The Shared-Thread Multiprocessor

Jeffery A. Brown
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404

Dean M. Tullsen
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404

ABSTRACT

This paper describes initial results for an architecture called the Shared-Thread Multiprocessor (STMP). The STMP combines features of a multithreaded processor and a chip multiprocessor; specifically, it enables distinct cores on a chip multiprocessor to share thread state. This shared thread state allows the system to schedule threads from a shared pool onto individual cores, allowing for rapid movement of threads between cores.

This paper demonstrates and evaluates three benefits of this architecture: (1) By providing more thread state storage than available in the cores themselves, the architecture enjoys the ILP benefits of many threads, but carries the in-core complexity of supporting just a few. (2) Threads can move between cores fast enough to hide long-latency events such as memory accesses. This enables very-short-term load balancing in response to such events. (3) The system can redistribute threads to maximize symbiotic behavior and balance load much more often than traditional operating system thread scheduling and context switching.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors; C.4 [**Performance of Systems**]: Performance attributes

General Terms

Design, Performance

Keywords

Chip multiprocessors, simultaneous multithreading

1. INTRODUCTION

As more processor manufacturers move to multi-core designs, the search for the optimal design point becomes a difficult one. The mix of hardware cores and threading contexts will determine aggregate throughput and latency for a particular workload. However, each workload will achieve the best performance with a different configuration, and determining the most appropriate overall configuration is not straightforward.

Designs with many relatively small cores offer high peak throughput, but provide high per-thread latencies and perform poorly when thread-level parallelism is low. A processor with fewer, more powerful cores will provide lower per-thread latencies and good few-thread performance, but will be comparatively inefficient when running many threads. If we add multithreading [18, 17] to the latter design, we achieve a better tradeoff: providing both low latency when offered few threads, and high throughput when running many threads. However, there is a limit to how far throughput gains will scale, since very large cores do not scale even their peak throughput linearly with area.

The desired architecture, particularly for general-purpose computing workloads wherein the amount of parallelism is not constant, is an architecture with modest hardware requirements that achieves good performance on a single thread, and with performance improving as threads are added (up to a relatively high number of threads). We describe an architecture which features relatively modest cores with only minimal support for on-core multithreading (i.e., simultaneous multithreading, or SMT [18]), yet have hardware support for more threads than allowed by the on-core SMT hardware. Peak performance on such an architecture will scale roughly linearly until the number of threads reaches the number of cores, will continue to scale well as the number of threads increases to the total number of SMT contexts, and continue to climb modestly as even more threads are added. We call this architecture the *Shared-Thread Multiprocessor* (STMP).

The Shared-Thread Multiprocessor enables distinct cores on a chip multiprocessor to share thread state. This shared thread state allows the system to not only mix on-core and off-core support for multithreading – providing high ILP with minimal design overhead – but, by scheduling threads from a shared pool onto individual cores, also allows for rapid movement of threads between cores. This approach enables, for the first time, low-latency "multithreading-type" context switches between distinct cores. In this way, it combines the simplicity of multicore designs with the flexibility and high utilization of aggressively multithreaded designs.

This paper demonstrates and evaluates three benefits of this architecture: (1) By providing more thread state storage than available in the cores themselves, the architecture enjoys the ILP benefits of many threads, but carries the in-core complexity of supporting just a few. (2) Threads

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.

can move between cores fast enough to hide long-latency events such as memory accesses. This enables very-short-term load balancing in response to such events. (3) The system can redistribute threads to maximize symbiotic behavior and balance load much more often than traditional operating system thread scheduling and context switching.

The rest of the paper is organized as follows. Section 2 discusses prior related work. Section 3 describes the baseline architecture, while Section 4 describes the Shared-Thread Multiprocessor implementation. Methodology for our experiments is presented in Section 5. Our evaluations and results are described in Section 6. We conclude with Section 7.

2. BACKGROUND AND RELATED WORK

Tune et al. describe an architecture, Balanced Multithreading, (BMT) [19], which allows a single processor core to combine the benefits of simultaneous multithreading and a form of coarse-grain multithreading. In their preferred design, the SMT core only supports two hardware threads, keeping complexity low and the register file small. However, this is supplemented with off-core thread storage, which allowed thread state to be brought into the core quickly when an SMT context became idle due to a long-latency event. This adds little or no complexity to the core because it relies on injected instructions to transfer context state in and out.

In this way, more than two threads time-share the two SMT contexts. BMT achieves the instruction throughput of an SMT core with additional hardware threads, without the full hardware costs: two SMT-scheduled threads augmented with two coarsely-scheduled threads can exceed the IPC of three SMT threads.

The Shared-Thread Multiprocessor is an extension of the BMT architecture. In STMP, the off-core thread storage is shared among a pool of cores. This brings several new capabilities not available to BMT: the ability to dynamically partition the extra threads among cores, the ability to share threads to hide latencies on multiple cores, and the opportunity to use the shared-thread mechanism to accelerate thread context switching between cores (enabling fast rebalancing of the workload when conditions change).

Other relevant work has examined thread scheduling policies to maximize the combined performance of threads on a multithreaded processor [11, 10]. Previously, we also identified the importance of accounting for long-latency loads and minimizing the negative interference between stalled threads and others on the core [16]. Both BMT and STMP address those loads, by first removing stalled threads and then injecting new threads which are not stalled.

Constantinou et al. [4] examine several implications of thread migration policies on a multi-core processor migration. However, they do not examine these issues in the context of the type of hardware support for fast switching that our architecture provides.

Torrellas et al. [14] is one of several papers that examine the importance of considering processor affinity when scheduling threads on a multiprocessor. In their work, they endeavor to reschedule threads where they last executed, to minimize cold cache effects. We find that processor affinity effects are also extremely important in the STMP.

Spracklen et al. [12] argue for the combination of multiple SMT cores within a CMP for efficient resource utilization in the face of high-TLP, low-ILP server workloads. Their focus is primarily on such workloads, sacrificing single-thread performance for a design tailored to maximize throughput, without paying particular attention to the movement of threads. The STMP exploits high ILP when it's available, and offers fast context-switching to other threads to boost throughput when ILP is lacking.

Dataflow architectures [1, 2] offer an alternative to traditional ILP-based designs. Stavrou et al. [13] consider the implementation of hardware support for data-driven multithreaded computation on top of a conventional CMP. Their design uses additional per-core hardware to coordinate data movement and the scheduling of threads as live-in data become ready; threads are explicitly compiled as slices of decomposed dataflow and synchronization graphs. Prefetching and cache conflict tracking are used to avoid long-latency memory stalls during a thread's execution. The STMP, in contrast, uses traditional functional unit scheduling and control-driven instruction streams, detecting and reacting to long-latency memory events as they occur.

3. THE BASELINE MULTITHREADED MULTICORE ARCHITECTURE

The next two sections describe the processor architecture that we use for this study. This section describes a conventional architecture which is the base upon which we build the Shared-Thread Multiprocessor described in Section 4. Our baseline is a multithreaded, multi-core architecture (referred to sometimes as chip multithreading [12]). Several examples of this architecture already exist in industry [3, 5, 8, 6].

3.1 Chip multiprocessor

We study a chip multiprocessor (CMP) design consisting of four homogeneous cores. Each core has an out-of-order execution engine and contains private first-level instruction and data caches; the four cores share a common second-level unified cache.

The cores communicate with each other and with the L2 cache over a shared bus; data caches are kept coherent with a snoop-based coherence protocol [7]. Off-chip main memory is accessed via a memory controller that is shared among cores. While this relatively simple interconnect does not scale up to many cores, it suffices for four cores, and still features the essential property that inter-core communication is much faster than memory access, with latency comparable to that of a L2 cache hit. Our STMP extensions can apply to more scalable interconnects, such as point-to-point mesh networks, but we utilize this simpler symmetric interconnect in order to focus on the STMP implementation itself.

3.2 Simultaneous-Multithreaded cores

Each core of our chip multiprocessor features Simultaneous Multithreading [18, 17] (SMT) execution, with two hardware execution contexts per core, similar to several recent processor designs [3, 9].

Two-way SMT allows for two threads to share a core's execution resources within any cycle, enabling efficient resource utilization in the face of stalls. SMT has been shown to effectively hide short-term latencies in a thread by executing instructions from other threads. It provides significant gains in instruction throughput with small increases in hardware

complexity. We evaluate our Shared-Thread Multiprocessor designs on top of SMT in order to demonstrate that there is additional potential for performance gain beyond what SMT is able to capture. SMT is less successful at hiding very long latencies, where at best the stalled thread becomes unavailable to contribute to available ILP, and at worst stalls other threads by occupying resources.

The inclusion of SMT support is not essential in the design of the STMP; single-threaded cores could apply coarse-grain multithreading by switching in threads from the shared off-core thread pool. Such an architecture would still offer significant advantages over a single-threaded CMP without inactive-thread storage, but we find the SMT-based architecture more attractive, because the second thread on each core allows the processor to hide the latency of the thread swap operations themselves.

3.3 Long-latency memory operations

A typical memory hierarchy features successively larger and slower memory units (caches) at each level. In such a system, individual memory instructions may vary widely in response times, ranging from a few cycles for a first-level cache hit, to hundreds of cycles for a main-memory access, to essentially unbounded latencies in the event of a page fault.

While out-of-order execution is able to make progress in the face of memory operations taking tens of cycles, those which take longer rapidly starve the processor of useful work: once the memory instruction becomes the oldest from that thread, there is a finite dynamic-execution distance following it beyond which the processor is unable to operate, due to resource limitations. Previous work [16] has explored this phenomenon, in particular the impact of long-latency loads on SMT processors, along with several mechanisms for detecting these situations and mitigating the impact on co-scheduled threads.

Our baseline SMT processors include a similar long-latency-load detection and flushing mechanism: when a hardware execution context is unable to commit any instructions in a given cycle, and the next instruction to commit from the resident thread is an outstanding memory operation which is older than a time threshold, it is classified as "long-latency", and action is taken. In the baseline system, instructions younger than the load are flushed while the load remains; the experimental STMP systems described below also use this signal as an input to scheduling decisions.

4. SHARED-THREAD STORAGE: MECHANISMS AND POLICIES

In our *Shared-Thread Multiprocessor* architecture, we augment the CMP-of-SMTs described in Section 3 with several relatively inexpensive storage and control elements. Some of these architectural assumptions are similar to that used in the BMT research [19].

4.1 Inactive-thread store

We add an off-core *inactive-thread store* – storage for the architected state of numerous threads; this storage is used to hold additional threads beyond those supported by the hardware execution contexts. Per-thread state consists primarily of logical register values, the program counter, and a system-wide unique thread ID. This state occupies a fixed-size store of a few hundred bytes per thread, which is accessed non-speculatively in a regular fashion; it can be implemented efficiently with a small amount of SRAM. This is shown in Figure 1.

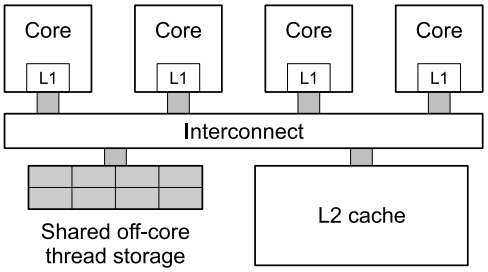

Figure 1: The Shared-thread Multiprocessor

4.2 Shared-thread control unit

We introduce additional control logic which coordinates the movement, activation, and deactivation of threads. The *shared-thread control unit* resides with the inactive-thread store and implements the various scheduling policies we explore. This control unit communicates with execution cores over the processor interconnect, receiving occasional notification messages from cores containing recent thread performance or status-change notifications, and sending messages which signal cores to trigger thread migration. This unit may be implemented in various ways, from a simple automaton to perhaps a small in-order processor (with some associated working memory), depending on the complexity of the scheduling policy desired.

As an alternative to using a discrete shared-thread control unit, its functionality may be implemented within the cores themselves in a scalable peer-to-peer fashion, which would be particularly beneficial as the number of cores is increased. We focus on a discrete implementation for this study.

4.3 Thread-transfer support

To each execution core, we add a mechanism to rapidly transfer thread state between the execution core with its internal data structures, and the simpler *inactive-thread store*.

Communication with the shared-thread control unit and its associated storage takes place across the processor interconnect, which is optimized for the transfer of cache blocks. To maximize communication efficiency, we add a small cache-block sized *spill/fill buffer* to each core, which is used for assembling thread-transfer data into cache-block sized messages. The spill/fill buffers are fixed-sized SRAM structures, which are accessed sequentially from beginning to end.

The primary concerns when removing a thread from a core are stabilizing a consistent, non-speculative view of the thread state, and then extracting that state from the on-core data structures into a concise form suitable for bulk transfers. We re-use mechanisms present on typical out-of-order processors to perform most of the detailed work involved with thread movement. We introduce two new micro-instructions, *spill* and *fill*, which each transfer a single register value to or from the next location in the spill/fill buffer. These new micro-instructions each specify a single logical register operand; they are injected into an execution core in a pre-decoded form just before the register-rename

stage, where they re-use the renaming and dependence logic to locate the appropriate values.

Each *spill* instruction extracts one architected register value, utilizing existing register-read ports and honoring any in-flight data dependences. When the *spill* completes execution, it appends the value to the spill/fill buffer. Each *fill* instruction reads one value from the spill/fill buffer as it executes, and writes to an architected register when it completes execution. When the buffer is filled by a write from a *spill* instruction, the entire buffer is bulk-transferred to the inactive-thread store, much like the delivery of a cache write-back. When the buffer is emptied by a read from a *fill* instruction, a flow-control indication is sent indicating that the core is ready for another block. (There is a small amount of ancillary data in these messages.)

In addition to the existing logic used to access registers, a new data pathway is necessary between the *spill/fill buffer* and whichever functional units it's most convenient to map *spill* and *fill* instructions to. Since *spill* and *fill* instructions are never introduced speculatively, are never used with overlapping register numbers, and always access the *spill/fill buffer* in order, adding this new data pathway is unlikely to introduce significant complexity to the processor.

Finally, each processor core is augmented with a small amount of *spill/fill control* logic, responsible for sending periodic performance counter samples to the shared-thread control unit, and for reacting to thread-swap requests. When a thread-swap request is received from the shared-thread control unit, the spill/fill control logic initiates a flush of the indicated hardware thread context – effectively causing an asynchronous trap – and then begins injecting *spill* or *fill* instructions (as appropriate), instead of vectoring to a conventional operating-system trap handler. This logic is also responsible for stalling fetch on the corresponding hardware context until the thread swap is complete. Note that, since each core supports two-way SMT execution, normal execution of a separate workload may proceed in one context while another is performing a swap.

Processors without explicit register renaming may still implement these *spill* and *fill* instructions, through whatever mechanism the processor uses to map between architected register numbers and run-time storage. We expect these new instructions to be trivial to represent in the existing internal instruction representation of most microarchitectures.

In summary, then, the bulk of the support for the STMP is in the thread storage and control, which is at about the same level (e.g., in distance and latency) from the cores as the L2 cache, but is much smaller. Other support that needs to be added to each core is relatively minor, and none of this functionality affects potential critical timing paths (such as renaming, instruction scheduling, register access, etc.) within a core.

4.4 Scaling the Shared-Thread Multiprocessor

In this research, we focus on a single implementation featuring four cores sharing thread state. As the number of cores increases, we could scale this architecture in two ways. We could increase the number of cores sharing a single centralized thread store, or we could increase the number of STMP instances. It is unclear that much would be gained by sharing thread storage directly among more than four cores, due to increases in latency and contention. Instead, we envision "islands" of STMPs, each a small set of cores clustered around their own inactive-thread store. A 64-core CMP, then, might have 16 thread storage units, each servicing four cores. Movement between thread stores would be supported, but with much less frequent movement, and need not be as fast. Future work is necessary to fully analyze these options. However, if we assume the approach which clusters STMP cores, then our results, which focus on four cores, are indicative of not just that configuration, but also the characteristics of each cluster of cores in a larger configuration.

The mechanisms presented are focused on the migration of registers and associated state, and incorporate fairly general assumptions about memory and interconnect layout. The underlying system described in Section 3.1 offers a shared L2 cache, with relatively low access latency; an architecture with private L2 caches, for example, would still support efficient register migration, but would experience more performance degradation due to cache effects subsequent to migration. We are currently performing related research which specifically addresses cache effects due to migration.

4.5 Thread control policies – Hiding long latencies

We consider a variety of policies for coordinating the motion of threads within the processor. The shared-thread control unit implements these policies, relying on status messages from individual cores for samples of performance data as well as notification of long-memory stall detection. For the sake of simplicity, these policies are centrally managed; more advanced implementations could coordinate peer-to-peer directly between cores.

We consider two distinct classes of policies, intended to exploit different opportunities enabled by the STMP. One opportunity is the ability to move threads quickly enough to react to very short-term thread imbalances resulting from long-latency operations; the other is the opportunity to perform coarser-grained load-balancing, but at time scales much shorter than possible in an operating system.

The first group of policies focuses on cases where threads are more numerous than cores, and such that some of the (multithreaded) cores will be more heavily loaded than others. When a thread on a less-loaded core stalls for a long-latency memory access – as characterized in Section 3.3 – it leaves the execution resources on that core under-utilized relative to the more heavily-loaded cores; in effect, hardware contexts on a core experiencing a stall will become relatively "faster" while the stall is outstanding. These policies seek to exploit the decreased contention on cores experiencing memory stalls. This can happen when thread load is balanced evenly, but requires two threads to be stalled on the same core to create a significant opportunity for thread movement. Therefore, the load-imbalance phenomenon exploited in this section is best illustrated when there are 5–7 threads on the (eight-thread) processor. In such scenarios, some cores will have one thread scheduled, and others two threads. If a thread scheduled by itself experiences a full-memory-latency stall, its parent core essentially sits idle for hundreds of cycles, while other cores are executing two threads. Because a four-wide superscalar processor running two threads typically achieves less than twice the throughput as when running a single thread, this imbalanced load is likely inefficient. If we can quickly move a co-scheduled

thread to the temporarily-idle core, these two threads executing on separate cores may execute faster than they would when co-scheduled on a single core.

It should be noted in this section and the next, that we only show a subset of the policies considered. Typically, the shown policy is a tuned, effective representative of a class of similar policies.

The specific policies evaluated in this paper are:

Conflict - the *conflict* policy migrates threads away from cores with execution-resource contention, towards those which are experiencing many memory stalls. It conservatively prefers to leave threads in-place, minimizing interference with normal execution. When the system detects that a thread has stalled for a long-latency memory access, it considers the amount of execution-resource contention that the corresponding core has experienced recently (in the last 10,000 cycles; this is a parameter). If the conflict rate – the mean number of ready instructions unable to issue per cycle due to resource exhaustion – exceeds a threshold, the already-stalled thread is evicted and sent to wait at the inactive-thread store. When the corresponding memory access completes, the thread is sent for execution on the core whose threads have experienced the highest rate of long-memory stalls in recent history.

In order to decrease spurious thread movement, additional conditions apply: threads are returned to their previous core if the stall-rate of the candidate core does not exceed that of the previous core by a threshold (5%); the initial thread swap-out is suppressed if a pre-evaluation of the core selection criteria indicates that it's already scheduled on the "best" core; and, stalled threads are never moved away from an otherwise-empty core.

Runner - under the *runner* policy, one or a subset of threads are designated as "runners", which will be opportunistically migrated toward stall-heavy cores; non-runner threads do not move, except in and out of inactive-thread storage, and back to the same core. This policy recognizes the high importance of processor affinity: most threads will run best when returning to the same core and finding a warm cache. The "runner" designation is used in the hope that some applications will inherently suffer less performance loss from frequent migrations than others.

This policy attempts to negotiate some of the more difficult tradeoffs facing these policies. If we seldom move threads between cores, then when we do move them, they tend to experience many L1 cache misses immediately as they enter a cold cache. If we move threads often enough that cold caches are not a problem, we are then asking all L1 caches to hold the working set of all threads, which puts excessive pressure on the L1 caches. By designating only a few threads as runners, we allow those threads to build up sufficient L1 cache state on the cores they visit frequently, decreasing the cost of individual migrations; additionally, each core's L1 cache now need hold the working set of only a few threads.

When a non-runner thread is detected as having stalled for a long-memory operation, the shared-thread scheduler decides whether to move the runner from its current location to the stalling core. Experimentally, interrupting the runner in order to migrate it whenever a long stall is detected on another core proved too disruptive to its forward progress; the results we present use a more relaxed policy, wherein the shared-thread scheduler records long-stall detection events, and when the runner itself stalls for a memory access, it is opportunistically migrated to the core with the most recent non-runner stalls. This decreases the amount of idle time which a runner may exploit for any one memory stall, but mitigates the performance impact on the runner itself. Over time, the runner gravitates towards those applications which stall for main memory most often.

While this policy is unfair over the short term to the runner threads – which suffer more than the threads pinned to particular cores – this unfairness can be mitigated by rotating which threads serve as runners over larger time intervals.

Stall-chase - the *stall-chase* policy aggressively targets individual memory stalls. When a thread is detected as having stalled for a miss to memory, active threads on other cores are considered for immediate migration to the site of the stall. The results presented use a policy which selects the thread with the lowest recent sampled IPC for migration; other selection policies we've evaluated include choosing the thread with the smallest L1 cache footprint, the thread which has has been running the longest since its last migration, and choosing a thread at random. The *stall-chase* policy is similar to the *runner* policy in that long-memory stalls "attract" other threads; they differ in the selection of the thread to be migrated.

For this set of policies we do not demonstrate the cases where the number of threads is less than the number of cores (this becomes simply multi-core execution) or where the number of threads is greater than the number of SMT contexts. In the latter case, the STMP architecture clearly provides gains over a conventional architecture, but we do not show those results because they mirror prior results obtained by BMT [19]. Consider the case where we have 12 threads; a reasonable setup would have each core executing a set of three threads over time, in round-robin fashion, using the BMT mechanisms to swap an active thread for an off-core thread when an L2 miss is encountered. There would be no immediate advantage in re-assigning threads to different cores when a thread stalls for a miss, because each core already has threads available to tolerate that miss.

On the other hand, we may want to re-balance those threads occasionally, to ensure that we are running a complementary set of applications on each core; those rebalancings are best done at a much coarser granularity than the polices described above, which target the covering of individual misses. Such rebalancing mechanisms are described in the next section.

4.6 Thread control policies – Rapid rebalancing

As stated previously, one advantage of the STMP over prior architectures, including BMT, is its ability to redistribute threads quickly between cores. Other architectures would rely on the operating system to make such changes; however, the operating system is not necessarily the best place to make such load balancing decisions. We advocate making some of these decisions in hardware, because in this architecture the hardware (1) has performance data indicating the progress of each thread, (2) has the mechanisms to context switch without software intervention, and (3) the cost of context switches is so low that it's economical to make them far more often than system software can consider. This rapid rescheduling allows system to (1) find a good schedule

more quickly initially, and (2) react very quickly to phase changes.

This group of policies considers, in particular, cases where there are more overall software threads than there are hardware execution contexts. These policies would also be effective when there are fewer threads than total contexts, as there would still be a need to consider which threads are co-scheduled and which run with a core all to themselves. We focus on the over-subscribed case, which combines stall-by-stall round-robin movement of threads to and from a core, with occasional re-shuffling among cores at a higher level. We evaluate several thread re-scheduling policies for this over-subscribed execution mode.

Our baseline for performance comparison is a model where the cores still share thread state, enabling hardware-assisted thread swapping, but the threads are partitioned exclusively among cores, and this partitioning rarely changes (i.e., at the scale of OS time-slice intervals).

The re-balancing policies considered are:

Symbiotic scheduler - this policy alternates through two phases, a "sampling" phase and a "run" phase, evaluating random schedules for short intervals and then using the best of the group for much longer execution intervals. Prior work [11] has found this style of policy effective for single-core SMT execution.

In a sampling phase, a sequence of random schedules – in our experiments, 19 random schedules as well as an instance of the best schedule from the previous run phase – are evaluated for performance. Each schedule is applied in turn; first, threads are migrated to the cores indicated by the schedule. Next, the threads are left undisturbed (i.e. no cross-core migrations are scheduled) for one sampling period in order to "warm up" execution resources. (Threads may still be rotated through a single core, as in Balanced Multithreading [19], during this time.) After the warm-up time has expired, the threads are left undisturbed for another sampling period, during which performance counters are collected. The counters are evaluated, then sampling continues at the next candidate schedule. The entire sequence of scheduling, warming up, and measuring each candidate in turn, makes up one sampling phase.

After the sequence of candidate schedules is exhausted, a "run" phase begins. The shared-thread control unit chooses the schedule from the sampling phase with the best observed performance, applies it, and then leaves that schedule to run undisturbed for much longer than the duration of an entire sampling phase (20 times longer, in our experiments). Afterward, the next sampling phase begins.

Medium-range predictor - this policy incorporates performance observations collected over many sampling periods, considering the correlation of overall performance with the pairs of applications that are co-scheduled in each. These observations are summarized in a compact data structure, a table from which predictions can be generated about arbitrary future schedules.

Simplified, the schedule performance predictor operates by maintaining a matrix that, with application IDs as coordinates, indicates the mean performance measured across all samples when those applications were resident on the same core. The predictor takes as input (*schedule*, *performance*) tuples of performance observations; the overall performance is added to the matrix cells corresponding to each pair of co-scheduled applications. A second matrix is maintained with sample counts to allow for proper scaling. Any candidate schedule can be evaluated against these matrices to yield a performance estimate. The matrices may also be continuously aged over time, so that more recent observations carry more significance than older ones.

To utilize this data, we've developed a straightforward greedy algorithm which can synthesize a new schedule that, when evaluated in the context of past measurements, yields a forecast performance that tends toward optimal. (Optimality bounds are not established.) Note that this scheduler doesn't guarantee near-optimal performance; in a sense, it runs the performance predictor in reverse, directly constructing a new schedule which the predictor considers to be good, tying the overall resulting performance to the accuracy of the predictor. The greedy schedule synthesizer constructs a schedule from the matrix of aggregated performance measurements by starting with an empty schedule, then successively co-scheduling the pair of applications which corresponds to the next-highest performance point in the matrix. (Several additional conditions apply, to ensure reasonable schedules result, to account for applications scheduled solo, etc.)

One additional use for the greedy schedule synthesizer is to generate schedules which represent the least-sampled portions of the schedule space. This is accomplished by creating a performance-sample matrix with each element set to the negative of the corresponding element in the sample-count matrix, and running the algorithm on that; since the greedy algorithm attempts to maximize the resulting expected-performance sum, it selects the near-minimal (negated) sample counts.

The *medium-range predictor* operates by using the greedy schedule synthesizer to construct a good schedule. This schedule is run for several sampling periods – 20, in our experiments – followed by an alternate schedule to ensure diversity in the measurement space. We generate alternate schedules either at random (*mrp-random*), or specifically targeting the least-sampled portion of the schedule space (*mrp-balance*).

5. METHODOLOGY

We evaluate the Shared-Thread Multiprocessor and a variety of scheduling policies through simulation, using a modified version of SMTSIM [15], an execution-driven out-of-order processor and memory hierarchy simulator. Starting with a multi-core version of SMTSIM implementing a system as described in Section 3, we have implemented the STMP mechanisms and policies described in Section 4.

We do not simulate the computation needed to implement the on-line thread control policies; however, these policies utilize very simple operations on small numbers of performance samples, which we expect could be performed with negligible overhead.

5.1 Simulator configuration

We assume a four-core multiprocessor, with the execution cores clocked at 2.0 GHz; for ease of accounting, all timing is calculated in terms of this clock rate. Table 1 lists the most significant of the baselines system parameters used in our experiments. While the 500 cycle main-memory access latency we specify would be high for a single-core system with two levels of caching, it's not unreasonable for a more complex system: in separate work, we've measured best-

Parameter	*Value*
Pipeline length	8 stages min.
Fetch width	4
Fetch threads	2
Fetch policy	ICOUNT
Scheduling	out-of-order
Reorder buffer	128 entries
Integer window	64 insts
FP window	64 insts
Max issue width	4
Integer ALUs	4
FP ALUs	2
Load/store units	2
Branch predictor	8 KB gshare
BTB	256-entry, 4-way
Cache block size	64B
Page size	8 KB
Cache replacement	LRU, write-back
L1 I-cache size/assoc.	64 KB/4-way
L1 D-cache size/assoc.	64 KB/4-way
L1 D-cache ports	2 read/write
L2 cache size/assoc.	8MB/8-way
L2 cache ports	8 banks, 1 port ea.
ITLB entries	48
DTLB entries	128
Load-use latency, L1 hit	2cyc
Load-use latency, L2 hit	13cyc
Load-use latency, memory	500cyc
TLB miss penalty	+500cyc

Table 1: Architecture Details

case memory access latencies of over 300 cycles on four-way multiprocessor hardware, with cores clocked at 1.8 GHz.

5.2 Workloads

We construct multithreaded workloads of competing threads by selecting subsets of the SPEC2000 benchmark suite. Starting with a selection of the 16 benchmarks as detailed in Table 2, we construct workloads for a given thread count by selecting subsets of the suite such that each subset contains the desired number of threads, and so that all subsets with a given thread count, when taken together, yield roughly the same number of instances of each benchmark. The 16 benchmarks were chosen arbitrarily to allow for this convenient partitioning, without requiring an excessive number of simulations to evenly represent individual benchmarks.

Table 3 details the composition of the workloads used in this study. We simulate each multithreaded suite until

Name	*Input*	*Fast-forward* ($\times 10^6$)
ammp		1700
art	c756hel, a10, hc	200
crafty		1000
eon	rushmeier	1000
galgel		391
gap		200
gcc	166	500
gzip	graphic	100
mcf		1300
mesa	-frames 1000	763
mgrid		375
parser		650
perl	perfect	100
twolf		1000
vortex	2	500
vpr	route	500

Table 2: Component Benchmarks

ID	*Components*
5a	ammp,art,crafty,eon,galgel
5b	gap,gcc,gzip,mcf,mesa
5c	mgrid,parser,perl,twolf,vortex
5d	ammp,crafty,galgel,gcc,mcf
5e	mgrid,perl,twolf,vortex,vpr
5f	art,eon,gap,gzip,mesa
6a	ammp,art,crafty,eon,galgel,gap
6b	gcc,gzip,mcf,mesa,mgrid,parser
6c	mgrid,parser,perl,twolf,vortex,vpr
6d	ammp,eon,gcc,mesa,vortex,vpr
6e	art,crafty,galgel,gzip,mcf,perl
7a	ammp,art,crafty,eon,galgel,gap,gcc
7b	gzip,mcf,mesa,mgrid,parser,perl,twolf
7c	art,eon,gap,gzip,mesa,vortex,vpr
7d	ammp,crafty,galgel,gcc,parser,twolf,vpr
7e	gap,mcf,mgrid,perl,twolf,vortex,vpr
8a	ammp,art,crafty,eon,galgel,gap,gcc,gzip
8b	mcf,mesa,mgrid,parser,perl,twolf,vortex,vpr
8c	ammp,crafty,galgel,gcc,mcf,mgrid,perl,vortex
8d	art,eon,gap,gzip,mesa,parser,twolf,vpr
10a	ammp,art,crafty,eon,galgel,gap,gcc,gzip,mcf,mesa

Table 3: Composite Workloads

($\# \ threads \times 200 \times 10^6$) overall instructions have been committed.

This evaluation of our thread-migration system does not make special accommodations for different classes of workloads – e.g. soft real-time, co-scheduled shared-memory – though such workloads are not intrinsically excluded. We focus on the SPEC2000 suite as a source of diverse execution behavior largely to streamline experimental evaluation; extending the system to consider explicit information about alternate workload types is an interesting avenue for future research.

5.3 Metrics

Evaluating the performance of concurrent execution of disparate workloads presents some challenges; using traditional metrics such as total instructions-per-cycle (IPC) tends to favor any architecture that prefers individual benchmarks which exhibit higher IPC, by starving lower-IPC workloads of execution resources for the duration of simulation. Such a policy is unlikely to yield performance gain in a real system: in a practical system, lower-IPC workloads could not be deferred forever, but simulation involves much smaller time-scales which do not capture this. A metric which introduces a fairness criterion is desirable; this is explored in more detail in previous work [11, 16]. We report multithreaded performance in terms of *weighted speedup* (WSU), as defined in [11], and used frequently in other studies of SMT or multicore architectures:

$$\text{Weighted Speedup} = \sum_{i \in threads} \frac{IPC_{i,exper}}{IPC_{i,single}} \qquad (1)$$

Each thread's experimental IPC is derated by its single-thread IPC, executing on a single core of the given architecture, over the *same dynamic instructions committed during the experimental run.*

In addition to this weighted speedup metric, which is calculated after-the-fact from experimental output and high-resolution logs of each component benchmark's single-thread execution, the policies detailed in Section 4.5 require on-line estimates of multithreaded performance. IPC is a dangerous

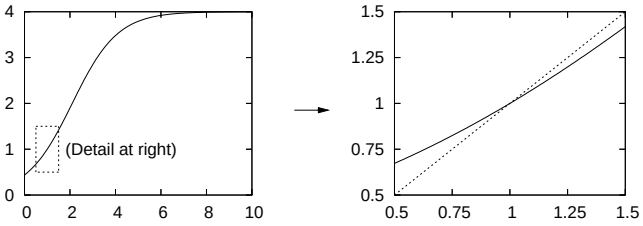

Figure 2: WSU compression function, $y = comp(x)$

metric on which to base runtime optimization decisions, for some of the same reasons that it is a misleading indicator of overall performance. We've experimented with several alternative runtime metrics; the results reported here perform on-line performance evaluation using a new metric, *interval weighted speedup*.

Weighted speedup as a goal function is not suitable for on-line decision-making, since it requires detailed single-threaded execution detail, $IPC_{i,single}$, which is not available at runtime; the challenge becomes finding a suitable basis for IPC samples which are available on-line, and which provide a reasonable baseline for the estimation of the changes in performance. For our experiments, we use the aggregate IPC of each thread over the entire previous round of scheduling as the basis – in place of $IPC_{i,single}$ – for evaluating the performance of the $IPC_{i,exper}$ samples taken during each sample interval. This strikes a balance, providing a measure of stability in values – particularly when comparing alternative schedules within a given round – yet still adapting over time to changes caused by earlier scheduling decisions.

We make one further modification to interval WSU: given a direct application of the weighted speedup Equation (1) over shorter time scales, it's possible for individual quotients which make up the overall sum to generate very large outputs, e.g. when the basis IPC for a an application is abnormally low due to execution conditions. It's important to prevent one such component from dominating the overall sum with very large values; while such a sample may cause only a short-term degradation when used for an individual scheduling decision, it can be disastrous when the aggregating *medium-range predictor* scheduler is used. We guard against this by compressing each thread's contribution to the interval WSU sum from $[0, \infty)$ down to the range $[0, 4]$, using a smooth sigmoid function which is nearly linear in the range near 1.0 (where samples are most frequent):

$$\text{Interval WSU} = \sum_{i \in threads} comp\left(\frac{IPC_{i,sample}}{IPC_{i,basis}}\right) \quad (2)$$

$$comp(x) = \frac{4.0}{1 + e^{(ln(3)+1-x)}} \quad (3)$$

Figure 2 shows the behavior of the range-compression function shown in Equation (3). Although used in this study, this function is not essential; linear scaling with clamping of values worked nearly as well.

Again, it is useful to distinguish the various metrics because of the close terminology. *Weighted speedup* is the metric we use to evaluate performance effectiveness. *Interval weighted speedup* is an online metric used by our architecture to evaluate or estimate the effectiveness of a schedule and make a scheduling decision.

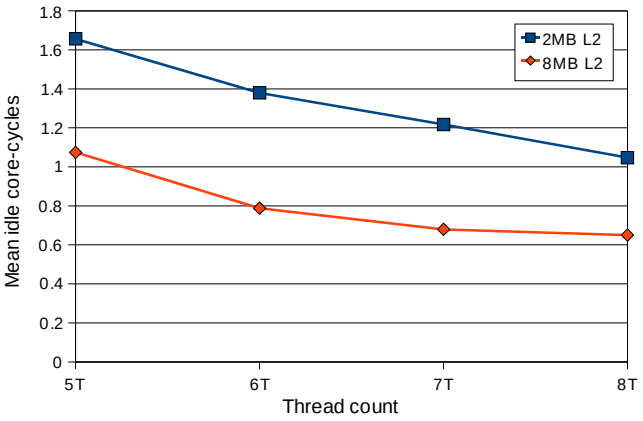

Figure 3: Total core-cycles (per processor cycle) spent effectively idle due to stalled memory instructions. Means are taken across all workloads for a given thread-count. Values above 0 indicate potential gains.

6. ANALYSIS AND RESULTS

In this section, we evaluate the performance of a *Shared-Thread Multiprocessor*, considering several distinct modes of operation. All results are reported for the hardware configuration as described in Section 5.1, unless otherwise noted.

6.1 Potential gains from memory stalls

This section demonstrates one axis of the opportunity for the STMP. Even a moderately-sized core with SMT support frequently experiences idle cycles where no execution progress is being made. We see in Figure 3 the total amount of time in which cores are effectively idle – with all active execution contexts unable to be used – due to long-latency memory operations, as described in Section 3.3. The values shown are the fraction of the overall execution time that each core spends completely idle, summed across all cores.

From this result we see that even when the full SMT capacity of the cores is used (8T), there remains significant idle execution capacity. Simply applying STMP, even without dynamic thread movement policies, should significantly reduce the idle cycles. Additionally, we see that the problem is more acute when not all of the cores are able to exploit SMT. We see that in many instances, the equivalent of an entire additional "virtual core" is available, if we can find a way to effectively utilize the combined lost cycles.

6.2 Rapid migration to cover memory latency

This section evaluates the use of the STMP to rapidly migrate individual threads between cores in response to stalls for memory access.

Figure 4 shows the weighted speedups achieved by several different thread-migration policies, with the five-threaded workloads described in Section 5.2. (Recall that our processor has four two-way SMT cores, for a total of eight execution contexts.) We use the five-thread case to exercise our policies because it is a region where we expect to see frequent imbalance.

For the sake of comparison, we also evaluate each workload under all possible static schedules, wherein threads do not migrate during execution. The *best-static* result shows the highest weighted speedup achieved by any static sched-

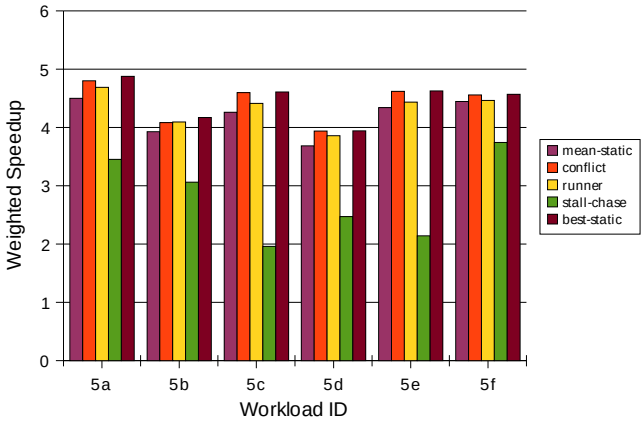

Figure 4: Performance of several stall-covering schemes on five-thread workloads.

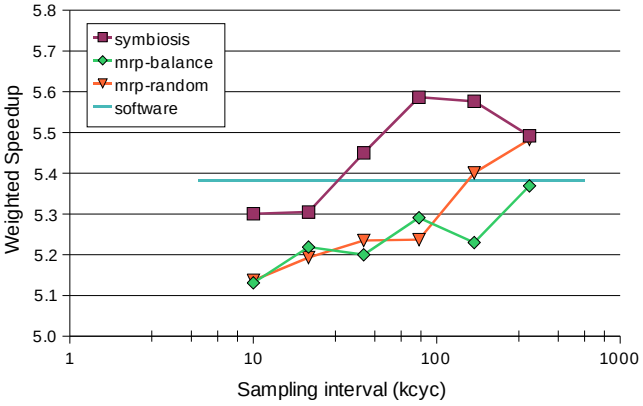

Figure 5: Performance impact of several dynamic-scheduling algorithms on a ten-thread workload, versus sampling interval.

ule for each workload. *mean-static* shows the arithmetic mean weighted speedup for each workload over all static schedules; it reflects the expected performance from assigning threads to cores at random, then not moving them during execution.

The *best-static* result represents a reasonable goal for our policies, as it is unattainable without oracle knowledge. In theory, we could beat *best-static* by exploiting dynamic changes in the workload, but none of these results achieve that. *mean-static* represents a reasonable baseline, as it represents what an OS scheduler might do, given the STMP hardware but no useful information about thread grouping. In these results, weighted speedup is computed relative to single-thread execution. Thus, for a five-thread, four-core configuration, a weighted speedup (WSU) of 4.0 is the minimum expected, and a WSU of 5.0 would indicate we're achieving the performance of five separate cores. Thus, the range of improvements we are seeing in these results is definitely constrained by the limited range of possible WSU values.

The *conflict* policy performs very close to the oracle static scheduler. This policy succeeds because it inhibits switching too frequently and it targets threads known to be on oversubscribed cores for movement. Because it only moves these threads, the instances where movement significantly degrades progress of a thread are lessened: the selected thread was struggling anyway.

We see that the *runner* policy performs significantly better than *stall-chase*. The two schemes are similar, with the primary difference that the former requires no core's L1 cache to support the working set of more than two threads at a time, significantly mitigating the lost L1 cache performance observed for the latter policy. Although *runner* provides performance gains in all cases over the baseline, it always suffers slightly compared to the less aggressive *conflict* because the frequency of ineffective migrations is still high.

Thus, we demonstrate two schemes that show improvement over a conventionally multithreaded chip multiprocessor. *conflict*, in particular, allows the processor to match the performance of an ideal oracle schedule: one which perfectly places threads in the optimal core assignment, with no overheads associated with identifying the optimal assignment at runtime. This result is very significant, because it shows that it is indeed possible to support multithreaded-style thread context-sharing between distinct CMP cores; in fact, they can share contexts quickly enough to even hide frequent memory latencies.

6.3 Rapid migration for improved scheduling

This section explores the other expected benefit of the STMP architecture, the ability to do frequent re-evaluation of the thread-to-core assignment. These results focus on policies which migrate threads between cores much less frequently than those of the previous section, but potentially far more frequently than could be done by system software alone. Figure 5 shows the performance of some of the policies discussed in Section 4.6.

The baseline software scheduler – which uses our best symbiotic scheduling mechanism, but at a timescale possible in software – is shown as the horizontal line. It should be noted that the baseline makes use of our STMP architecture, e.g. allowing three threads to share two SMT contexts an a core, and it uses our best mechanism to re-evaluate schedules; the baseline only lacks the ability to re-evaluate schedules at a rate faster than OS time-slice intervals.

We see that the symbiotic scheduler is able to achieve some gains over the software scheduler for intermediate levels of rescheduling. The more often we re-sample and re-schedule the threads, the quicker we are able to react to changes in the workload.

However, the reason that the results peak in the middle is that there are offsetting costs to rescheduling too frequently. The first reason, which our results have shown to be less of a factor, is the cost of migrating threads (primarily the cold cache effects).

The second, more important but less obvious factor, is our reliance on performance predictions, which become inherently less reliable as the sampling intervals shrink. All of our schemes depend on some kind of predictor to estimate the schedule quality. For the symbiotic scheduler, the predictor is a direct sample of each schedule; for the others, they use predictors based on some kind of history. The inherent advantages of rescheduling more quickly are offset by the inaccuracy of the predictor. For example, the shorter our sampling intervals in the symbiosis scheduler, the more noisy and less reliable the sample is as a predictor of the long-term behavior of that schedule.

The *medium-range predictor* provided more accurate predictions of future performance. While predictions proved relatively accurate, it did not outperform the much simpler *symbiotic scheduling* policy; the predictor-based scheduler still tended to schedule more thread movement than the symbiosis policy, losing performance to scheduling overhead. Of the two predictor-based policies shown, *mrp-random* tended to marginally outperform *mrp-balance*. This is counterintuitive: both policies explicitly deviate from the current "expected best schedule" in order to explore new ground, with the latter explicitly targeting the least-sampled regions of the schedule space; one would suspect that *mrp-balance* would thus be more successful at introducing useful diversity at each step. However, as the predictor-based scheduler learns more about the performance correlation between threads, it naturally begins to avoid co-scheduling threads which don't perform well together. Over time, then, some of the the under-sampled regions of the sample space are avoided specifically due to bad performance, and the *mrp-balance* scheduler can force execution back into these regions. The *mrp-random* scheduler has no such bias.

These results demonstrate that with a very simple hardware re-scheduler, there is clear value to load-balancing and re-grouping threads at rates significantly faster than possible in a software scheduler. These results indicate that in this type of aggressive multithreaded, multicore processor, an operating system is no longer the best place to make all scheduling decisions.

7. SUMMARY AND CONCLUSIONS

This paper presents the Shared Thread Multiprocessor, which extends multithreading-like interaction to distinct processor cores on a chip multiprocessor. We show that, beyond the advantages shown in previous work, where additional external thread storage improves the performance of a single SMT processor, the STMP can be exploited in two unique ways.

First, when long-latency memory stall events create transient load imbalances, we can improve overall throughput by quickly moving a thread between cores to exploit otherwise idle resources.

At a coarser time scale, we can repartition the mapping of threads to cores very quickly and efficiently, in order to re-evaluate scheduling decisions and react to emerging application behavior. While sampling too quickly tends to render scheduling decisions unreliable, there exists a middle ground at which it is profitable to make decisions more quickly than possible with software mechanisms alone.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF grant CCF-0702349 and Semiconductor Research Corporation Grant 2005-HJ-1313.

8. REFERENCES

[1] Arvind. Data flow languages and architecture. In *Proceedings of the 8th International Symposium on Computer Architecture*, 1981.

[2] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, 39(3):300–318, 1990.

[3] J. Clabes, J. Friedrich, and M. S. et al. Design and implementation of the Power5 microprocessor. In *International Solid-State Circuits Conference*, 2004.

[4] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News*, pages 80–91, 2005.

[5] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti. Design of the Power6 microprocessor. In *International Solid-State Circuits Conference*, Feb. 2007.

[6] T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit power efficient SPARC SoC (Niagara2). In *Proceedings of the 2007 International Symposium on Physical Design*, 2007.

[7] R. Katz, S. Eggers, D. Wood, C. Perkins, and R. Sheldon. Implementing a cache consistency protocol. In *12th Annual International Symposium on Computer Architecture*, pages 276–283, 1985.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. In *IEEE MICRO Magazine*, Mar. 2005.

[9] D. Koufaty and D. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, pages 56–65, April 2003.

[10] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, 2000.

[11] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[12] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[13] K. Stavrou, C. Kyriacou, P. Evripidou, and P. Trancoso. Chip multiprocessor based on data-driven multithreading model. In *International Journal of High Performance System Architecture*, 2007.

[14] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[15] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[16] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[17] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[18] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[19] E. Tune, R. Kumar, D. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *34th International Symposium on Microarchitecture*, pages 183–194, Dec. 2004.