consumes x bytes of data produced by f. The next step is to search for strongly interconnected functions. That is, functions that share a large amount of data. This allows us to cluster functions sharing data structures (strongly connected functions). Clustering the functions divides the data streams in two categories: *intercluster data streams* and *intracluster data streams*. In Figure 3 the function clusters are shown with grey rectangles. To indicate that a function f belongs to cluster N we use the notation f_N. Consequently an intercluster data stream is noted as $f_N \rightarrow g_M$, while an intracluster data stream has the form $f_N \rightarrow g_N$.

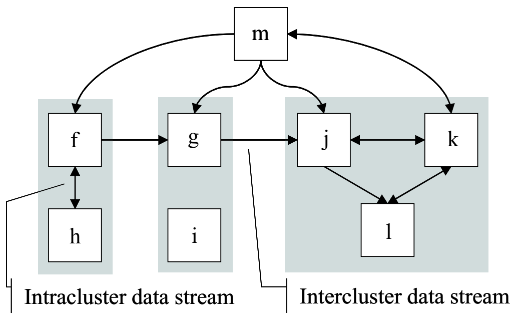


Figure 3. Example of interprocedural data flow graph

One of the aims of this clustering is to limit the intercluster data streams, in order to facilitate parallelization. This does not necessarily means that intercluster data streams should be non-existent or small in capacity. Such a requirement would be too restrictive, especially when we are dealing with sequential programs. A more reasonable goal is to impede the amount of bidirectional intercluster data streams. If data streams between two clusters of functions go in both directions, it is hard to parallelize them further on. Unidirectional intercluster data streams form less of a problem, since these are suitable for certain parallel constructs, as will be explained in Section 2.2. Bidirectional intracluster data streams cause less trouble, since the functions in that cluster are executed sequentially.

The clustering of functions is also guided by the call graph, since this graph imposes a certain hierarchy between different functions. For example in the call graph of Figure 1 we see that function h is only called by function f, so it may be a good choice to put these two functions in the same cluster. On the other hand if the fraction of execution time of these two functions is too large, they are not put together in consideration of finding a balanced solution. So the second role of the call graph is to find clusters that are balanced in execution time.

Data sharing graph While the previous representation showed the existing data streams between functions, it does not show *how* the data is shared. In other words, in order to show which function uses what, we have to make another abstraction of the profiled information. The idea is to show both the data dependencies as well as the involved data structures. The resulting graph has two kinds of nodes: function nodes and data nodes. An edge from a function node to a data node indicates the number of write accesses made by the function. We use the notation $f \xrightarrow{w} ds_1$. The opposite, an edge from a data node to a function node, is seen as the number of read accesses from that function to the data structure, which is noted as $ds_1 \xrightarrow{r} f$.

Memory addresses that are used by the same functions with a similar read/write behavior are saved in the same data structure. This reduces the number of data nodes compared to the profiling phase, where the behavior of each memory address was stored in a different structure. It also forms a good heuristic for reconstructing the original data structures used in the program source.

If we examine one function and all the data structures it accesses, we can distinguish 4 types of data usage, which are represented in Figure 4. If a function reads data, that is written by another function, it is a *consumer*. On the other hand, if it writes the data, which is read by another function, it is a *producer*. Data that is both read and written by the same function, is seen as *private consumption*. Sometimes data is read, without a traceable origin, which is considered as *constant consumption*. If we do this classification for each function, we get a graph, the *data sharing graph* (Figure 5), which shows how the data is shared between different functions. Rectangular nodes represent functions, while elliptic nodes are data structures. Note that there can be function nodes that are not connected with any other functions (for example function i). These functions only communicate with other functions by passing arguments via registers and the stack.

If we map the clustering of functions, obtained with the interprocedural data flow graph, we detect which data structures become private within a cluster and which ones are shared between different clusters, respectively called *cluster private* and *cluster shared*. This will prove to be useful when the actual parallelization is performed, as described in section 2.3.

2.2 Parallel constructs

There are numerous techniques to make a program work in parallel. Each of them having different synchronization requirements and useful under different circumstances. In the software world there are three paradigms that are commonly used [5].

The first is the *Master-Slave*. In this paradigm the main master thread launches several slave threads and allocates to each slave a portion of the work to be done. Once the work of all the slaves is done, a new batch of them can be started. Synchronization can be achieved by using a *barrier*.

Another paradigm is the *Workpile*. In this case each thread fetches a portion of work from some form of a queue, called the *workpile*, until there are no more entries in the workpile. The worker threads can also push extra work on the workpile.

A third paradigm is called the *Pipeline* paradigm. It is based on the simple producer/consumer relation: one pipeline stage produces data for the next stage, which digests this data and on its turn passes it on to the next stage.

For our purpose we need to detect parallel constructs based on the information we can extract from our interprocedural data flow graph and data sharing graph. In this paragraph we look at the last paradigm, the pipeline, and show that this construct can easily be detected in our interprocedural data flow graph and this in a formalized fashion.

The first requirement is that between several clusters of functions the intercluster data streams are unidirectional. Also there are no dependencies from f_m to g_n with $m < n$. This last requirement is interpreted as a function from cluster m that in sequential executions comes before all the functions of cluster n, is not dependent from a previous execution of a function from cluster n. The shared data between the cluster represents the pipeline registers. Depending on the partitioning of work, we can consider two cases. The first is the *heterogeneous pipeline* in which each stage of the pipeline handles a different function clusters. The second is called the *homogeneous pipeline* where each stage executes the same code. This is similar to the master-slave paradigm, but the synchronization is different.

2.3 Parallelization

If the previous analysis succeeded in finding parallel constructs, the next step consists of the actual parallelization. We define the pieces of code that are involved in the parallelization. This can involve the whole program or sometimes only certain parts, depending on the findings of the previous analysis. Since we looked at data dependencies between functions, the marked parts are a single function or a cluster of functions.

The memory addresses are also mapped on the corresponding data structures from the original program. For static data structures this mapping is trivial. For mapping dynamic data structures we have to record the addresses during profiling when these data structures are allocated.

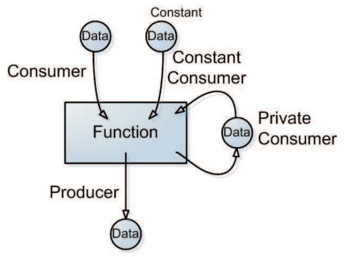


Figure 4. Classification of data dependencies.

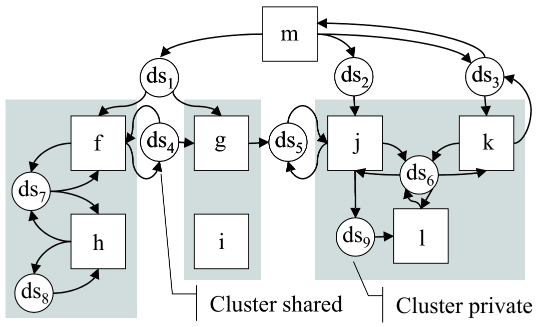


Figure 5. Example of data sharing graph.

Furthermore we detect the data structures that can be private for the different threads. Given the fact that sharing involves communication, which on its turn entails sequentiality, we want to minimize the amount of shared data. The information of which data is shared and which can be made private, can be obtained from the data sharing graph described in Section 2.1. It is clear that cluster private data of cluster N are placed in the thread that executes the functions of cluster N and need not be visible to other threads.

The necessary initialization and startup code for the threads is generated in order to preserve a correct execution of the program. Also, some code is required to complete the work after the threads are finished. New data structures are introduced to allow passing on shared data between different threads. Again the data sharing graph helps in this task. When there exists only one version of a shared data structure, this data has to be locked when it has to be altered. This mutual exclusion will prevent data races. Sometimes, a shared data structures is read and written multiple times before being passed on to the next pipeline stage. In this case, it is advantageous to create multiple instances of the shared data structure, so the data itself needs no locking. In case of a pipeline construct, the pipeline registers (the data that is shared between two pipeline stages) can be duplicated. Each pipeline stage works on its own pipeline register and passes it on to the next stage when ready. To enforce the correct execution the necessary synchronization needs to be added.

3 Results

3.1 Experimental setup

We profile the program with a modified version of Dynamic SimpleScalar [3]. The profiling code was incorporated in this simulator. The program we used as a test case was *bzip2* from the SPEC2000 benchmark suite. For our experiments during profiling, we used the reference input *program*. The parallelized version of the program was tested on a multiprocessor platform with four Itanium processors.

3.2 Compression part

Analysis In a first effort to parallelize *bzip2*, we let the profiler concentrate on the compression part of the program. In Figure 6 the part of the call graph that is responsible for more than 99% of the execution time during compression is depicted. To prevent the graph from being overloaded, we have left out the library functions that were called inside these functions. This call graph already gives a big hint in how we could cluster our functions, nevertheless without more information we can't know for sure that there are no intercluster data streams that restrain possible parallelism.

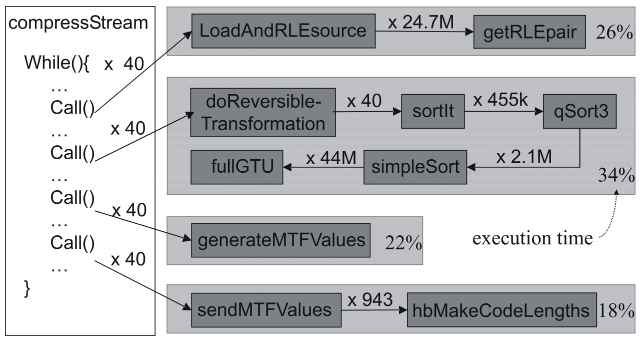


Figure 6. Partial call graph of compress procedure in *bzip2*

This is where the interprocedural data flow graph comes in. Figure 7 shows this information. Again in order to keep the overview, we discarded the library functions and their data streams. Based on the call graph and the interprocedural data flow graph of *bzip2*, we can safely identify four function clusters. The first cluster is dedicated to applying run length encoding on the input. The second block performs a reversible transformation. The third block is a move-to-front encoding. The last block is responsible for the final mapping and output. Based on this clustering we see that the intercluster data streams are unidirectional (Figure 7), which provides opportunities for parallelization using the pipeline paradigm. Furthermore, the balancing of execution time between the different clusters (Figure 6) is acceptable.

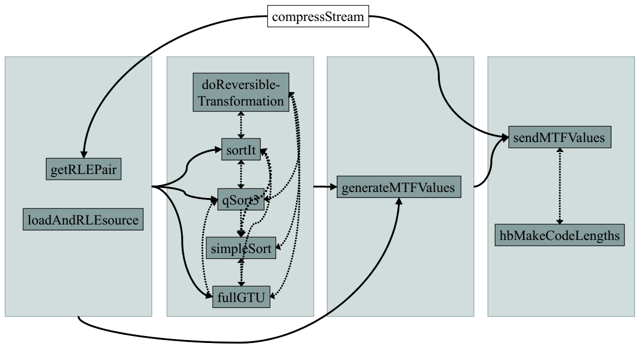


Figure 7. Simplified interprocedural data flow graph of compression procedure

A first parallelized version of the compression, *heterogeneous*, is a pipeline where each function cluster is assigned to a different thread (Figure 8(a)). Consecutive pipeline stages are connected with a synchronized pipeline register. A second parallel version, *homogeneous*, is depicted in Figure 8(b), in this case each thread runs the same code. This requires that there are no dependencies between two successive runs of the second and third cluster (dotted arrows). After closer code inspection it was verified that these dependencies were void: data structures that were shared between the two stages, were initialized (reset to zero) before use in the second cluster. This second approach has two benefits over the first: firstly the number of threads is not limited to the number of clusters and secondly it requires less synchronization (only for reading input and writing output).

Performance To compare the performance of our parallel versions of the compression, we compare the execution time of the original sequential version to the execution time of the code that is now parallelized. The results for the compression are shown in the left of Figure 9. As was expected, the homogeneous version is faster (speedup 3.60) than the heterogeneous (speedup 2.64), due to less synchronization overhead. We can also look at the speedup of the homogeneous version in function of the number of used threads (light bars in Figure 10). When we use only one thread, there is a slowdown compared to the base version, caused by the overhead of thread related code. If the number of threads is larger then the number of processor cores we first see a small increase in speedup (6 threads), but afterwards the speedup decreases (due to insufficient resources).

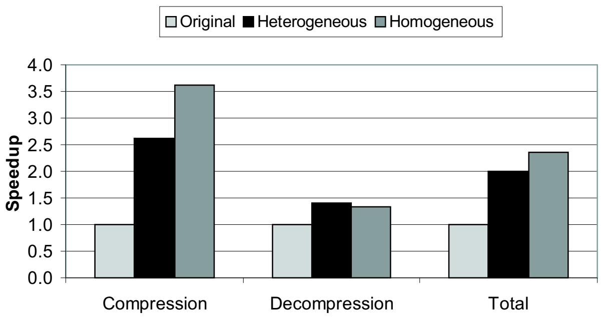


Figure 9. Speedup results of parallelized *bzip2* using 4 threads for compression and 2 for decompression

3.3 Decompression part

Analysis After the fruitful parallelization of the compression part, we also take a look at the decompression part. The analysis is completely analogous to the former. However, in this case the code is more restricted by memory dependencies for reading the input file. Therefore it was only possible to split the decompression procedure in two parts. Once again we make a homogeneous and a heterogeneous version.

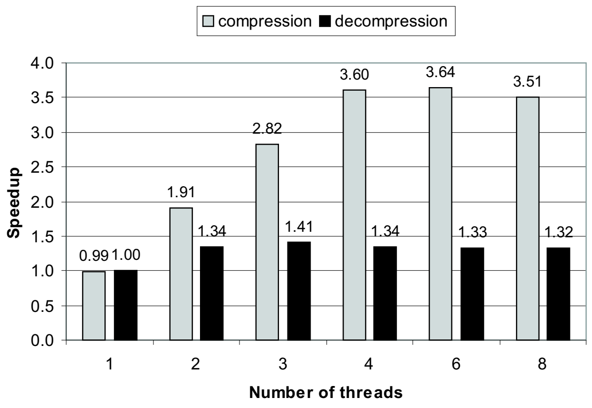
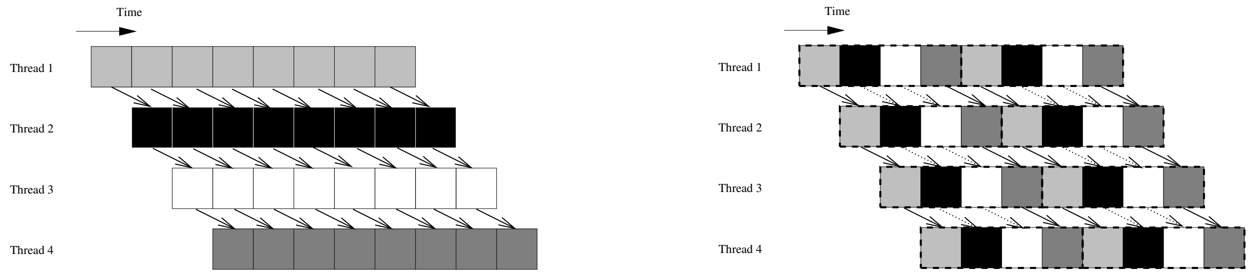


Figure 10. Speedup results for homogeneous parallelized version in function of used threads

Performance The speedup of the parallel version is illustrated in the middle of Figure 9. In this case the speedup is only about 1.41, partly due to less balanced threads. The



(a) Heterogeneous pipeline (b) Homogeneous pipeline

Figure 8. Parallel versions of compression. Each color represents a cluster of functions, the arrows indicate synchronization.

dark bars in Figure 10 show the speedup of the homogeneous version in function of the number of used threads. The results show a similar behavior as in the case of compression.

3.4 Entire program

Finally we look at the results when we put both our parallel version for compression and decompression together. The speedup of the total execution time is represented on the right in Figure 9. The heterogeneous version has a speedup of 1.99, while the homogeneous version can run 2.45 times faster than the original version. Similar speedups are achieved when using other input files and different options such as the blocksize of the compression.

4 Related work

When it comes to parallelization a lot of research went into loop transformations. Wolf and Lam showed how parallelism in perfectly nested loops could be extracted and automatically transformed [16]. In case the loops are not perfectly nested other techniques can be applied that allow transformations with a minimum of synchronization [6]. However, most of this work is focused on numerical programs and works on the level of loop iterations, while our work is more concerned with sequential programs and looks at the function level.

In order to parallelize programs there have been several proposals using speculative techniques. For example, *thread-level speculation* (TLS) [14] determines unlikely dependencies and based on this information speculatively parallelizes a program. Roth and Sohi [12] introduced *speculative data-driven multithreading*, in which critical sections are pre-executed in parallel in order to speedup the original program. In most cases these methods are based on heuristics. There also have been proposed profiling techniques [7, 11] that use profile information to determine the interesting spawning points for speculative threads.

The performance achieved with TLS is, however, disappointing. E.g. it is reported [4] that at most 31% of the execution of bzip2 can be parallelized using TLS, implying that the maximum speedup of TLS is 1.44 assuming perfect speculation and no threading overhead. In contrast, we obtain a realistic speed-up of 2.45. Prabhu and Olukotun [10] manually select regions to apply TLS and manually transform source code to extend the speedup obtainable with TLS. Nonetheless, they achieve speedups larger than 2 only when they can restructure the program such that traditional loop-level parallelism can be exploited.

The graphs introduced in this paper are new types of data flow graphs. Data flow graphs are categorized as either static data flow graphs or dynamic data flow graphs. Static data flow graphs are used by a compiler to make decisions in register allocation and optimizations. Zhao introduced the so called *Multithreaded Dependence Graph* [17] for program slicing. The dynamic version measures the data dependencies during a particular program run. *Redux* [8] is such a profiling tool, introduced by Nethercore *et al*. It measures the data dependencies at the level of assembler instructions and uses this information for debugging and program slicing.

5 Conclusion and future work

In this paper we presented a framework for extracting potential thread-level parallelism from sequential programs. We assume perfect knowledge of control flow and data flow by measuring all data dependencies occuring in a profiled execution of a program. Using the profiled information, we construct the function call graph and an interprocedural data flow graph to detect function-level parallelism. The functions in a program are clustered such that strongly communicating (dependent) functions are members of the same clusters. Thread-level parallelism between function clusters is detected by analyzing inter-cluster data flow.

Explicit synchronization between threads is required to guide the correct communication of data between parallelized functions. The data structures facilitating this communication are identified by means of the data sharing graph.

We applied our framework to the *bzip2* benchmark from the SPECcpu2000 suite. For the compression part, a 4-stage pipeline was detected. Pipelining compression yielded a speedup of 3.74. Decompression was split in a 2-stage pipeline, yielding a 1.41 speedup. Overall, the parallel version of bzip2 is 2.45 times faster then the base version.

This work is still in its initial phase, so several aspects in our approach need to be worked out further and be formalized in the near future. Once the steps are more formalized, the process can also become more automated. A point of special interest goes into finding more parallel constructs. In this paper we mainly focused on pipeline constructs, but it is clear that other constructs can be extracted from our graphs that may uncover more parallelism. More parallel constructs will also enable us to produce results for more programs and make the presented technique more versatile. Furthermore bidirectional data streams need to be investigated more accurately, due to their frequent occurrence. This could be achieved by collecting some timing information during the profiling phase. Depending on the distribution in time these data streams could also prove to be useful for certain parallel constructs.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. Sean Rul is supported by a grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Hans Vandierendonck is a post-doctoral research fellow with the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

References

[1] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, 2001.

[2] B. Flachs, S. Asano, H. Dhong, et al. The microarchitecture of the synergistic processor for a cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):63–70, Jan. 2006.

[3] X. Huang, J. E. B. Moss, K. S. McKinley, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Feb. 2003.

[4] A. Kejariwal, X. Tian, W. Li, et al. On the performance potential of different types of speculative thread-level parallelism. In *ICS06: Proceedings of the 2006 International Conference on Supercomputing*, pages 24–35, 2006.

[5] S. Kleiman, D. Shah, and B. Smaalders. *Programming with threads*. SunSoft Press, Mountain View, CA, USA, 1996.

[6] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, 1997.

[7] P. Marcuello and A. González. Thread-Spawning Schemes for Speculative Multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 55–64, 2002.

[8] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2):1–22, Oct. 2003.

[9] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, 1999.

[10] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 142–152, 2005.

[11] C. G. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on precomputation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, June 2005.

[12] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 37, 2001.

[13] IMEC's new SPRINT design methodology transforms sequential code to multi-threaded code. http://www.imec.be/wwwinter/mediacenter/en/Sprint 2004.shtml.

[14] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[15] D. M. Tullsen and J. S. Seng. Storageless Value Prediction Using Prior Register Values. In *ISCA'99: Proceedings of the 26th International Symposium on Computer Architecture,*, pages 270–279, 1999.

[16] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.

[17] J. Zhao. Multithreaded dependence graphs for concurrent java programs. In *Proceedings of 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 13–23, May 1999.

NOTES