8th Annual Workshop on Duplicating, Deconstructing, and Debunking

http://www.ece.wisc.edu/~wddd

Held in conjunction with the 36th Annual International Symposium on Computer Architecture (ISCA-36)

Austin, Texas
June 21, 2009

Workshop Organizers:

Bryan Black, AMD, (bryan.black@amd.com)
Natalie Enright Jerger, University of Toronto,
(enright@eecg.toronto.edu)
Gabriel Loh, Georgia Tech, (loh@cc.gatech.edu)

Eighth Annual Workshop on Duplicating, Deconstructing, and Debunking

Sunday, June 21 2009

Final Program

Session I: 1:30 -3:30 PM

The Performance Potential for Single Application Heterogeneous Systems
Henry Wong and Tor Aamodt
University of Toronto and University of British Columbia

Our Many-Core Benchmarks Do Not Use that Many Cores
Paul Bryan, Jesse Beu, Thomas Conte, Paolo Faraboschi and Daniel Ortega
Georgia Institute of Technology and HP Labs

Session II: 4:00 – 5:30 PM

Considerations for Mondriaan-like Systems
Emmett Witchel
University of Texas at Austin

Is Transactional Memory Programming Actually Easier?
Christopher Rossbach and Owen Hofmann and Emmett Witchel
University of Texas at Austin

The Performance Potential for Single Application Heterogeneous Systems*

Henry Wong
University of Toronto
henry@eecg.utoronto.ca

Tor M. Aamodt
University of British Columbia
aamodt@ece.ubc.ca

Abstract

*A consideration of Amdahl's Law [9] suggests a single-chip multiprocessor with asymmetric cores is a promising way to improve performance [16]. In this paper, we conduct a limit study of the potential benefit of the tighter integration of a fast sequential core designed for instruction level parallelism (e.g., an out-of-order superscalar) and a large number of smaller cores designed for thread-level parallelism (e.g., a graphics processor). We optimally schedule instructions across cores under assumptions used in past ILP limit studies. We measure sensitivity to the sequential performance (instruction read-after-write latency) of the low-cost parallel cores, and latency and bandwidth of the communication channel between these cores and the fast sequential core. We find that the potential speedup of traditional "general purpose" applications (e.g., those from SpecCPU) as well as a heterogeneous workload (game physics) on a CPU+GPU system is low (2.2× to 12.7×), due to poor sequential performance of the parallel cores. Communication latency and bandwidth have comparatively small performance impact (*1.07× *to* 1.48×*) calling into question whether integrating onto one chip both an array of small parallel cores and a larger core will, in practice, benefit the performance of these workloads significantly when compared to a system using two separate specialized chips.*

1 Introduction

As the number of cores integrated on a single chip continues to increase, the question of how useful additional cores will be is of intense interest. Recently, Hill and Marty [16] combined Amdahl's Law [9] and Pollack's Rule [28] to quantify the notion that single-chip asymmetric multicore processors may provide better performance than using the same silicon area for a single core or some number of identical cores. In this paper we take a step towards refining this analysis by considering real workloads and their behavior scheduled on an idealized machine while modeling communication latency and bandwidth limits.

Heterogeneous systems typically use a traditional microprocessor core optimized for extracting *instruction level parallelism* (ILP) for serial tasks, while offloading parallel sections of algorithms to an array of smaller cores to efficiently exploit available data and/or thread level parallelism. The Cell processor [10] is a heterogeneous multicore system, where a traditional PowerPC core resides on the same die as an array of eight smaller cores. Existing GPU compute systems [22, 2] typically consist of a GPU with a discrete GPU attached via a card on a PCI Express bus. Although development of CPU-GPU single-chip systems has been announced [1], there is little published information quantifying the benefits of such integration.

One common characteristic of heterogeneous multicore systems employing GPUs is that the small multicores for exploiting parallelism are unable to execute a single thread of execution as fast as the larger sequential processor in the system. For example, recent GPUs from NVIDIA have a register to register read-after-write latency equivalent to 24 shader clock cycles [25][1]. This latency is due in part to the use of fine grained multithreading [32] to hide memory access and arithmetic logic unit latency [13]. Our limit study is designed to capture this effect.

While there have been previous limit studies on parallelism in the context of single-threaded machines [7, 17, 15], and homogeneous multicore machines [21], a heterogeneous system presents a different set of trade-

*Work done while the first author was at the University of British Columbia.

[1]The CUDA programming manual indicates 192 threads are required to hide read-after-write latency within a single thread, there are 32-threads per warp, and each warp is issued over four clock cycles.

offs. It is no longer merely a question of how much parallelism can be extracted, but also whether the parallelism is sufficient considering the lower sequential performance (higher register read-after-write latency) and communication overheads between processors. Furthermore, applications with sufficient thread-level parallelism to hide communication latencies may diminish the need for a single-chip heterogeneous system except where system cost considerations limit total silicon area to that available on a single-chip.

This paper makes the following contributions:

- We perform a limit study of an optimistic heterogeneous system consisting of a sequential processor and a parallel processor, modeling a traditional CPU and an array of simpler cores for exploiting parallelism. We use a dynamic programming algorithm to choose points along the instruction trace where mode switches should occur such that the total runtime of the trace, including the penalties incurred for switching modes, is minimized.

- We show the parallel processor array's *sequential performance* (read-after-write latency) relative to the performance of the sequential processor (CPU core) is a significant limitation on achievable speedup for a set of general-purpose applications. Note this is *not* the same as saying performance is limited by the serial portion of the computation [9].

- We find that latency and bandwidth between the two processors have comparatively minor effects on speedup.

In the case of a heterogeneous system using a GPU-like parallel processor, speedup is limited to only 12.7× for SPECfp 2000, 2.2× for SPECint 2000, and 2.5× for PhysicsBench [31]. When connecting the GPU using an off-chip PCI Express-like bus, SPECfp achieves 74%, SPECint 94%, and PhysicsBench 82% of the speedup achievable without latency and bandwidth limitations.

We present our processor model in Section 2, methodology in Section 3, analyze our results in Section 4, review previous limit studies in Section 5, and conclude in Section 6.

2 Modeling a Heterogeneous System

We model heterogeneous systems as having two processors with different characteristics (Figure 1). The *sequential processor* models a traditional processor core optimized for ILP, while the *parallel processor* models an array of cores for exploiting thread-level parallelism. The parallel processor models an array of low-cost cores by allowing parallelism, but with a longer register read-after-write latency than the sequential processor. The two processors may communicate over a communication channel whose latency is high and bandwidth is limited when the two processors are on separate chips. We assume that the processors are attached to ideal memory systems. Specifically, for dependencies between instructions within a given core (sequential or parallel) we assume store-to-load communication has the same latency as communication via registers (register read-after-write latency) on the same core. Thus, the effects of long latency memory access for the parallel core (assuming GPU-like fine-grained multi-threading to tolerate cache misses) is captured in the long read-to-write delay. The effects of caches and prefetching on the sequential processor core are captured by its relatively short read-to-write delay. We model a single-chip system (Figure 1(a)) with shared memory by only considering synchronization latency, potentially accomplished via shared memory (❶) and on-chip coherent caches [33]. We model a system with private memory (Figure 1(b)) by limiting the communication channel's bandwidth and imposing a latency when data needs to be copied across the link (❷) between the two processors.

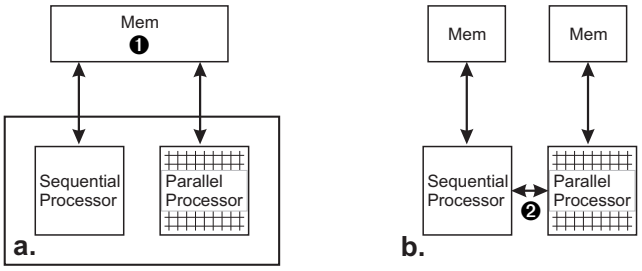

Figure 1. Conceptual Model of a Heterogeneous System. Two processors with different characteristics (a) may, or (b) may not share memory.

Section 2.1 and 2.2 describe each portion of our model in more detail. In Section 2.3 we describe our algorithm for partitioning and scheduling an instruction trace to optimize its runtime on the sequential and parallel processors.

2.1 Sequential Processor

We model the sequential processor as being able to execute one instruction per cycle (CPI of one). This simple model has the advantage of having predictable performance characteristics that make the optimal scheduling (Section 2.3) of work between sequential and parallel processors feasible. It preserves the

essential characteristic of high-ILP processors that a program is executed serially, while avoiding the modeling complexity of a more detailed model. Although this simple model does not capture the CPI effects of a sequential processor which exploits ILP, we are mainly interested in the relative speeds between the sequential and parallel processors. We account for sequential processor performance due to ILP by making the parallel processor relatively slower. In the remainder of this paper, all time periods are expressed in terms of the sequential processor's cycle time.

2.2 Parallel Processor

We model the parallel processor as a dataflow processor, where a data dependency takes multiple cycles to resolve. This dataflow model is driven by our trace based limit study methodology described in Section 3.2, which assumes perfectly predicted branches to uncover parallelism. Using a dataflow model, we avoid the requirement of partitioning instructions into threads, as done in the thread-programming model. This allows us to model the upper bound of parallelism for future programming models that may be more flexible than threads.

The parallel processor can execute multiple instructions in parallel, provided data dependencies are satisfied. Slower sequential performance of the parallel processor is modeled by increasing the latency from the beginning of an instruction's execution until the time its result is available for a dependent instruction. We do not limit the parallelism that can be used by the program, as we are interested in the amount of parallelism available in algorithms.

Our model can represent a variety of methods of building parallel hardware. In addition to an array of single-threaded cores, it can also model cores using fine-grain multithreading, like current GPUs. Note that modern GPUs from AMD [3] and NVIDIA [23, 24] provide a *scalar* multithreaded programming abstraction even though the underlying hardware is single-instruction, multiple data (SIMD). This execution mode has been called *single-instruction, multiple thread* (SIMT) [14].

In GPUs, fine-grain multithreading creates the illusion of a large amount of parallelism (>10,000s of threads) with low per-thread performance, although physically there is a lower amount of parallelism (100s of operations per cycle), high utilization of the ALUs, and frequent thread switching. GPUs use the large number of threads to "hide" register read-after-write latencies and memory access latencies by switching to a ready thread. From the perspective of the algorithm, a GPU appears as a highly-parallel, low-sequential-performance parallel processor.

To model current GPUs, we use a register read-after-write latency of 100 cycles. For example, current Nvidia GPUs have a read-after-write latency of 24 shader clocks [25] and a shader clock frequency of 1.3-1.5 GHz [23, 24]. The 100 cycle estimates includes the effect of instruction latency (24×), the difference between the shader clock and current CPU clock speeds (about 2×), and the ability of current CPUs to extract ILP—we assume an average IPC of 2 on current CPUs, resulting in another factor of 2×. We ignore factors such as SIMT branch divergence [8].

We note that the SPE cores on the Cell processor have comparable read-after-write latency to the more general purpose PPE core. However, the SPE cores are not optimized for control-flow intensive code [10] and thus may potentially suffer a higher "effective" read-after-write latency on some general purpose code (although quantifying such effects is beyond the scope of this work).

2.3 Heterogeneity

We model a heterogeneous system by allowing an algorithm to choose between executing on the sequential processor or parallel processor and to switch between them (which we refer to as a "mode switch"). We do not allow concurrent execution of both processors. This is a common paradigm, where a parallel section of work is spawned off to a co-processor while the main processor waits for the results. The runtime difference for optimal concurrent processing (e.g., as in the asymmetric multicore chips analysis given by Hill and Marty [16]) is no better than 2× compared to not allowing concurrency.

We schedule an instruction trace for alternating execution on the two processors. Execution of a trace on each type of core was described in Sections 2.1 and 2.2. For each mode switch, we impose a "mode switch cost", intuitively modeling synchronization time during which no useful work is performed. The mode switch cost is used to model communication latency and bandwidth as described in Sections 2.3.2 and 2.3.3, respectively. Next we describe our scheduling algorithm in more detail.

2.3.1 Scheduling Algorithm

Dynamic Programming is often applied to find optimal solutions to optimization problems. The paradigm requires that an optimal solution to a problem be recursively decomposable into optimal solutions of smaller

sub-problems, with the solutions to the sub-problems computed first and saved in a table to avoid recomputation [6].

In our dynamic programming algorithm, we aim to compute the set of mode switches (i.e., scheduling) of the given instruction trace that will minimize execution time, given the constraints of our model. We decompose the optimal solution to the whole trace into sub-problems that are optimal solutions to shorter traces with the same beginning, with the ultimate sub-problem being the trace with only the first instruction that can be trivially scheduled. We recursively define a solution to a longer trace by observing that a solution to a long trace is composed of a solution to a shorter sub-trace, followed by a decision on whether to perform a mode switch, followed by execution of the remaining instructions in the chosen mode.

The dynamic programming algorithm keeps a N×2 state table when given an input trace of N instructions. Each entry in the state table records the cost of an optimal scheduling for every sub-trace (N of them) and mode that was last used in those sub-traces (2 modes). At each step of the algorithm, a solution for the next sub-trace requires examining all possible locations of the previous mode switch to find the one that gives the best schedule. For each possible mode switch location, the corresponding entry of the state table is examined to retrieve the optimal solution for the sub-trace that executes all instructions up to that entry in the corresponding state (execution on the sequential, or the parallel core, respectively). This value is used to compute a candidate state table entry for the current step by adding the mode switch cost (if switching modes), and the cost to execute the remaining section of the trace from the candidate switch point up to the current instruction in the current mode (sequential, parallel). The lowest cost candidate over all earlier sub-traces is chosen for the current sub-trace.

The naive optimal algorithm described above runs in quadratic time with respect to the instruction trace length. For traces of millions of instructions in length, quadratic time is too slow. We make an approximation to enable the algorithm to run in time linear in the length of the instruction trace. Instead of looking back at all past instructions for each potential mode switch point, we only look back 30,000 instructions. The modified algorithm is no longer optimal. We mitigate this sub-optimality by first reordering instructions before scheduling. We observed that the amount of sub-optimality using this approach is insignificant.

To overcome the limitation of looking back only 30,000 instructions in our algorithm, we reorder instructions in dataflow order before scheduling. Dataflow order is the order in which instructions would execute if scheduled with our optimal scheduling algorithm. This linear-time preprocessing step exposes parallelism found anywhere in the instruction trace by grouping together instructions that can execute in parallel.

We remove instructions from the trace that do not depend on the result of any other instruction. Most of these instructions are dead code created by our method of exposing loop- and function-level parallelism, described in Section 3.2. Since dead code can execute in parallel, we remove these instructions to avoid having them inflate the amount of parallelism we observe. Across our benchmark set, 27% of instructions are removed by this mechanism. Note that the dead code we are removing is not necessarily dynamically dead [5], but rather overhead related to sequential execution of parallel code. The large number of instructions removed results from, for example, the expansion of x86 push and pop instructions (for register spills/fills) into a load or store micro-op (which we keep) and a stack-pointer update micro-op (which we do not keep).

2.3.2 Latency

We model the latency of migrating tasks between processors by imposing a constant runtime cost for each mode switch. This cost is intended to model the latency of spawning a task, as well as transferring of data between the processors. If the amount of data transferred is large relative to the bandwidth of the link between processors, this is not a good model for the cost of a mode switch. This model is reasonable when the mode switch is dominated by latency, for example in a heterogeneous multicore system where the memory hierarchy is shared (Figure 1(a)), so very little data needs to be copied between the processors.

As described in Section 2.3, our scheduling algorithm considers the cost of mode switches. A mode switch cost of zero would allow freely switching between modes, while a very high cost would constrain the scheduler to choose to run the entire trace on one processor or the other, whichever was faster.

2.3.3 Bandwidth

Bandwidth is a constraint that limits the rate that data can be transferred between processors in our model. Note that this does not apply to the processors' link to its memory (Figure 1), which we assume to be unconstrained. In our shared-memory model (Figure 1(a)) mode switches do not need to copy large amounts of data so only latency (Section 2.3.2) is a relevant constraint. In our private-memory model (Figure 1(b)),

bandwidth is consumed on the link connecting processors as a result of a mode switch.

If a data value is produced by an instruction in one processor and consumed by one or more instructions in the other processor, then that data value needs to be communicated to the other processor. A consequence of exceeding the imposed bandwidth limitation is the addition of idle computation cycles while an instruction waits for its required operand to be transferred. In our model, we assume opportunistic use of bandwidth, allowing communication of a value as soon as it is ready, in parallel with computation.

Each data value to be transferred is sent sequentially and occupies the communication channel for a specific amount of time. Data values can be sent any time after the instruction producing the value executes, but must arrive before the first instruction that consumes the value is executed. Data transfers are scheduled onto the communication channel using an "earliest deadline first" algorithm, which produces a scheduling with a minimum of added idle cycles.

Bandwidth constraints are applied by changing the amount of time each data value occupies on the communication channel. Communication latency is applied by setting the deadline for a value some number of cycles after the value is produced.

Computing the bandwidth requirements and idle cycles needed, and thus the cost to switch modes, requires a scheduling of the instruction trace, but the optimal instruction trace scheduling is affected by the cost of switching modes. We approximate the ideal behavior by iteratively performing scheduling using a constant mode switch overhead for each mode switch and then updating the average penalty due to bandwidth consumption across all mode switches, then using the new estimate of average switch cost as input into the scheduling algorithm, until convergence.

3 Simulation Infrastructure

We evaluate performance using micro-op traces extracted from execution of a set of x86-64 benchmarks on the PTLsim [18] simulator. Each micro-op trace was then scheduled using our scheduling algorithm for execution on the heterogeneous system.

3.1 Benchmark Set

We chose our benchmarks with a focus towards general-purpose computing. We used the reference workloads for SPECint and SPECfp 2000 v1.3.1 (23 benchmarks, except 253.perlbmk and 255.vortex which did not run in our simulation environment), PhysicsBench 2.0 [31] (8 benchmarks), SimpleScalar 3.0 [29] (used here as a benchmark), and four small microbenchmarks (described in Table 1).

Benchmark	*Description*
`linear`	Compute average of 9 input pixels for each output pixel. Each pixel is independent.
`sepia`	3×3 constant matrix multiply on each pixel's 3 components. Each pixel is independent.
`serial`	A long chain of dependent instructions, has parallelism approximately 1 (no parallelism).
`twophase`	Loops through two alternating phases, one with no parallelism, one with high parallelism. Needs to switch between processor types for high speedup.

Table 1. Microbenchmarks

We chose PhysicsBench because it contains both sequential and parallel phases in the benchmark, and would be a likely candidate to benefit from heterogeneity, as it would be unsatisfactory if both types of phases were constrained to one processor type [31].

Our SimpleScalar benchmark used the out-of-order processor simulator from SimpleScalar/PISA, running `go` from SPECint 95, compiled for PISA.

We used four microbenchmarks to observe behavior at extremes of parallelism, as shown in Table 1. `Linear` and `sepia` are highly parallel, `serial` is serial, and `twophase` has alternating highly parallel and serial phases.

Figure 2 shows the average parallelism present in our benchmark set. As expected, SPECfp has more parallelism (611) than SPECint (116) and PhysicsBench (83). `Linear` (4790) and `sepia` (6815) have the highest parallelism, while `serial` has essentially no parallelism.

3.2 Traces

Micro-op traces were collected from PTLsim running x86-64 benchmarks, compiled with gcc 4.1.2 -O2. Four microbenchmarks were run in their entirety, while the 32 real benchmarks were run through SimPoint [30] to choose representative sub-traces to analyze. Our traces are captured at the micro-op level, so in this paper instruction and micro-op are used interchangeably.

We used SimPoint to select simulation points of 10-million micro-ops in length from complete runs of benchmarks. As recommended [30], we allowed Sim-

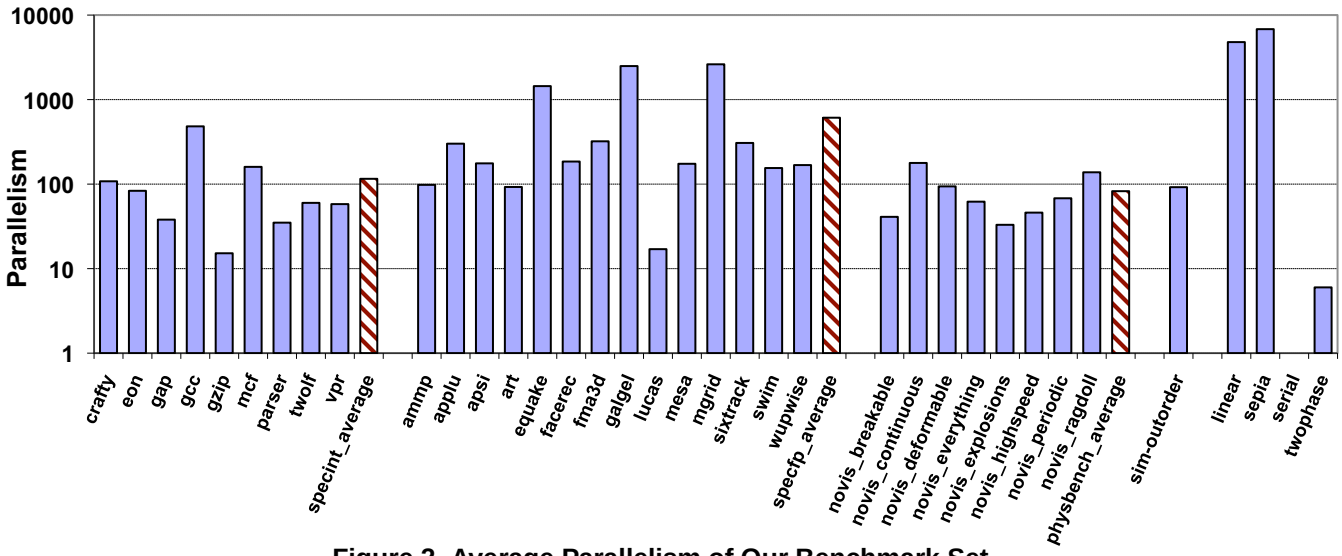

Figure 2. Average Parallelism of Our Benchmark Set

Point to decide how many simulation points should be used to approximate the entire benchmark run. We averaged 12.9 simulation points per benchmark. This is a significant savings over the complete benchmarks which were typically several hundred billion instructions long. The weighted average of the results over each set of SimPoint traces are presented for each benchmark.

We assume branches are correctly predicted. Many branches, like loops, can often be easily predicted or speculated or even restructured away during manual parallelization. As we are trying to evaluate the upper-bound of parallelism in an algorithm, we avoid limiting parallelism by not imposing the branch-handling characteristics of sequential machines. This is somewhat optimistic as true data-dependent branches would at least need be converted into speculation or predicated instructions.

Each trace is analyzed for true data dependencies. Register dependencies are recognized if an instruction consumes a value produced by an earlier instruction (read-after-write). Dependencies on values carried by the instruction pointer register are ignored, to avoid dependencies due to instruction-pointer-relative data addressing. Like earlier limit studies [17, 15], stack pointer register manipulations are ignored, to extract parallelism across function calls. Memory disambiguation is perfect: Dependencies are carried through memory only if an instruction loads a value from memory actually written by an earlier instruction.

It is also important to be able to extract loop-level parallelism and avoid serialization of loops through the loop induction variable. We implemented a generic solution to prevent this type of serialization. We identify instructions that produce result values that are statically known, which are instructions that have no input operands (e.g. load constant). We then repeatedly look for instructions dependent only on values that are statically known and mark the values they produce as statically known as well. We then remove dependencies on all statically-known values. This is similar to repeatedly applying constant folding and constant propagation optimizations [20] to the instruction trace. The dead code that results is removed as described in Section 2.3.

A loop induction variable [20] is often initialized with a constant (e.g. 0). Incrementing the induction variable by a constant depends only on the initialization value of the induction variable, so the incremented value is also statically known. Each subsequent increment is likewise statically known. This removes serialization caused by the loop control variable, but preserves genuine data dependencies between loop iterations, including loop induction variable updates that depend on a variable computed value.

4 Results

In this section, we present our analysis of our experimental results. First, we look at the speedup that can be achieved when adding a parallel co-processor to a sequential machine and show that the speedup is highly dependent on the parallel instruction latency. We define *parallel instruction latency* as the ratio of the read-after-write latency of the parallel cores (recall we assume a CPI of one for the sequential core). We then look at the effect of communication latency and bandwidth as parallel instruction latency is varied, and see that the effect is significant, but small.

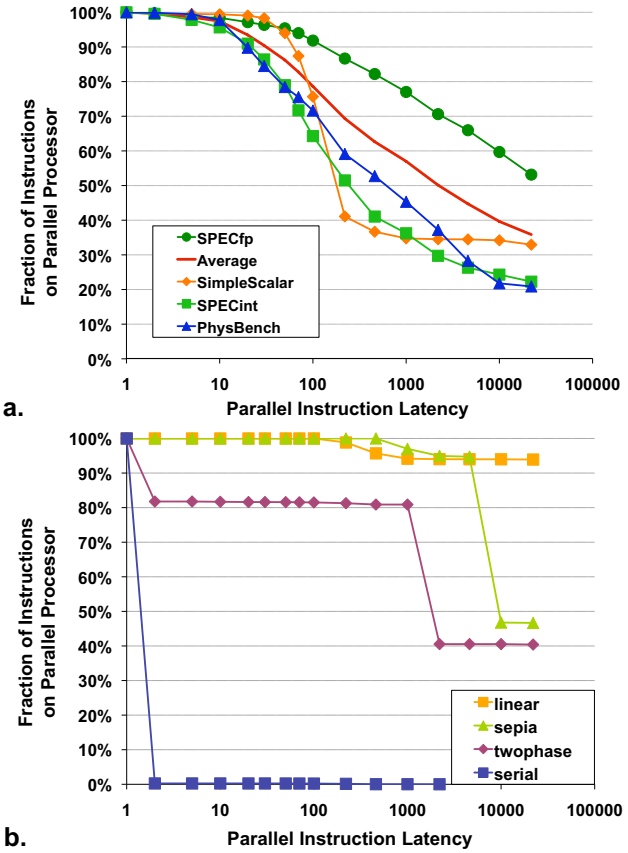

Figure 3. Proportion of Instructions Scheduled on Parallel Core. Real benchmarks (a), Microbenchmarks(b)

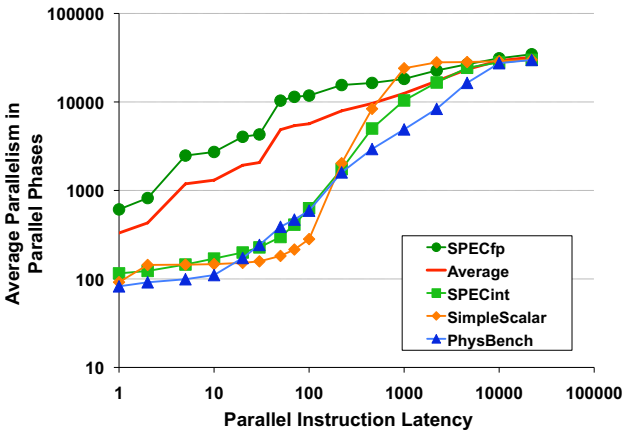

Figure 4. Parallelism on Parallel Processor

4.1 Why Heterogeneous?

Figures 3 and 4 give some intuition for the characteristics of the scheduling algorithm. Figure 4 shows the parallelism of the instructions that are scheduled to use the parallel processor when our workloads are scheduled for best performance. Figure 3(a) shows the proportion of instructions that are assigned to execute on the parallel processor. As the instruction latency increases, sections of the workload where the benefit of parallelism does not outweigh the cost of slower sequential performance become scheduled onto the sequential processor, raising the average parallelism of those portions that remain on the parallel processor, while reducing the proportion of instructions that are scheduled on the parallel processor. The instructions that are scheduled to run on the sequential processor receive no speedup, but scheduling more instructions on the parallel processor in an attempt to increase parallelism will only decrease speedup.

The microbenchmarks in Figure 3(b) show our scheduling algorithm works as expected. `Serial` has nearly no instructions scheduled for the parallel core. `Twophase` has about 18.5% of instructions in its serial component that are scheduled on the sequential processor leaving 81.5% on the parallel processor, while `sepia` and `linear` highly prefer the parallel processor.

We look at the potential speedup of adding a parallel processor to an existing sequential machine. Figures 5(a) and (b) show the speedup of our benchmarks for varying parallel instruction latency, as a speedup over a single sequential processor. Two plots for each benchmark group are shown: The solid plots show the speedup of a heterogeneous system where communication has no cost, while the dashed plot shows speedup when communication is very expensive. We focus on the solid plots in this section.

It can be observed from Figures 5(a) and (b) that as the instruction latency increases, there is a significant loss in the potential speedup provided by the extra parallel processor, becoming limited by the amount of parallelism available in the workload that can be extracted, as seen in Figure 3. Since our parallel processor model is somewhat optimistic, the speedups shown here should be regarded as an upper bound of what can be achieved.

With a parallel processor with GPU-like instruction latency of 100 cycles, SPECint would be limited to a speedup of 2.2×, SPECfp to 12.7×, PhysicsBench to 2.5×, with 64%, 92%, and 72% of instructions scheduled on the parallel processor, respectively. The speedup is much lower than the peak relative throughput of a GPU compared to a sequential CPU ($\approx 50\times$), which shows that if a GPU-like processor were used as the parallel processor in a heterogeneous system, the speedup on these workloads would be limited by the parallelism available in the workload, while still leaving much of the GPU hardware idle.

In contrast, for highly-parallel workloads, the speedups achieved at an instruction latency of 100 are

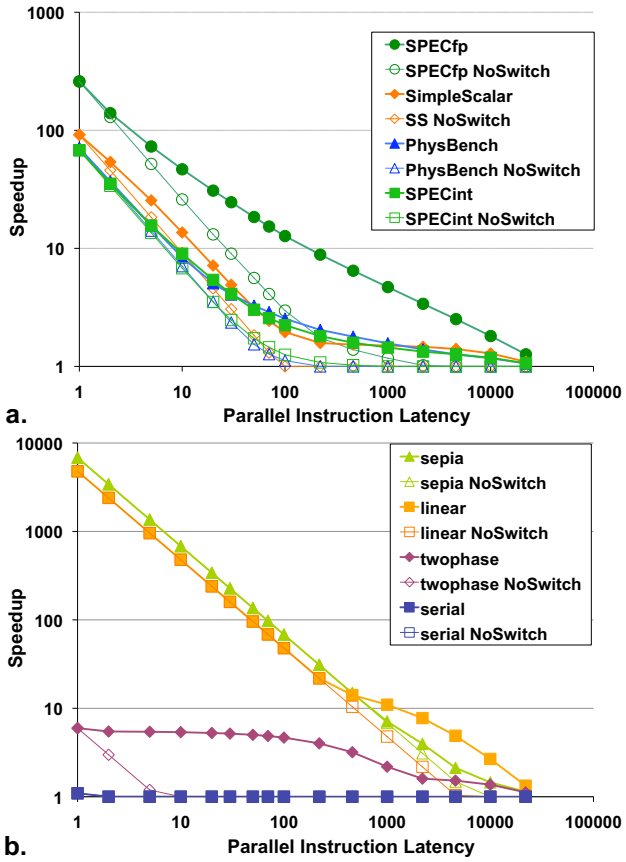

Figure 5. Speedup of Heterogeneous System: (a) Real benchmarks, (b) Microbenchmarks. Ideal communication (solid), communication forbidden (dashed, NoSwitch).

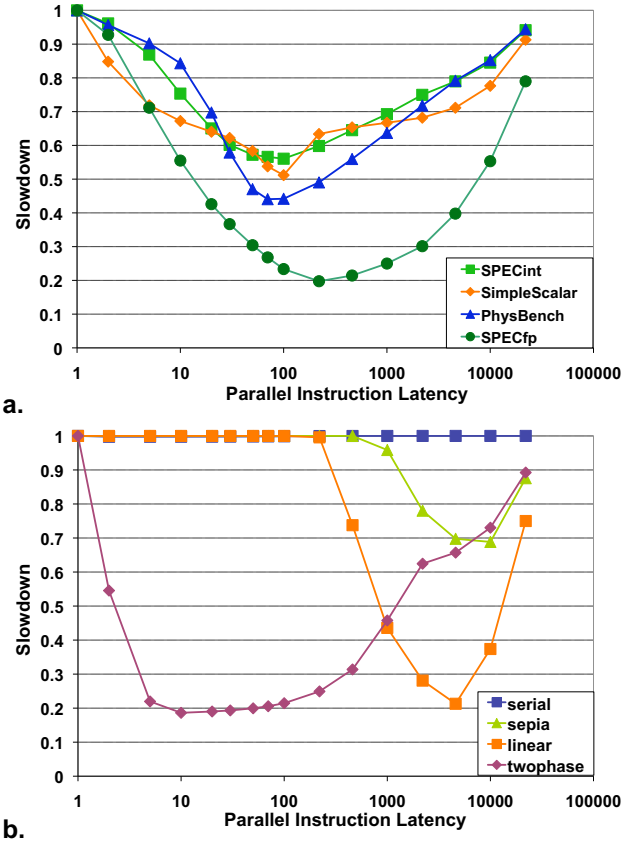

Figure 6. Slowdown of infinite communication cost (NoSwitch) compared to zero communication cost. Real benchmarks (a), Microbenchmarks (b).

similar to the peak throughput available in a GPU. The highly-parallel linear filter and sepia tone filter (Figure 5(b)) kernels have enough parallelism to achieve 50-70× speedup at an instruction latency of 100. A highly-serial workload (serial) does not benefit from the parallel processor.

Although current GPU compute solutions built with efficient low-complexity multi-threaded cores are sufficient to accelerate algorithms with large amounts of thread-level parallelism, general-purpose algorithms would be unable to utilize the large number of thread contexts provided by the GPU, while under-utilizing the arithmetic hardware available.

4.2 Communication

In this section, we evaluate the impact of communication latency and bandwidth on the potential speedup, comparing performance between the extreme cases where communication is unrestricted and communication is forbidden. The solid plots in Figure 5 show speedup when there are no limitations on communication, while the dashed plots (marked NoSwitch) has communication so expensive that the scheduler chooses to run the workload entirely on the sequential processor or parallel processor, never switching between them. Figures 6(a) and (b) show the ratio between the solid and dashed plots in Figures 5(a) and (b), respectively, to highlight the impact of communication. At both extremes of instruction latency, where the workload is mostly sequential or mostly parallel, communication has little impact. It is in the moderate range around 100-200 where communication potentially matters most.

The potential impact of expensive (latency and bandwidth) communication is significant. For example, at a GPU-like instruction latency of 100, SPECint achieves only 56%, SPECfp 23%, and PhysicsBench 44% of the performance of no communication, as can be seen in Figure 6(a). From our microbenchmark set (Figures 5(b) and 6(b)), `twophase` is particularly sensitive to communication costs, and gets no speedup for instruction latency above 10. We look at more realistic

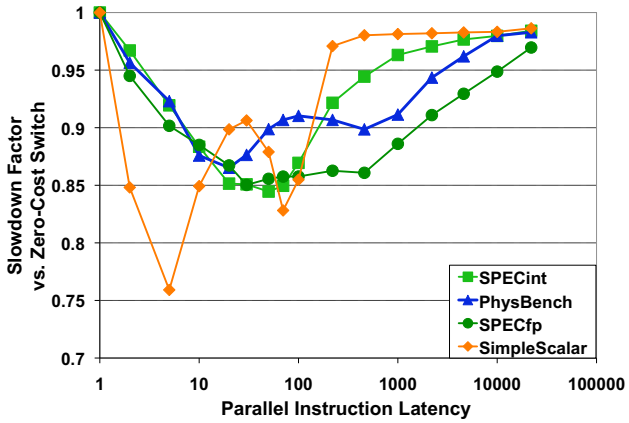

Figure 7. Slowdown due to 100,000 cycles of mode-switch latency. Real benchmarks.

constraints on latency and bandwidth in the following sections.

4.2.1 Latency

Poor parallel performance is often attributed to high communication latency [21]. Heterogeneous processing adds a new communication requirement—the communication channel between sequential and parallel processors (Figure 1). In this section, we measure the impact of the latency of this communication channel.

We model this latency by requiring that switching modes between the two processor types causes a fixed amount of idle computation time. In this section, we do not consider the bandwidth of the data that needs to be transferred. This model represents a heterogeneous system with shared memory (Figure 1(a)), where migrating a task does not involve data copying, but only involves a pipeline flush, notification to the other processor of work, and potentially flushing private caches if caches are not coherent.

Figure 7 shows the slowdown when we include 100,000 cycles of mode-switch latency in our performance model and scheduling, when compared to zero-latency mode switch.

The impact of imposing a delay for every mode switch has only a minor effect on runtime. Although Figure 6(a) suggested that the potential for performance loss due to latency is great, even when each mode switch costs 100,000 cycles (greater than 10us at current clock rates), most of the speedup remains. We can achieve ≈85% of the performance of a heterogeneous system with zero-cost communication. Stated another way, reducing latency between sequential and parallel cores might provide an average ≈ 18% performance improvement.

To gain further insight into the impact of mode

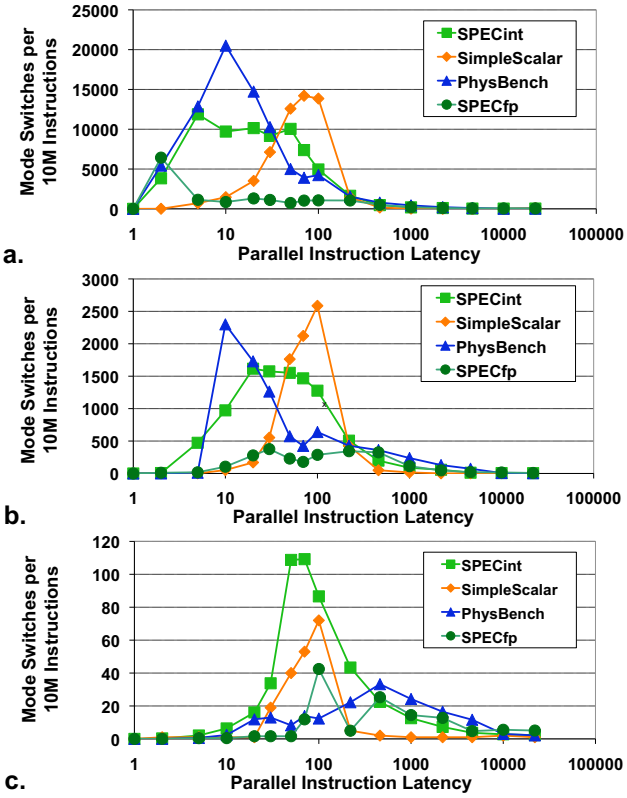

Figure 8. Mode switches as switch latency varies: (a) zero cycles, (b) 10 cycles, (c) 1000 cycles.

switch latency, Figure 8 illustrates the number of mode switches per 10 million instructions as we vary the cost of switching from zero to 1000 cycles. As the cost of a mode switch increases the number of mode switches decreases. Also, more mode switches occur at intermediate values of parallel instruction latency where the benefit of being able to use both types processors outweighs the cost of switching modes.

For systems with private memory (e.g. discrete GPU), data copying is required when migrating a task between processors at mode switches. We consider bandwidth constraints in the next section.

4.2.2 Bandwidth

In the previous section, we saw that high communication latency had only a minor effect on achievable performance. Here, we place a bandwidth constraint on the communication between processors. Data that needs to be communicated between processors is restricted to a maximum rate, and the processors are forced to wait if data is not available in time for an instruction to use it, as described in Section 2.3.3. We also include 1,000 cycles of latency as part of the model.

We first construct a model to represent PCI Express,

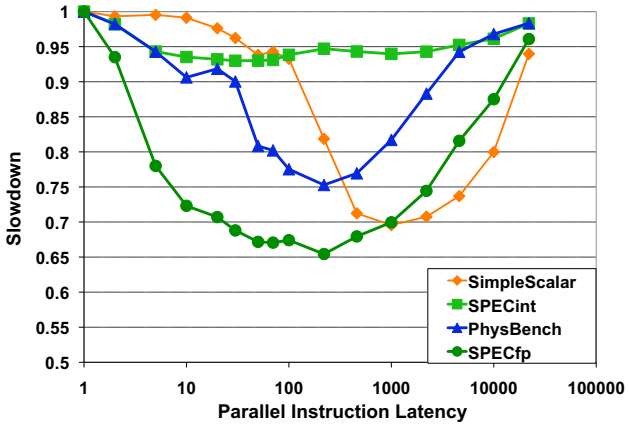

Figure 9. Slowdown due to a bandwidth constraint of 8 cycles per 32-bit value and 1,000 cycles latency, similar to PCI Express x16. Real benchmarks.

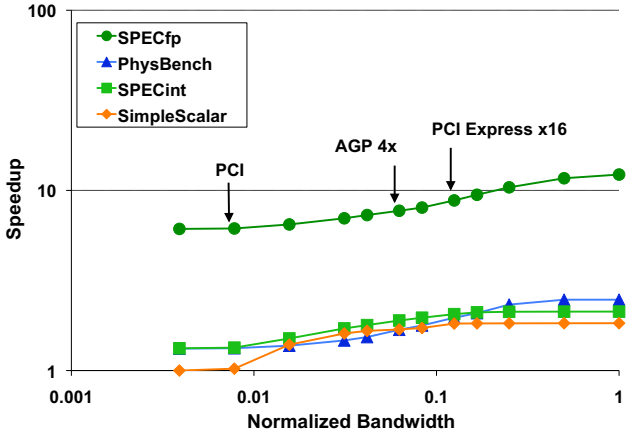

Figure 10. Speedup over sequential processor for varying bandwidth constraints. Real benchmarks.

as discrete GPUs are often attached to the system this way. PCI Express x16 has a peak bandwidth of 4GB/s and latency around 250ns [4]. Assuming current processors perform about 4 billion instructions per second on 32-bit data values, we can model PCI Express using a latency of about 1,000 cycles and bandwidth of 4 cycles per 32-bit value. Being somewhat pessimistic to account for overheads, we use a bandwidth of 8 cycles per 32-bit value (about 2GB/s).

Figure 9 shows the performance impact of restricting bandwidth to one 32-bit value every 8 clocks with 1,000 cycles of latency. Slowdown is worse than with 100,000 cycles of latency, but the benchmark set affected the most (SPECfp) can still achieve ≈67.4% of the ideal performance at a parallel instruction latency of 100. Stated another way, increasing bandwidth between sequential and parallel cores might provide an average 1.48× performance improvement for workloads like SPECfp. For workloads such as PhysicsBench and SPECint the potential benefits appear lower (1.33× and 1.07× potential speedup, respectively). Comparing latency (Figure 7) to bandwidth (Figure 9) constraints, SPECfp and PhysicsBench has more performance degradation than under a pure-latency constraint, but SPECint performs better, suggesting that SPECint is less sensitive to bandwidth.

The above plots suggest that a heterogeneous system attached without a potentially-expensive, low-latency, high-bandwidth communication channel can still achieve much of the potential speedup.

To further evaluate whether GPU-like systems could be usefully attached using even lower bandwidth interconnect, we measure the sensitivity of performance to bandwidth for instruction latency 100. Figure 10 shows the speedup for varying bandwidth. Bandwidth (x-axis) is normalized to 1 cycle per datum, equivalent to about 16GB/s in today's systems. Speedup (y-axis) is relative to the workload running on a sequential processor.

SPECfp and PhysicsBench have similar sensitivity to reduced bandwidth, while SPECint's speedup loss at low bandwidth is less significant (Figure 10). Although there is some loss of performance at PCI Express speeds (normalized bandwidth = 1/8), about half of the potential benefit of heterogeneity remains at PCI-like speeds (normalized bandwidth = 1/128). At PCI Express x16 speeds, SPECint can achieve 92%, SPECfp 69%, and PhysicsBench 78% of the speedup achievable without latency and bandwidth limitations.

As can be seen from the above data, heterogeneous systems can potentially provide significant performance improvements on a wide range of applications, even when system cost sensitivity demands high-latency, low-bandwidth interconnect. However, it also shows that applications are not entirely insensitive to latency and bandwidth, so high-performance systems will still need to worry about increasing bandwidth and lowering latency.

The lower sensitivity to latency than to bandwidth suggests that a shared-memory multicore heterogeneous system would be of benefit, as sharing a single memory system avoids data copying when migrating tasks between processors, leaving only synchronization latency. This could increase costs, as die size would increase, and the memory system would then need to support the needs of both sequential and parallel processors. A high-performance off-chip interconnect like PCI Express or HyperTransport may be a good compromise.

5 Related Work

There have been many limit studies on the amount of parallelism within sequential programs.

Wall [7] studies parallelism in SPEC92 under various limitations in branch prediction, register renaming, and memory disambiguation. Lam et al. [17] studies parallelism under branch prediction, condition dependence analysis, and multiple-fetch. Postiff et al. [15] perform a similar analysis on the SPEC95 suite of benchmarks. These studies showed that significant amounts of parallelism exist in typical applications under optimistic assumptions. These studies focused on extracting instruction-level parallelism on a single processor. As it becomes increasingly difficult to extract ILP out of a single processor, performance increases often comes from multicore systems.

As we move towards multicore systems, there are new constraints, such as communication latency, that are now applicable. Vachharajani et al. [21] studies speedup available on homogeneous multiprocessor systems. They use a greedy scheduling algorithm to assign instructions to cores. They also scale communication latency between cores in the array of cores and find that it is a significant limit on available parallelism.

In our study, we extend these analyses to heterogeneous systems, where there are two types of processors. Vachharajani examined the impact of communication between processors within a homogeneous processor array. We examine the impact of communication between a sequential processor and an array of cores. In our model, we roughly account for communication latency between cores within an array of cores by using higher instruction read-after-write latency.

Heterogeneous systems are interesting because they are commercially available [10, 25, 2] and, for GPU compute systems, can leverage the existing software ecosystem by using the traditional CPU as its sequential processor. They have also been shown to be more area and power efficient [16, 26, 27] than homogeneous multicore systems.

Hill and Marty [16] uses Amdahl's Law to show that there are limits to parallel speedup, and makes the case that when one must trade per-core performance for more cores, heterogeneous multiprocessor systems perform better than homogeneous ones because non-parallelizable fragments of code do not benefit from more cores, but do suffer when all cores are made slower to accommodate more cores. They indicate that more research should be done to explore "the scheduling and overhead challenges that Amdahl's model doesn't capture". Our work can be viewed as an attempt to further quantify the impact that these challenges present.

6 Conclusion

We conducted a limit study to analyze the behavior of a set of general purpose applications on a heterogeneous system consisting of a sequential processor and a parallel processor with higher instruction latency.

We showed that instruction read-after-write latency of the *parallel* processor was a significant factor in performance. In order to be useful for applications without copious amounts of parallelism, we believe that instruction read-after-write latencies of GPUs will need to decrease and thus GPUs can no longer rely exclusively on fine-grain multithreading to keep utilization high. We note that VLIW or superscalar issue combined with fine-grained multithreading [3, 19] do not inherently mitigate this read-after-write latency, though adding forwarding [12] might. Our data shows that latency and bandwidth of communication between the parallel cores and the sequential core, while significant factors, have comparatively minor effects on performance. Latency and bandwidth characteristics of PCI Express was sufficient to achieve most of the available performance.

Note that since our results are normalized to the sequential processor, our results scale as processor designs improve. As sequential processor performance improves in the future, the read-after-write latency of the parallel processor will also need to improve to match.

Manufacturers have and will likely continue to build single-chip heterogeneous multicore processors. The data presented in this paper may suggest the reasons for doing so are other than to obtain higher performance from reduced communication overheads on general purpose workloads. A subject for future work is evaluating whether such conclusions hold under more realistic evaluation scenarios (limited hardware parallelism, detailed simulations, real hardware) along with exploration of a wider set of applications (ideally including real workloads carefully tuned specifically for a tightly coupled single-chip heterogeneous system). As well, this work does not quantify the effect that increasing problem size [11] may have on the question of the benefits of heterogeneous (or asymmetric) multicore performance.

Acknowledgements

We thank Dean Tullsen, Bob Dreyer, Hong Wang, Wilson Fung and the anonymous reviewers for their comments on this work. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] AMD Inc. The future is fusion. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf, 2008.

[2] AMD Inc. *ATI Stream Computing User Guide*, 2009.

[3] AMD Inc. *R700-Family Instruction Set Architecture*, 2009.

[4] B. Holden. Latency Comparison Between HyperTransport and PCI-Express in Communications Systems. http://www.hypertransport.org/.

[5] J. A. Butts and G. Sohi. Dynamic Dead-Instruction Detection and Elimination. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–210, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 2001.

[7] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report 93/6, DEC WRL, 1993.

[8] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *To appear in: ACM Trans. Architec. Code Optim. (TACO)*, 2009.

[9] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc. vol. 30*, pages 483–485, 1967.

[10] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy. Introduction to the Cell multiprocessor. *IBM J. R&D*, 49(4/5):589–604, 2005.

[11] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of ACM*, 31(5):532–533, 1988.

[12] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *ASPLOS*, pages 308–318, 1994.

[13] E. Lindholm, M. J. Kligard, and H. P. Moreton. A user-programmable vertex engine. In *SIGGRAPH*, pages 149–158, 2001.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[15] M. A. Postiff, D. A. Greene, G. S. Tyson, T. N. Mudge. The Limits of Instruction Level Parallelism in SPEC95 Applications. *ACM SIGARCH Comp. Arch. News*, 27(1):31–34, 1999.

[16] M. D. Hill, M. R. Marty. Amdahlś Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[17] M. S. Lam, R. P. Wilson. Limits of control flow on parallelism. In *Proc. 19th Int'l Symp. on Computer Architecture*, pages 46–57, 1992.

[18] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software ISPASS*, 2007.

[19] J. Montrym and H. Moreton. The GeForce 6800. *IEEE Micro*, 25(2):41–51, 2005.

[20] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kauffman, 1997.

[21] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, D. Connors. Chip multi-processor scalability for single-threaded applications. *ACM SIGARCH Comp. Arch. News*, 33(4):44–53, 2005.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[23] NVIDIA Corp. GeForce 8 Series. http://www.nvidia.com/page/geforce8.html.

[24] NVIDIA Corp. GeForce GTX 280. http://www.nvidia.com/object/geforce_gtx_280.html.

[25] NVIDIA Corp. *CUDA Programming Guide*, 2.2 edition, 2009.

[26] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. 31st Int'l Symp. on Computer Architecture*, page 64, 2004.

[27] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. 36th IEEE/ACM Int'l Symp. on Microarchitecture*, page 81, 2003.

[28] S. Borkar. Thousand Core Chips — A Technology Perspective. In *Proc. 44th Annual Conf. on Design Automation*, pages 746–749, 2007.

[29] T. Austin, E. Larson, D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[30] T. Sherwood, E Perelman, G. Hamerly, B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems ASPLOS*, 2002.

[31] T. Y. Yeh, P. Faloutsos, S. J. Patel, G. Reinman. ParallAX: an architecture for real-time physics. In *Proc. 34th Int'l Symp. on Computer Architecture*, pages 232–243, 2007.

[32] J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS Proc. FJCC*, volume 26, pages 33–40, 1964.

[33] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT 2008)*, pages 52–61, Oct. 2008.

Our Many-core Benchmarks Do Not Use That Many Cores

Paul D. Bryan Jesse G. Beu Thomas M. Conte Paolo Faraboschi Daniel Ortega
Georgia Institute of Technology
{paul.bryan, jgbeu3, conte}@gatech.edu
HP Labs, Exascale Computing Lab
{paolo.faraboschi, daniel.ortega}@hp.com

Abstract

Increased processor performance can no longer be achieved through reduced feature size due to power density issues. As a result, high performance designs increasingly rely upon multiple processors in order to extract parallelism. Over the next few generations of designs, the core count of such machines is expected to increase, ultimately reaching thousands of processors. As the performance of multi-threaded programs has become increasingly important, the contemporary benchmarks of choice are PARSEC and Splash-2. Both of these suites offer great scalability from the program's perspective, where approximate ideal speedup can be obtained up to a certain processor threshold.

PARSEC and Splash-2 are attractive to many in the community because they offer programs that are highly scalable in the evaluation of large CMP designs. Both benchmark suites offer the *inherent* parallelization characteristics present within their programs as a metric for scalability. However, the assessment of program characteristics including system-level effects may skew overall program scalability results. Because speedup computed through Amdahl's law is extremely sensitive to the serial fraction of any program on the upper end, a minute change in the ratio of serial to parallel work may have dramatic effects upon the maximum obtainable speedup. The operating system, shared libraries, and thread synchronization incur serializing costs upon the overall program execution, which add to the serial code sections, and limit the scalability of these workloads to even a small number of cores. In this study, we show how these effects tragically limit multi-threaded performance, which must be overcome in order for new systems with a large number of cores to be viable.

1. Introduction

Power consumption and heat dissipation have become limiting factors to increased performance of single-processor designs. These constraints have limited the viability of feature size reduction as a technique to produce faster processors. In order to mitigate these effects, current high performance designs have shifted to include multiple processors on chip to compensate for decreased processor frequencies (and increased cycle times). The number of cores is expected to climb with each new design cycle, with the ultimate goal being thousands of processors.

In the past, SPEC [1] benchmarks were sufficient in the evaluation of single processor designs and multi-processor designs including a modest number of cores. As the number of processors has increased, it has become exceedingly difficult to evaluate chip multiprocessors in the context of throughput computing (where separate workloads are executed on individual processors). Typically, this would involve the arbitrary selection of workloads equal to the number of processors, where the total number of "CMP workloads" would be equal to ${}_{B}C_{P}$, which is the combination of B benchmarks taken P at a time (where B is the number of workloads contained in the suite and P is the number of processors in the design). Many problems stem from this type of evaluation and are described in [2].

As the number of processors have increased, PARSEC [3] and Splash-2 [4] have become the benchmark suites of choice. These multi-threaded workloads not only test the characteristics associated with synchronization and coherence, but also provide a standard substrate for individuals to compare their results (as well as mitigating the problems previously mentioned with throughput oriented workloads). Each of these benchmark suites is notably attractive because they advertise highly scalable workloads up to a specified processor threshold. PARSEC v1.0 reports indicate that their workloads approximately scale with the ideal up until 16 processors [5], and Splash-2 reports indicate scalability up until 64 processors [4]. Each of these reports determines the scalability of the program in isolation, which include program characteristics internally. Any external effects associated with the operating system and shared system libraries (which implement synchronization) are excluded from the analysis.

Multi-threaded programs may be broken down into two separate pieces, corresponding to their parallel and serial regions. According to Amdahl's law parallel program speedup is ultimately limited by the serial code sections of an application. Represented by the fraction of the enhanced ($F_{enhanced}$), inherent parallelism indicates the percentage of the program that is fully parallelizable. The remaining serial portion of the program may be calculated by $1\text{-}F_{enhanced}$. Once

the fraction of the enhanced is known, the maximum achievable speedup obtainable by parallelization may be calculated. Figure 1 shows the maximum obtainable speedup for fraction of the enhanced ($F_{enhanced}$) values between 0% and 99%. Assuming *perfect communication* and *no parallelization overhead*, a program must have at least 93.75% fully parallel code in order to obtain 16x speedup, and at least 99% for 100x speedup. Thus, for a real multi-processor system with 16 processors to obtain ideal speedup and utilize all processors, the target program must contain at least 93.75% parallel code. The underlying system cannot diminish the observed parallelization below that percentage (due to synchronization, additional coherence overhead, additional interconnect communication delay, shared libraries, OS scheduler events, load balancing, etc.). As the number of processors increase, multi-threaded program system performance will increasingly rely upon efficient thread execution and communication.

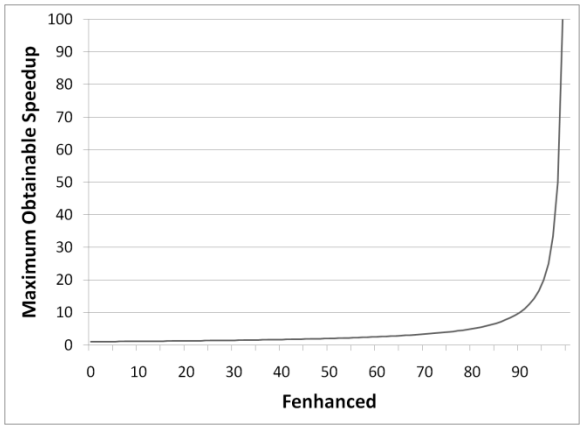

Figure 1: Maximum Speedup (0-99%)

In this study, we evaluate the observed speedups of various PARSEC and Splash-2 workloads. In order to understand the limitations of thread synchronization upon system performance, we characterize thread behaviors for an abstract machine model that incorporates operating system effects. Section 2 briefly describes basic thread synchronization mechanics studied in this work. The experimental framework is described in Section 3. Section 4 describes the characterization of synchronization events within the tested workloads and discusses its impact upon scalability. Section 5 discusses the measured speedups of workloads at varying workload/processor configurations and describes inherent obstacles to large multi-processor systems. Section 6 discusses related work followed by conclusions in Section 7.

2. Thread Synchronization

In multi-core systems, multi-threaded programs result in many concurrently executing tasks, or threads. In the POSIX implementation, each task resides within the context of a single process, allowing certain process resources to be shared, but each task to have separate program counters, registers, and local stack. Multi-threaded programs may spawn more threads than processors in order to avoid blocking while threads are waiting for a resource. However, too many active threads may hurt system performance due to OS context switching.

Thread interaction is governed by the use of synchronization primitives. The correct use of synchronization can prevent race conditions and other unexpected program behavior. Synchronization primitives may be implemented in the kernel or within user level libraries, and usually incorporate atomic operations to perform bus locking and cache coherency management. Atomic operations are a set of operations that are combined so the system perceives it as a single operation that succeeds or fails. Upon failure the effects of all operations are undone so that system state remains invariant. Examples of atomic operations include test-and-set, fetch-and-add, and compare-and-swap.

After a thread has been spawned it will run independent from the parent program unless dictated by synchronization. Shared resources that could be dynamically accessed simultaneously by multiple threads are often placed in critical sections to guarantee mutual exclusivity, typically guarded by locks or semaphores. If no threads are using a particular shared resource, then a thread may enter the critical section and is in a running state. Otherwise, the thread transitions to a blocked state while waiting for critical section access. Previous techniques have been proposed to mitigate the serializing effects due to locks [6-8].

System execution of multiple threads introduces potential non-deterministic effects due to the complex interactions between the scheduler, memory hierarchy and interconnect. Threads competing for a lock request resources from the memory hierarchy, and interconnect if the request misses. The first thread to obtain a cache line in exclusive access will win the lock and continue execution. Waiting threads will then obtain the lock based upon physical machine characteristics. Threads may also be preempted by other processes and/or threads which may change the observed synchronization events and program ordering. Non-deterministic effects may cause a program to behave differently over multiple runs, making debugging difficult.

The concurrent execution of threads may yield significant speedup compared to its single-threaded cousin if: 1) sufficient parallel code regions exist in the program, and 2) they can be executed efficiently with minimal synchronization bottlenecks and other system overheads. In this work, it is shown that

synchronization is an important factor that must be considered when evaluating workload scalability.

3. Simulation Environment

In this study, the linux GNU C library (glibc v2.8) was instrumented to profile synchronization events. Each type of synchronization provided by the Native POSIX Threading Library (pthreads) was instrumented to provide detailed information regarding thread behaviors. The pthreads library API provides for the following types of synchronization: mutexes, read-write locks, barriers, condition variables, thread joining, and semaphores. Read-write locks are multiple readers, single writer locks. Multiple threads can acquire a lock for reading purposes, but exclusive (write) accesses are serialized similar to mutex-based critical sections.

Using the instrumented libc, a number of workloads were simulated using HP Labs' COTSon simulator [9]. COTSon is a system-level simulator that models execution holistically, including: peripheral devices, disk I/O, network devices, and the operating system. Inclusion of the operating system allows for system calls and system interaction to be measured. Internally, COTSon uses the AMD's SimNow simulator [10] to provide fast native emulation of instructions, which are fed as a trace to COTSon timers for detailed measurement. For this work, multi-threaded workloads were simulated on top a 64-bit Debian operating system (kernel 2.6.26) using the instrumented libc.

In order to reduce context switching effects from other running processes, a stripped down Debian operating system was utilized. All non-necessary daemons were killed, and the X server was removed. To ensure that comparisons would be consistent across processor configurations, each workload was executed from the same operating system checkpoint image.

In order to classify synchronization events and their effects on thread behaviors, workloads from the Splash-2 [4] and PARSEC v1.0 [3] benchmark suites were simulated. COTSon provided timestamps for pthread events in order to determine detailed timing information for each type of synchronization event. For locks and mutexes, times were recorded for resource request, acquire, and release. Additionally, wait and release times were also recorded for thread joins, barriers, and conditional variables. The thread id and internal pthread data structure addresses were recorded to isolate individual thread behaviors and interaction based upon specific synchronization instances. Event timestamps were used to calculate the time that an individual thread spent within library synchronization calls. Individual thread wait times were determined as the cumulative difference between request/acquire and acquire/release events (in the case of locks) and wait/release events for joins, barriers, and condition variables. Although code instrumentation may affect the timing behavior of any program, the instrumentation of glibc was very lightweight (an event id was added to a buffer) and should not significantly alter the actual program behavior. Since our simulation environment was based on a 64-bit linux OS, we verified measurements on real systems containing eight processors for both the AMD Opteron and Intel Xeon architectures. Collected measurements for both architectures yielded speedups consistent with those in our simulated environment. Larger input sets were also executed on real systems and exhibited performance results similar to the smaller input sets.

Data were collected using a COTSon timer that functionally executed instructions in a single time unit. The use of a functional simulator has the same effect as if a cycle-accurate model were used with a perfect cache, branch predictor, TLB, pipeline (1-issue, in-order), interconnect, and coherence. The purpose of using this configuration was to approximate a lower bound of synchronization overheads. Assuming perfect communication across the cache hierarchy and interconnect, program performance is bounded by the serial code regions, OS interaction, and synchronization overheads. Synchronization overheads are based upon the ordering imposed by the benchmark algorithm as well as costs incurred within the pthreads library. The use of functional models has been incorporated in other studies to assess inherent program parallelization [3, 4]. Previous work has shown that Splash-2 and PARSEC have the inherent parallelization necessary to scale. However, when including the operating system and threading library effects within these measurements, application speedup may differ significantly from the ideal case.

4. Synchronization Characterization

PARSEC and Splash-2 workloads discussed in this section were simulated for 1, 2, 4, 8, 16, and 32 processors. At each processor configuration, the number of threads equaled the number of processors. From these data, average synchronization wait times were calculated (excluding one thread configurations because a serial program should not wait due to synchronization). All experimental workloads implement parallelization using either pthreads or OpenMP. Because the linux version of gcc internally uses POSIX threads by default to implement OpenMP pragmas, the instrumentation of pthreads was sufficient to capture the behavior for both types of workloads. In this study, PARSEC v1.0 benchmarks were simulated using the simlarge input set. Splash-2 workloads were also simulated using the default input size. Vips was excluded because it segfaulted for the simlarge input. FFT was also excluded from Splash-2 because its execution time was too small to be reliably measured by the time command.

Figure 2 shows the average time threads spent waiting as a percentage of program execution for the studied PARSEC and Splash-2 workloads. For each of these workloads, wait times have been decomposed into their synchronization constituents. These figures show the average percentage time that threads spent waiting for condition variables, barriers, and mutexes (including read-write locks) for all tested workload/processor configurations.

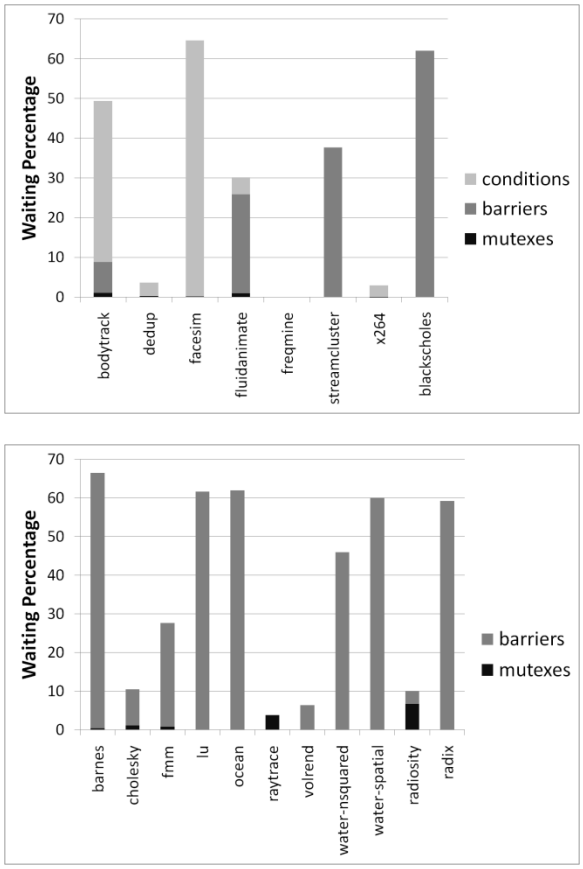

Figure 2: Thread Waiting Decomposition for PARSEC and Splash-2

For PARSEC, certain workloads spent considerable time waiting on synchronization events. Bodytrack waited for 49%, blackscholes for 62%, facesim for 65%, fluidanimate for 29%, and streamcluster for 38%. Others such as dedup, freqmine, and x264 all wait for less than 4%. Interestingly, condition variables and barriers dominate mutexes as the cause of waiting threads. Condition variables are barriers that can be selectively applied to individual threads. Once a thread reaches a condition variable, it must wait until that condition becomes true. Upon receiving the wakeup signal, thread execution will continue. For all workloads, mutexes account for less than 0.34% of thread execution. Barriers and condition variables account for 8.7% and 14.3%, respectively. On average, 17.3% of PARSEC workload execution was consumed on synchronization.

For Splash-2 workloads, the average synchronization overheads for mutexes and barriers are 1.3% and 34.2%, respectively. No condition variable activity was measured for these workloads. Similar to PARSEC, mutexes had minimal impact upon overall thread wait times. On average, Splash-2 wait times consumed 35.4% of program execution. In part, this is due to the short execution times of Splash-2 workloads. On average, all Splash-2 permutations executed in 0.68 seconds within the simulated operating system. The short execution times of Splash-2 workloads may have issues associated with constant timeslice interruption. However, these workloads are over a decade old and may be outdated for contemporary system evaluation. But, even when these workloads were relatively new, scalability issues were observed [11] for NUMA architectures.

Wait times for mutex synchronization was extremely low, implying threads can regularly acquire locks uncontested. Similar behavior for the low contention rate of mutexes in cycle-accurate simulation environments has been observed in [6, 7]. Instrumented barriers are extremely costly even for our simulated abstract machine because all system threads must wait for the slowest thread to reach a specified execution point. If threads have common algorithmic tasks and similar performance, then the slack time between the highest and lowest performing thread should be minimal. Threads with dissimilar tasks or whose performance varies greatly will incur greater barrier costs.

The scheduler may also impact associated barrier overhead because it may preempt thread execution in lieu of another system process. Assuming homogenous execution among the remaining threads, the evicted process will then become the slowest thread, and program execution cannot continue until it is both rescheduled and reaches the barrier. Here, the scheduler overheads provide a non-intuitive trade-off. If the OS time slice is too short, then threads could be preempted frequently by other system processes. Increasing the time slice interval could reduce preemption, but could also increase the penalties of preemption when it occurs. It is currently unclear which scheme would most benefit the performance of barrier execution and is left for future research.

A detailed decomposition of thread behaviors at the varying thread counts is shown in Figure 3 and Figure 4 for interesting workloads with the highest average wait times. For these workloads, the cost of synchronization increases with the number of threads. Bodytrack and facesim both result in increased wait times for conditional variables. Fluidanimate, streamcluster, barnes, lu, ocean, and water-spatial all result in greater barrier costs at higher thread counts. Wait times for bodytrack are in contrast with the other workloads (which exhibited decreased barrier wait

times as the number of threads increased) because one thread spent the majority of its time waiting for all other worker threads to complete. The addition of threads in this workload caused the overall wait times to decrease because it was averaged over more running threads.

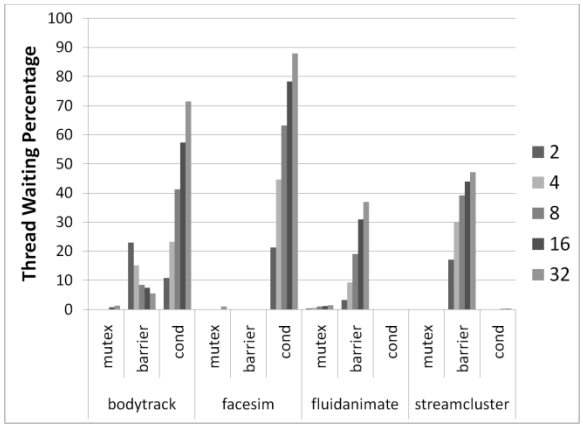

Figure 3: Thread Waiting vs. Thread Count for a subset of PARSEC

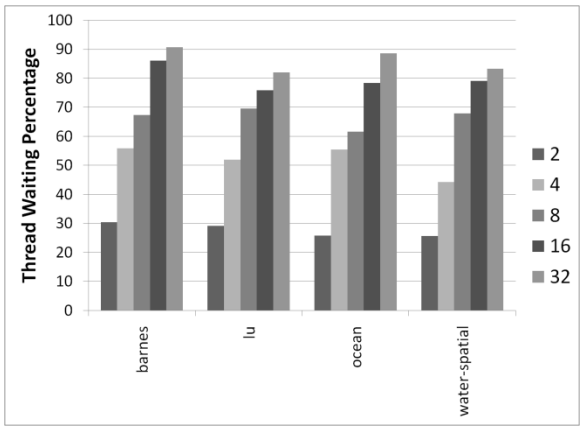

Figure 4: Barrier Overhead vs. Thread Count for a subset of Splash-2

Measured wait times were dependent upon the time spent within synchronization library functions, other miscellaneous system calls, and thread execution. As the numbers of threads are increased, highly parallel workloads that scale well have significantly less execution time. If the synchronization costs are fixed, then this would result in linear wait percentage increases. Insufficient parallel code or poor synchronization performance may cause workloads to scale poorly. In either case, the addition of threads that synchronize over the same shared structures (via condition variables or barriers) increases the overhead of the pthreads library. Such overheads, however minimal, may have dramatic impacts upon the observable scalability that can be extracted from a multi-threaded program at high thread counts, and are discussed in Section 6. In this study, many programs contain inflection points where the addition of threads will no longer help performance (or worse, hurt performance). A more detailed discussion of simulation times is discussed in Section 5.

The synchronization penalties of mutexes among the tested workloads were numerous and light. Although the specific behavior is workload dependent, the wait times associated with mutexes generally were very small and contributed little to overall thread waiting. Additionally, dynamic instances of barriers and condition variables were much less frequent than mutexes but had dramatically higher overhead.

5. Observed Speedup

Simulated speedups of each program were compared against the ideal case and were based upon the "real" execution from the perspective of the simulated OS. Execution times for each of the different processor counts were compared against the single-threaded case to derive the parallelizable fraction of execution. This resulted in five $F_{enhanced}$ ratios for each workload: one calculated by the observed speedup between 1 thread and 2 threads, between 1 thread and 4 threads, between 1 thread and 8 threads, etc. For each of the five computed fractions, the maximum was selected because it revealed the best parallelism that was observed at the system level. Measured fractions lower than the maximum indicate overheads that inhibited concurrency. While the program in isolation may inherently contain a fraction higher than the observed, it is important to include the serializing system-level effects in the measurement. Derived $F_{enhanced}$ values for the PARSEC and Splash-2 benchmark suites are shown in Table 1 and Table 2 and indicate the maximum projected speedup for the largest $F_{enhanced}$ measurement.

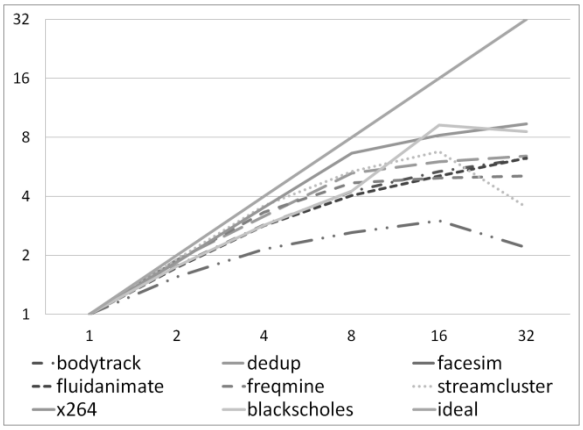

Figure 5: PARSEC Threads vs. Speedup

Figure 5 and Figure 6 show measured speedups of simulated workloads as the number of threads and processors are increased logarithmically. Surprisingly, even if a perfect processor model is incorporated, no workload is able to scale past 10x once system-level effects are incorporated. Associated speedup for the

two benchmark suites were dramatically impacted by the wait times of synchronization. Splash-2 suffered higher synchronization event wait times than PARSEC, which accounts for the lower obtained speedup values for these workloads.

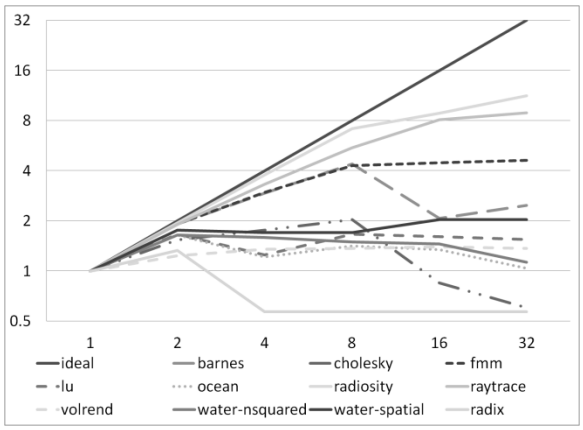

Figure 6: Splash-2 Speedup vs. Processor Count

Benchmark	$F_{enhanced}$	Projected Speedup
bodytrack	0.8740	6.52x
dedup	0.9481	12.27x
facesim	0.7127	3.23x
fluidanimate	0.8676	6.27x
freqmine	0.9331	10.41x
streamcluster	0.9615	14.59x
x264	0.9709	16.83x
blackscholes	0.9512	12.73x

Table 1: PARSEC Parallelizable Fraction

Benchmark	$F_{enhanced}$	Projected Speedup
barnes	0.9533	13.08x
cholesky	0.6949	3.06x
fmm	0.9600	14.28x
lu	0.8000	4.44x
ocean	0.7843	4.16x
radiosity	0.9835	21.20x
raytrace	0.9520	12.87x
volrend	0.3864	1.59x
water-nsquared	0.7843	4.16x
water-spatial	0.8627	6.08x
radix	0.5000	1.85x

Table 2: Splash-2 Parallelizable Fraction

At 32 processors, x264 had the highest speedup of 9.38x and facesim had the smallest speedup of 2.13x. On average, workloads with 2 processors had a speedup of 1.79. Increasing the processor counts to 4, 8, 16, and 32 had average speedups of 3.06x, 4.69x, 5.62x, and 5.58x, respectively. The performance of all workloads increased up until 16 processors. After this point, the simulation times for certain workloads actually increased. When comparing 16 and 32 processors, the program execution time of facesim increased from 13.8s to 14.2s, and streamcluster increased from 3.26s to 6.29s. Splash-2 workloads exhibit similar behavior and are included in Figure 6. Radiosity, raytrace, fmm, and barnes all scaled well relative to the ideal case up until two processors. Beyond two processors, only radiosity and raytrace scaled up to 8 before saturating. Cholesky suffered slowdowns at 16 and 32 processors, and radix suffered slowdowns beyond 2. An average speedup factor for both suites was less than 6x at 32 processors.

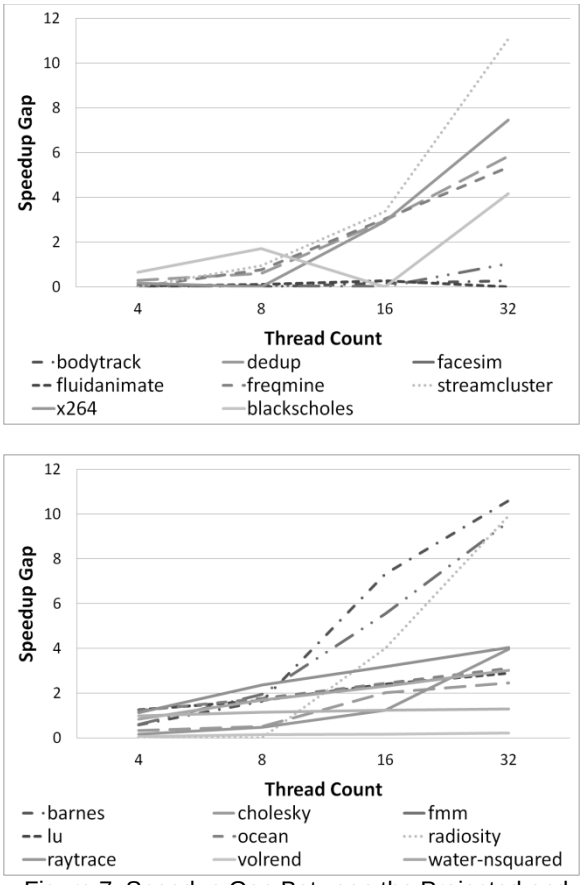

Figure 7: Speedup Gap Between the Projected and Measured

Figure 7 shows the gap between the measured speedup of benchmark performance relative to the predicted speedup from calculated $F_{enhanced}$ values. Parallelization fractions represent the maximum amount of observed parallelism from the simulated workloads, including all system overheads. As the number of processors and threads increase, the difference between the projected speedup and the simulated speedup is directly attributable to additional system overheads. The parallelization projected by the PARSEC and Splash-2 workloads did not yield sufficient scalability for even a small number of cores. Up to 8 processors PARSEC had adequate scaling, but saturated at 16. Most workloads obtained little gain

beyond 16 processors, and some exhibited degradation. Reported speedups for PARSEC workloads indicate that inherent program parallelism will scale up to 16 processors. When OS and synchronization overheads are taken into consideration, ideal scaling was not seen. Furthermore, as designs increasingly incorporate 32 processors, scalability for these workloads will become a much greater issue. For Splash-2, only two workloads were resistant to the OS and synchronization effects. The remaining workloads began to show saturation or performance degradation after two processors.

6. Putting It All Together

In order to more adequately understand the system bottlenecks associated with workload scalability, additional experiments were conducted within the simulated OS for each workload/processor combination using OProfile. On average, 33% of program execution of PARSEC workloads and 52% of Splash-2 workloads were spent within the linux kernel. This does not imply inefficiencies within the OS, but rather is an artifact of the interaction between waiting threads and the scheduler. Due to potentially high wait times associated with thread execution (shown in Section 4), threads often cannot make forward progress due to synchronization stalls. (Also, newly spawned threads that have yet to be given any useful work are started in the idle state.) If a thread is stalled, and thus not performing any useful work, it will eventually be removed from the run queue by the scheduler, and sent to a wait queue.

Decomposition of the time spent within the linux kernel shows that approximately 97% of the time is spent within the *default_idle* kernel function and 3% is spent for all other OS services (e.g. memory mapping, scheduling, I/O, filesystem bookkeeping, etc.). Outliers from this type of behavior are dedup and streamcluster from PARSEC, and water-nsquared from Splash-2. Dedup and water-nsquared spend 12% and 9% of their execution within the OS on non-idle services and can be attributed to poor paging behavior. Water-nsquared also exhibits behavior similar to dedup. Streamcluster spends approximately 31.2% of its execution with the OS on non-idle services due to load imbalancing issues. The behaviors of barriers within streamcluster indicate that approximately half of the threads never wait for barriers, while the other half spend considerable time waiting.

In general, time spent within shared libraries was small, however significant times were measured for the standard C++/C, math, and pthread shared libraries and varied between less than 1% and 5% (excluding thread wait times that were measured within the default_idle kernel function). When excluding the default_idle kernel function from the OS measurements, the OS component of workload execution varied between less than 1% and 31%.

Previous work [12] has demonstrated the impact that system calls and kernel code may have upon system performance, and advocate its inclusion within single-threaded simulation. We are advocating that synchronization, system calls and OS behavior *must* be considered when evaluating possible speedups that could be obtained from multi-threaded workloads. Future systems containing hundreds or even thousands of cores will increasingly rely upon massively parallel code in order to obtain speedup. In these systems, effective processor utilization is dependent upon input workloads that converge upon 100% parallelization. For example, if a system can execute 99.99% of a program in parallel, then the maximum attainable speedup is 10000x. Decreasing the enhanced fraction by 0.09 to 99.9% reduces the maximum obtainable speedup by 9000x!

The parallelization requirements of contemporary workload/system pairs are modest in comparison to that of the hypothetical 1000 core machine. In our experiments, measured $F_{enhanced}$ values indicate that at least three of the benchmark inputs contained the inherent parallelism necessary to scale to 16 processors, and two benchmark inputs contained the parallelism to scale above 20x. Yet, no input for any benchmark was able to obtain speedup past 11x when considering the additional system overheads included in our model.

7. Related Work

When discussing parallel benchmarking sets, various methods have been proposed to discuss the inherent parallelization characteristics. Woo et al [4] measure speedup for Splash-2 workloads using perfect caches and communication. All instructions executed in their environment complete in one cycle. The authors note that non-deterministic behaviors of programs make it difficult to compare data when architectural parameters are varied because the execution path may change. Bienia et al [3] measure the inherent program concurrency based upon the number of instructions executed in the parallel and serial regions of code. Delays due to blocking and load imbalance are not studied because they focus on the fundamental program characteristics. Our characterization differs in that we consider the operating system, shared system libraries, and detailed thread synchronization in our analysis.

8. Conclusion

As more processors are added to next generation designs, it is important to identify the capabilities of application parallelization. If a new system has a large number of cores, then programs must be able to adequately leverage its resources in order to be

effective. For the studied workloads, parallel execution was insufficient to scale along with the ideal case. This is in contrast to other work which describes the identified parallelism found within the workloads in isolation. However, discrepancies between the two can be explained by synchronization, the OS and other shared system libraries that are measured within our infrastructure. Synchronization incurs significant overheads which must be measured to obtain realistic performance projections as the number of cores scale. The effect of pthread calls causes many threads to block, thereby increasing serial sections of the multi-threaded program and decreasing $F_{enhanced}$. Furthermore, additional OS overheads increase serial code sections and limit parallelization opportunities. As more processors are added to commodity systems, the OS and shared libraries will play an increasingly important role in the available parallelism that can be achieved in a multi-threaded workload.

9. References

[1] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1-17, 2006.

[2] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "FAME: FAirly MEasuring Multithreaded Architectures," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, 2007, pp. 305-316.

[3] C. Bienia, S. Kumar, and L. Kai, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008, pp. 47-56.

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture* S. Margherita Ligure, Italy: ACM, 1995.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* Toronto, Ontario, Canada: ACM, 2008.[6] R. Rajwar and J. R. Goodman, "Speculative lock elision: enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture* Austin, Texas: IEEE Computer Society, 2001.

[7] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* Washington, DC, USA: ACM, 2009.

[8] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems* San Jose, California: ACM, 2002.

[9] E. Argollo, A. Falc, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52-61, 2009.

[10] R. Bedicheck, "SimNow: Fast platform simulation purely in software," *Hot Chips 16*, 2004.

[11] A. Chauhan, B. Sheraw, and C. Ding, "Scability and Data Placement on SGI Origin," *Technical Report TR98-305*, 1998.

[12] J. A. Redstone, S. J. Eggers, and H. M. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture," *SIGPLAN Not.*, vol. 35, pp. 245-256, 2000.

Considerations for Mondriaan-like Systems

Emmett Witchel
Department of Computer Science, The University of Texas at Austin
witchel@cs.utexas.edu

Abstract

Mondriaan memory protection is a hardware/software system that provides efficient fine-grained memory protection. Other researchers have embraced the Mondriaan design as an efficient way to associate metadata with each 32-bit word of memory. However, the Mondriaan design is efficient only when the metadata has certain properties. This paper tries to clarify when a Mondriaan-like design is appropriate for a particular problem. It explains how to reason about the space overhead of a Mondriaan design and identifies the significant time overheads as refills to the on-chip metadata cache and the time for software to encode and write metadata table entries.

1 Introduction

Mondriaan memory protection (MMP) is hardware/-software co-design for fine-grained memory protection. Like page tables, the heart of MMP is a set of hardware structures and software-written data structures that efficiently associate protection metadata with user data. Other researchers have used Mondriaan-like structures when they need to efficiently associate non-protection metadata with user data [ZKDK08, CMvPT07]. However, MMP is only efficient under certain assumptions about the metadata and data. This paper tries to clarify these assumptions to guide researchers in when MMP can be useful to address their problems.

While the computer science publication system is effective at creating incentives for researchers to publish innovative results, it is less effective at encouraging researchers to reflect on, and publicly critique, their own work. Students of the field are often left wondering why a promising sounding idea was left unimplemented. Or they wonder why certain ideas from an early paper on a subject are left out of follow-on work. Did those ideas fail or were they simply not explored?

While journals provide some outlet to summarize the progress of a research project, they often default to extended versions of conference papers. This paper is much shorter than a journal paper and tries to convey insights and experience, rather than rigorous quantitative evidence for its conclusions.

While providing a compact summary of MMP research, this paper highlights the assumptions made by various MMP implementations that are required for high performance. These assumptions do not always apply to systems developed by other researchers. The purpose of this paper is to allow other researchers to quickly determine if their application is likely to perform well with MMP-like hardware or what they would have to modify to make it perform well.

This paper discusses the interplay between the following design decisions.

1. **Space overhead.** The space overhead for MMP is approximately the average number of metadata bits per data item. MMP keeps space overhead low by storing 2 bits of protection information per 32-bit word (approximately a 6% overhead). Tables can be encoded for greater space efficiency if there are long stretches of memory with identical metadata values.
2. **PLB reach.** MMP includes an on-chip associative memory for its metadata called the protection lookaside buffer (PLB). For the PLB hardware to be an effective cache, the metadata must have particular properties, either much of it is coarse-grained, or it has long segments with identical metadata values.
3. **Software overheads.** MMP requires system software to write the metadata tables. The meta-

data format must be simple enough and written infrequently enough to prevent software from significantly reducing performance.

2 MMP history

MMP started in 2002 as follow-on work to low-power data caches [WLAA01]. Our idea was to automatically migrate unused program data to a portion of the cache/memory hierarchy that requires lower power to maintain state. To track program objects, which tend to be small and not naturally aligned, we needed a data structure. The data structure would be written by software and read by hardware because we thought the hardware would make frequent decisions about what data belongs in high-power fast memory and what can reside in low-power slow memory.

During the design of the hardware data structure, we realized that solving the basic problem of having hardware track user-defined data structures was more profound than the application of moving data to save energy. We soon left that motivation and chose fine-grained protection. The plugin model for program functionality extension made the motivation for fine-grained protection clear. Programs (like the OS and a web browser) load user-supplied code directly into their address space to extend functionality. The problem with this approach is that a bug can crash the entire program—plugins are fast, but not safe. Fine-grained protection can restore the safety without reducing the speed of the plugin extensibility model.

The first MMP paper focused on the format of the hardware tables [WCA02]. This paper is most often cited by those interested in MMP. It introduces the basic idea of MMP and presents both a simple table format and a more advanced table format, a technical innovation explained in §3.2. It also contains a design for fine-grained memory remapping, which allows a user to stitch together bits of memory into a contiguous buffer. The design for remapping is a bit complicated, but provides good support for zero-copy networking. The issues for supporting protection dominated the project after this paper and the remapping was dropped, simply for lack of space.

The follow-on paper [WA03] describes how the OS support for fine-grained protection domains would work and how to support safe calling between protection domains. Though our experience was limited at the time, much of our design ended up in our final implementation. My thesis [Wit04] continued to refine the OS support and added ideas for protecting the stack. MMP culminated in an SOSP paper [WRA05], which is the most complete implementation of the system, though it is not often cited. Most of the interest in MMP comes from computer architects, many of whom do not regularly read the proceedings of SOSP.

3 MMP technical summary

This section provides a high-level summary of how MMP works, with a focus on how MMP-like hardware would be used for other applications. The three main features of MMP are memory protection, protected cross-domain calling, and stack protection. The feature most attractive for other uses is a generalization of memory protection, which associates metadata with every word of user data. This section focuses on the general design of that protection mechanism.

3.1 CPU modifications

MMP consists of hardware and software to provide fine-grained memory protection. MMP modifies the processor pipeline to check permissions on every load, store, and instruction fetch. MMP is designed to be simple enough to allow an efficient implementation for modern processors, but powerful enough to allow a variety of software services to be built on top of it. The permissions information managed by MMP could be generalized to any metadata.

Figure 1 shows the basics of the MMP hardware. MMP adds a *protection lookaside buffer* (PLB) that is an associative memory, like a TLB. The PLB caches entries of a memory-resident permissions (or metadata) table, just as a TLB caches entries of a memory-resident page table. The PLB is indexed by virtual address. MMP also adds two registers, the protection domain ID, and a pointer to the base of the permissions table. The protection domain ID identifies the protection (or metadata) context of a particular kernel thread to the PLB, just as an address space identifier identifies a kernel thread to a TLB.

The protection domain ID register is not necessary, but without it, the entire PLB must be flushed on

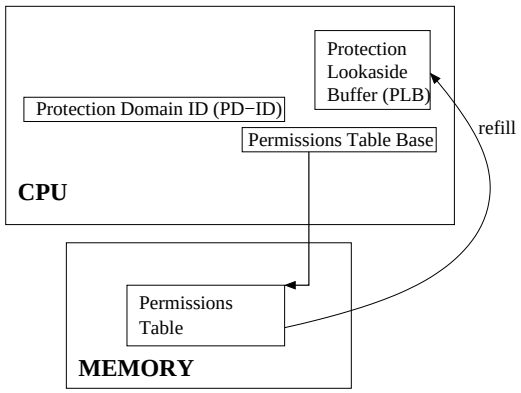

Figure 1: The major components of the Mondriaan memory protection system.

every domain switch. For some kinds of metadata, this might be acceptable. For example, if each kernel thread is its own domain, then domain switches only happen on context switches, which are relatively rare. In this case, the protection domain ID can be dispensed with, just as the x86 does not tag its TLB. However, many consider the lack of tags in the x86 TLB a major design flaw.

On a memory reference, the processor checks the PLB for permissions (or performs whatever metadata check is specified by a system using an MMP-like structure). If the PLB does not have the permissions information, either hardware or software looks it up in the permissions table residing in memory. The reload mechanism caches the matching entry from the permissions table in the PLB, and possibly writes it to the address register sidecar registers. Sidecar registers were a feature of the early MMP design. They are a cache for the PLB, meant to reduce the energy cost of indexing the PLB. They are simply an energy optimization, and because they are inessential, we do not mention them further.

Just like the TLB, the PLB is indexed by virtual address. The PLB lookup can happen in parallel with address translation because MMP stores its metadata per virtual address. Virtual addresses that alias to the same physical address can have different permissions values in MMP.

One of the hardware efficiencies of MMP is that the permissions check can start early in the pipeline and can overlap most of the address translation stages and computational steps of the pipeline. The permissions check need finish only before instruction retirement.

3.2 Permissions table

The MMP protection table represents each *user segment*, using one or more *table segments*. A user segment is a contiguous run of memory words with a single permissions value that has some meaning to the user. For example, a memory block returned from `kmalloc` would be a user segment. User segments start at any word boundary and do not have to be aligned. A table segment is a unit of permissions representation convenient for the permissions table. MMP is not efficient for arbitrary user segments, it assumes certain properties of user segments to achieve efficient execution (§3.2).

System software converts user segments into table entries when permissions are set on a memory region. As explained in §4.3, the frequency and complexity of transforming user segments into table segments determines whether software is an appropriate choice for encoding table segments. Some table entry formats are inefficient for software to write at the update rates required of applications.

Effective address (bits 31−0)

Root Index (10)	Mid Index (10)	Leaf Index (6)	Leaf Offset (6)
Bits (31−22)	Bits (21−12)	Bits (11−6)	Bits (5−0)

Figure 2: How an address indexes the trie.

MMP uses a trie to store metadata, just like a page table. The top bits of an address index into a table, whose entry can be a pointer to another table which is indexed by the most significant bits remaining in the address.

Figure 2 shows which bits of a 32-bit virtual address are used to index a particular level of the MMP permissions table trie. Three loads are sufficient to find the metadata for any user 32-bit word. The lookup algorithm (that can be implemented in software or hardware, just like a TLB) is shown in pseudo-code in Figure 3. The root table has 1024 entries, each of which maps a 4 MB block. Entries in the mid-level table map 4 KB blocks. The leaf level tables have 64 entries, each providing individual permissions for at least 16 four-byte words. The table indices are expanded for 64-bit address spaces [Wit04].

```
PERM_ENTRY lookup(addr_t addr) {
  PERM_ENTRY e = root[addr >> 22];
  if(is_tbl_ptr(e)) {
    PERM_TABLE* mid = e<<2;
    e = mid[(addr >> 12) & 0x3FF];
    if(is_tbl_ptr(e)) {
      PERM_TABLE* leaf = e<<2;
      e = leaf[(addr >> 6) & 0x3F];
    }
  }
  return e;
}
```

Figure 3: Pseudo-code for the trie table lookup algorithm. The table is indexed with an address and returns a permissions table entry. The base of the root table is held in a dedicated CPU register. The implementation of `is_tbl_ptr` depends on the encoding of the permission entries.

The granularity of the metadata is determined by the level at which it appears. Each two-bit entry in a leaf table encodes permissions for a user word of memory. At one level higher, each mid-level entry represents permissions for an entire 4KB page. Regions of at least 4KB that share a single permissions value can therefore be represented with less space overhead. This space savings happens regardless of the entry format.

MMP designs have used different permissions entry formats, notably bitmaps and run-length encoded (RLE) entries (shown in Figure 4). Each leaf entry in bitmap format has 16 two-bit values indicating the permissions for each of 16 words. Run-length encoded entries encode permissions as 4 regions with distinct permissions values, dedicating 8 out of 32 bits to metadata.

RLE entries cannot represent arbitrary word-level metadata. They assume that contiguous words have the same metadata value. For MMP's RLE entries, there can be no more than 4 distinct metadata regions in the entry's 16 data words. If each word has a metadata value distinct from its immediate neighbors, then there are 16 metadata regions and that cannot be represented with an RLE entry. Bitmap entries are used as backup in this case.

The permissions data in MMP run-length encoded entries overlaps with previous and succeeding entries. In addition to permissions information for the 16 words, they can also contain permissions for up to 31 words previous and 32 words subsequent to the 16. In the best case a single RLE entry can contain permissions from 5 distinct bitmap entries.

MMP uses RLE entries to overlap permissions information, but they can be used to save space in leaf-level tables. A 32-bit RLE entry can represent permissions information about 79 words. Taking into account alignment, each entry could encode permissions for 64 words instead of 16, bringing down the average space overheads for leaf-level tables from 6% to 1.6%. Doing so would change the lookup algorithm in Figure 3, because the leaf index would require only 4 bits, leaving 8 bits for the leaf offset. This new RLE format would be more restrictive, only allowing 4 permissions regions in every aligned 64 word block.

4 Requirements for good MMP performance

This section distills our observations on the factors salient for a particular instantiation of an MMP-like system to have good performance.

4.1 Space overhead

While physical memory capacity continues to grow at an impressive rate, MMP-like systems consume memory in proportion to the virtual memory used by a process. As processes use more memory, MMP uses more memory to hold the metadata associated with the data. Keeping the size of the metadata tables reasonable is a first order concern for the practicality of the system.

For the simplest Mondriaan system [WCA02], the space overhead of the most fine-grained tables is approximately 6%, for 2 bits of metadata per 32-bit data word. Both bitmaps and run-length encoded entries dedicate two bits of table entry per user word in leaf-level tables that manage permissions for 32-bit words. The run-length encoding could be adjusted for lower space overhead (§3.2). The mid-level entries that manage permissions for 4KB pages specify 2 bits of metadata per aligned 512 bytes of data, for a space overhead of 0.8%.

Mondrix [WRA05] (the application of Mondriaan

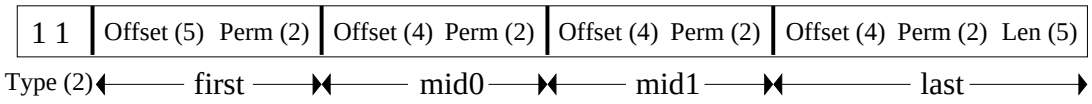

Figure 4: The bit allocation for a run-length encoded (RLE) permission table entry.

memory protection to the Linux kernel), modifies the kernel memory allocator to create larger, aligned data regions. The slab [Bon94] allocator takes a memory page and breaks it into equal sized chunks that are doled out by `kmalloc`, the general-purpose kernel memory allocator. By turning read/write permissions on and off for the entire page, rather than for each individual call to `kmalloc`, Mondrix greatly reduced the space overheads of the permissions. The cost is less memory protection. A read or write into the unallocated area of a page being used as a slab is an error can be detected if the page's permissions are managed on a per-word basis. However, Mondrix forgoes this protection, enabling read and write permissions to the entire page once any of it is used.

Colorama [CMvPT07] uses a Mondriaan design and extends the permission table entries with 12-bit color identifiers that allow the processor to infer synchronization for user data. The color identifiers bring the overhead from 2 bits per 32-bit word to 14 bits per 32-bit word, which is a 44% space overhead. Run-length encoding can bring down this overhead. Using MMP's RLE entry expands the 8 permissions bits to 48 for a 14% space overhead (though keeping entries aligned, which is necessary for a realistic design, would increase the space overhead to nearly 19%). Furthermore, not every data item needs to be colored, only those accessed by multiple threads. The Colorama implementation has a measured overhead of 0–28% space overhead.

Loki [ZKDK08] uses tagged memory to reduce the amount of trusted code in the HiStar operating system. Loki differs from MMP in that the tags are for physical memory. Additionally, Loki maintains two distinct maps, one from physical memory address to tag and another from tag to access permissions.

Loki segregates pages on the basis of whether they need fine-grained tags. Pages with fine-grained tags have 100% space overhead (a 32-bit tag for a 32-bit word), while pages without fine-grained tags (one tag for the entire page) have 0.1% space overhead. The authors see a variable fraction of memory pages that use fine-grained tags, from 3–65%. For this scenario, the fraction of memory pages using fine-grained tags dictates the memory overhead, so the application that uses fine-grained tags for 65% of its pages, experiences a space overhead of 65%. Loki does not use an MMP design for its tags, but the designers note that MMP's RLE entries could save space.

4.2 PLB reach

MMP uses a protection lookaside buffer (PLB) to cache permissions information for data accessed by the CPU, avoiding long walks through the memory resident permissions table. A high hit rate for the PLB is essential for low latency performance. Without a high hit rate, the processor is constantly fetching data from the permissions tables, which increases memory pressure, cache area pressure and decreases the rate at which instructions can retire.

MMP contains several features to enhance the hit rate in the PLB that can be adopted as-is by other projects. The PLB allows different entries to apply to different power-of-two sized ranges. This mechanism allows large granularity entries to co-exist with word-granularity entries (much like super-pages in TLBs). The PLB tags also include protection domain IDs to avoid flushing the PLB on domain switches. Tags are important for Mondrix because its fine-grained protection domains can be crossed as frequently as every 664 cycles [WRA05]. Other applications of MMP might not have such frequent domain crossings.

The main technique for MMP to increase PLB reach is to use large granularity entries (which is done by Mondrix) or run-length encoded entries (e.g., vpr and twolf from SPEC2000 [WCA02]). The PLB miss rate for Mondrix was lower than 1% for all workloads and the execution penalty for PLB refill was less than 4% of execution time because kernel text and data sections are represented with a single entry, and as mentioned in the previous section, the kernel memory allocator was modified to manage protections at the granularity of a page.

When each user allocation for vpr and twolf is protected by inaccessible words at the start and end of the allocation, the system spends 10–20% of its memory references refilling the PLB [WCA02] when using bitmap entries. Run-length encoded entries increase PLB reach by effectively encoding large user segments that share a permissions value with the start of the entry and/or its end. Using run-length encoded entries, memory accesses to the permissions table drop to 7.5% for both SPEC2000 benchmarks. As §3.2 discusses, RLE entries encode overlapping permissions information. A single RLE entry can contain the permissions information from multiple bitmapped entries, eliminating PLB refills.

Because Colorama only monitors shared data accesses, that decreases traffic to the on-chip metadata cache (the Colorama PLB). The Colorama implementation uses run-length encoded entries (called mini-SST entries in the original design [WCA02]), which should be effective at making PLB reach large enough for high performance. Additionally, the authors mention that color metadata could be aggregated into larger granularity chunks, by doing pooled memory allocation.

Loki's support for page-granularity metadata tags is crucial to keeping its runtime overheads low. For one `fork`/`exec` benchmark, page-granularity tags reduces the time overhead from 55% to 1%. It has an 8-entry cache to map from physical page to tag or pointer to a page of fine-grained tags. The fine-grained tags are stored in the CPU's cache. The physical address to tag map does not need to be flushed on a context switch. Loki also has a 32-entry 2-way associative cache that maps tags to permissions. This cache does need to be flushed on context switches, but does not need to be flushed when memory changes tags.

4.3 Software overheads

In an MMP-like design, metadata is managed like page tables are managed, software writes table entries that are read by hardware. Mondrix writes protection tables frequently to protect memory allocations, to protect network packets, and to protect arguments to cross-domain calls. The time for software to write the tables can become a significant performance cost, up to 10% of the kernel execution time in one Mondrix workload.

It is possible that a given application for an MMP-like system will have infrequent metadata updates. Having software encode table entries is a good choice for systems that update metadata infrequently because software is so flexible. However, we found that the only reliable technique for evaluating the cost of the software encoding is to implement it and run it on realistic inputs. The software entry encoding does not need to play a functional role in the system, but it is necessary for benchmarking.

The MMP ASPLOS paper [WCA02] does not evaluate the cost of writing table entries in software as it is a typical hardware evaluation paper that lacks system software support. In its defense, the system software required years of development effort, though effort to develop the table-entry encoder was a small fraction of that time. One unexpected consequence of writing the software to encode table entries is measuring the high runtime cost of writing run-length encoded entries. On one trace of memory protection calls extracted from Mondrix execution, writing run-length encoded entries is three times slower than writing bitmap entries. The run-length encoded entries are slow for software to write because they are complicated to encode, and because they overlap updates to an entry requires complicated logic for breaking and coalescing adjacent entries. While we developed and debugged the code to write run-length encoded entries (a task that required a solid month), we never deployed it in Mondrix because of its poor performance. Also, Mondrix had enough coarse-grained allocations that it did not need run-length encoded entries. Because of our experience with the software, we believe that any MMP implementation with run-length encoded entries will require hardware to encode the table entries.

Run-length encoded entries might be effective for Colorama, because the metadata update rate should be lower than Mondrix's. Mondrix writes the permissions table on memory allocations, and also during data structure processing (e.g., packet reception) and for cross-domain calls. The Colorama implementation measures low allocation rates for some applications (every 129K-288M instructions), and high rates for others, every 2–4K instructions. The authors conservatively assume that every allocation is for colored data, while the true rate for changing the color table

might be lower. The encoding costs for the run-length encoded entries might be an issue if the color tables are actually updated every 2–4K instructions.

Loki's simple data layout can be efficiently written by software. Page-granularity tags are held in an array, and fine-grained tags occupy the same offset in a page as their associated data.

Summary. The trade-offs among space overhead, PLB reach and software overheads are complex for a high-performance MMP-like system. Applying MMP to SPEC2000 and to the Linux kernel resulted in different trade-offs. For projects that only tangentially involve an MMP-like structure, the details of these trade-offs is out of scope. However, a high-level argument for the plausibility of a specific application is necessary to make an argument for the efficiencies of an MMP implementation.

5 Conclusion

The hardware and software designs for Mondriaan memory protection can be used to associate arbitrary metadata with individual user words at reasonable storage and execution time costs. However, keeping those costs limited requires careful design. The original MMP design makes assumptions that follow-on work may violate.

We encourage others to use MMP-like structures, and to include a discussion about space overhead, PLB reach, and software overheads. We hope this paper can act as a guide. The original MMP design limits its space overhead to 6% by using 2 metadata bits for each data word. It increases PLB reach either by using run-length encoded entries or by relying on large user segments. MMP limits software overheads by writing bitmaps in software and run-length encoded entries in hardware.

Acknowledgments

We thank Nickolai Zeldovich and Luis Ceze for their considered comments on a draft of this paper. Thanks also to Owen Hofmann and the anonymous referees for their constructive comments.

References

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, 1994.

[CMvPT07] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectural support for data-centric synchronization. In *IEEE International Symposium on High Performance Computer Architecture*, 2007.

[WA03] Emmett Witchel and Krste Asanović. Hardware works, software doesn't: Enforcing modularity with Mondriaan memory protection. In *HotOS*, 2003.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[Wit04] Emmett Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachussetts Institute of Technology, January 2004.

[WLAA01] Emmett Witchel, Sam Larsen, C. Scott Ananian, and Krste Asanović. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[WRA05] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[ZKDK08] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Operating Systems Design and Implementation*, 2008.

Is Transactional Programming Actually Easier?

Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel
Department of Computer Science, University of Texas at Austin
{rossbach,osh,witchel}@cs.utexas.edu

Abstract

Chip multi-processors (CMPs) have become ubiquitous, while tools that ease concurrent programming have not. The promise of increased performance for all applications through ever more parallel hardware requires good tools for concurrent programming, especially for average programmers. Transactional memory (TM) has enjoyed recent interest as a tool that can help programmers program concurrently.

The TM research community claims that programming with transactional memory is easier than alternatives (like locks), but evidence is scant. In this paper, we describe a user-study in which 147 undergraduate students in an operating systems course implemented the same programs using coarse and fine-grain locks, monitors, and transactions. We surveyed the students after the assignment, and examined their code to determine the types and frequency of programming errors for each synchronization technique. Inexperienced programmers found baroque syntax a barrier to entry for transactional programming. On average, subjective evaluation showed that students found transactions harder to use than coarse-grain locks, but slightly easier to use than fine-grained locks. Detailed examination of synchronization errors in the students' code tells a rather different story. Overwhelmingly, the number and types of programming errors the students made was much lower for transactions than for locks. On a similar programming problem, over 70% of students made errors with fine-grained locking, while less than 10% made errors with transactions.

1 Introduction

Transactional memory (TM) has enjoyed a wave of attention from the research community. The increasing ubiquity of chip multiprocessors has resulted in a high availability of parallel hardware resources, without many concurrent programs. TM researchers position TM as an enabling technology for concurrent programming for the "average" programmer.

Transactional memory allows the programmer to delimit regions of code that must execute atomically and in isolation. It promises the performance of fine-grain locking with the code simplicity of coarse-grain locking. In contrast to locks, which use mutual exclusion to serialize access to critical sections, TM is typically implemented using optimistic concurrency techniques, allowing critical sections to proceed in parallel. Because this technique dramatically reduces serialization when dynamic read-write and write-write sharing is rare, it can translate directly to improved performance without additional effort from the programmer. Moreover, because transactions eliminate many of the pitfalls commonly associated with locks (e.g. deadlock, convoys, poor composability), transactional programming is touted as being easier than lock based programming.

Evaluating the ease of transactional programming relative to locks is largely uncharted territory. Naturally, the question of whether transactions are easier to use than locks is qualitative. Moreover, since transactional memory is still a nascent technology, the only available transactional programs are research benchmarks, and the population of programmers familiar with both transactional memory and locks for synchronization is vanishingly small.

To address the absence of evidence, we developed a concurrent programming project for students of an undergraduate Operating Systems course at the University of Texas at Austin, in which students were required to implement the same concurrent program using coarse and fine-grained locks, monitors, and transactions. We surveyed students about the relative ease of transactional programming as well as their investment of development effort using each synchronization technique. Additionally, we examined students' solutions in detail to characterize and classify the types and frequency of programming errors students made with each programming technique.

This paper makes the following contributions:

- A project and design for collecting data relevant to the question of the relative ease of programming with different synchronization primitives.
- Data from 147 student surveys that constitute the first (to our knowledge) empirical data relevant to the question of whether transactions are, in fact, easier to use than locks.

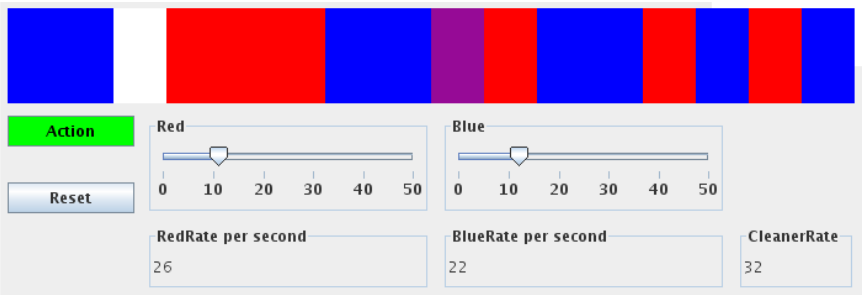

Figure 1: A screen-shot of sync-gallery, the program undergraduate OS students were asked to implement. In the figure the colored boxes represent 16 shooting lanes in a gallery populated by shooters, or *rogues*. A red or blue box represents a box in which a rogue has shot either a red or blue paint ball. A white box represents a box in which no shooting has yet taken place. A purple box indicates a line in which both a red and blue shot have occurred, indicating a race condition in the program. Sliders control the rate at which shooting and cleaning threads perform their work.

- A taxonomy of synchronization errors made with different synchronization techniques, and a characterization of the frequency with which such errors occur in student programs.

2 Sync-gallery

In this section, we describe sync-gallery, the Java programming project we assigned to students in an undergraduate operating systems course. The project is designed to familiarize students with concurrent programming in general, and with techniques and idioms for using a variety of synchronization primitives to manage data structure consistency. Figure 1 shows a screen shot from the sync-gallery program.

The project asks students to consider the metaphor of a shooting gallery, with a fixed number of lanes in which *rogues* (shooters) can shoot in individual lanes. Being pacifists, we insist that shooters in this gallery use red or blue paint balls rather than bullets. Targets are white, so that lanes will change color when a rogue has shot in one. Paint is messy, necessitating *cleaners* to clean the gallery when all lanes have been shot. Rogues and cleaners are implemented as threads that must check the state of one or more lanes in the gallery to decide whether it is safe to carry out their work. For rogues, this work amounts to shooting at some number of randomly chosen lanes. Cleaners must return the gallery to it's initial state with all lanes white. The students must use various synchronization primitives to enforce a number of program invariants:

1. **Only one rogue may shoot in a given lane at a time.**
2. **Rogues may only shoot in a lane if it is white.**
3. **Cleaners should only clean when all lanes have been shot (are non-white).**
4. **Only one thread can be engaged in the process of cleaning at any given time.**

If a student writes code for a rogue that fails to respect the first two invariants, the lane can be shot with both red and blue, and will therefore turn purple, giving the student instant visual feedback that a race condition exists in the program. If the code fails to respect to the second two invariants, no visual feedback is given (indeed these invariants can only be checked by inspection of the code in the current implementation).

We ask the students to implement 9 different versions of rogues (Java classes) that are instructive for different approaches to synchronization. Table 1 summarizes the rogue variations. Gaining exclusive access to one or two lanes of the gallery in order to test the lane's state and then modify it corresponds directly to the real-world programming task of locking some number of resources in order to test and modify them safely in the presence of concurrent threads.

2.1 Locking

We ask the students to synchronize rogue and cleaner threads in the sync-gallery using locks to teach them about coarse and fine-grain locking. To ensure that students write code that explicitly performs locking and unlocking operations, we require them to use the Java `ReentrantLock` class and do not allow use of the `synchronized` keyword. In locking rogue variations, cleaners do not use dedicated threads; the rogue that colors the last white lane in the gallery is responsible for becoming a cleaner and subsequently cleaning all lanes. There are four variations on this rogue type: **Coarse, Fine, Coarse2** and **Fine2**. In the coarse implementation, students are allowed to use a single global lock which is acquired before attempting to shoot or clean. In the fine-grain implementation, we require the students to implement individual locks for each lane. The Coarse2 and Fine2 variations require the same mapping of locks to ob-

```
final int x = 10;
Callable c = new Callable<Void> {
  public Void call() {
    // txnl code
    y = x * 2;
    return null;
  }
}
Thread.doIt(c);
```

```
Transaction tx = new Transaction(id);
boolean done = false;
while(!done) {
  try {
    tx.BeginTransaction();
    // txnl code
    done = tx.CommitTransaction();
  } catch(AbortException e) {
    tx.AbortTransaction();
    done = false;
  }
}
```

Figure 2: Examples of (left) DSTM2 concrete syntax, and (right) JDASTM concrete syntax.

jects in the gallery as their counterparts above, but introduce the additional stipulation that rogues must acquire access to and shoot at two random lanes rather than one. The pedagogical value is illustration that fine-grain locking requires a lock-ordering discipline to avoid deadlock, while a single coarse lock does not. Naturally, the use of fine grain lane locks complicates the enforcement of invariants 3 and 4 above.

2.2 Monitor implementations

Students must use condition variables along with signal/wait to implement both fine and coarse locking versions of the rogue programs. These two variations introduce dedicated threads for cleaners: shooters and cleaners must use condition variables to coordinate shooting and cleaning phases. In the coarse version (**CoarseCleaner**), students use a single global lock, while the fine-grain version (**FineCleaner**) requires per-lane locks.

2.3 Transactions

Finally, the students are asked to implement 3 TM-based variants of the rogues that share semantics with some locking versions, but use transactional memory for synchronization instead of locks. The most basic TM-based rogue, **TM**, is analogous to the Coarse and Fine versions: rogue and cleaner threads are not distinct, and shooters need shoot only one lane, while the **TM2** variation requires that rogues shoot at two lanes rather than one. In the **TMCleaner**, rogues and cleaners have dedicated threads. Students can rely on the TM subsystem to detect conflicts and restart transactions to enforce all invariants, so no condition synchronization is required.

2.4 Transactional Memory Support

Since sync-gallery is a Java program, we were faced with the question of how to support transactional memory. The ideal case would have been to use a software transactional memory (STM) that provides support for atomic blocks, allowing students to write transactional code of the form:

```
void shoot() {
  atomic {
    Lane l = getLane(rand());
    if(l.getColor() == WHITE)
      l.shoot(this.color);
  }
}
```

Rogue name	Technique	R/C Threads	Additional Requirements
Coarse	Single global lock	not distinct.	
Coarse2	Single global lock	not distinct	rogues shoot at 2 random lanes
CoarseCleaner	Single global lock, conditions	distinct	conditions, wait/notify
Fine	Per lane locks	not distinct	
Fine2	Per lane locks	not distinct	rogues shoot at 2 random lanes
FineCleaner	Per lane locks, conditions	distinct	conditions, wait/notify
TM	TM	not distinct	
TM2	TM	not distinct	rogues shoot at 2 random lanes
TMCleaner	TM	distinct	

Table 1: The nine different rogue implementations required for the sync-gallery project. The technique column indicates what synchronization technique was required. The R/C Threads column indicates whether coordination was required between dedicated rogue and cleaner threads or not. A value of “distinct” means that rogue and cleaner instances run in their own thread, while a value of “not distinct” means that the last rogue to shoot an empty (white) lane is responsible for cleaning the gallery.

No such tool is yet available; implementing compiler support for atomic blocks, or use of a a source-to-source compiler such as spoon [1] were considered out-of-scope for the project. The trade-off is that students are forced to deal directly with the concrete syntax of our TM implementation, and must manage read and write barriers explicitly. We assigned the lab to 4 classes over 2 semesters. During the first semester both classes used DSTM2 [14]. For the second semester, both classes used JDASTM [24].

The concrete syntax has a direct impact on ease of programming, as seen in Figure 2. Both examples pepper the actual data structure manipulation with code that explicitly manages transactions. We replaced DSTM2 in the second semester because we felt that JDASTM syntax was somewhat less baroque and did not require students to deal directly with programming constructs like generics. Also, DSTM2 binds transactional execution to specialized thread classes. However, both DSTM2 and JDASTM require explicit read and write barrier calls for transactional reads and writes.

3 Methodology

Students completed the sync-gallery program as a programming assignment as part of several operating systems classes at the University of Texas at Austin. In total, 147 students completed the assignment, spanning two sections each in classes from two different semesters of the course. The semesters were separated by a year. We provided an implementation of the shooting gallery, and asked students to write the rogue classes described in the previous sections, respecting the given invariants.

We asked students to record the amount of time they spent designing, coding, and debugging each programming task (rogue). We use the amount of time spent on each task as a measure of the difficulty that task presented to the students. This data is presented in Section 4.1. After completing the assignment, students rated their familiarity with concurrent programming concepts prior to the assignment. Students then rated their experience with the various tasks, ranking synchronization methods with respect to ease of development, debugging, and reasoning (Section 4.2).

While grading the assignment, we recorded the type and frequency of synchronization errors students made. These are the errors still present in the student's final version of the code. We use the frequency with which students made errors as another metric of the difficulty of various synchronization constructs.

To prevent experience with the assignment as a whole from influencing the difficulty of each task, we asked students to complete the tasks in different orders. In each group of rogues (single-lane, two-lane, and separate cleaner thread), students completed the coarse-grained lock version first. Students then either completed the fine-grained or TM version second, depending on their assigned group. We asked students to randomly assign themselves to groups based on hashes of their name. Due to an error, nearly twice as many students were assigned to the group completing the fine-grained version first. However, there were no significant differences in programming time between the two groups, suggesting that the order in which students implemented the tasks did not affect the difficulty of each task.

3.1 Limitations

Perhaps the most important limitation of the study is the much greater availability of documentation and tutorial information about locking than about transactions. The novelty of transactional memory made it more difficult both to teach and learn. The concrete syntax of transactions is also a barrier to ease of understanding and use (see §4.2). Lectures about locking drew on a larger body of understanding that has existed for a longer time. It is unlikely that students from one year influenced students from the

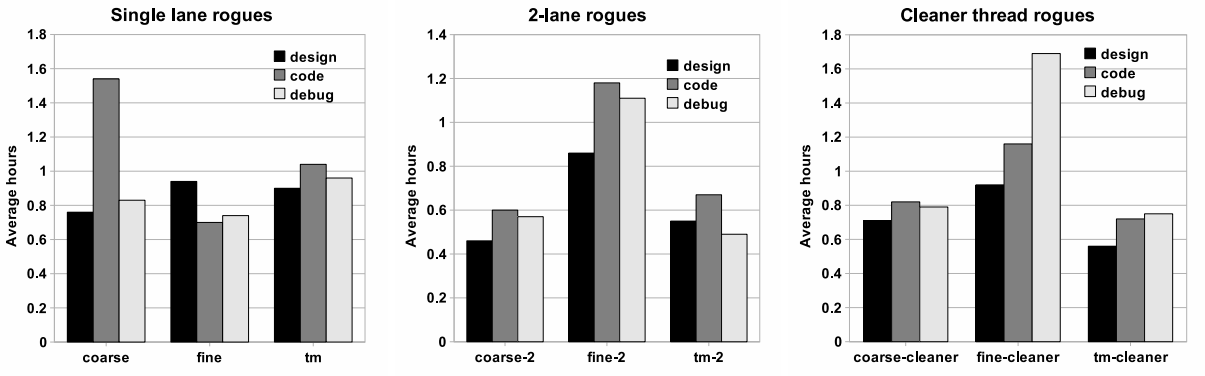

Figure 3: Average design, coding, and debugging time spent for analogous rogue variations.

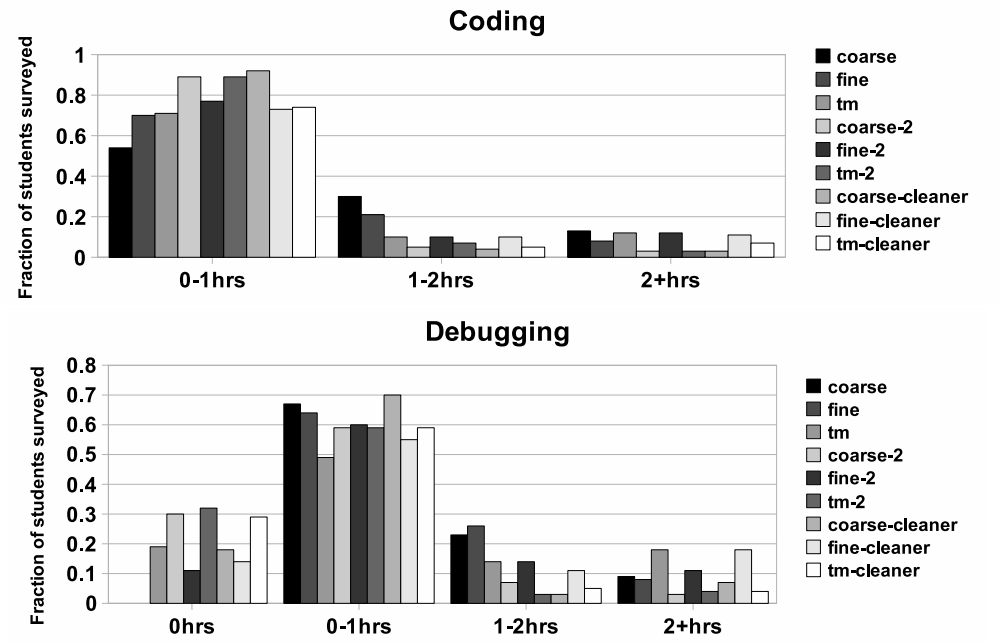

Figure 4: Distributions for the amount of time students spent coding and debugging, for all rogue variations.

next year given the difference in concrete syntax between the two courses.

4 Evaluation

We examined development time, user experiences, and programming errors to determine the difficulty of programming with various synchronization primitives. In general, we found that a single coarse-grained lock had similar complexity to transactions. Both of these primitives were less difficult, caused fewer errors, and had better student responses than fine-grained locking.

4.1 Development time

Figures 4 and 3 characterize the amount of time the students spent designing, coding and debugging with each synchronization primitive. On average, transactional memory required more development time than coarse locks, but less than required for fine-grain locks and condition synchronization. With more complex synchronization tasks, such as coloring two lanes and condition synchronization, the amount of time required for debugging increases relative to the time required for design and coding (Figure 3).

We evaluate the statistical significance of differences in development time in Table 2. Using a Wilcoxon signed-rank test, we evaluated the alternative hypothesis on each pair of synchronization tasks that the row task required less time than the column task. Pairs for which the signed-rank test reports a p-value of $< .05$ are considered statistically significant, indicating that the row task required less time than the column. If the p-value is greater than .05, the difference in time for the tasks is not statistically significant or the row task required more time than the column task. Results for the different class years are separated due to differences in the TM part of the assignment(Section 2.4).

We found that students took more time to develop the initial tasks while familiarizing themselves with the assignment. Except for fine-grain locks, later versions of similar synchronization primitives took less time than earlier, e.g. the Coarse2 task took less time than the Coarse task. In addition, condition synchronization is difficult. For both rogues with less complex synchronization (Coarse and TM), adding condition synchronization increases the time required for development. For fine-grain locking, students simply replace one complex problem with a second, and so do not require significant additional time.

In both years, we found that coarse locks and transactions required less time than fine-grain locks on the more complex two-lane assignments. This echoes the promise of transactions, removing the coding and debugging complexity of fine-grain locking and lock ordering when more than one lock is required.

4.2 User experience

To gain insight into the students' perceptions about the relative ease of using different synchronization techniques we asked the students to respond to a survey after completing the sync-gallery project. The survey ends with 6 questions asking students to rank their favorite technique with respect to ease of development, debugging, reasoning about, and so on.

A version of the complete survey can be viewed at [2].

In student opinions, we found that the more baroque syntax of the DSTM2 system was a barrier to entry for new transactional programmers. Figure 5 shows student responses to questions about syntax and ease of thinking about different transactional primitives. In the first class year, students found transactions more difficult to think about and had syntax more difficult than that of fine-grain locks. In the second year, when the TM implementation was replaced with one less cumbersome, student opinions aligned with our other findings: TM ranked behind coarse locks, but ahead of fine-grain. For both years, other questions on ease of design and implementation mirrored these results, with TM ranked ahead of fine-grain locks.

4.3 Synchronization Error Characterization

We examined the solutions from the second year's class in detail to classify the types of synchronization errors students made along with their frequency. This involved both a thorough reading of every student's final solutions and automated testing. While the students' subjective evaluation of the ease of transactional programming does not clearly indicate that transactional programming is easier, the types and frequency of programming errors does.

While the students showed an impressive level of creativity with respect to synchronization errors, we found that all errors fit within the taxonomy described below.

1. **Lock ordering (lock-ord)**. In fine-grain locking solutions, a program failed to use a lock ordering discipline to acquire locks, admitting the possibility of deadlock.
2. **Checking conditions outside a critical section (lock-cond)**. This type of error occurs when code checks a program condition with no locks held, and subsequently acts on that condition after acquiring locks. This was the most common error in sync-gallery, and usually occurred when students would check whether to clean the gallery with no locks held, subsequently acquiring lane locks and proceeding to clean. The result is a violation of invariant 4 (§2). This type of error may be more common because no visual feedback is given when it is violated (unlike races for shooting lanes, which can result in purple lanes).
3. **Forgotten synchronization (lock-forgot)**. This class of errors includes all cases where the programmer forgot to acquire locks, or simply did not realize that a particular region would require mutual exclusion to be correct.
4. **Exotic use of condition variables (cv-exotic)**. We encountered a good deal of signal/wait usage on condition variables that indicates no clear understanding of what the primitives actually do. The canonical example of this is signaling and waiting the same con-

Year 1

Best syntax

Answers	1	2	3	4
Coarse	**69.6%**	17.4%	0%	8.7%
Fine	13.0%	**43.5%**	17.4%	21.7%
TM	8.7%	21.7%	21.7%	**43.5%**
Conditions	0%	21.7%	**52.1%**	21.7%

Easiest to think about

Answers	1	2	3	4
Coarse	**78.2%**	13.0%	4.3%	0%
Fine	4.3%	**39.1%**	34.8%	17.4%
TM	8.7%	21.7%	26.1%	**39.1%**
Conditions	4.3%	21.7%	30.4%	**39.1%**

Year 2

Best syntax

Answers	1	2	3	4
Coarse	**61.6%**	30.1%	1.3%	4.1%
Fine	5.5%	20.5%	**45.2%**	26.0%
TM	26.0%	**31.5%**	19.2%	20.5%
Cond.	5.5%	20.5%	28.8%	**39.7%**

Easiest to think about

Answers	1	2	3	4
Coarse	**80.8%**	13.7%	1.3%	2.7%
Fine	1.3%	**38.4%**	30.1%	28.8%
TM	16.4%	**31.5%**	30.1%	20.5%
Cond.	4.1%	13.7%	**39.7%**	**39.7%**

Figure 5: Selected results from student surveys. Column numbers represent rank order, and entries represent what percentage of students assigned a particular synchronization technique a given rank (e.g. 80.8% of students ranked Coarse locks first in the "Easiest to think about category"). In the first year the assignment was presented, the more complex syntax of DSTM made TM more difficult to think about. In the second year, simpler syntax alleviated this problem.

		Coarse	Fine	TM	Coarse2	Fine2	TM2	CoarseCleaner	FineCleaner	TMCleaner
Coarse	Y1	1.00	**0.03**	**0.02**	1.00	**0.02**	1.00	0.95	0.47	0.73
	Y2	1.00	0.33	0.12	1.00	0.38	1.00	1.00	0.18	1.00
Fine	Y1	0.97	1.00	0.33	1.00	0.24	1.00	1.00	0.97	0.88
	Y2	0.68	1.00	0.58	1.00	0.51	1.00	1.00	0.40	1.00
TM	Y1	0.98	0.68	1.00	1.00	0.13	1.00	1.00	0.98	0.92
	Y2	0.88	0.43	1.00	1.00	0.68	1.00	1.00	0.41	1.00
Coarse2	Y1	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	**<0.01**	**<0.01**	**<0.01**	**<0.01**
	Y2	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	0.45	**<0.01**	**<0.01**	**<0.01**
Fine2	Y1	0.98	0.77	0.87	1.00	1.00	1.00	1.00	1.00	0.98
	Y2	0.62	0.49	0.32	1.00	1.00	1.00	0.99	0.59	1.00
TM2	Y1	**<0.01**	**<0.01**	**<0.01**	0.99	**<0.01**	1.00	**0.04**	**<0.01**	**<0.01**
	Y2	**<0.01**	**<0.01**	**<0.01**	0.55	**<0.01**	1.00	**<0.01**	**<0.01**	**<0.01**
CoarseCleaner	Y1	0.05	**<0.01**	**<0.01**	1.00	**<0.01**	0.96	1.00	**<0.01**	0.08
	Y2	**<0.01**	**<0.01**	**<0.01**	1.00	**<0.01**	1.00	1.00	**<0.01**	0.96
FineCleaner	Y1	0.53	**0.03**	**0.02**	1.00	**<0.01**	1.00	0.99	1.00	0.46
	Y2	0.83	0.60	0.59	1.00	0.42	1.00	1.00	1.00	1.00
TMCleaner	Y1	0.28	0.12	0.08	1.00	**0.03**	1.00	0.92	0.55	1.00
	Y2	**<0.01**	**<0.01**	**<0.01**	0.99	**<0.01**	1.00	**0.04**	**<0.01**	1.00

Table 2: Comparison of time taken to complete programming tasks for all students. The time to complete the task on the row is compared to the time for the task on the column. Each cell contains p-values for a Wilcoxon signed-rank test, testing the hypothesis that the row task took less time than the column task. Entries are considered statistically significant when $p < .05$, meaning that the row task did take less time to complete than the column task, and are marked in bold. Results for first and second class years are reported separately, due to differing transactional memory implementations.

dition in the same thread.

5. **Condition variable use errors (cv-use)**. These types of errors indicate a failure to use condition variables properly, but do indicate a certain level of understanding. This class includes use of `if` instead of `while` when checking conditions on a decision to wait, or failure to check the condition at all before waiting.
6. **TM primitive misuse (TM-exotic)**. This class of error includes any misuse of transactional primitives. Technically, this class includes mis-use of the API, but in practice the only errors of this form we saw were failure to call `BeginTransaction` before calling `EndTransaction`. Omission of read/write barriers falls within this class as well, but it is interesting to note that we found no bugs of this form.
7. **TM ordering (TM-order)**. This class of errors represents attempts by the programmer to follow some sort of locking discipline in the presence of transactions, where they are strictly unnecessary. Such errors do not result in an incorrect program, but do represent a misunderstanding of the primitive.
8. **Forgotten TM synchronization (TM-forgot)**. Like the forgotten synchronization class above (lock-forgot), these errors occur when a programmer failed to recognize the need for synchronization and did not use transactions to protect a data structure.

Table 3 shows the characterization of synchronization for programs submitted in year 2. Figure 6 shows the overall portion of students that made an error on each programming task. Students were far more likely to make an error on fine-grain synchronization than on coarse or TM.

	lock-ord	**lock-cond**	**lock-forgot**	**cv-exotic**	**cv-use**	**TM-exotic**	**TM-order**	**TM-forgot**
occurrences	11	62	26	11	14	5	4	1
opportunities	134	402	402	134	134	201	201	201
rate	8.2%	6.5%	15.4%	8.2%	10.5%	2.5%	2.0%	0.5%

Table 3: Synchronization error rates for year 2. The occurrences row indicates the number of programs in which at least one bug of the type indicated by the column header occurred. The **opportunities** row indicates the sample size (the number of programs we examined in which that type of bug could arise: e.g. lock-ordering bugs cannot occur in with a single coarse lock). The **rate** column expresses the percentage of examined programs containing that type of bug. Bug types are explained in Section 4.3.

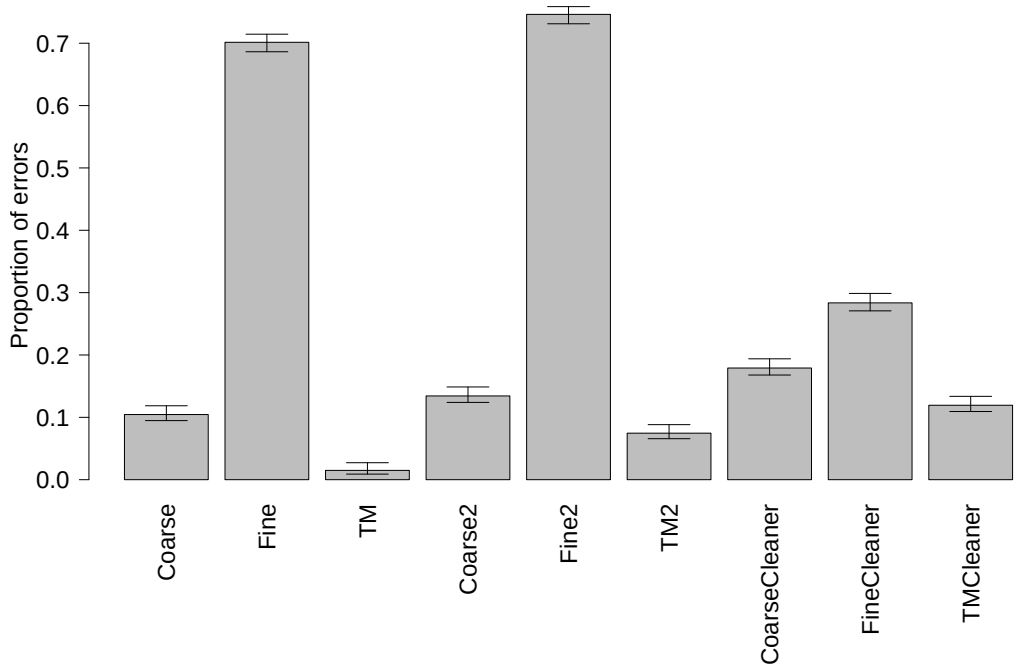

Figure 6: Overall error rates for programming tasks. Error bars show a 95% confidence interval on the error rate. Fine-grained locking tasks were more likely to contain errors than coarse-grained or transactional memory (TM).

About 70% of students made at least one error on the Fine and Fine2 portions of the assignment.

5 Related work

Hardware transactional memory research is an active research field with many competing proposals [4–7, 9–11, 15–17, 19–23, 26]. All this research on hardware mechanism is the cart leading the horse if researchers never validate the assumption that transactional programming is actually easier than lock-based programming.

This research uses software transactional memory (which has no shortage of proposals [3, 12–14, 18, 25]), but its purpose is to validate how untrained programmers learn to write correct and performant concurrent programs with locks and transactions. The programming interface for STM systems is the same as HTM systems, but without compiler support, STM implementations require explicit read-write barriers, which are not required in an HTM. Compiler integration is easier to program than using a TM library [8]. Future work research could investigate whether compiler integration lowers the perceived programmer difficulty in using transactions.

6 Conclusion

To our knowledge, no previous work directly addresses the question of whether transactional memory actually delivers on its promise of being easier to use than locks. This paper offers evidence that transactional programming really is less error-prone than high-performance locking, even if newbie programmers have some trouble understanding transactions. Students subjective evaluation showed that they found transactional memory slightly harder to use than coarse locks, and easier to use than fine-grain locks and condition synchronization. However, analysis of synchronization error rates in students' code yields a more dramatic result, showing that for similar programming tasks, transactions are considerably easier to get correct than locks.

References

[1] *Spoon*, 2009. `http://spoon.gforge.inria.fr/`.

[2] *Sync-gallery survey: http://www.cs.utexas.edu/users/witchel/tx/sync-gallery-survey.html*, 2009.

[3] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, Jun 2006.

[4] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *Proceedings of the 35th*

Annual International Symposium on Computer Architecture. June 2008.

[5] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[6] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[7] J. Chung, C. Minh, A. McDonald, T. Skare, H. Chafi, B. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS*, 2006.

[8] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.

[9] L. Yen et al. Logtm-SE: Decoupling hardware transactional memory from caches. In *HPCA*. 2007.

[10] Mark Moir et. al. Experiences with a commercial processor supporting htm. *ASPLOS 2009*.

[11] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[12] T. Harris, M. Plesko, A. Shinnar, and D.Tarditi. Optimizing memory transactions. In *PLDI*, Jun 2006.

[13] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, Oct 2003.

[14] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[15] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[16] Owen S. Hofmann, Christopher J. Rossbach, and Emmett Witchel. Maximal benefit from a minimal tm. *ASPLOS*.

[17] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 197–206, New York, NY, USA, 2008. ACM.

[18] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT*, 2006.

[19] A. McDonald, J. Chung, B. Carlstrom, C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA*, Jun 2006.

[20] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, , and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA*, 2006.

[21] R. Rajwar and M. Herlihy K. Lai. Virtualizing transactional memory. In *ISCA*, Jun 2005.

[22] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: Transactional memory for an operating system. In *ISCA*, 2007.

[23] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, 2008.

[24] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an STM. In *PPoPP*, 2009.

[25] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug 1995.

[26] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.