Multithreaded Instruction Sharing

Mark Dechene, Elliott Forbes, and Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{mjdechen, jeforbe2, ericro}@ece.ncsu.edu

Technical Report
December 3, 2010

Abstract

We show that when multi-threaded benchmarks are executed on a Chip Multiprocessor (CMP), the threads typically execute identical instructions at nearly the same time. When multiple threads are all executing identical instructions (same PC, same source operands, and same source values) at nearly the same time, we recognize that the computation can be performed by one thread, and the results can be shared with the other threads, saving critical execution resources and bandwidth for other instructions.

We study these thread properties, and evaluate a hardware implementation that recognizes and exploits instruction-similarity. In our experiments, we find that for one thread of a multi-threaded benchmark, about 20% of instructions are identical to nearby instructions in other running threads. Evaluation of our proposed sharing techniques on a high throughput, in-order, Simultaneous Multithreaded architecture achieves a 10% mean (32% peak) increase in processor throughput. As with most performance enhancing techniques, the performance is achieved at the expense of additional power. In Instruction Sharing the increase in per-core power is about 12%, which is due in part to hardware modifications but also to higher utilization of existing hardware. The core area is increased by approximately 15%, of which 13% is due to increasing core contexts. 2% area increase is due to banking the register file, adding match logic, and adding small associative match tables.

1. Introduction

In this work, we propose Multithreaded Instruction Sharing (MIS), a mechanism to increase throughput of a processor core when different threads of the same parallel application are executing in tandem. As more threads of the same program are added to a single SMT core, it becomes more likely that two threads are executing the same instructions – possibly with identical data – at the same time. MIS exploits this property to boost instruction throughput.

Current multi-core processor configurations span an architectural spectrum. On one side of the spectrum, machines have a relatively small number of threads, but the performance of a single thread is high. Throughput processors, on the other end of the spectrum, have a high number of threads at the expense of the performance of any one thread. Simultaneous Multithreading (SMT) is a microarchitectural feature that enables high efficiency in throughput architectures. In SMT, thread contexts provide resiliency against events that typically degrade performance, such as stalls due to branches and cache misses. While one thread stalls to resolve the performance degrading event, other threads continue to execute instructions and make forward progress. Without SMT, or with too few threads, during these events instructions are unable to issue and execution resources are under utilized. Therefore, both throughput and efficiency are direct effects of provisioning processing cores with sufficient contexts.

Throughput processor cores have resources provisioned by first determining the processor core backend width. The backend width directly dictates the register read, execute, bypass, and register write widths – it is a first-order design choice constrained by processor frequency targets. After determining the backend width, the fetch bandwidth and the number of SMT contexts are separately increased until the execution unit utilization remain saturated with instructions. Once execution units are fully utilized, there is no benefit to increasing fetch bandwidth or adding additional thread contexts.

Generally, fetch bandwidth and context count can be increased if there is a performance benefit in doing so. While increasing fetch bandwidth or context count may increase the pipeline depth of the processor, these enhancements do not impact the critical back-to-back execution loop. Further, increasing either fetch bandwidth or context count naturally makes throughput processor cores more tolerant to increased pipeline latencies, such as a longer fetch to branch resolution loop. In single threaded applications, increasing fetch bandwidth beyond a single cache line requires exotic hardware. However, Kumar et al. [13] note that parallel applications often have threads fetching the same static instruction, hence multiple contexts can consume the result of a single instruction cache access. Thus, we contend that (1) increasing fetch bandwidth is largely an inexpensive design optimization for parallel applications, and (2) increasing contexts may increase the pipeline depth due to additional register storage requirements, but it also adds latency tolerance, so the added depth has negligible effect.

Our MIS architecture is built onto a throughput core modeled after the Sun Niagara [3] [12] pipeline. After instructions issue, they are decoded and their register source operands are read from the register file for that thread. In MIS, the same source registers are read from the register file of every other thread in the core. Since MIS is built on a pipeline with no register renaming, this is a logically trivial operation – the same register identifiers are presented to each register file. In parallel with execute, the source register operands in the instruction are compared with the register state of all other threads. If the register state matches, the instruction can be shared with the matching context – provided an instruction with the same PC is fetched by the matching context and reaches the head of the instruction issue buffer before the matching source registers are killed.

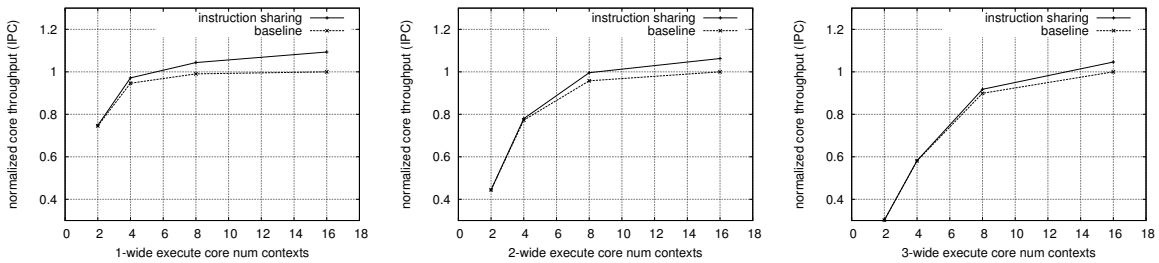

Fig. 1. Throughput impact of adding contexts to a 1-wide execute core, a 2-wide execute core, and a 3-wide execute core – both with and without multithreaded instruction sharing. Throughputs normalized to the 16-context baseline case.

Matches are placed into small, associative match buffers. As instructions issue, the logical destination register identifier is used to invalidate match table entries that may no longer be valid. This prevents future instructions from sharing data produced with different source values. Similarly, the addresses of stores and L1 data cache evictions are used to invalidate matched loads that may no longer be consistent. When a new instruction reaches the head of the instruction buffer, before issue, the match table is indexed with the instructions PC. If the instruction hits an entry in the match table, the stored result value is written back into the register file for that context. The consumer instruction is retired from the head of the instruction buffer without issuing, without reading the register file, and without consuming execution pipeline resources.

We show that MIS is enabled by key properties of SMT to provide an efficient, implementable design that increases core throughput without adding execution resources or increasing bypass pressure. Importantly, MIS provides a means whereby increasing fetch bandwidth and core contexts can improve throughput beyond the point where execution resources have been saturated.

MIS is similar to Sodani's Dynamic Instruction Reuse (DIR) [26], with the following key differences. DIR is a technique for reusing identical instructions within a single-threaded application, while MIS is a technique to share instructions across threads in parallel applications. In DIR, a value producing instruction writes data into a reuse table, including source values or logical register dependencies – consuming instructions read their register values or logical register names and test the table to determine if results can be reused. In MIS, value producing instructions proactively perform match tests on other threads in parallel with execute. Upon successful match tests, the execution result and invalidation data are written into a match table for the matched thread. Since DIR is reusing within a thread, identical instructions are found many cycles apart. Thus, DIR may need to buffer results for a significant period of time in a large reuse table. Conversely, MIS is sharing across threads – identical instructions are frequently found in the pipeline at the same time. Thus, MIS must only buffer results for a short period of time in small match tables.

The performance of MIS lies in the ability to recognize when instruction results from one thread can be used by other threads. Threads that consume instruction results from other threads can non-speculatively retire those matching instructions without executing them. The execution resources that would have been used for those matching instructions can instead be used for instructions that are unique. The key benefit of SMT style pipelines is that any thread can quickly use the freed execution resources, since co-scheduled threads can execute independent of other threads and without any scheduling latency.

This work contributes a study of program behavior that functionally simulated parallel benchmarks in lock-step. Instructions in each thread were scanned against instructions in all other threads, up to a certain history depth. We found that, depending on the number of threads and the length of history, roughly 20% (up to 50%) of instructions are identical (same instruction, same data) to at least one instruction in another thread.

A cycle accurate, execute-at-execute, cache coherent, multicore simulator was developed to evaluate MIS on various processor configurations. A single-core, in-order 1-wide, 16-context SMT processor using MIS techniques achieved a mean speedup of 10%, with several individual benchmarks achieving speedups of 15% - 32%, over a baseline processor without MIS techniques. This performance comes with an increase in power. We find that with the above configuration, the power increases up to 12%, (peak increase of 23%). Furthermore, this increase doesn't strictly correlate with speedup. For benchmarks that are not significantly impacted by sharing, the optimization can be disabled to avoid the power cost. We estimate the core area cost to be 15%, of which 13% is due to increasing the number of SMT contexts from 4 to 16.

Finally, as Figure 1 illustrates, MIS achieves the goal of enabling additional contexts to increase core throughput – beyond the point where execution units become saturated without MIS. This data was collected using our timing simulation with 4 match table entries.

The rest of the paper is organized as follows. Section 2 provides background and discusses related work. Section 3 presents a program behavior study illustrating the degree to which instructions may be shared for parallel benchmarks that are functionally simulated in lock-step. Section 4 proposes a microarchitecture to exploit MIS. Section 5 details our simulation methodology. Section 6 presents results. Section 7 concludes.

2. Background

Since the initial proposals for Simultaneous Multithreading, there has been significant work to increase its performance and efficiency. Several bodies of work focus specifically on the pressure imposed on various resources of an SMT design. Several of these proposals attempt to either merge hardware resources such as 1) function units by time-multiplexing the use of the unit, or 2) caches by using common cache blocks among threads [5] [13]. Other proposals partition structures to ensure fair usage by all threads [21] [28]. Our approach does not consider concurrent resource usage, necessarily, as a problem. Instead, we acknowledge that this sharing occurs and attempts to leverage the sharing for the benefit of all active threads. We share execution results, not the hardware.

To alleviate pressure on shared hardware, many research proposals avoid co-scheduling threads that tend to use the same resources [6] [8] [9] [24]. To increase performance, these proposals adjust the scheduling of threads to either offset similar threads from one another in time or to schedule different types of threads at the same time. The benefit manifests when threads alternatively use structures, ideally a thread will leave a phase that stresses one resource just as another thread enters a phase that needs that resource. MIS, conversely, does not try to avoid similar program phases of a parallel application. Indeed MIS works best when threads are executing very similar instructions by providing the opportunity to collapse execution of the instructions that are the same.

Instruction and trace reuse [16] [10] [25] [26] [18] [11] [7] are single-thread performance enhancements that avoid execution of an instruction or group of instructions. Instruction and trace reuse buffer the results of instructions or sequences of instructions as they execute. If the operands/live-ins of instruction(s) match a previous execution, then those instruction(s) can be collapsed and the live-out values can be written directly. These proposals are similar in spirit to MIS since they potentially obviate executing instructions. However, these reuse schemes suffer from 1) potentially high buffering requirements and 2) difficulty providing results before the instructions could have been re-executed. As will be discussed in section 4, MIS does not require large buffering (at the expense of missed opportunity should co-scheduled threads not have matching instructions) and has a match test performed by value producers on behalf of value consumers.

Squash reuse [22] [20] is an extension of reuse that specifically targets reusing instructions squashed due to a branch misprediction. This is an attractive proposition in large-window single-threaded processors, as it allows the pipeline to recover more quickly in the shadow of a misprediction, however, throughput processors do not build large windows, and may avoid predicting branches.

Collange, et al. [23] recognize a similar program property as that which we are exploiting in MIS. The application class for which they measure Affine and Uniform vectors is in GPGPU codes. Of particular interest in their work, Uniform vectors are single-instruction multiple-data (SIMD) instructions in which the elements of the instruction all have the same value. Although they implement no specific architecture, they measure the scale of Uniform vector usage in GPGPU codes and suggest that by only performing parts of the vector calculation there can be a power savings. Our work differs in two key ways. First, the underlying programming model of GPGPU codes forces the alignment of data into vectors. Thus, the programmer is exposed to, and is expected to utilize, this property for the best performance. Our proposal dynamically detects the alignment of instructions without an exotic programming model; our benchmarks are simply multi-threaded codes. Second, we propose an architecture that exploits the alignment of instructions for performance. SIMD instructions require lock-step execution of all elements of the data vector. In the event that Uniform vectors are executed, the GPU [15] [17] architecture can not fill unused execution resources with subsequent instructions. This ability of threads to execute independent of each other is crucial to MIS, since unused execution resources can be used for the unique computation from any thread. Additionally, MIS has a benefit not possible in Uniform vectors in that Uniform vectors require all elements in a vector to be equal (or follow a strided pattern for their Affine vectors) for any benefit. In MIS, we minimally require two matching instructions for a performance benefit.

3. Program Behavior

To determine the potential of MIS we devised a series of studies to measure the properties of multi-threaded programs. First, we determine how often instructions from a thread match instructions from any other thread. With these instructions identified, we then measure several of the properties which characterize their nature. These studies remove any effects imposed by any particular implementation details and measure the sharing properties inherent to the benchmarks.

3.1 Amount of Sharing

We use a trace-driven simulator that maintains a queue for each thread holding the instruction information (opcode, operands, etc) for the recent history of the thread. Once all instructions for the current cycle have been added to the head of the queue, then for each instruction executed during that cycle a comparison is made to all of the instructions in each other thread queue. Matching instructions are those for which the instruction opcode, all operands, and the operand values are the same. These instructions are identical dynamic instances of computation that appear in two different threads at nearly the same time.

Note that, for this series of measurements, stores are not considered when scanning for matching instructions, and the queue entry for stores will be marked invalid. NOP and prefetch instructions are also invalidated and not counted in these experiments.

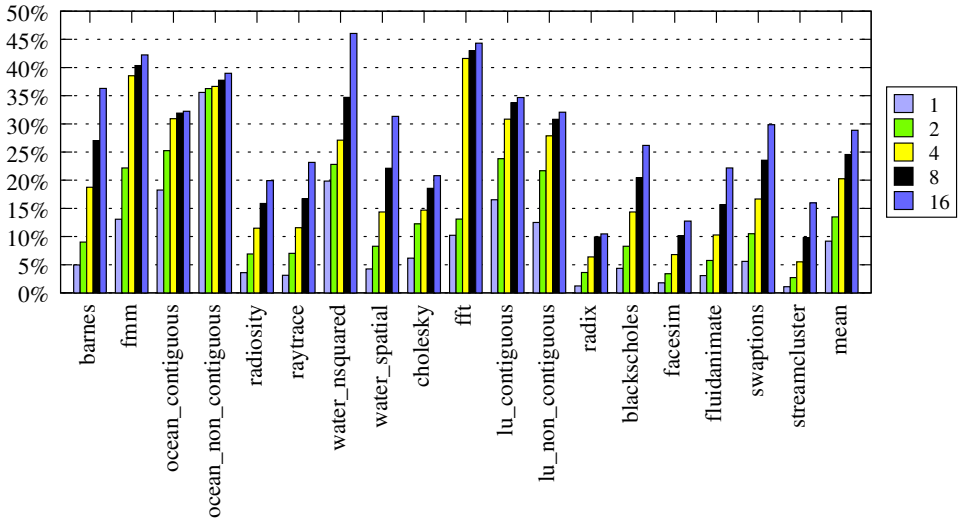

Fig. 2. Instruction matches when holding number of threads at 8 while varying the buffer depth. For each benchmark, the buffer depth from left to right is 1, 2, 4, 8, and 16.

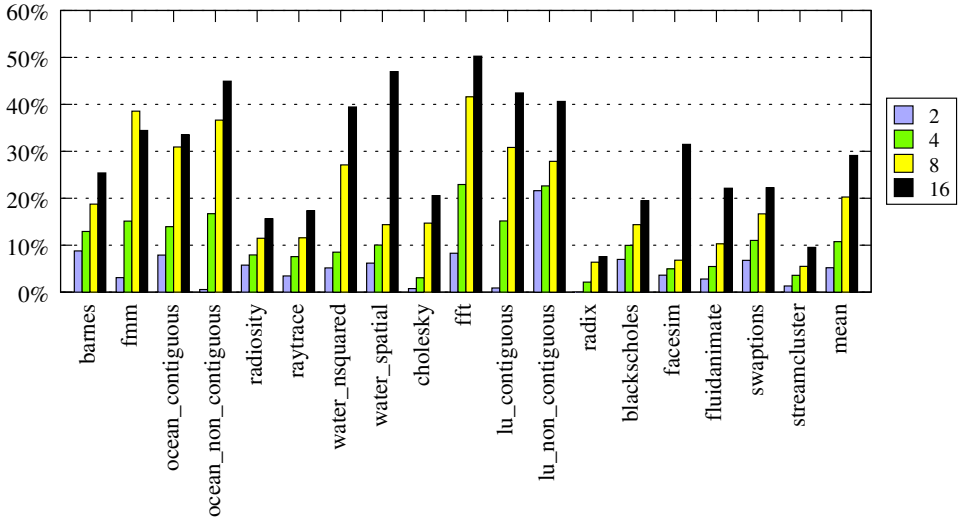

Fig. 3. Instruction matches when varying the number of threads while holding the buffer depth at a constant depth of 4. For each benchmark, the number of threads from left to right are 2, 4, 8, and 16.

Loads, branches and other control instructions, and all arithmetic (including floating-point) instructions are considered for potential matches.

Figure 2 and 3 show the results of these experiments. Each bar of these graphs shows the percentage of all instructions for one thread that match an instruction either within the same thread or in at least one other thread. The groups of bars for each benchmark hold the number of threads constant at 8 threads and vary the amount of history maintained for each thread (Fig. 2), or hold the history depth the same at 4 instructions and vary the number of threads (Fig. 3).

We notice that there is a significant percentage of instructions that match instructions from other threads. Figure 2 shows that as instruction history is increased, more instructions are able to find matching instructions from other threads. This follows naturally, since the opportunity for matching instructions increases as the number of instructions tracked increases. Figure 3 shows that as the number of threads increases, so does the percentage of matching instructions. This means that as the number of threads increase, it is more likely that another running thread happens to be executing in a similar region of code. Parallel benchmarks re-align the code region of their threads at barriers, so this conclusion is also not too surprising.

What perhaps is surprising is the extent to which instructions are shared. Depending on the number of threads and the amount of buffered instruction data, a thread can expect to find on average between 5% and 29% of its instructions have recently executed in other threads. Instruction reuse similarly measures a high number of instructions reusing register inputs. But in the threaded applications that we are targeting, the shared instructions are very close together temporally – a maximum

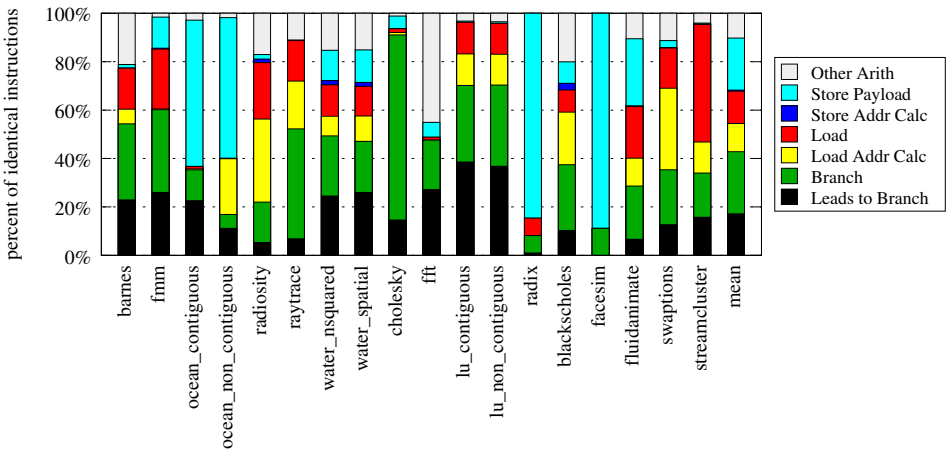

Fig. 4. Breakdown of shared instructions. The possible instruction types are based on either the instruction opcode, or the instructions dataflow contribution.

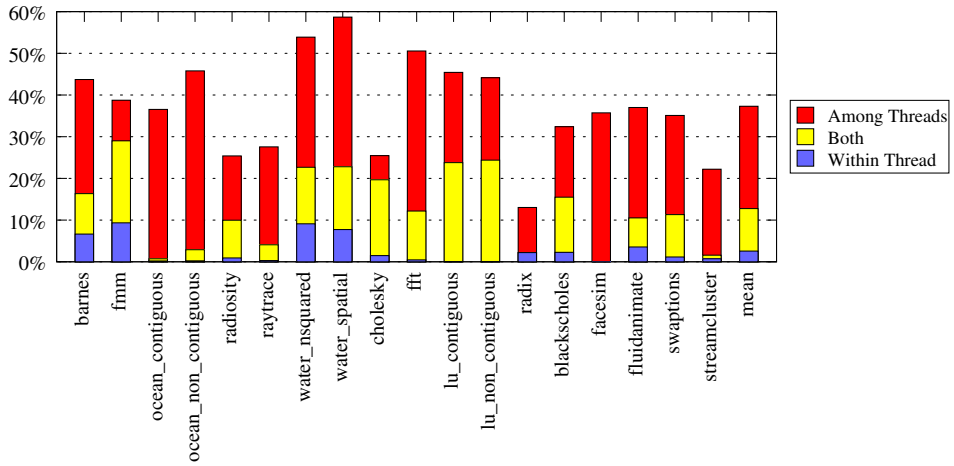

Fig. 5. The availability of matching instructions within a thread, among multiple threads, or both.

of 16 cycles distant in these experiments. Conversely, reuse is typically found in temporally distant dynamic instructions.

3.2 Sharing Characteristics

In this experiment, the simulator was modified to track the *types* of instructions that commonly match. The rationale is that simplifications to the MIS architecture may be possible if most sharing occurs between particular instruction types. For instance if only arithmetic instructions are shared, then MIS can ignore loads and the additional hardware required for them. In this experiment, once an instruction is determined to be identical to an instruction in any other thread, that instructions type is determined based on the following categories:

- Loads
- Branches (and other control instructions)
- Arithmetic instructions whose result is used for a load address
- Arithmetic instructions whose result is used for a store address
- Arithmetic instructions whose result is stored to memory (i.e. determines the store payload)
- Arithmetic instructions that determine branch conditions and targets
- All other arithmetic

Figure 4 quantifies these categories for a simulation of 16 threads with a history depth of 16 instructions. These results show that most instruction types are represented, particularly when considering all benchmarks. This discourages simplifying the hardware design to favor or track only certain types of instructions.

One final modification to the simulator determines how often an identical instructions match is within the same thread or in a different thread. Figure 5 shows the results of a 16 thread, 16 instruction history depth configuration. The three categories

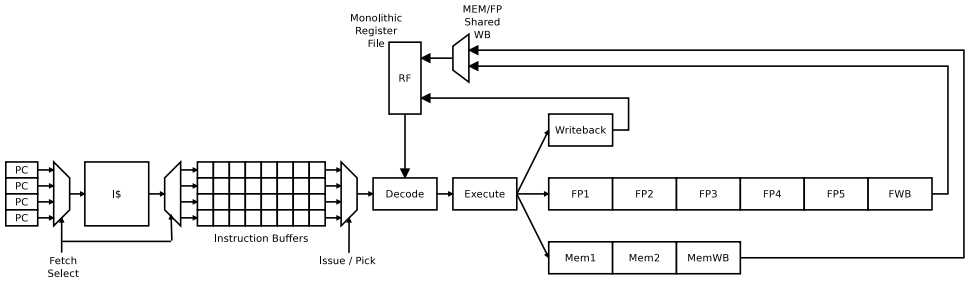

Fig. 6. Baseline throughput processor pipeline. The pipeline is modeled after the Sun Niagara (Ultrasparc) processor family. In this diagram, 4 threads are supported, although we model similar cores supporting 8 and 16 threads as well.

measure when an instruction matches another instruction found only in its own thread, found only in a different thread, or found multiple times both within its own thread and in a different thread. The outcome shows that instructions do occasionally match an instruction within its own thread, but a significant amount of matching also occurs between threads. This highlights the importance of a hardware architecture that can identify these cross-thread opportunities without incurring long detection or update latencies between threads.

4. Core Microarchitecture

To evaluate Multithreaded Instruction Sharing, we use an in-order throughput processor pipeline design, similar to the Niagara processor, as a baseline. The baseline is described in section 4.1.

Four key modifications are made to enable Multithreaded Instruction Sharing. First, the register file is split from a thread-shared monolithic structure to a set of smaller, thread-dedicated register files – to support concurrent access to registers in each context. Second, producer instructions perform a match test to identify contexts that are potential consumers. Matches are placed into small, associative, per-context match tables. Third, the match tables observe instructions in the decode stage of the pipeline to determine if match table entries need to be killed. Finally, a PC check is made against the each head instruction in the instruction buffer (IB). This check occurs in an efficient manner – the match table is checked once, when an instruction reaches the head of the IB, and subsequent writes into the match table for a context are directly checked against the PC at the head of the IB for that context. When an instruction at the head of an IB matches an entry in the match table, the value in the match table entry is written directly into the register file. These mechanisms are described in detail below.

4.1 Baseline Model

Our baseline model, as shown in Figure 6 consists of an in-order throughput-execution core. The pipeline is similar to that found in the Niagara processor family. We vary the number of thread contexts in a core between 4 and 16. Instructions are fetched in round-robin thread order without branch prediction. When a control transfer instruction is encountered, that thread is removed from fetch arbitration until execute resolves the PC of the thread. This mechanism prevents wasting fetch bandwidth in the shadow of a branch misprediction. Up to four sequential instructions are fetched from a single cache line each cycle.

Fetched instructions are placed into an thread-specific Instruction Buffer (IB) awaiting issue. Instructions are issued out of the IB in program order. Threads arbitrate for instruction issue using eldest-first arbitration. The oldest ready instruction in the IB will be selected for issue, provided that it will writeback after all earlier instructions in that thread. Short latency instructions are not allowed to bypass long latency instructions within a thread, unless the long latency instruction has no effect upon correct execution. E.g. instructions are allowed to bypass software prefetch instructions that have cache missed.

Up to one instruction per core is selected to issue per cycle. The instruction undergoes register read, then is diverted into the appropriate execution unit. After execution completes, the results write back into the register file and the instruction is considered to be retired.

NOP instructions (instructions that do not produce values or side effects) present in some benchmarks are fetched but bypass the IB, issue and execution stages and do not consume resources. This pipeline feature prevents NOPs from artificially inflating the sharing statistics since NOPs are generally easy sharing candidates, but also easily targeted by simpler pipeline enhancements.

4.2 Dedicated Per-Context Register Files

Instruction Sharing requires identifying all contexts that could potentially execute an instruction based upon their current register file values. This implies that a large number of register reads will need to be performed every cycle. To support these reads, without creating a large many-ported monolithic structure, the register file in Instruction Sharing is banked – one bank per context. Each bank of the contains two read ports and two write ports, similar to the number of read and write ports to the original monolithic structure.

Because the processor pipeline does not perform register renaming, the decoded logical registers can be efficiently applied to all register banks without a translation step. That is, all banks are read with the same physical register identifiers. The results of register read are sent to the match stage, which occurs in parallel with execute. Execute receives only the correct register values, for the context that was issued.

Banking the register file has an important additional effect used to write entries from the match table into the register file. One of the write ports into each register file is shared, via multiplexer, between the match table and the output of the execute stage. The output of execute has priority over the match table. That is, if an execute pipeline is writing a value into the register file, the match table may not commit a value that cycle. The match table is allowed to write into the register file while execute is processing other threads.

4.3 Match Test

In parallel with execute, the register values driven to the match stage are tested via comparators. In our pipeline, we implement two comparators per context – one comparator per potential source register. One input to each comparator is wired to a read port of a bank of the register file, while the other input is wired to the input to the execute stage for that register. While this reduces the complexity of the steering logic, one set of comparators will redundantly check the correct source register against itself.

If a match test was successful, a match table entry is allocated and populated for that context. As shown in Fig. 8, the match table entry contains a valid bit, LRU bits, the PC, the source register identifiers, the destination register identifier, a portion of the physical address (for invalidating inconsistent loads), and the result of the matched instruction.

As a power saving optimization, the match test can be restricted to a subset of instructions. In this work, we evaluated a reduction where only n-byte sign-extended sources could be matched. The value of n was varied between 1 (a single byte) and 8 (full 64-bit values). This directly reduces the size of the comparators, and can be used to implement a register file that reads a portion of the register instead of the full register when the read is being used for a match test instead of full execution.

4.4 Match Table Invalidation

The match table is designed to logically contain instructions that could be written back without reading the register file. To maintain this property, the match tables must observe instructions as they are decoded. When an instruction reaches decode, the destination register identifier is checked against the source registers in the match table corresponding to that thread. Any match table entries that use the destination register as a source are invalidated. Note that this is not a perfect scheme – entries may be invalidated on silent productions.

4.5 Match Table Load Invalidation

The match table also contains the result of load instructions. Match table entries containing load data act, effectively, as logical extensions of cache memory. Correctness requires that these match table entries maintain consistency with the processor cache. Match table load consistency is implemented in the following fashion. First, a portion of the shared load physical address is populated into the match table when the entry is written. Second, the match tables must observe the physical address of store instructions after address translation (to prevent aliasing from inducing correctness problems). This observation must occur for stores from all threads in the same core. Third, local L1 data cache evictions must also be observed by the match tables, to ensure coherency on a global scale.

Matched load instructions are not allowed leave the IB and writeback results until after store instructions in the same thread have been observed. This induces a one-cycle delay in the issue scoreboard for a thread. However, this delay only blocks core issue slots very rarely – due to SMT's inherent ability to tolerate instruction delays. Note there are no scoreboard stall conditions created by checking for stores in other threads impacting load issue, only stalls within a thread, despite the possibility that a store in the pipeline from a different thread will write to the address being loaded. In these situations, the load is still allowed to reuse the data during this brief window. Because the load and store are from different threads, the load remains sequentially consistent, appearing to execute the immediately before the store. Consistency between threads is ensured by the in-order pipeline computing all store physical addresses from one thread in order, and the match tables observing the addresses in the same order they were computed (and retired).

In our implementation, we use 26 bits of the physical address in the match table. There are two sources of needless eviction that this choice entails. First, the lower 6 bits have been removed to force match table eviction on cache line granularity. This decision was made to simplify the invalidation process when a data cache line was evicted, i.e. due to a remote core requesting write ownership of a line. This choice creates the possibility of false invalidation when a local store writes a different portion of a cache line as the address being shared. The upper 32 bits have been removed to reduce the size of the match table entry and to reduce the complexity of the invalidation test. This choice creates the possibility of a store or cache line eviction from a different region of memory aliasing to the shared load and thus evicting it needlessly.

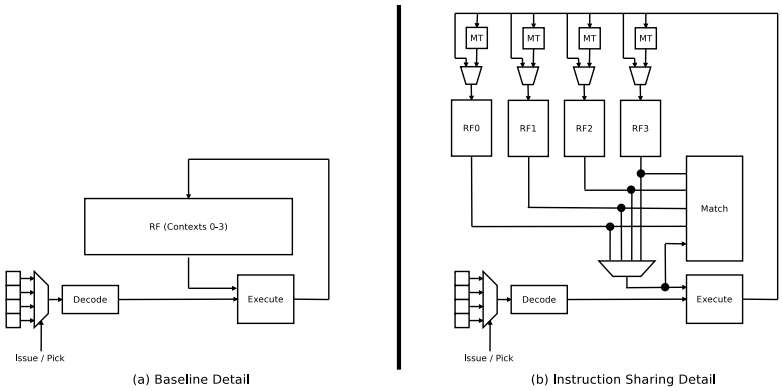

Fig. 7. (a) Detail of the baseline processor pipeline, illustrating portions of issue, decode, execute, and writeback. (b) Detail of proposed modifications to support instruction sharing. The monolithic register file has been split into context-specific register files. A match stage has been added to compare the contents of registers in each context for each issued instruction. Small fully-associative match tables have been added.

1 bit	2 bits	31 bits	6 bits	6 bits	6 bits	64 bits
Valid	LRU	PC	RSRC	RSRC	RDST	Result Value

Fig. 8. Contents of each match table entry. The PC of the matched instruction is used by the instruction buffer to identify when a match table entry can be committed to the register file. The two rsrc and the load address are used for invalidation. The rdst in the entry indicates the register number corresponding to the stored result value.

4.6 Match Table Writeback

The first cycle that an instruction is at the head of the IB, the match table for that context is checked for an entry with the same PC as the ready instruction. If the instruction hits the match table, the matching thread is taken out of contention for issue. Rather, the instruction schedules a writeback from the match table into the register file at the earliest possible correct time. That is, to maintain proper in-order execution semantics, the instruction cannot write into the register file until the first cycle after all prior instructions from this context have written into the register file. While this is occurring, other threads may continue to write into their register files without contention.

If the instruction did not hit the match table during its first cycle at the head of the IB, it does not check the match table in subsequent cycles. However, it does observe writes into the match table and can writeback if a new match table entry is allocated with a matching PC.

In our design, the IB is provisioned to either issue or writeback up to one instruction per cycle. Specifically, if the instruction at the head of the IB writes back from the match table, an instruction will not issue from that context during that cycle.

5. METHODOLOGY

5.1 Simulator

The microarchitecture presented in section 4 was evaluated in cycle accurate, execute-at-execute, cache coherent multicore simulator. Three processor models were constructed with 1, 2, and 4 cores such that the total number of threads simultaneously executing on the processor was equal to 16: 1-core/16-wide SMT, 2-core/8-wide SMT, and 4-core/4-wide SMT. System calls are emulated in our simulator. Additionally, all pthread calls are converted into system calls, and our simulator emulates the internal state and mechanisms of the pthreads library (see section 5.3).

5.2 Benchmarks

The benchmarks simulated were drawn from the SPLASH2 [29] and PARSEC [4] benchmark suites. The benchmarks were selected based on those that were able to compile and execute in our simulation environment. The threading library used to thread the benchmarks was based on POSIX threads (pthreads), the details of which appear in the next section. All benchmarks were compiled to the Alpha ISA using the GCC 4.0.2 toolchain and glibc 2.3.6, with the -O3 optimization level.

Throughput processors are currently typically used in large datacenter servers. Our tested benchmarks do not include any transaction benchmarks such as TPC [2] or SPEC WEB2009 [1]. Instruction Sharing is a technique to increase the performance of a throughput processor. Transaction and server benchmarks usually require full-system or virtual machine support to execute. While it is possible to measure performance, the overhead of the operating system or virtual machine obscures the core performance that we target.

To verify the correct execution of our benchmarks, we ran the SPLASH2 kernels using the -t option, which tests that the outputs are as expected. The SPLASH2 apps and PARSEC benchmarks were run and compared against the expected output.

Parameter	Value/Type
I-cache	64kB (64B lines, 4-ways, 256 sets), 16-entry victim cache, 1 cycle access latency, 32 MHSRs, 32 write out buffers, private, MESI coherence.
D-cache	64kB (64B lines, 4-ways, 256 sets), 16-entry victim cache, 1 cycle access latency, 32 MHSRs, 32 write out buffers, private, write-back, MESI coherence.
Per-Core L2	512kB (64B lines, 8-ways, 1024 sets), 16-entry victim cache, 20 cycle pipelined internal access latency, 32 MHSRs, 32 write out buffers, inclusive of core I$ and D$, private, write-back, MESI coherence.
Shared L3	8MB (64B lines, 16-ways, 8192 sets), 16 entry victim cache, 40 cycle pipelined internal access latency, 32 MHSRs, 16 write out buffers, inclusive of all caches on chip, write-back, MESI coherence.
Chip, Network and Routers	1 physical link between routers, 3-logical networks (command, response, and data), 4-entry ingress queues, 4-entry egress queues, 1-packet routed per network per router per 2 core cycles, minimum of 2 core cycle network ingress, minimum 2 core cycle network egress, minimum 2 core cycle per hop, static routing tables.
RAM	200 cycle access latency.

TABLE I
COMMON SIMULATION MODEL DETAILS

For each benchmark, the simulator quickly skipped from the beginning of the program until all threads had been spawned and are executing in earnest. After this skip, simulation transitioned to our detailed execute-at-execute simulator. The first 25 million instructions were used to warm up the system state. Results were collected over the next 100 million instructions. All benchmarks were configured to spawn 16 computation threads.

5.3 Threading Library

To simulate benchmarks that utilize the pthread threading library, without running an operating system inside the simulator, we replaced the pthreads implementation code in the glibc library with a series of system calls, one system call per pthread function. All benchmarks were compiled using the modified glibc library. During compilation, pthread calls made by the benchmarks either directly or through glibc code were converted into operating system calls with a unique function code specific to that pthread call. The simulator intercepts all pthread calls (in an identical fashion to all other system calls) and emulates them, writing the results of the pthread call back into the timing simulator register file and memory and updating internal state regarding threads, mutexes, barriers, and other pthread mechanisms.

5.4 Power and Area Modeling

Instruction Sharing incurs additional power and die area due to 1) the additional hardware and 2) the microarchitectural impact on the existing processor hardware. We used a suite of tools to quantify this impact as thoroughly as possible.

The McPAT tool [14] is a complex analytical modeling framework used to establish the die area and power of processors. McPAT uses a description of the processor pipeline, including most typical pipeline structures, in addition to event counts to produce its result. McPAT is able to handle descriptions of several pipeline features key to our proposed implementation, such as SMT support for many in-order cores. Unfortunately, McPAT does not have the ability to describe the features unique to Instruction Sharing.

The two main additions that must be separately modeled are the match tables and the comparators used to determine when matches occur. To find the area and power of the match tables, we used CACTI [19]. CACTI uses a complex analytical model to determine characteristics of a diverse range of memories. In fact, McPAT itself uses an integrated CACTI model for all memory structures, so the results of our McPAT description and our match table are comparable. Finally, neither McPAT nor CACTI provide a way to determine the area and power of combinational logic. We implemented the register value comparators necessary for the match test in Verilog and synthesized them using Synopsys Design Compiler. To match the 45nm process technology specified to McPAT and CACTI, we synthesized using the FreePDK [27] open source 45nm library.

6. RESULTS

Figure 9 presents the speedup of instruction sharing on a 4-core/4-context-per-core processor, a 2-core/8-context-per-core processor, and a 1-core/16-context-per-core processor, while Figure 10 presents the percent of the dynamic instruction stream that was shared. The two graphs are highly correlated. As the number of contexts per core are increased, the opportunity for sharing increases, and hence speedup grows. In these graphs, 4 match table entries are used per context. The 4-core/4-thread configuration does not see significant performance gains on most benchmarks, although it does achieve a 10% speedup for lu. The 2-core/8-thread configuration achieves modest average speedup, with 5 benchmarks near or above 10% speedup. The 1-core/16-thread configuration achieves appreciable speedup, with a 10% mean improvement across the simulated benchmarks, and 8 benchmarks achieving at or above 10% speedup, with lu-contiguous reaching the peak speedup of 30%.

Figure. 11 tempers these results with the effect on per-core power. There are two distinct sources for the increased power. First, Instruction Sharing consumes power via additional register reads, performing the match test, and maintaining and accessing

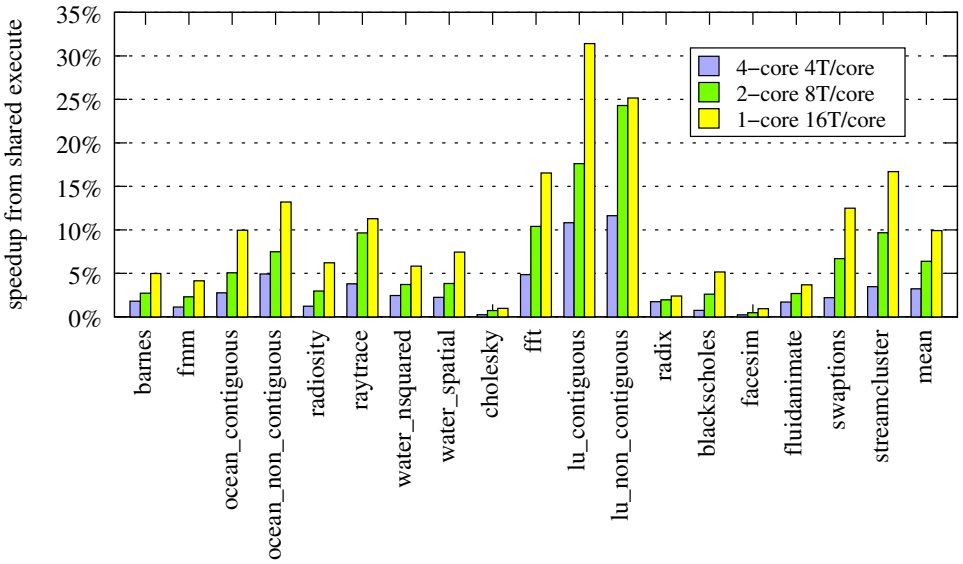

Fig. 9. Speedup of instruction sharing techniques on a 4-core/4-context-per-core processor, a 2-core/8-context-per-core processor, and a 1-core-16/context-per-core processor. Speedup is presented by comparing the throughput of each processor with shared execution enabled against each processor without shared execution. Match tables have 4 entries.

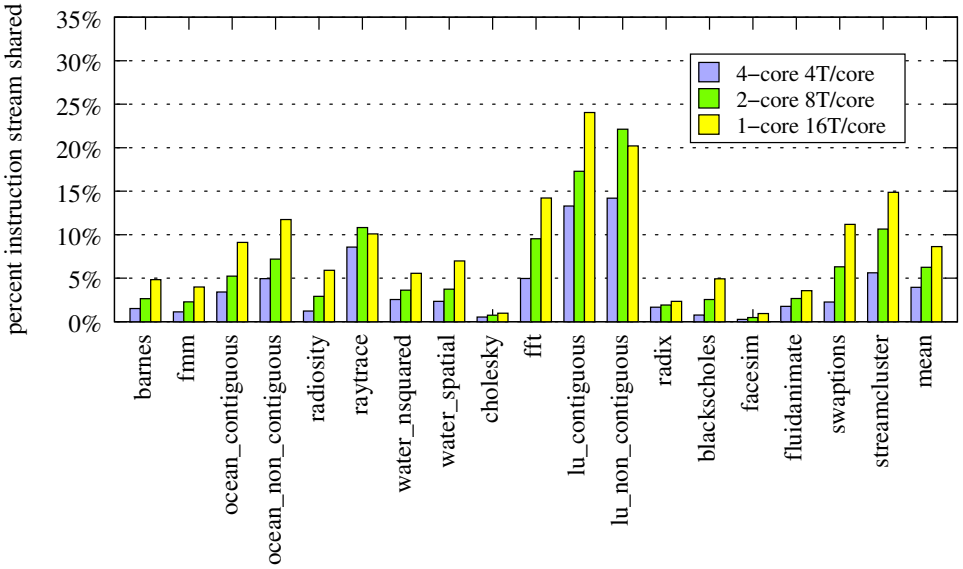

Fig. 10. Percent dynamic instruction stream shared using presented instruction sharing hardware on a 4-core/4-context-per-core processor, a 2-core/8-context-per-core processor, and a 1-core/16-context-per-core processor. Generally, the percent of the instruction stream shared correlates highly with the speedups obtained in Fig. 9. Match tables have 4 entries.

the match table. Second, as core performance increases, the higher instruction rate increases the power demands on all core structures. Instructions need to be fetched and written into the instruction buffer at a higher rate. More useful instructions generally issue down the execute pipeline than in the baseline processor, contributing to more power due to increased work.

We find that in the average case, Instruction Sharing incurs a slightly greater power increase than the performance increase, e.g. a 12% mean power increase for a 10% mean performance increase. However, several benchmarks achieve significantly greater performance than the power increase, while other benchmarks incur significantly greater power cost than the performance benefit. This indicates that Instruction Sharing should probably be accompanied by a method to disable it for benchmarks where it does not perform well compared to the power budget, favoring other performance enhancements.

The performance of Instruction Sharing is directly affected by the number of match table entries. Figure 12 illustrates this by varying the number of match table entries per table from 1 entry to 8 entries, for a 2-core/8-context per core processor.

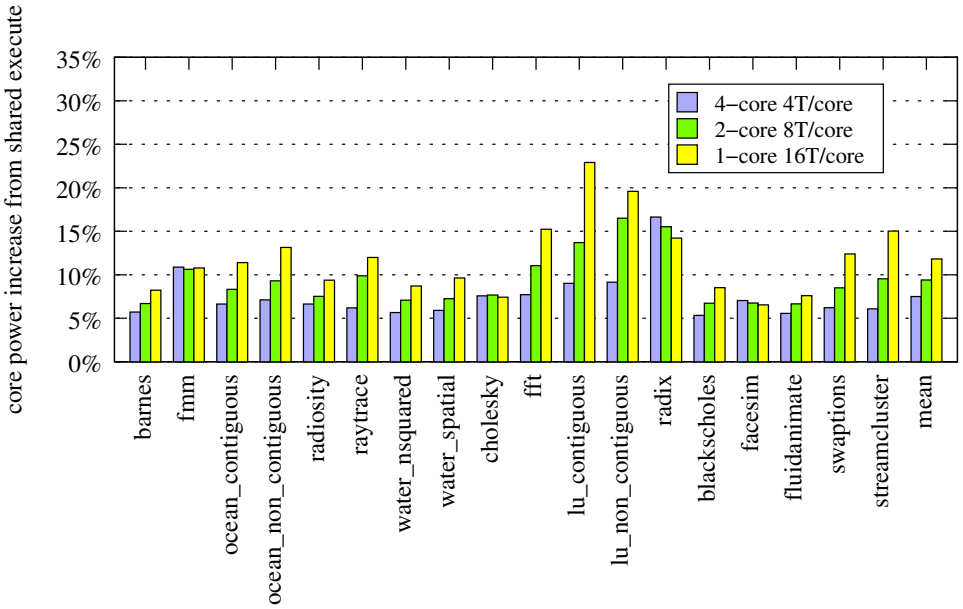

Fig. 11. Core power impact of implementing instruction sharing. Caches, interconnect, and memory controller power are excluded from this data. Power is increased by two sources. First, the additional register read process, and the match test and match table hardware contribute power. Second, by increasing performance, power demands on most core structures are also affected. E.g., by increasing fetch rate and IB write rates. Match tables have 4 entries.

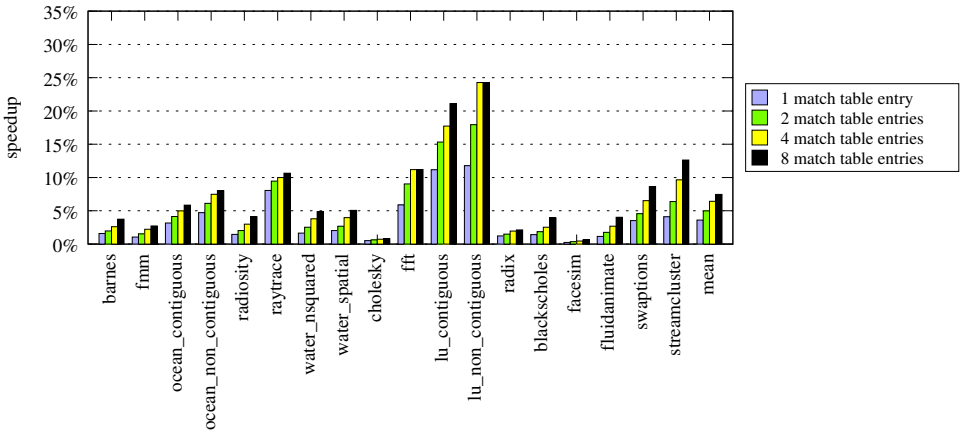

Fig. 12. Effects of various numbers of match table entries on the speedup afforded by instruction sharing. Match table entries are per-thread, as the processor maintains separate fully-associative match tables for each context. Data is presented for a 2-core/8-context-per-core processor.

Generally, speedup monotonically increases as match table entries are increased, up to a saturation point.

In Fig. 13, the performance of a power-saving scheme is evaluated. Instead of performing full 64-bit register reads and 64-bit value comparisons for the match test, a limited number of bytes are read out of the register file, along with a sign-extension bit. The sign extension bit must be true on both the correct value sent to execute (as a 64-bit value) and the registers being matched against it (as n-byte values), as well as the lower bytes. Based on these results, sharing can likely be restricted to 4 or 5 byte sign extended source register values, providing modest power savings with negligible performance impact compared to a full 8-byte match test.

Without instruction sharing, designers would provision cores with sufficient contexts to saturate the execution units, and no more. Hence, we consider the costs of multithreaded instruction sharing to arise from two places: increasing the number of SMT contexts on the core and adding instruction match hardware and tables. We estimate the total area overhead of adding instruction sharing hardware and increasing contexts to 16 contexts to be to be approximately 15% of the core area, compared to a 4-context baseline core. We find that the greatest portion of this cost, 13%, is caused by the additional SMT contexts. The instruction sharing hardware itself incurred 2% of the total cost. The total cost of 16 4-entry match tables was on the order of 1104 bytes. The register value matching comparators incurred negligible area cost.

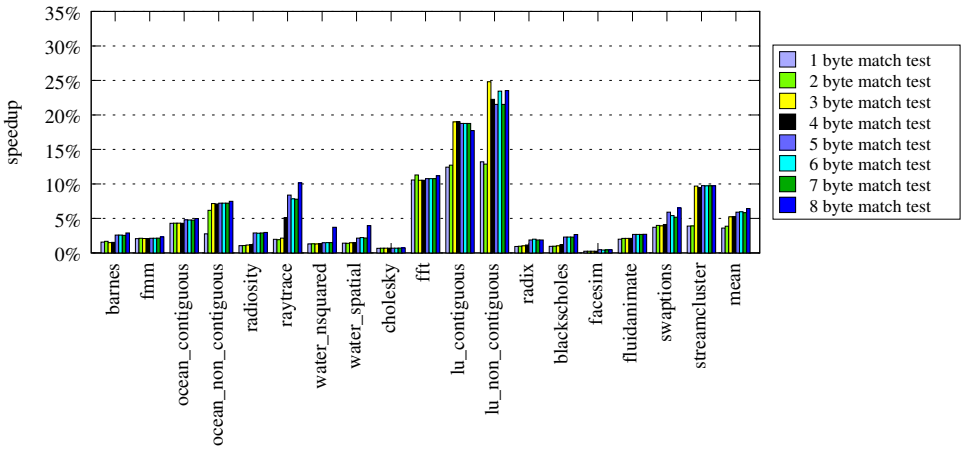

Fig. 13. Effect of limited match test. Instructions are only matched if they use source registers that contain sign-extended n-byte values. One additional bit is added to each register to indicate whether they meet this criteria. The sign extension bit and the lower bytes are tested instead of the full 64-bits.

7. Conclusion

As programs become more multithreaded, the likelihood that different threads of the same parallel program will execute identical instructions increases. In this work, we illustrated this trend with a application behavior study that showed that depending on the benchmark, the number of threads, and the length of instruction history tracked, up to about 50% of instructions of a thread match an instruction in another thread.

We proposed a Multithreaded Instruction Sharing technique whereby identical instructions awaiting issue in the same core execution. When an instruction is undergoing decode, all contexts in the core undergo an instruction sharing test. Matched instructions are removed from the instruction buffer and do not execute. Rather, they share the value produced by their matching instruction. On a full-processor simulation of a 1-core/16-context processor, execution sharing techniques achieved a mean speedup of 10%, with several individual benchmarks achieving peak speedups of 15%-32%. Simulation of a pared down design consisting of a 2-core/8-contexts-per-core processor, execution sharing techniques achieved a mean speedup of 6%, with several benchmarks achieving speedups of 10%-20%. The techniques presented in this work increase core area by approximately 15% (13% additional contexts, 2% instruction match hardware). MIS increases core power by a mean of 12%, but we propose techniques to reduce the power cost by sharing only instructions that use can be matched as sign-extended registers of fewer bytes – thus reducing the number of bits that need to be read out of the register file to perform the match test while negligibly impacting performance.

Acknowledgments

This project is supported by NSF grant CCF-0702632, SRC grant 2007-HJ-1594, and an Intel gift. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] "SPEC WEB2009 Benchmark Suite," 2009. [Online]. Available: http://www.spec.org/web2009/

[2] "TPC Benchmark Suite," 2010. [Online]. Available: http://www.tpc.org/

[3] J. Barreh, J. Brooks, R. Golla, G. Grohoski, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, and M. Shah, "Niagara-2: A Highly Threaded Server-on-a-Chip," in *Proceedings of the 18th Annual Hot Chips Symposium*, August 2006.

[4] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th Annual International Conference on Parallel Architectures and Compilation Techniques*, October 2008, pp. 72–81.

[5] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. Chong, "Multi-Execution: Multicore Caching for Data-Similar Executions," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009, pp. 164–173.

[6] S. Chen, P. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. Mowry, and C. Wilkerson, "Scheduling Threads for Constructive Cache Sharing on CMPs," in *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, June 2007, pp. 105–115.

[7] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998, pp. 252–261.

[8] M. Curtis-Maury, T. Wang, C. Antonopoulos, and D. Nikolopoulos, "Integrating Multiple Forms of Multithreaded Execution on Multi-SMT Systems: A Study with Scientific Applications," in *Proceedings of the 2nd International Conference on the Quantitative Evaluation of Systems*, September 2005, pp. 199–208.

[9] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors," in *Proceedings of International Workshop on Parallel Symbolic Computation (held in conjunction with the 20th Annual International Symposium on Symbolic and Algebraic Computation)*, July 2007, pp. 15–23.

[10] A. González, J. Tubella, and C. Molina, "Trace-Level Reuse," in *Proceedings of the International Conference on Parallel Processing*, September 1999, pp. 30–37.

[11] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1998, pp. 216–225.

[12] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded SPARC Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, March 2005.

[13] R. Kumar, N. Jouppi, and D. Tullsen, "Conjoined-core Chip Multiprocessing," in *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, December 2004, pp. 195–206.

[14] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual International Symposium on Microarchitecture*, December 2009, pp. 469–480.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[16] C. Molina, A. González, and J. Tubella, "Dynamic Removal of Redundant Computations," in *Proceedings of the 13th International Conference on Supercomputing*, June 1999, pp. 474–481.

[17] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41–51, March 2005.

[18] A. Moshovos and G. Sohi, "Speculative Memory Cloaking and Bypassing," *International Journal of Parallel Programming*, vol. 27, no. 6, pp. 427–456, December 1999.

[19] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009.

[20] V. Petric, A. Bracy, and A. Roth, "Three Extensions to Register Integration," in *Proceedings of the 35rd Annual International Symposium on Microarchitecture*, November 2002, pp. 37–47.

[21] S. Raasch and S. Reinhardt, "The Impact of Resource Partitioning on SMT Processors," in *Proceedings of the 12th Annual International Conference on Parallel Architectures and Compilation Techniques*, September 2003, pp. 15–25.

[22] A. Roth and G. Sohi, "Register Integration: A Simple and Efficient Implementation of Squash Reuse," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, December 2000, pp. 223–234.

[23] Y. Z. S. Collange, D. Defour, "Dynamic Detection of Uniform and Affine Vectors in GPGPU Computations," in *Proceedings of the Workshop on Highly Parallel Processing on a Chip (held in conjunction with the 16th International European Conference on Parallel and Distributed Computing)*, August 2009.

[24] A. Snavely and D. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000, pp. 234–244.

[25] A. Sodani and G. Sohi, "An Empirical Analysis of Instruction Repetition," *ACM SIGOPS Operating System Review*, vol. 32, no. 5, pp. 35–45, December 1998.

[26] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 194–205.

[27] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An Open-Source Variation-Aware Design Kit," in *Proceedings of the IEEE International Conference on Microelectronic Systems Education*, June 2007, pp. 173–174.

[28] H. Wang, I. Koren, and C. Krishna, "An Adaptive Resource Partitioning Algorithm in SMT Processors," in *Proceedings of the 17th Annual International Conference on Parallel Architectures and Compilation Techniques*, October 2008, pp. 230–239.

[29] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24–36.