Minimal Multi-Threading: Finding and Removing Redundant Instructions in Multi-Threaded Processors

Guoping Long†[1], Diana Franklin‡, Susmit Biswas‡,
Pablo Ortiz‡, Jason Oberg§, Dongrui Fan†, Frederic T. Chong‡
† Institute of Computing Technology, Chinese Academy of Sciences
‡ Department of Computer Science, UC Santa Barbara
§ Department of Computer Science, UC San Diego

Abstract – **Parallelism is the key to continued performance scaling in modern microprocessors. Yet we observe that this parallelism can often contain a surprising amount of instruction redundancy. We propose to exploit this redundancy to improve performance and decrease energy consumption.**

We propose a multi-threading micro-architecture, Minimal Multi-Threading (MMT), that leverages register renaming and the instruction window to combine the fetch and execution of identical instructions between threads in SPMD applications. While many techniques exploit intra-thread similarities by detecting when a later instruction may use an earlier result, MMT exploits inter-thread similarities by, whenever possible, fetching instructions from different threads together and only splitting them if the computation is unique. With two threads, our design achieves a speedup of 1.15 (geometric mean) over a two-thread traditional SMT with a trace cache. With four threads, our design achieves a speedup of 1.25 (geometric mean) over a traditional SMT processor with four-threads and a trace cache. These correspond to speedups of 1.5 and 1.84 over a traditional out-of-order processor. Moreover, our performance increases in most applications with no power increase because the increase in overhead is countered with a decrease in cache accesses, leading to a decrease in energy consumption for all applications.

Keywords: SMT, Instruction Redundancy, Parallel Processing

1. INTRODUCTION

As clock speeds plateau, all forms of parallelism have become critical to increasing microprocessor performance. Research on SMT and CMP processors has largely focused on multi-programmed workloads, taking advantage of the different program characteristics to better utilize cache space [1, 2, 3, 4], load-balance threads on an SMT [5], and use thread priorities to execute efficiently [6].

While these differences in multi-programmed workloads allow more efficient use of resources, we focus upon another important class of of workloads: multi-threaded applications programmed in a single-program-multiple-data (SPMD) style. The SPMD style is a common way for programmers to simplify the task of writing parallel code, focusing on data parallelism rather than control parallelism. Programming paradigms in this category include Pthreads-style multi-threaded programming, MPI-style message-passing programming, and multi-execution programs whose executions and results analysis are all separate programs. MMT provides purely hardware mechanisms to target the largest number of application classes, even those the programmer did not explicitly program in parallel.

We observe that the SPMD multi-threaded codes involve threads which often execute the same instructions at approximately the same time. Many of these instructions even operate on identical data. MMT fetches these instructions together, only splitting instructions that may produce different results. We further observe that existing out-of-order mechanisms on multi-threaded microprocessors can be leveraged to inexpensively achieve this. Specifically, we present minor modifications to an SMT core that use register renaming and the instruction window to implement combined instruction fetch, combined instruction execute, and instruction synchronization for SPMD multi-threaded workloads.

Our goal is to execute separate instructions for each thread only when either the instructions are different or the inputs to those instructions are different *without requiring program recompilation to give hints as to when they are identical*. We begin with the premise that SPMD programs start with the same instructions in each thread and decide instruction-by-instruction which must be split to accommodate the differences between the threads. This provides two distinct optimizations. First, if multiple threads need to fetch the same set of instructions, we only need to fetch them once, though we may execute them multiple times (once for each thread). Second, if these identical instructions also have identical input values, we only need to execute them once, applying the execution results to all threads. In this paper, we focus on dynamic instantaneous instruction reuse in two categories of parallel workloads - multi-threaded and multi-execution, described in Section 3.1.

This work makes two main contributions. First, we present a mechanism that facilitates the synchronization of multiple threads without software hints to increase the time they spend fetching and executing identical instructions. Second, we propose changes to the decode and register renaming stages to identify and manage instructions with the same inputs, executing and committing those instructions once for all threads.

We evaluated our design on a cycle-accurate simulation of an SMT architecture. Compared to a traditional SMT core with a trace cache running the *same number of threads*, we achieve average speedups of 1.15 and 1.25 (geometric mean) for two and four hardware threads, respectively.

The paper is organized as follows. We present related research in Section 2. Section 3 motivates our research by profiling the potential of instruction sharing for our benchmark programs. Section 4 presents our instruction fetch mechanism to track instructions shared by multiple threads, and discusses the details on how to execute common instructions for multiple threads. Sections 5 and 6 discuss our experimental setup and results, followed by our conclusions.

2. RELATED WORK

This is not the first, and certainly will not be the last, attempt to exploit the redundancy in instructions and data on a modern processor. Our work focuses on removing inter-thread instruction redundancy in SPMD programs.

Intra-thread instruction redundancy has been exploited by memoizing values for later reuse, increasing performance and reducing energy. Hardware approaches have been used to reuse results of long-latency alu operations [7, 8, 9, 10, 11] and compress functions or arbitrary sequences of instructions into a single memoized result [12, 7, 8, 13, 14, 15, 16, 17, 18]. Further studies were performed to analyze the effectiveness and propose solutions for instruction reuse in the hardware [19, 20, 21, 22, 23, 24, 25, 26, 27]. Others have proposed using compiler hints to find reuse [28, 29, 30, 31]. Inter-thread instruction sharing has also been exploited to reduce cache misses [32, 33, 34]. Our work can be used in conjunction with these intra-thread techniques.

Some prior work attacked the same problem: identifying and exploiting instruction redundancy in multi-execution workloads. Umut et al [35] proposed a software approach by maintaining the dependence graph of the computation and using the graph to propagate changes from inputs to the output. By far the closest work to ours is Thread Fusion[36], which proposes a hardware/software system to merge the fetch portion of multi-threaded programs. We have extended this idea in two significant ways. First, we have proposed hardware that removes any use of software hints provided by either the compiler or the programmer, greatly expanding the application classes that can be targeted (adding legacy codes, message-passing and multi-execution to multi-threaded apps). Our hardware could be used in conjunction with their software hints system to provide even better performance. Second, we have proposed a system that allows identical execution in addition to the fetch, which is a significant overall improvement.

3. MOTIVATION

The goal of SMT is simple: retain single-thread performance while allowing a multi-threaded program to execute as quickly as possible by dynamically sharing resources. We observe that when an SMT core runs an SPMD program, the different threads fetch large sections of the same instructions, with some subset of those having identical input values. Our goal is to design and extend an SMT core, so that it can fetch and possibly execute these identical instructions with low overhead. In this section, we present our two application classes followed by profiling results that motivate our design.

3.1 Workloads

Our work focuses on the execution of instructions in data-parallel SPMD workloads, not memory access, so we consider both shared-memory and non-shared-memory workloads. There are three such types of programs: multi-threaded, in which threads communicate through shared memory; message-passing, in which threads communicate through explicit messages; and multi-execution, in which threads do not communicate during execution (a separate program accumulates results once all threads have completed). This work evaluates two categories of workloads: multi-threaded and multi-execution. Multi-execution workloads are, in normal use, applications that require many instances of the program with *slightly different input values*. Simulations such as circuit routing, processor verification, and earthquake simulation require the application to be run hundreds of times with different inputs. The applications we use are summarized in Figure 1. Seven applications are multi-execution workloads from SPEC2000 and SVN[37] with varying data inputs drawn from [34]. The next five are multi-threaded applications from the SPLASH-2 benchmark suite [38], using the same inputs as in [38]. For Parsec programs, we use sim-small input sets for our experimentation.

Suite	Type	Applications
SPLASH-2[38]	MT	LU, FFT, Water-Spatial, Ocean, Water-Nsquared
Parsec[39]	MT	swaptions, fluidanimate, blackscholes, canneal
SPEC2000	ME	ammp, twolf, vpr, equake, mcf, vortex
SVM[37]	ME	Libsvm

Table 1: Summary of Applications

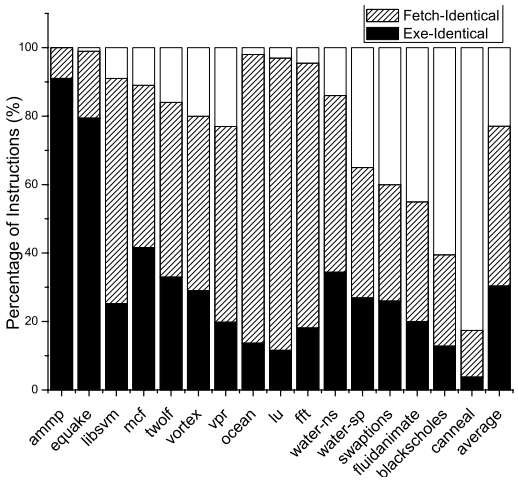


Figure 1: Breakdown of Instruction Sharing Characteristics

While multi-threaded and multi-execution workloads share the characteristic that many identical instructions are potentially executed in different contexts at the same time, they have three important differences. The first is that different threads of multi-threaded programs do not start with identical state - the stack pointers are different. Multi-execution workloads, on the other hand, begin with all registers identical. The difference is in the input parameters, which are stored in memory. The second difference is that threads in a multi-threaded program share memory, so a load to the same virtual address in multiple threads will always return the same value (if executed without an intervening write). In a multi-execution workload, this may not be the case. No memory is shared, so a load from the same virtual address in different threads may or may not return the same data. Third, threads in a multi-threaded program are all the same process, and instances of programs in a multi-execution workload are different processes. In the rest of this paper, when the word *thread* is used, it is a thread of execution, which can either refer to a single *thread of a multi-threaded program* or a single *instance of a multi-execution workload*.

3.2 Instruction Redundancy

For each application we want to see how much redundancy there

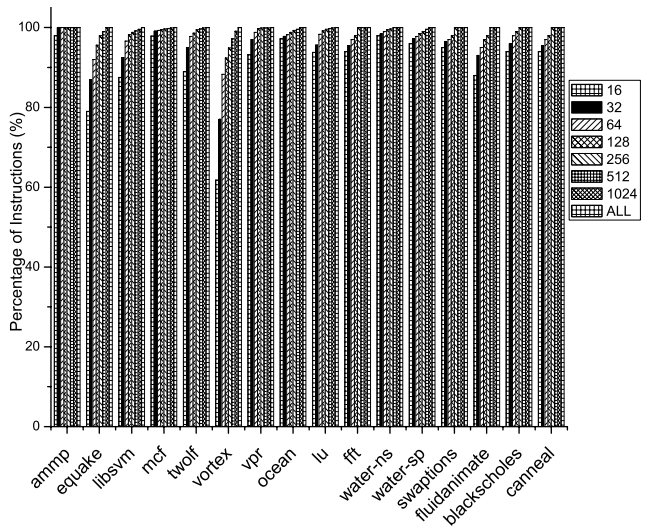


Figure 2: Distribution of the difference in length of divergent execution paths

is in the instructions executed. Because our goal is to reduce inter-thread redundancy, not intra-thread redundancy, we only measure when instructions from different threads are identical at the same point during execution, taking into account that execution paths may diverge for different amounts of time before coming back together. We do this by finding all of the common subtraces of each trace.

We want to answer two questions for each application. First, how often is the same instruction *fetched* by all threads? We call this fetch-identical. Second, how many of these fetch-identical instructions have identical operand values? We call these execute-identical. Note that execute-identical are also fetch-identical instructions, even though we no longer refer to them in this way.

Figure 3.1 shows the instruction breakdown for each application: Execute-identical, fetch-identical, and not identical. The average bar of the figure shows the algorithmic average of all applications. Although the distribution of instructions varies for each application, all applications have large amounts of fetch-identical instructions (since execute-identical instructions are also fetch identical). About 88% of instructions, on average, can be fetched together, making a shared fetch mechanism very promising. Previous work also observed a similar phenomenon for server workloads [32] and transaction processing in database systems [33].

For each application, we are particularly interested in execute-identical instructions. Multi-execution applications have execute-identical instructions mainly because part of the execution graph is the same, whereas in multi-threaded applications, execute-identical instructions exist because some data are shared by all threads. As can be seen, this property varies greatly from application to application. Both equake and ammp have lots of such instructions. There are also applications (vpr, lu, fft, ocean, etc) with limited execute-identical. On average, approximately 35% are execute-identical instructions. This observation motivates us to design hardware mechanisms to exploit this program property to boost performance.

3.3 Synchronization

The most efficient way to take advantage of this sharing is to synchronize the threads as much as possible. Each time the execution paths diverge, fetch resumes as normal. We want to gather data as to the likelihood that we can design a mechanism to detect when the threads begin fetching the same instructions again, given that they are unlikely to return to the same point at the same time. Therefore, we measure the *difference* between the lengths of the two divergent execution paths between the common subtraces, where the length is calculated in *taken branches* rather than instructions. Figure 2 presents a per-application histogram of this length for two threads. That is, the leftmost bar in each cluster measures the percentage of divergent dynamic paths that were within 16 taken branches or less of each other. The next bar measures the percentage that were within 32 taken branches or less, and so on.

For all programs except equake and vortex, more than 85% of all diverged paths have a difference in length of no more than 16 taken branches. This implies that if we track the paths while threads are fetching different instructions, we need only a short history of taken branches in order to detect when the threads begin executing on the same path again.

4. ARCHITECTURE

In order to share the fetch and possibly the execution of identical instructions, we face several design challenges. Both optimizations depend on both threads fetching the same instruction at the same time. Therefore, the first design challenge is to remerge the execution path when two threads follow divergent paths. The second challenge is to provide an efficient way to detect whether instructions that were fetched together can also be executed together and manage both types of instructions through the machine.

We present an initial hardware-only design for this problem, the goal being to make changes to as few stages as possible. Each mechanism has a relatively simple hardware implementation, and the design integrates well into an SMT or hyperthreaded processor. The fetch optimization is performed in parallel with instruction fetches, adding negligible delay, and the execute optimization requires a single pipeline stage between decode and register renaming. Register merging requires changes to the commit stage. Gate counts and delays are summarized in Table 3 at the end of this section.

4.1 Instruction Fetch

In this section, we present a mechanism to remove redundant fetches of fetch-identical instructions. This requires two parts. First, when the PCs of two or more threads are identical, we perform a single fetch. The instruction window is enlarged by 4 bits, and a bit is set for each thread with the corresponding PC. We call this 4-bit pattern, identifying the sharing of the instruction, the Instruction Thread ID (ITID) of the instruction. Second, when two threads diverge, the hardware needs to detect when the thread paths remerge and synchronize.

When the threads begin, all begin fetching at the same location. We call this mode of instruction fetch MERGE. At some point in execution, two threads will have a different outcome for the same branch, resulting in divergent execution paths. We will present our mechanism for remerging two threads, but it can be easily translated to four threads.

We need a mechanism to track the two threads' progress in order to detect when the thread paths remerge. This is called DETECT mode, during which threads fetch instructions independently. We introduce a Fetch History Buffer (FHB) for each thread to record the fetch history. Every taken branch, a thread in DETECT mode records its target PC into its Fetch History Buffer. In parallel, it

checks the other thread's history to see if its target PC is in another thread's FHB, as shown in Figure 3(b). If the PC is found, the two threads attempt to synchronize, transitioning to CATCHUP mode. Figure 3(a) shows the transition between three instruction fetch modes.

In CATCHUP mode, in order to resynchrnonize the two threads, we increase the fetch priority of the "behind" thread, whose target PC was found in the others' history. The other thread(s) receive lower fetch priority. Because of the priority changes, CATCHUP mode represents a performance penalty, so we want to stay in it for as little time as necessary. It is also important to detect false positives - when the merged path was too short to resynchronize. Thus, we continue as we did in DETECT mode, recording branch targets and continuing to search for them in the other thread's history. If, while in CATCHUP mode, the "behind" thread's branch target PC is not in the FHB of the "ahead" thread, the state transitions back to DETECT mode, with both threads receiving equal fetch priority again.

The size of the FHB provides an interesting design tradeoff. A larger FHB increases the chance that a path will remerge, providing a large boost in performance, but it also increases the chances of a false positive, resulting in wasted time in CATCHUP mode, a performance penalty. We will explore this tradeoff in Section 6.4

Prior work used software hints for remerge points. A hardware mechanism was provided to find and store a small set of remerge points that were limited to the target of the branch that caused the divergence. The scheme described above provides much more flexibility in determining merge points and does not require space to store successful merge points.

4.2 Detecting Identical Operands

During the fetch stage, a single instruction may be fetched for multiple threads. The four bits that identify those threads are the Instruction Thread ID, or the ITID. A later stage must determine whether or not the instructions are execute-identical. This is what determines whether to split the instructions into two (or more) distinct instructions, each with their own ITID, or to continue through to the commit stage as a single instruction, applying the result to all threads. Thus, there are two distinct tasks. First, given an ITID corresponding to one or more threads, produce the minimal set of 1-4 ITIDs representing the instructions required to be executed for the different threads. Second, for any merged instructions, apply the result to all threads to which it corresponds.

Except for load instructions in multi-execution workloads, as long as the register input values of an instruction are identical, then the output value will be identical, so the instruction may be executed only once. For loads with non-shared memory, the address calculation may be performed merged, but the ld/st queue must split the actual loads and stores. The decision for when to split a fetch-identical instruction is summarized in Table 1. We will address why the multi-programmed loads have a predictor in Section 4.2.5

We add a stage between the decode stage and register alias table lookup to split any fetch-identical instructions that do not have identical inputs. The purpose of this stage is, given a single fetched instruction, to produce the minimal set of 1-4 identical instructions to produce correct execution. Each instruction will be executed for different threads. We must account for four instructions in the case when an incoming thread with ITID 1111 (indicating that it was fetched for all threads) turns into four instructions with ITIDs 1000, 0100, 0010, and 0001, because all threads had a different register value for their inputs.

Stage	Inst	App	Type	Operation
Decode	ALU/Ld Branch	Both	F-id	SPLIT
	ALU/Br	Both	X-id	MERGE
	Load	MT	X-id	MERGE
	Load	ME	X-id	Check LVIP
Ld/St Q	Store	ME	Both	SPLIT
	Ld/St	MT	Both	No Change
	Load	ME	Both	SPLIT; Verify LVIP Pred

Table 2: Logic for splitting instructions. ME is Multi-Execution, MT is Multi-Threaded. F-id is Fetch-identical, and X-id is Execute-identical.

4.2.1 Register Sharing Table

The Register Sharing Table (RS) holds one entry for each architected register. Each entry contains a bit for each potential sharing pair. For a 4-thread MMT, there are 6 combinations of paired sharing, so there are 6 bits. A 1 indicates that the two threads share the register (or the registers hold identical values), and a 0 indicates that the register may contain different values.

4.2.2 Splitting Instructions

The first step is to read the bit in every entry corresponding to each source register, regardless of whether the entry contains information relevant to this particular instruction (ITID). Then different combinations of bits are ANDed to produce the sharing information for all combinations of sharing, 2-4 threads.

Filter.

The filter takes the ITID and filters out (setting to 0) any shared bits from entries that are not possible outcomes of this ITID. For example, if the ITID is 0110, then it could either stay merged at 0110 or be split into two instructions with ITID 0100 and ITID 0010. Thus, the three entries with EID 0100, 0010, and 0110 are the relevant entries. All other entries are set to 0. When anded with the sharing bits, this produces all possible valid sharing combinations.

Chooser.

The purpose of the chooser is to output the EID corresponding to the entry with filtered output 1 that has the most threads sharing it.

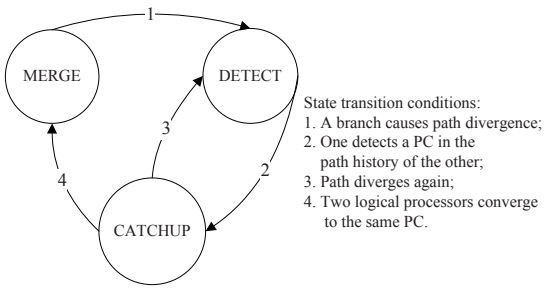
With these components, we can split the instruction up to three times, resulting in new instructions with decreasing amounts of sharing. This produces the minimal set of instructions since, in each stage, we choose the ITID with the greatest amount of sharing.

4.2.3 Updating Register Sharing Table

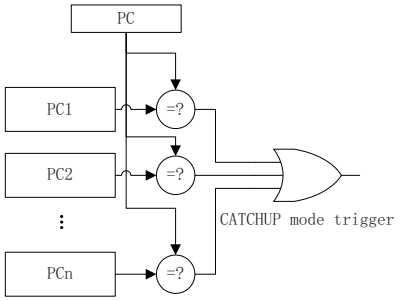
Each cycle, the sharing information must be updated for relevant entries. If this is the destination register for a particular instruction and at least one of the threads' ID was in the ITID, then the bit will be set to a 1 if a resulting ITID has both threads' ID's and 0 otherwise.

4.2.4 Register Alias Table

The Register Alias Table (RAT) remains largely the same. For



9(a) State Transitions of Instruction Fetch



9(b) Fetch History Table Lookup (CAM) Used in DETECT and CATCHUP Mode

Figure 3: Hardware Added for Synchronizing Fetch

each instruction it receives, it needs to read the operands once (regardless of the sharing) and create a single physical destination register. For execute-identical instructions, however, that destination register gets recorded in the RAT of all corresponding threads, not just one.

4.2.5 Load/Store Queue

For multi-execution workloads, even when the address calculation is merged, loads and stores must be performed independently. The load/store queue expands the loads and stores to the appropriate number and performs them serially.

This complicates the instruction splitting mechanisms because even if the inputs are the same, the loaded value may be different, which is what determines whether or not to split an instruction. This loaded value will not be known until much later, and we need to know during this stage in order to a) keep current sharing information to decide whether or not to split instructions that use the loaded value and b) know whether to allocate separate registers.

Previous work [34] found that loads from the same virtual address in multi-execution workloads often contain the same value across instances. For this reason, we use a predictor that predicts whether the current load, which has the same inputs (and therefore virtual address), is going to load the same value from the different processes. We maintain a table of PC's whose loads have been previously mispredicted. We begin by predicting the value will be identical. This is the Load Values Identical Predictor (LVIP). It is stored with the Register Sharing Table to provide an extra check for load instructions.

Therefore, when running a multi-execution program, the queue must wait for both loads to return, check the values, compare the result to the prediction, and possibly trigger a rollback.

4.2.6 Initial Design Summary

In order to illustrate how this works, we begin with the start of program execution. When two threads begin, in a multi-execution workload, all architected registers are mapped to the same physical registers - memory is different, not register values. The RST bits are all set to 1, and the RAT values are identical. In a multi-threaded workload, all architected registers except the stack pointer are mapped to physical registers. The RST and RAT values are all identical except the stack pointer.

There are three ways that the register values become different for the same architected register in two threads. First, if the threads diverge in their execution paths, their instructions will not be fetch-identical, so each instruction's destination register is set to 0. Second, if a fetch-identical instruction has input values in different physical registers, then the instruction will be split, and the two results will be placed into different physical registers (and the corresponding bit will be set to 0 in the RST). Third, if the threads are running a multi-execution program, a load to the same virtual address may load a different value because no memory is shared between different instances of the same program.

4.2.7 Register Merging

Using the Register Sharing Table is not always perfect, because the values within the registers are not checked, only the mapping is checked. Two instructions on divergent paths may write the same value into the same architected register. Because these were on divergent paths, the bit will be set to 0 in the RST, and future fetch-identical instructions will consider these input registers to be different. It would not be unlikely to have the entire register set become divergent for this reason, resulting in no detection of execute-identical instructions.

It would be clearly impractical to check all non-execute-identical results to check the contents of all registers between threads. We observe that it is only during DETECT and CATCHUP mode that two instructions are likely to write the *same value* to the *same architected register* but *different physical registers*, and we only care about the comparison between the current result and the value in the same architected registers of the other threads. In addition, if a later instruction has already passed the RST and writes to this register, it is too late to update the entry. Thus, we only check the destination registers of instructions fetched in DETECT or CATCHUP mode whose destination register is not being overwritten by a younger instruction in this or another thread.

In order to make sure a later instruction in the current thread is not writing to the same architected register, we would need to access the register mapping in the register alias table. This would require too many read ports, so we keep a copy of the mapping table for this purpose. This involves no addition to the RAT's critical path since we are only taking its results, not sending information to it. If the instruction being committed is writing to the same physical register as its architected destination register still maps to, then it is the only instruction in the pipeline writing to this register for this thread. We refer to this as the instruction's mapping still being valid.

In order to track the other threads' use of the architected registers, we keep a single bit for each architected register for each thread.

Component	Description	Area	Delay
Inst Win	ITID/entry	4b/entr	0
FHB	CAM	32*32 b	1 cyc
RST Inst Split RST Update	Ident Reg Info Make ITIDs Update dest reg	11*50 b 80k um^2	0.5ns
Reg State	Thread owners	256*4 b	N/A
LVIP	Pred table	4B*4K entr	1 cyc
Track Reg	Reg Map bit vector	4*50*9 b	1 cyc

Table 3: Conservative Estimate of Hardware Requirements

This bit is a 1 if no active instruction is writing to the register and 0 otherwise. When an instruction is assigned its physical register, it sets the architected register bit to 0. If its mapping is still valid when it commits, it sets the bit back to 1.

When an instruction commits, it checks to make sure its mapping is valid. If it is, then it looks at the bit vector for the other threads. For any other thread whose bit is 1, the architected to physical register mapping is used to determine which register to read. If there are read ports available this cycle in the register file, the physical register is read and compared to the committed value. If they are the same, the appropriate bit(s) are set to 1 in the Register Sharing Table to indicate that the register values are identical.

4.3 Summary

Figure 4 shows the high-level, logical view of the processor pipeline, highlighting the components which are closely related to our design. There are a total of four instructions shown in this figure before renaming. For each instruction, we give the operation type along with a bit vector, indicating which thread owns the instruction. As can be seen, there are three fetch-identical instructions. The last one (MUL) belongs to only one thread.

After register renaming, instructions are put into the issue queue. In this example, only SUB is split because its source registers have different architecture-to-physical mappings. The LOAD is not split, because all source registers share the same mapping. But for multi-execution workloads, the LSQ fetches data for each thread separately and checks if previous prediction on the load is correct.

Table 3 gives details on the hardware requirements for the major components we are adding to the SMT core. The major storage components are the CAM containing the Fetch History Buffers, RST for register sharing tracking, and the LVIP to predict which loads should share a destination register. The major delay introduced is in splitting instructions, which can be accomplished within a single cycle. Note that the storage requirement for the RST shows only 11 entries. This is because the first four entries are hard-coded to 1 and not stored in the table. The table shows the hardware requirements for an optimized implementation of the RST and instruction splitting logic.

As a sanity check, we implemented the RST components in structural VHDL and evaluated power and area using the Synopsis design tools and their academic 90 *nm* technology library. To scale to 32 *nm*, we assumed a reduction in 2x power, 7.9x power[40], and 9x delay[41].

4.4 Operating System Support

In order to realize the performance gains illustrated in the results section, the operating system must be aware and support the system.

Threads	4
Issue/Commit Width	8/8
LVIP/CAM Size	4KB/32 entries
LSQ Size	64
ROB Size	256
ALU/FPU units	6/3
Branch Predictor	2-level, 1024 Entry History Length 10
BTB/RAS Size	2048/16
LVIP	4K
FHB	32 entries
Trace Cache Size	1MB
L1I/L1D Cache	64KB+64KB, 4 way,64B lines
L1 Latency	1 cycle
L2 Cache	4MB, 8 way, 64B lines
L2 Latency	6 cycles
DRAM Latency	200

Table 4: Simulator Configuration

Name	Description
Base	Traditional SMT
MMT-F	MMT, shared fetch only
MMT-FX	MMT, shared fetch and execute
MMT-FXR	MMT-FX with register merging
Limit	MMT-FXR running two instances with identical inputs

Table 5: Summary of Minimal Multi-Threading (MMT) and baseline Configurations

The scheduler needs to gang schedule the threads in pairs or larger groups. If an exception occurs, threads that are currently merged might benefit from both being suspended and restarted together. Synchronization is a rich area to study the trade-offs between the performance suffered from stopping threads that need not suspend and gaining performance from being merged when restarted.

5. EXPERIMENTAL SETUP

Our SMT simulation infrastructure is extended from Godson-T [42], a many-core architecture with a validated simulator built on the simplescalar toolset [43]. Table 5 shows the detailed simulator configuration. We chose this aggressive core for two reasons. First, this width has been used in prior well-known SMT research[6]. Second, the speedups of our system increase as the system is scaled down, so we chose an aggressive baseline in order to not give our system an unfair advantage. We also chose an aggressive fetch mechanism, a trace cache[44, 45] with perfect trace prediction for a similar reason. The worse the fetch performance, the more our system benefits from the shared fetch and execution. We found that the trace cache actually had a negligible effect on the results, so the results with a traditional cache are virtually identical to our presented results.

In order to evaluate the effectiveness of our design, we ran several different configurations, summarized in Table 5. Our baseline is a traditional SMT with a trace cache. We began by running MMT-F, with only the shared fetch capability, always splitting into different instructions in the decode stage. Next, we executed MMT-FX

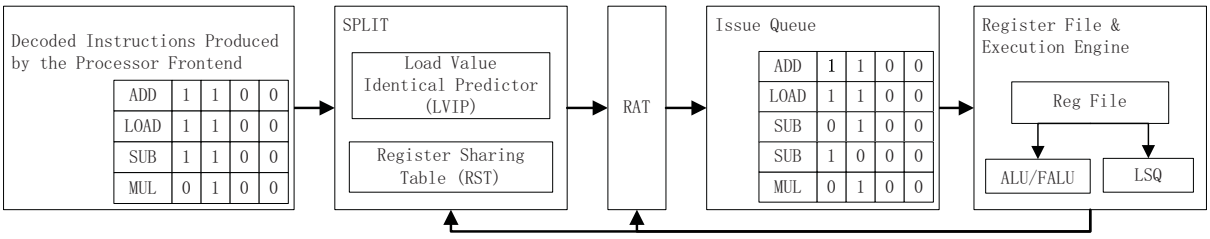


Figure 4: High-Level View of the Pipeline

with shared execution, allowing instructions with identical register mappings to continue executing as a single instruction. Finally, we implemented MMT-FXR with register merging, which occasionally compares the value being written to the register file with the same architected register in the other threads in order to increase the number of registers identified as identical, in turn increasing the number of instructions executed as one. As a comparison point, we also executed a Limit configuration. In this configuration, we execute two identical threads. This represents the case in which all instructions are fetched and executed together, though memory operations may still be performed separately. This is an upper limit on potential performance, though no application will ever attain this performance, since not all instructions can possibly be execute-identical.

We also varied the number of threads executed. As the number of threads increases, we expect an increase in savings due to the opportunities for greater degrees of identical instructions. The way in which we increase the threads differs for each workload. For a multi-threaded program, when we double the number of threads, we are still solving the same problem, so each thread performs less work than before. For a multi-execution workload, on the other hand, the executable itself has no notion of parallelism. When we double the number of threads, we are doubling the amount of work performed.

6. EXPERIMENTAL RESULTS

The goal of our work is to increase performance and decrease energy consumption in workloads with threads or processes that are nearly-identical. We expect to increase performance, because the instruction window is easier to keep full, fewer instructions need to be executed, and fewer cache accesses must occur. We expect to reduce energy consumption by performing fewer cache accesses and executing fewer instructions. We begin by analyzing the performance and energy consumption. We then evaluate our remerging mechanism by comparing our observed results with the profiling results. Finally, we perform sensitivity analysis to see how our design works as the size of the remerging history buffer is changed as well as the width of the instruction fetch.

6.1 Performance Benefit Analysis

We first analyze the performance of the overall design, as well as the contributions of different elements of the design. Figures 5(a) and 5(c) show the performance improvement of our design for all applications. The configurations were described in Table 5.

The overall performance gain for different workloads varies substantially. Some programs, such as libsvm, twolf, vortex, vpf, ocean, lu, fft, waterscholes, and canneal, gain between 0-10% with two threads and 8-20% with four threads. Other programs, such as ammp, equake, mcf, water-ns, water-sp, swaptions, fluidanimate, gain between 20-42% with two threads and 20-80% with four threads. This results in an average performance improvement of 15% and 25% for two and four threads, respectively.

The performance results show that every element of our design was helpful to some applications. With two threads, each element (fetch, execute, and register merging) contributes equally- about 5%. With four threads, shared fetch contributes 10%, shared execution 9%, and register merging 6%. The contribution varies widely by application - ammp benefits most by shared execution, whereas equake and water gain significantly from register merging.

Finally, while a majority of the applications show little difference between the SMT-FXR configuration and the Limit configuration, four applications (libsvm, twolf, vortex, vpr) do well but have a large potential for improvement. This can be explained by looking at Figure 5(b), which depicts the percentage of instructions executed in fetch-identical and execute-identical mode, and comparing it with Figure 3.1, which depicts the percentage of instructions that our profiling showed could be executed in fetch-identical and execute-identical mode. Figure 5(b) introduces one additional category of interest: Exe-Identical+RegMerge. This category means that with register merging, these instructions are identified as execute-identical, but without it, they would only be identified as fetch-identical.

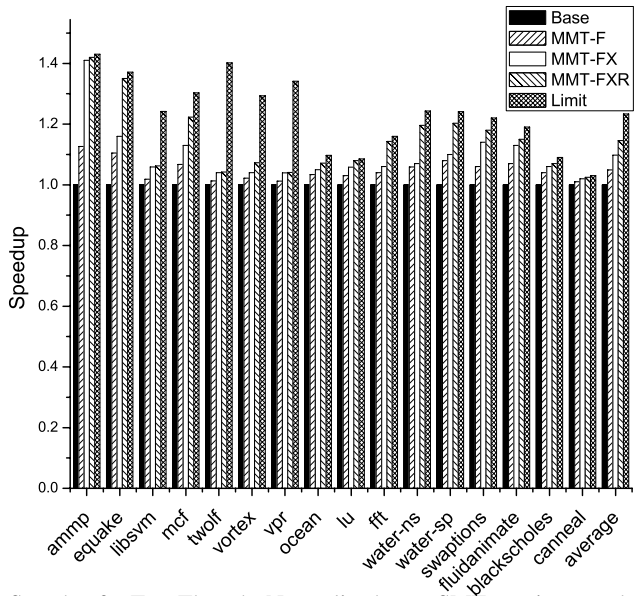
We see that for several of the applications (ammp, equake, mcf, lu, and fft), our mechanism was able to find most of the identical instructions. With our current design, we can track approximately 60% of fetch-identical instructions on average, almost half of which are execute-identical instructions. In four applications, libsvm, twolf, vortex, and vpr, there is a large gap between the identical instructions found and the existing identical instructions. This explains the gap between the realized performance and the limit of the performance.

This figure also shows why shared execution and register merging are important design components. Almost half of fetch-identical instructions are execute-identical instructions. For applications such as equake, mcf, fft, and water-ns, the Exe-Identical+RegMerge section is a noticeable component, meaning that register merging is necessary to identify these instructions as execute-identical. These are the applications that exhibit the most speedup when using register merging.

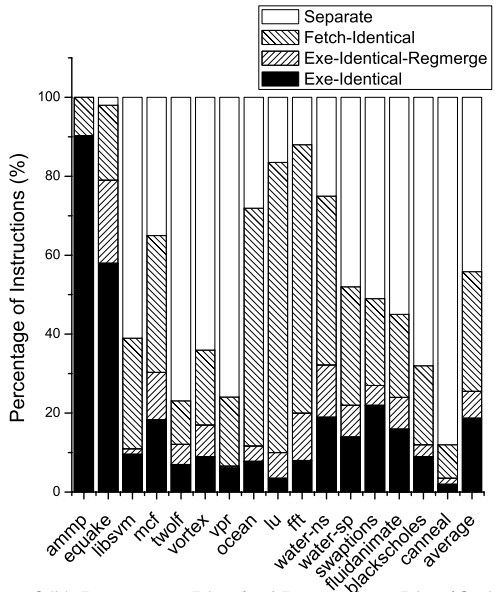
In conclusion, we show good initial results on many applications. In addition, even our applications that showed less improvement have the potential for much better performance if more instructions can be identified as fetch-identical and execute-identical.

6.2 Energy Savings

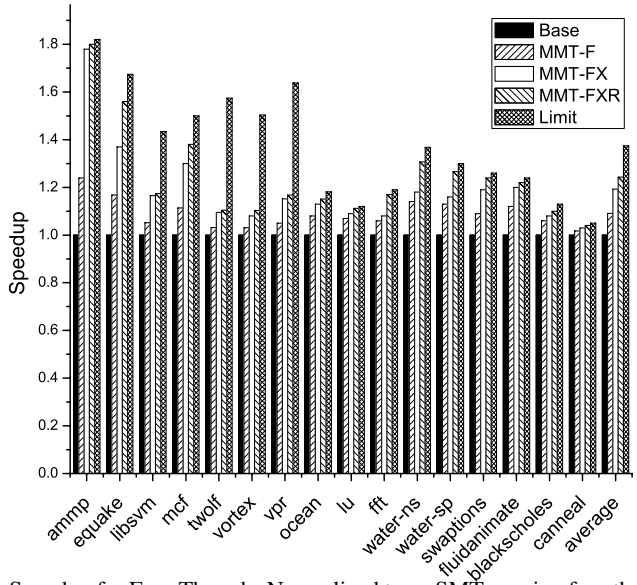
Energy savings are realized, because our mechanisms have low power requirements, yet they result in fewer cache accesses (fetch-identical instructions) and fewer instructions executed, registers read and written, and instructions committed (fetch-identical instructions). For most applications, this results in lower average power and lower



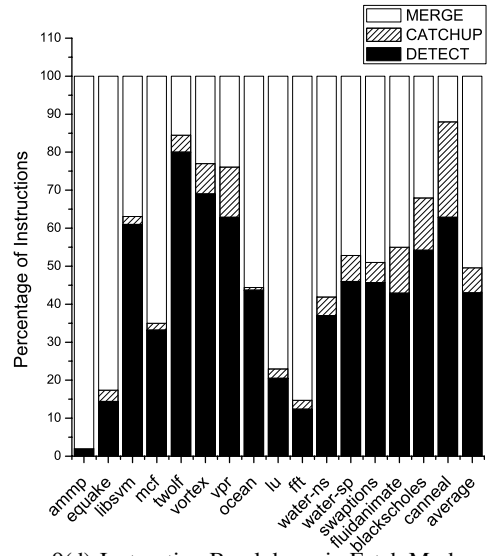
9(a) Speedup for Two Threads, Normalized to an SMT running two threads



9(b) Percentage Identical Instructions Identified



9(c) Speedup for Four Threads, Normalized to an SMT running four threads



9(d) Instruction Breakdown in Fetch Modes

Figure 5: Overall Speedup and Instruction Behavior Results

total energy, with ammp being the only exception. Ammp results in higher average power, because the machine can be used so much more efficiently (resulting in a dramatic reduction in execution time and thus lower total energy).

In order to show the combined effect of the increase in power due to the scheme's overhead and the decrease in energy due to a decrease in total work performed, we graph the energy consumption of each application. We modeled the power with Wattch [46] and our conservative Synopsis power estimates from Section 4.3. We assume 32nm technology throughout our power modeling to reflect future trends.

Figure 6 shows the energy consumption per job completed of a SMT and MMT cores running two and four threads, normalized to a traditional SMT core running two threads. For each application, we graph four bars - SMT core with 2 threads, MMT core with 2 threads, SMT core with 4 threads, and MMT core with 4 threads. We also present the breakdown of total energy consumption in three components: cache power, overhead of our scheme, and the power of other components of the processor.

We can make two observations from these results. First, although we introduce several hardware components, the power consumption contributed by the overhead is negligible. Even without

power gating, the power contributed by the overhead is less than 2% of total processor power. Note that the Load Value Identical Predictor (LVIP) is only accessed when instructions are fetched in MERGE mode, and the Fetch History Buffers and Register Merge Hardware are only accessed when instructions are not in MERGE mode. The Register Sharing Table is updated every cycle, regardless of whether it is used to split instructions. Because of this, the FHB's are used less than 30% of the time, on average. With power gating, the power consumption contributed by these components can be further reduced to less than 1%.

Finally, the total energy consumption of our design is significantly lower than the traditional SMT core. We gain by both reducing cache accesses and reducing instruction execution. As the number of threads increases, the energy reduction continues to increase across both workloads, especially the multi-execution workloads. With four threads, the MMT core consumes 50-90% as much energy as the traditional SMT core, with a geometric mean of 66%, though a majority experience 10%-20% savings.

6.3 Instruction Fetch Merging

The MMT design results in performance improvement and energy reduction, but we need to analyze where there is room for improvement. In this section, we analyze the effectiveness of our instruction fetch synchronization mechanism.

Figure 5(d) is a different view of the instruction breakdown from the instruction fetch mode. The best possible performance will be attained if the percentage of instructions fetched in MERGE mode is the same as the percentage of instructions that are fetch-identical or execute-identical, and the rest is in DETECT mode. On the other hand, during CATCHUP mode, most of the instructions in the pipeline are from one thread. This degrades performance.

Luckily, CATCHUP mode is rare in most programs. This is because the processor enters the mode only when a potential remerge point has already been detected. Unfortunately, three programs (vpr, twolf and vortex) spend less time in MERGE mode than other programs. This also indicates that our instruction fetch logic can be further improved. We will leave this for our future work.

Finally, although not graphed, we analyzed the number of fetched branches encountered before the hardware remerged the threads. We found that in 90% of the cases, the remerge point was found within 512 branches, though many were found in less time.

6.4 Fetch History Buffer Size

A major design question for the instruction merging design is how large to make the CAM for the Fetch History Buffer. The CAM uses space, consumes power (when in use), and must be small enough to be accessed in a single cycle. With 32 entries, in more than 90% of the cases, the remerge point is found within 512 taken branches. Increasing the FHB size has the potential to improve the number of fetch-identical instructions that can be tracked. It may, however, also make the catchup process longer, hurting performance. These two effects are depicted in Figures 7(c) and 7(a).

The direct effect the FHB size has is to the time spent in DETECT, CATCHUP, and MERGE modes. Figure 7(c) shows that several applications, including equake, vortex, ocean, lu, FFT, and water-ns, spend more time in MERGE mode with a larger FHB, because it captures merge points the small FHB did not. Other applications, such as twolf, vortex, vpr, and water-sp, spend an increasing amount of time in CATCHUP mode, potentially decreasing their performance.

As the FHB size increases, we expect from these statistics to see a performance improvement in equake, FFT, and water-ns, but perhaps not for vortex. We also have the potential for a decrease in performance for twolf, vpr and water-sp.

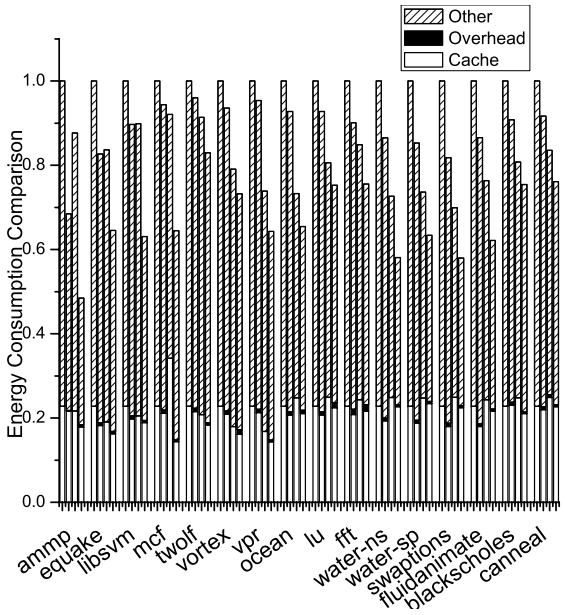


Figure 6: Energy Consumption Comparison. The four bars, from left to right, are: SMT - 2 threads, MMT - 2 threads, SMT - 4 threads, MMT - 4 threads

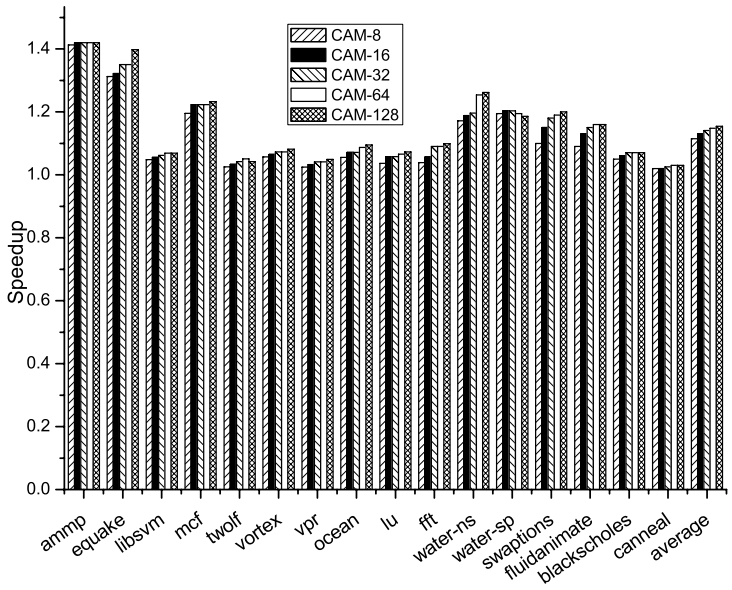
Figure 7(a) shows the performance results when we vary the History Buffer size from 8 entries to 128 entries. As predicted, the performance increases slightly as the percentage of MERGE instructions increases. We see small increases in vortex, vpr, ocean, lu, and fft, and greater increases in equake and water-ns. As expected, two applications, twolf and water-sp, exhibit a small decrease in performance at large FHB sizes.

Because there is an increase in performance for all applications through 32 entries, and a larger FHB has the danger of not being accessed in a single cycle, we chose an FHB size of 32 entries. The average performance does continue increasing slightly as the size is doubled, so, depending on the cycle time of the computer, an FHB size up to 128 entries would be beneficial.

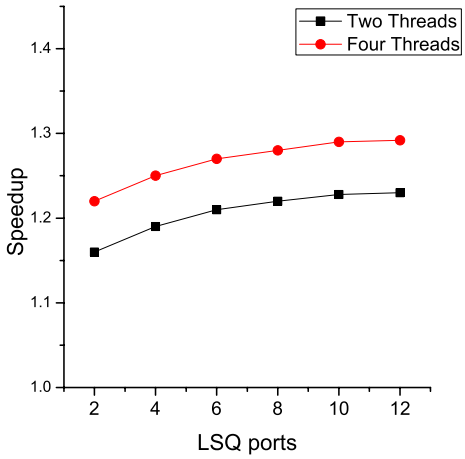
6.5 MMT Sensitivity Analysis

Finally, we see how dependent our results are on the width of the instruction fetch. Figure 7(d) shows the performance sensitivity when the instruction fetch width varies from 4 to 32. Depicted is the geometric mean of the results from all applications. The gains are reduced as the width of the instruction fetch is increased, because the fetch is entirely removed as a bottleneck. Even at 32 instructions per cycle and a trace cache with perfect prediction, we still observe average speedups of 11%. The design point we chose in our design is 8 instructions per cycle for fetch, issue, and commit. Although not shown, similar trends exist if the entire machine width is varied from 4 to 32.

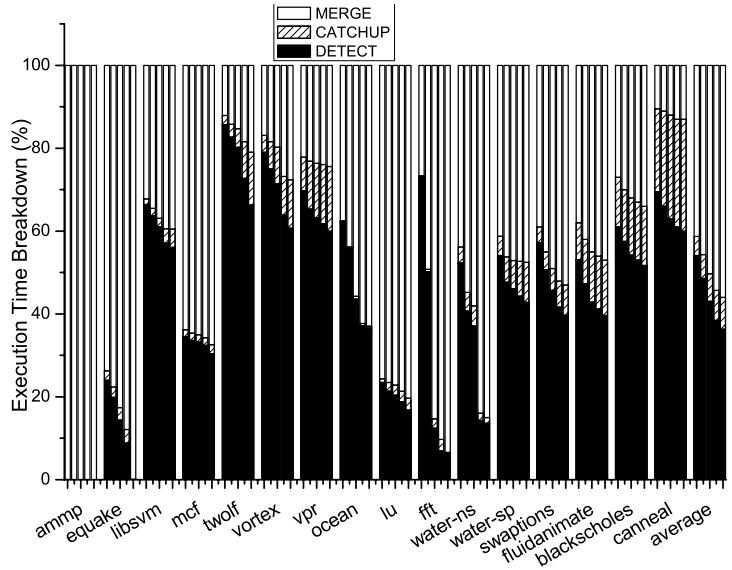
Figure 7(b) shows the performance sensitivity when the number of load/store ports varies from 2 to 12. When the number of load/store ports increases, we also increase the number of MSHRs accordingly to understand the impact of memory bandwidth on the performance gain. The results show that, more load/store ports, or more aggressive memory bandwidth, leads to better performance



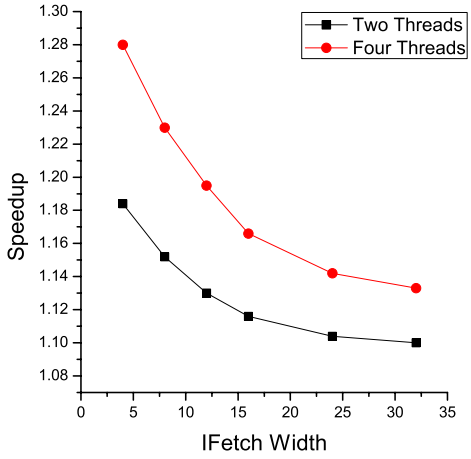
9(a) Fetch History Buffer Size Sensitivity Results



9(b) MMT Ld/St Queue Ports Sensitivity



9(c) Synchronization Behavior as FHB size is varied from 8 to 128 entries



9(d) MMT Fetch Bandwidth Sensitivity

Figure 7: Energy and Sensitivity Results

increases due to shared fetch and execution. The observation is that if the system is less constrained by the memory hierarchy performance, the performance advantage of instruction merging is more promising. This makes sense because, if we reduce contention in parts of the processor other than the fetch unit, then the fetch unit becomes more of a bottleneck. We already showed that as the fetch unit becomes more of a bottleneck, our system is more beneficial.

7. CONCLUSION

In this work, we present Minimal Multi-Threading (MMT), hardware designed to take advantage of the similarities in multiple threads of SPMD programs. Specifically, our design provides three optimizations. First, it removes redundant fetches by allowing multiple threads to fetch the same instruction at the same time. Second, it removes redundant execution by checking for identical inputs on fetch-identical instructions, allowing instructions with identical inputs to be executed together in the machine. Finally, it manipulates fetch priority to increase the time that two threads fetch and execute identical instructions. We show that with these low-overhead modifications to an SMT architecture, execution time and energy consumption decrease significantly. Compared to a traditional SMT

core with a trace cache running the same number of threads as an MMT core, our design achieves average speedups of 1.15 and 1.25 for two and four threads, respectively.

Despite these successes, we still have room for improvement. With a minority of our applications, although they showed improvement, the MMT was unable to synchronize the threads enough to identify many of the identical instructions. In addition, different workloads, such as software diversity in the security domain, have similar execution but different executables, requiring a new, but similar, approach. Finally, we have not evaluated another application class that would benefit greatly from our MMT hardware: message-passing applications.

8. ACKNOWLEDGMENTS

We would like to thank Joel Emer, Bobbie Manne, Janet Franklin, and our anonymous reviewers for their thoughtful comments. This material is based upon work supported in part by the National Science Foundation under Grant No.CCF-1017578, CAREER Award CCF-0855889 to Diana Franklin and a directed research grant from Google to Fred Chong. Prof. Dongrui Fan is supported by the 863 (2009AA01Z103) and 973 (2005CB321600, 2011CB302500) plans of China, NSFC (60925009, 60736012, 60921002), and EU MULTICUBE project (FP7-216693).

9. REFERENCES

[1] B. Beckmann, M. Marty, and D. Wood, "ASR: adaptive selective replication for cmp caches," in *Proceedings of Annual IEEE/ACM Symposium on Microarchitecture*, 2006.

[2] J. Chang and G. Sohi, "Cooperative caching for chip multiprocessors," in *Proceedings of International Symposium on Computer Architecture*, 2006.

[3] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *Proceedings of International Symposium on Computer Architecture*, 2005.

[4] Z. Chishti, M. Powell, and T. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *Proceedings of International Symposium on Computer Architecture*, 2005.

[5] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of International Symposium on Computer Architecture*, 1995.

[6] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of International Symposium on Computer Architecture*, 1996.

[7] S. E. Richardson, "Caching function results: Faster arithmetic by avoiding unnecessary computation," Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, 1992.

[8] S. E. Richardson, "Exploiting trivial and redundant computation," in *Proceedings of International Symposium on Computer Arithmetic*, 1993.

[9] S. F. Oberman and M. J. Flynn, "On division and reciprocal caches. technical report," Technical Report CSL-TR-95-666, Stanford University, 1995.

[10] S. F. Oberman and M. J. Flynn, "Reducing division latency with reciprocal caches," *Reliable Computing, pages 147-153* , 1996.

[11] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating multi-media processing by implementing memoing in multiplication and division units," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[12] S. P. Harbison, "An architectural alternative to optimized compilers," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 1982.

[13] N. Weinberg and D. Nagle, "Dynamic elimination of pointer-expressions," in *Proceedings of International Symposium on Parallel Architectures and Compilation Techniques*, 1998.

[14] A. T. Costa, F. M. G. Franca, and E. M. C. Filho, "The dynamic trace memorization reuse technique," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[15] M. L. Pilla, P. O. A. Navaux, and B. R. Childers, "Value predictors for reuse through speculation on traces," in *Proceedings of Symposium on Computer Architecture and High Performance Computing*, 2004.

[16] W. D. Wang and A. Raghunathan, "Profiling driven computation reuse: An embedded software synthesis technique for energy and performance optimization," in *Proceedings of International Conference on VLSI Design*, 2004.

[17] A. Golander and S. Weiss, "Reexecution and selective reuse in checkpoint processors," *Transactions on High-Performance Embedded Architectures and Compilers, pages 242-268* , 2009.

[18] A. Gonzalez, J. Tubella, and C. Molina, "The performance potential of data value reuse," Technical Report UPC-DAC-1998-23, UPC, 1998.

[19] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," in *Proceedings of International Symposium on Computer Architecture*, 1997.

[20] A. Sodani and G. S. Sohi, "An empirical analysis of instruction repetition," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[21] A. Sodani and G. S. Sohi, "Understanding the differences between value prediction and instruction reuse," in *Proceedings of International Symposium on Microarchitecture*, 1998.

[22] C. Molina, A. Gonzalez, and J. Tubella, "Dynamic removal of redundant computations," in *Proceedings of International Conference of Supercomputing*, 1999.

[23] D. Citron and D. G. Feitelson, "Hardware memorization of mathematical and trigonometric functions," Technical Report -2000-5, Hebrew University of Jerusalem, 2000.

[24] G. Surendra, S. Banerjee, and S. K. Nandy, "Enhancing speedup in network processing applications by exploiting instruction reuse with flow aggregation," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2003.

[25] M. Azam, P. Franzon, and W. T. Liu, "Low power data processing by elimination of redundant computations," in *Proceedings of International Symposium on Lower Power Electronics and Design*, 1997.

[26] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "A complexity effective approach to alu bandwidth enhancement for instruction level temporal redundancy," in *Proceedings of International Symposium on Computer Architecture*, 2004.

[27] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor," *IEEE Computer Architecture Letters* , 2005.

[28] D. A. Connors and W. M. Hwu, "Compiler directed dynamic computation reuse: Rational and initial results," in *Proceedings of International Symposium on Microarchitecture*, 1999.

[29] J. Huang and D. J. Lilja, "Exploiting basic block value locality with block reuse," in *Proceedings of International Symposium on High-Performance Computer Architecture*, 1998.

[30] J. Huang and D. J. Lilja, "Exploring sub-block value reuse for superscalar processors," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[31] Y. H. Ding and Z. Y. Li, "A compiler scheme for reusing intermediate computation results," in *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, 2004.

[32] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: Employing hardware migration to specialize cmp cores on-the-fly," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[33] S. Harizopoulos and A. Ailamaki, "Steps towards cache resident transaction processing," in *Proceedings of the 30th Very Large Database (VLDB) conference*, 2004.

[34] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong, "Multi-execution: Multi-core caching for data similar executions," in *Proceedings of International Symposium on Computer Architecture*, 2009.

[35] U. Acar, G. Blelloch, and R. Harper, "Adaptive functional programming," in *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, January 2002.

[36] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, and A. González, "Thread fusion," in *ISLPED '08: Proceeding of the 13th international symposium on Low power electronics and design*, pp. 363–368, ACM, (New York, NY, USA), 2008.

[37] C. C. Chang and C. J. Lin, "Libsvm: a library for support vector machines," 2001.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of International Symposium on Computer Architecture*, 1995.

[39] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[40] ITRS, "International technology roadmap for semiconductors." http://www.itrs.net/Links/2009ITRS/Home2009.htm, 2009.

[41] M. Horowitz *et al*., "Scaling, power, and the future of CMOS," in *Proceedings of the IEEE International Electron Devices Meeting*, 2005.

[42] D. R. Fan, N. Yuan, J. C. Zhang, Y. B. Zhou, W. Lin, F. L. Song, X. C. Ye, H. Huang, L. Yu, G. P. Long, and H. Zhang, "Godson-t: A many-core processor for efficient multithreaded program execution," *Journal of Computer Science and Technology* , 2010.

[43] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News, Volume 25, Issue 3, Pages: 13 - 25* , 1997.

[44] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of International Symposium on Microarchitecture*, 1996.

[45] B. Black, B. Rychilik, and J. P. Shen, "The block-based trace cache," in *Proceedings of International Symposium on Computer Architecture*, 1999.

[46] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of International Symposium on Computer Architecture*, 2000.