AN ABSTRACT OF THE THESIS OF

Adam W. Browning for the degree of Master of Science in Electrical and Computer Engineering presented on September 29, 2006.

Title: Instruction Fetching, Scheduling, and Forwarding in a Dynamic Multithreaded Processor

Abstract approved: _____
Ben Lee

Dynamic multithreaded processors attempt to increase the performance of a single sequential program by dynamically extracting threads from sources such as loop iterations. The scheduling of instructions in such a processor plays a vital role in the amount of thread level parallelism that can be extracted and thus the overall system performance. Three new systems are presented in this thesis to increase the performance of instruction scheduling and value forwarding in a dynamic multithreaded processor.

Conflicts within the instruction cache from multiple threads requesting the same cache blocks reduces instruction fetch performance. A new instruction scheduling and fetching method is presented that uses the unique nature of dynamically generated threads to increase fetch performance while keeping the complexity of the instruction cache low. Performance for this new fetching scheme is on par or better than the current instruction fetching method used by the simulated processor.

The overall performance of a dynamic multithreaded processor is limited by inter-thread dependencies that arise from generating threads that are not fully independent or parallel. A new inter-thread forwarding system is presented that speeds up the forwarding of values between threads, thus reducing the number of stalls from inter-thread dependencies. To further reduce the number of stalls, a critical path system is implemented that dynamically identifies and prioritizes instructions that produce inter-thread dependency values.

Instruction Fetching, Scheduling, and Forwarding in a Dynamic Multithreaded Processor

by
Adam W. Browning

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 29, 2006
Commencement June 2007

Master of Science thesis of Adam W. Browning presented on September 29, 2006

APPROVED

Major Professor, representing Electrical and Computer Engineering

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Adam W. Browning, Author

ACKNOWLEDGEMENTS

The author expresses sincere appreciation to the following individuals: David Zier for all his help on getting the simulator working. My family for all their support and help through the difficult times, and letting me monopolize their computers for simulations. Finally, my wonderful girlfriend Allison for keeping me sane and being my oasis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Instruction Fetching, Scheduling, and Forwarding in a Dynamic Multithreaded Processor

1 Introduction

The high performance of modern superscalar processors is a direct result of exploiting *instruction level parallelism* (ILP) in programs by executing multiple instructions per cycle. However, the performance of superscalar processors has become limited due in part to *true dependencies* and *control flow dependencies*. The concept of *multithreading,* which uses *thread level parallelism* (TLP), has attracted great interest in both academia and industry as a new source of exploitable parallelism. A multithreaded processor makes use of multiple threads of execution to exploit both ILP and TLP [1]. TLP compensates for a lack of ILP per thread and improves the utilization of execution resources and thus performance.

Several multithreaded architectures have been proposed in the past, but most rely on generating threads from multiple independent programs [2][3]. These multithreaded processors, however, do no increase the performance of the individual programs. Alternatively, threads can be generated from a single sequential program, allowing its performance to be increased using multithreading techniques [4][5]. Static and Dynamic Multithreading are two methods for extracting this type of TLP. *Static* methods include the use of compilers that parallelize source codes or binary annotators [6] that modify already compiled programs. However, static methods cannot exploit the dynamic behavior of programs, and thus can only produce conservative levels of TLP. On the other hand, *dynamic* multithreading is performed by identifying and spawning threads at run-time to take advantage of dynamic behavior, such as loops and subroutine calls. Processors that use dynamic thread generation techniques will be referred to as *dynamic multithreaded processors* throughout the rest of this thesis.

There are several bottlenecks that prevent dynamic multithreaded processors from reaching their maximum performance. One bottleneck, which is shared by both superscalar and multithreaded processors, is the underutilization of the instruction fetch bandwidth due to cache misses, branch misspeculations, and partial cache-line

fills [7]. Since multithreaded processors are dependent on TLP for performance, reduced instruction fetch performance for the threads reduces both ILP and TLP. This thesis proposes a new instruction scheduling and fetching method called *Fetch Block Grouping* (FBG) to increase the instruction fetch bandwidth and performance of a dynamic multithreaded processor. The idea behind FBG is to exploit threads that share the same instructions and cache blocks within the instruction cache. FBG uses this characteristic to allow multiple threads requesting the same cache block to be serviced as a single request. This allows FBG to increase the number of instructions fetched per cycle without increasing the number of ports on the instruction cache.

Another bottleneck inherent to dynamic multithreaded processors is the handling of *inter-thread dependencies* in the form of register and memory values, which occur when the threads are not fully independent (i.e., parallel). This thesis proposes two new methods for managing inter-thread dependencies. The first is a new inter-thread forwarding system (ITFS) which speeds up values forwarded between threads. Careful tracking is needed when using ITFS in order for the system to recover from branch and other misspeculations. The second method involves a scheme to decrease the time threads must stall while waiting for values to be produced by other threads. The *Critical-path-based Instruction Scheduler* (CIS) uses *critical path* techniques to identify and prioritize instructions that belong to the *inter-thread critical path* and the *intra-thread critical path*. Execution time for the threads and the program should be reduced by prioritizing instructions on the inter-thread and intra-thread critical paths.

To determine the performance benefits of the proposed solutions, the Dynamic Simultaneous Multithreading (DSMT) processor [8] is modified and simulated using a new object-oriented simulator.

The remainder of this thesis is organized as follows. Section 2 will present related works on increasing instruction fetch performance, both in superscalar and multithreaded processors, and critical path instruction scheduling. Section 3 will present background information on the DSMT architecture and its current inter-thread dependency resolution. The new FBG technique will be presented in Section 4. CIS

will be presented in Section 5, followed by the new inter-thread forwarding system in Section 6. In Section 7, simulation study of FBG, CIS, and the new inter-thread forwarding system will be presented. Future work will be suggested in Section 8, followed by conclusions in Section 9.

2 Related Works

Instruction fetch performance is critical to the overall processor performance. Superscalar processors require sufficient instructions fetched per cycle to keep their execution units busy. Multithreaded processors are more sensitive to fetch performance since they rely on TLP from multiple threads, which is limited by low fetch performance. There have been many proposals over the years for increasing fetch performance, for both superscalar and multithreading processors.

One architecture proposed to increase the effective fetch bandwidth in superscalar processors is the *decoupled front-end* [9][10]. In such an architecture, the branch predictor is decoupled from the instruction fetch unit using a *fetch target queue* (FTQ). This allows both the branch predictor and the fetch unit to operate independently; i.e., a stall in one will not affect the other. Reinman *et al*. [9] extended the decoupled front-end by adding a *fetch target buffer* (FTB). The FTB extends the *branch target buffer* (BTB) to store variable length *fetch blocks*. A fetch block begins at a branch target and ends in a strongly biased taken branch. This allows several branches that are strongly biased as not taken to contain within the fetch block, effectively ignoring them and increasing the instruction fetch performance.

A very well known method for instruction fetch performance is through the use of a *trace cache* [11][12], which is implemented in the Pentium 4 processor [13]. The trace cache captures instruction segments in *execution order* instead of the *program order* stored in higher levels of memory. Each segment is called a *trace* and can contain several blocks of instructions and branches, both taken and not taken, and is stored in a decoded format. If the same segment of code is requested in the future, the trace cache can service the request in a single cycle without further processing or accessing other levels of memory, thereby increasing the instruction fetch performance. For example, the Pentium 4 processor can provide three decoded micro-ops per cycle using its L1 trace cache[13].

Many different methods have been explored for increasing instruction fetch

performance in multithreaded processors. Some have ranged from simple modifications to fetch units found in superscalar processors [14], to completely new font-end designs that use multiple fetch units [15][16]. Tullsen *et al*. [17] proposed a modification to the instruction cache and fetch units in a SMT processor that splits the instruction fetch bandwidth between two threads. Due to the high occurrence of branch instructions and branch misalignments found in typical programs, it is hard for a single thread to fill a wide instruction fetch bandwidth. By splitting the fetch bandwidth between two threads, it is more likely that each can fill its allotment. For example, it is more likely for two threads to fetch four instructions than it is for a single thread to fetch eight instructions. The proposed instruction scheduling policy selects the threads to fetch using a heuristic, such as the number of instructions for each thread in the ROB. The ICOUNT2.8 algorithm provided the best performance by fetching for the two threads with the least number of instructions in the instruction queue and instruction decode/rename stages. The 2.8 signifies a dual-port instruction cache was used, each port able to fetch eight instructions per cycle.

Moure *et al*. [15] proposed a new instruction cache and fetch structure for an SMT processor consisting of *fetch clusters*. Each cluster is a self-contained instruction fetching system, with an instruction cache, branch predictor, fetch buffer, and instruction decoder. During multithreaded execution, each thread is given its own fetch cluster. However, the fetch clusters can be used in different ways during single thread execution. This includes grouping them together to create a larger effective instruction cache, performing *multi-path execution* [18] on hard to predict branches, and increasing the effective issue width to the functional units.

Falcon *et al*. studied the application of fetch performance enhancements for superscalar processors on SMT processors [14]. Their studies included the use of the decoupled front-end previously described, and other modifications to increase branch prediction bandwidth and accuracy. Their results showed how instruction fetch performance can be improved in SMT by fetching more instructions from a single thread each cycle, eschewing the common solution of fetching from multiple threads.

This had the benefit of using simpler instruction caches and keeping the front-end of the processor simple.

While instruction fetching plays a significant role in the performance of a processor, deciding which instruction to issue also affects performance. Several papers have been published in the past describing the *critical path* in a program and its affects on program performance. The critical path is the longest path through a program or code segment made of chains of dependencies between the instructions. Execution time for the program or code segment is related directly to the execution time of the instructions on the critical path.

Identifying the critical path in a program can be done at compile time [19][20] or performed by the processor during run-time[21][22]. Tune *et al*. presented a method for identifying and utilizing instructions that belong to the critical path during execution of a program [21]. In their proposal, a heuristic is used to identify instructions that possibly belong to the critical path, which is used to increment or decrement *saturating counters* for the instructions. Once the counter value for an instruction passes a *threshold*, it is considered critical and *flagged* during future instruction fetches. Flagged instructions are given priority within the pipeline, such as access to branch and data predictors and issuing priority to the function units. This was expanded upon by Fields *et al*. to include the affects of the microarchitecture on the instruction execution [22]. Instead of using a heuristic, a *token* system was implemented. Instructions are randomly given a token, and if they produce a last-arriving operand to another instruction, the token is also propagated to the new instruction. If the token is still in the system after a specific number of instructions have been committed, the original instruction has its saturating counter incremented, similar to the system proposed by Tune *et al*.

3 DSMT Background

The DSMT processor was proposed by Ortiz-Arroyo and Lee [8] as a way to increase individual program performance on an SMT processor using dynamic threading. Threads are produced from loop iterations in a program and are used to fill the *contexts* in the SMT execution back-end. The DSMT microarchitecture is shown in Figure 1 below. Modifications to the base SMT architecture are shown in light gray, while new additions are in dark gray.

Each thread in DSMT is represented by a *Context*, which contains the register file and information for managing the thread. An additional unit to the base SMT microarchitecture is the *Loop Detection Unit* (LDU), which is responsible for detecting and collecting information on loops. The LDU supplies this information to the *Thread Creation and Initialization Unit* (TCIU). As the name suggests, the TCIU is responsible for creating and initializing the individual threads in DSMT, and communicating with the ROB and Contexts. Lastly, the Scheduler unit controls how the processor fetches instructions and is the focus of the changes for FBG.

During the execution of a program, DSMT operates in either non-DSMT, pre-

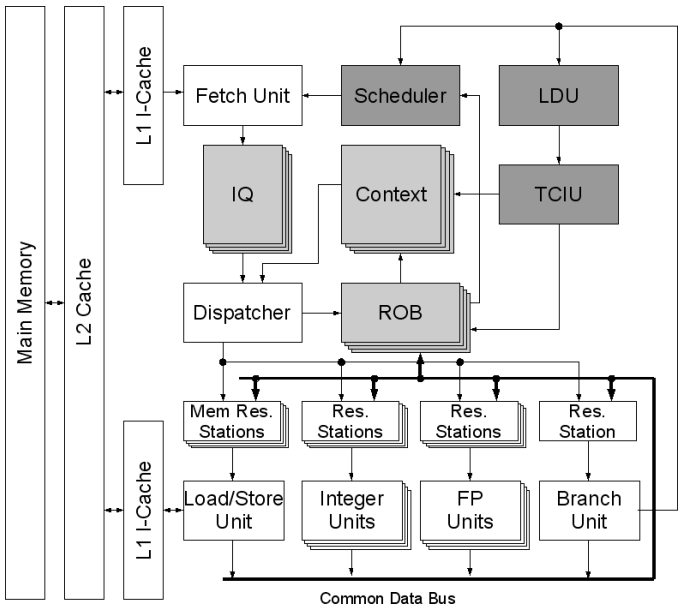


Figure 1: DSMT microarchitecture

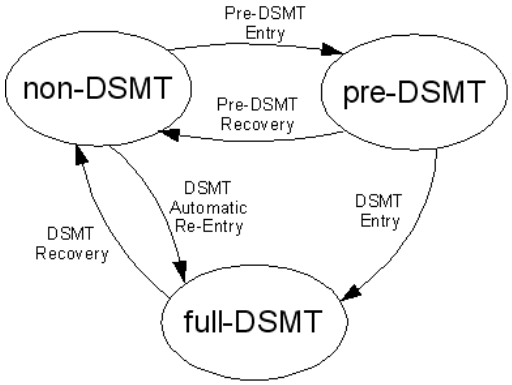


Figure 2: DSMT transition graph.

DSMT, or full-DSMT mode. The system can transition between the three modes as indicated in Figure 2. In the non-DSMT or Normal mode there is a single active thread and the processor behaves like a wide-issue superscalar processor. Once a loop is detected by the LDU, the system enters the pre-DSMT mode in which information on the loop and possible threaded performance are computed. If the system predicts that performance for the loop would increase if threaded, the system enters full-DSMT mode. If the system performs well with the loop threaded, it is marked as Good in the LDU and the next time the system encounters the loop it will transition to full-DSMT mode immediately.

During full-DSMT mode, the simultaneous execution of loop iterations is performed. A single *non-speculative* thread, or *head context*, always exists while in full-DSMT mode, which is the only thread from which speculative threads are spawned. The threads are connected in a ring fashion with a Head and Tail field in the TCIU to indicate the first and last threads, respectively.

DSMT adds several utility bits to the registers in the Contexts beyond the traditional Valid (V) bit and ROB tag field to help resolve inter-thread dependencies. The first is the Ready (R) bit, which indicates that the value in the register was produced by an instruction in the thread logically preceding the current instruction. If the R-bit is cleared, then a speculative thread cannot use the value and must look to a predecessor thread for the correct value. The second bit is the Dependency (D) bit. When the D-bit is set an inter-thread dependency exists for the register, and if the R-

bit is not set, the value must be read from a predecessor thread. The D-bits from previous threads are used to set the D_Anchor register within the TCIU to help resolve inter-thread dependencies in future threads. Finally, the last utility bit is the Load (L) bit, which is set on a register whenever its value is read from another thread. When a thread commits an instruction to its register file, a quick check is made of all the successor threads to see if they have the L-bit set on the same register. If so, then an inter-thread misspeculation has occurred and all successor threads are squashed and their execution restarted at the beginning of the iterations.

Inter-thread dependency resolution occurs during the dispatching of instructions. Each operand of the instruction is checked within the register file, checking the R-bit to see if it is safe to read from its own Context. If the R-bits are set, the values are read from within the Context and the instruction is dispatched. However, if an R-bit is not set, first level speculation, called *register dependence speculation*, is performed. The D_Anchor within the TCIU is checked to determine from where the register value should be read. If the D_Anchor is set for the register, then an inter-thread dependency occurred in previous executions and there is a good chance it will occur again, so the immediate predecessor thread is checked. At this point, if the R-bit is set on the register in the immediate predecessor and it is Valid, the register value is forwarded. However, if the register is not ready, the *consumer* thread, or thread attempting to read the value, is stalled for a set number of cycles to allow the *producer* or predecessor thread time to execute an instruction which writes to the register. If the register is still not ready after this time, or if the D_Anchor bit was not set, second level speculation is performed in which a search is made for the last thread to write to the register. Finally, if the value has not be produced by any thread, the value is read from the head context and it is assumed that the value was produced outside of the loop.

In DSMT, there are two levels in which values can be forwarded, (1) intra-thread, and (2) inter-thread. Intra-thread forwarding occurs similarly to value forwarding in a superscalar processor. Inter-thread forwarding occurs between threads, as previously

described, which is significantly slower than intra-thread forwarding. While values can be passed within a thread from the common data bus and ROB entries, values must first be committed to a register file before they can be forwarded to another thread. This creates a performance bottleneck when handling inter-thread dependencies, which is addressed by the new inter-thread forwarding system described in Section 6.

4 Fetch Block Grouping

In multi-program multithreaded processors, such as SMT [3], instruction fetching bandwidth is split between multiple threads. This allows for a better utilization of the instruction fetch bandwidth and for multiple threads to be serviced each cycle. However, implementing a dual-ported instruction cache is complex and requires additional hardware. Multiple banks are needed to avoid, or at least reduce, conflicts from multiple threads accessing the same cache line or bank. A popular method used is interleaved banks [23]. Additional logic is required to prevent bank conflicts and to make the cache non-blocking. All these additions increase the complexity of the cache in terms of decoders, sense amplifiers, multiplexers, and memory cells. When using a multi-ported instruction cache in a dynamic multithreading processor, such as DSMT, bank conflicts significantly increase. Threads that share instructions, such as those generated from loop iterations, will attempt to access the same instructions and cache blocks in the shared instruction cache. Handling these conflicts requires either stalling and servicing threads sequentially, thus removing the benefits to using a multi-ported cache, or additional hardware must be added, further increasing the complexity of the cache.

The FBG scheme does not require a dual or multi-ported instruction cache to increase instruction fetch performance. Instead, FBG uses a simple, single-ported cache and makes modifications to the instruction Scheduler and Fetch unit to manage the cache block conflicts. Currently, DSMT uses a dual-ported instruction cache and a modified version of the ICOUNT2.8 scheduling policy [3], called ICOUNT2.8-modified [8]. Each cycle, the two threads with the fewest number of instructions in their instruction queues are chosen and scheduled for fetching. During scheduling, the cache block addresses that each thread will access are computed by taking the next PC addresses for each thread and applying a *block mask*. For example, if each port on the instruction cache is eight instructions wide, and each instruction is 64 bits wide, then each cache block is 64-bytes and the block mask will clear the last six bits of the next PCs to compute the block addresses.

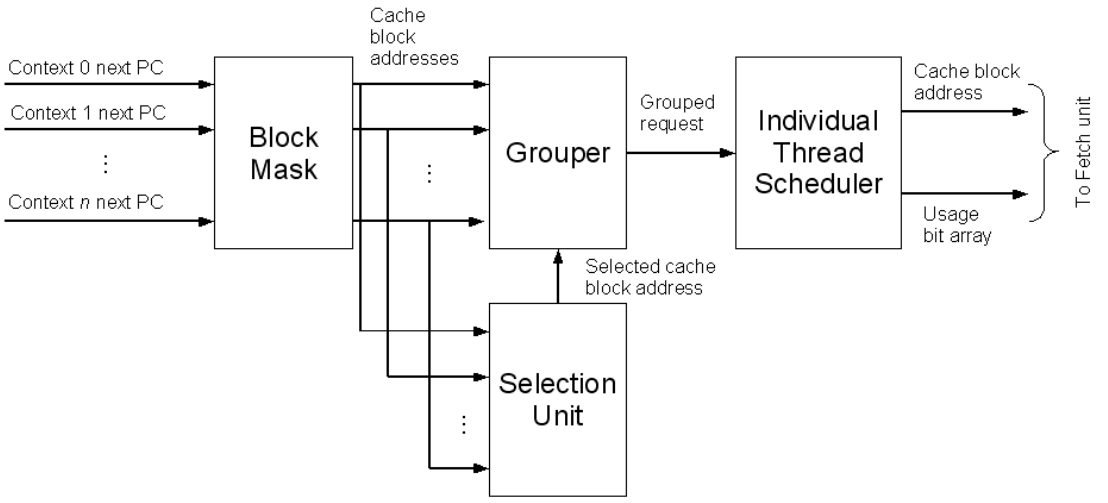


Figure 3: FBG block diagram.

The diagram shown in Figure 3 illustrates how FBG works. The block mask produces the cache block addresses for the individual threads, similar to the current Scheduler in DSMT. These block addresses are then fed into the *selection unit*. Since a single-ported instruction cache is used, only a single cache block may be fetched per cycle, thus a selection unit is needed to determine which cache block from the available requests will be fetched. Several selection algorithms have been tested with FBG, which are described in Table 1.

Table 1: FBG Selection Algorithms

Name	Description
CtxOrder	The head (i.e., non-speculative) thread is selected. If the head thread is not ready for fetching, the first speculative thread is chosen and so on. By prioritizing the head thread over the speculative threads, its instruction queue will be kept filled and execution of the thread will not be stalled.
MemOrder	The thread that requests the cache block with the address earliest in program memory is selected. This prioritizes the speculative threads and prevents them from being starved of instructions.
ICOrder	Similar to ICOUNT proposed by Tullsen *et al* [3]. The thread with the least number of instructions in its instruction queue is selected. This thread will most likely provide the best throughput and therefore increase the TLP of the system.

	Ctx 0	Ctx 1	...	Ctx *n*
Inst 0	T	T		T
Inst 1	T	F		T
...				
Inst *m*	T	F		F

Figure 4: FBG instruction usage bit array.

Once the selection unit has chosen a cache block to fetch, the selected address and the cache block addresses from all the threads are fed into the *grouper*. The grouper performs comparisons between the cache block addresses and the block address from the selection unit. The output of the grouper indicates which threads are requesting the same cache block as the one selected by the selection algorithm. Thus the threads included in the grouping should have scheduling performed.

The final stage of FBG, *individual thread scheduler*, takes the grouped requests from the grouper and performs scheduling for the individual threads within the group. For each thread, branch prediction is made and the offset into the cache block, instruction count from the offset, and the next PC are computed. These are used to create the *instruction usage bit array*, as shown in Figure 4. This bit array is used by the Fetch unit to determine which thread or threads each instruction should be copied to once the cache block has been fetched. The rows in the array correspond to the individual instructions within the cache block, while the columns represent the threads. For example, instruction 0 has a value of True (T) for all the threads in Figure 4, indicating that all threads will have the first instruction in the cache block copied to their instruction queues. Instruction 1 has its value for thread 1 set to False (F), meaning thread 1 will not receive a copy of the second instruction in the cache block.

While FBG was designed to leverage a single-ported instruction cache, it can also be modified to make use of an instruction cache with multiple ports. Rather than having the selection unit choose a single cache block from the available requests, it chooses multiple cache blocks. In the event that all threads are requesting the same cache block, there is no need to select additional block addresses. The grouper is expanded to perform comparisons for all the selection block addresses provided to it

by the selection algorithm. Similarly, multiple usage bit arrays and cache block addresses are produced and given to the Fetch unit. This modified FBG scheme does not require the instruction cache to manage bank contentions more complex than what is currently seen in multiported instruction caches, giving FBG an advantage over the ICOUNT2.8 fetching method. Performance for FBG when using a single-ported and dual-ported instruction cache is presented in Section 7 and compared to the performance of ICOUNT1.8 and ICOUNT2.8.

5 Critical-Path-Based Instruction Scheduling

CIS attempts to increase the performance of inter-thread dependency resolution by prioritizing instructions that produce dependency values. In Figure 5 the diagram of CIS is illustrated as well as how predictions for the two critical paths are made. Each cycle, CIS uses a heuristic to search through the reservation stations within DSMT for the oldest instruction or instructions that were not issued in the current cycle. This is a modified version of the QOld algorithm proposed by Tune *et al.* [21] in predicting instruction criticality in a superscalar processor. The QOld heuristic also searched for the oldest instructions that did not issue, but it searched the instruction queue since the system Tune *et al*. used was based on register renaming and issued instructions out-of-order from the instruction queue. Instructions found by the heuristic have their entries in the ROB *marked*. When instructions are committed, the ROB informs CIS if an instruction is mark or unmarked. If an instruction is marked when it is committed, a saturating counter is incremented, while unmarked instructions have their counters decremented or removed if the value drops to zero. Once a counter passes a threshold value, the instruction is considered critical and is flagged as being on the intra-thread critical path.

Once flagged instructions are given priority within the system, there is a chance they will no longer meet the requirements to be marked by the heuristic during future executions. Tune *et al*. showed how using separate increment and decrement values for the saturating counters allows the critical path predictor to continue predicting an instruction as critical for several executions, even when the heuristic has stopped marking the instruction. Using different increment and decrement values allows the saturating counter to remain above the threshold for several executions of the instruction before it drops below and is not longer considered critical.

While the QOld-modified heuristic is used to predict instructions on the intra-thread critical path, a different method is needed to determine the inter-thread critical path. Instructions on the inter-thread critical path can easily be identified as the

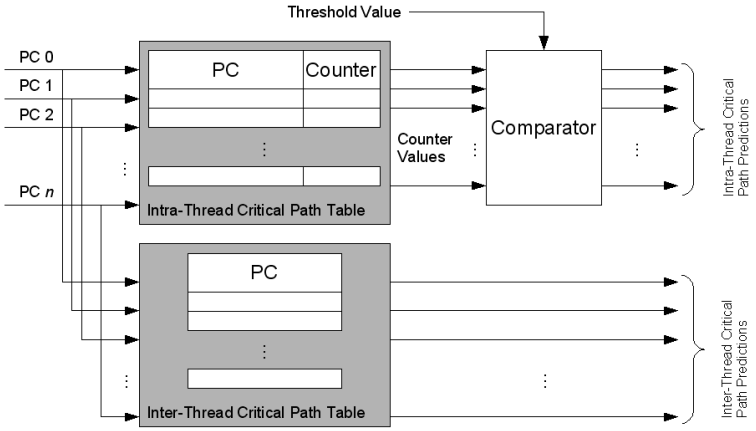


Figure 5: Critical path prediction using CIS.

instructions that produce values forwarded to other threads. If a consumer thread is forced to stall waiting for a specific instruction from another thread to produce a value, the producer instruction is marked as belonging to the inter-thread critical path. On the other hand, if the value can be immediately read from a register file from another thread, no instruction is marked. The instruction producing the needed value in those cases are already executing fast enough; therefore, further prioritizing is not needed. However, if this situation changes, the producer instruction will be marked. When instructions are committed that are marked for the inter-thread critical path, no saturation counters are used. Instead, an entry is added to the inter-thread critical path table and the instruction will always be flagged as on the inter-thread critical path.

When using CIS there are two different critical paths, one for the critical path within a thread and the critical path between threads. A choice must be made about which path is more important when multiple instructions are ready to issue that are flagged for one or both of the critical paths. When only a single critical path is used, as was done by Tune *et al*. [21], all instructions on the critical path are issued to the functional units first, followed by non-flagged instructions. Instructions on the inter-thread critical path produce values for inter-thread dependencies, which affect the amount of TLP in the system. Therefore, issuing priority is always given to instructions that are on the inter-thread critical path. Only after all instructions that are flagged for the inter-thread critical path have been issued, are instruction on the intra-

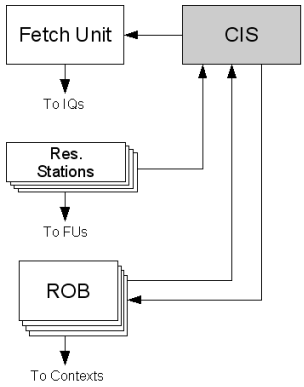


Figure 6: CIS in the DSMT architecture.

thread critical path issued. Finally, if there are any more available function units, non-flagged instructions are issued.

In the original work by Tune *et al*. [21], instruction criticality was predicted when the instructions were fetched. This was appropriate since their simulation environment used a system based on register renaming and issued instructions from the instruction queue. In DSMT, however, instructions are dispatched to reservation stations, presenting a new stage in which criticality predictions could be made. Instead of immediately predicting if a fetched instruction is on the intra-thread or inter-thread critical path, the system could wait until the instructions are dispatched from the instruction queue.

The final placement of CIS within the DSMT architecture is shown in Figure 6. The Dispatch unit in DSMT can dispatch several instructions from each thread per cycle. In configurations with a high number of threads, such as four and eight contexts, the number of instructions dispatched can be very high. For example, if the Dispatch unit can dispatch four instructions per thread and there are four active threads, there could be up to 16 instructions dispatched that would require criticality predictions. On the other hand, if criticality predictions were made when the instructions are fetched, the number of predictions would be much less. For example, in a configuration using a single-ported instruction cache that can fetch a cache block eight instructions wide, only eight predictions would be needed per cycle. If FBG is used in the example, then it is possible that those eight instructions could be copied to

the instruction queues of multiple threads, thus increasing the number of effective predictions made.

6 New Inter-Thread Forwarding System

When inter-thread dependency resolution is performed by the Dispatch unit in DSMT, register values are forwarded between threads. As previously described, this method is significantly slower than intra-thread value forwarding and has an adverse affect on the overall performance. The proposed ITFS in this thesis expands the current inter-thread forwarding system to allow forwarding from the forwarding bus and ROB entries of other threads. Rather than stall threads when waiting on a register in another thread to become Valid, the forwarding bus and ROB are checked for the desired value. If the register value is not available, the system stalls for a cycle, similar to the current inter-thread forwarding system. However, if the value is present, ITFS can forward the value immediately and continue executing the consumer thread.

The biggest obstacle to forwarding values from the forwarding bus and ROB of other threads is the managing of misspeculations, such as branch and inter-thread misspeculations. These misspeculations can cause *producer instructions*, or the instructions which have had their values forwarded from the ROB, to be squashed before they have committed their values to the register file. When this occurs, the consumer instruction, and possibly the entire consumer thread, must be flushed and the execution restarted at the point of misspeculation.

In order to recover from misspeculations, both the consumer and producer instructions must be tracked. Whenever a value is forwarded from the ROB or forwarding bus, a *forwarded pair* is generated and placed in a *Forwarded Table*, as illustrated in Figure 7. A forwarded pair consists of the ROB tags and context numbers for both the consumer and producer instructions, along with the register number of the forwarded value. Entries are kept in the table until either the consumer instruction is squashed or the producer instruction is committed.

Extra bits are added to the entries in the ROB to help manage the forwarded values. Whenever an entry in the ROB is read, a Forward (F) bit is set on the entry. Similarly, the consumer instruction has a Read (R) bit set on its ROB entry whenever it is

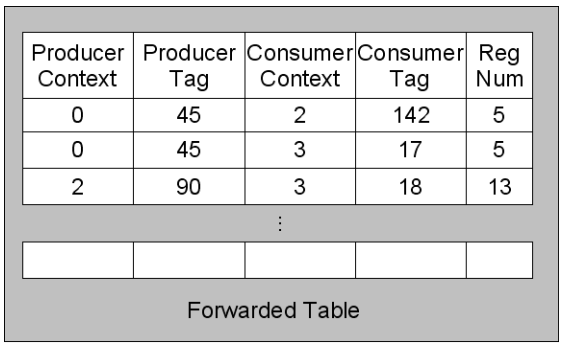


Figure 7: Forwarded Table with forward pairs.

involved in a forwarded pair. These additional bits on the ROB are used to ensure program correctness and to prevent corrupting the register files. Entries in the ROB with set R-bits are not allowed to commit until the inter-thread forwarding is resolved.

Two tables are needed in the new inter-thread forwarding system to track the number of forwarded pairs in which each instruction is participating and is shown in Figure 8. Since an instruction can be a producer instruction for multiple consumers, a table, indexed by the context number and ROB tag of the producer instruction, tracks the number of outstanding forwards for the instruction. Another table is used to track the number of forwarded pairs in which each consumer instruction is participating. When producer instructions are committed or consumer instructions are squashed, the counters in the tables are decremented. If the values drop to 0, the entries are removed and the appropriate bits on the ROB entries are cleared. This helps to ensure that a consumer instruction that has multiple operands forwarded from other threads does not have its R-bit cleared when the first forwarded pair is resolved.

There are several types of misspeculations that can cause either producer or consumer instructions to be squashed. The first type of misspeculation, and most simple to manage, is a branch misspeculation that squashes a consumer instruction. To resolve this misspeculation, the producer instruction has its F-bit cleared, if it is not part of any other forwarded pairs, and the forwarded pair entry in the Forwarded Table is removed. If a producer instruction is squashed, on the other hand, the resolution is more complicated. The consumer thread must be flushed beginning at the consumer instruction and execution restarted at the PC of the consumer instruction. Figure 9(a)

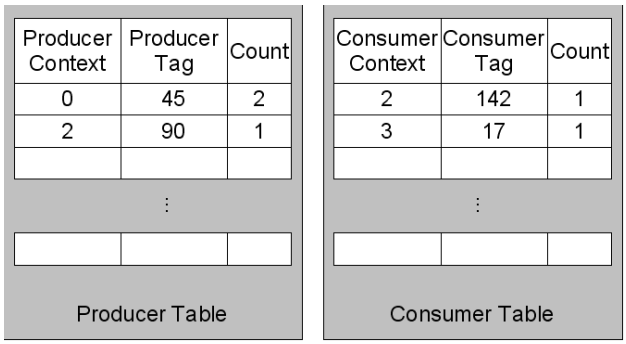


Figure 8: Producer and consumer tracking tables in new inter-thread forwarding system.

shows an example of a branch misspeculation in the head thread causing a speculative thread to flush its pipeline beginning at a consumer instruction.

When there are multiple active speculative threads, a thread can be a consumer thread for one forwarded pair and a producer thread for another forwarded pair. In such a situation, a misspeculation in one thread can lead to multiple misspeculations in other threads, which is referred to as a *cascading misspeculation*. Figure 9(b) illustrates a simple branch misspeculation in the head thread can lead to a cascading misspeculation that affects all other threads.

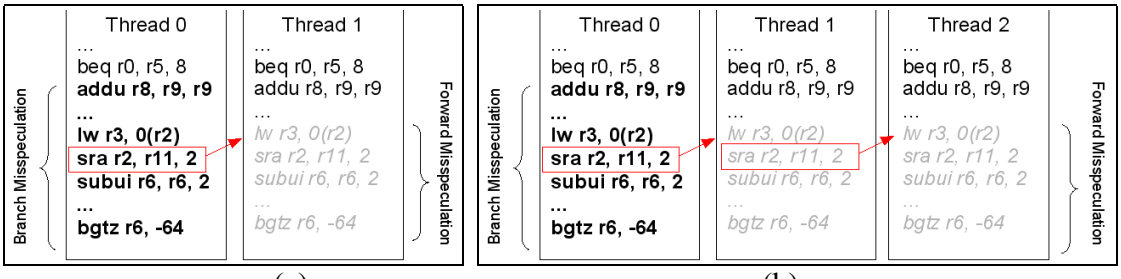


Figure 9: (a) Branch misspeculation causing a forwarding misspeculation and (b) a more complex cascading misspeculation

7 Simulation Results

In order to test the performance of the proposed techniques, the DSMTSim simulator, an object-oriented simulator created using the NetSim architectural simulator suite [24], was modified and used to simulate benchmarks from SPEC2000 and MediaBench [25]. DSMTSim is a cycle-accurate, execution-based simulator of the DSMT microarchitecture with extensive statistics tracking capabilities. Each benchmark is run with two, four, and eight threads, along with a baseline simulation using a single context. The baseline configuration is used to compare the relative performance increase for each benchmark. Each benchmark is run from the start of the program to the end, unless otherwise specified. The specific configuration settings used in all simulations are listed in Table 2 and Table 3.

Table 2: DSMT configuration.

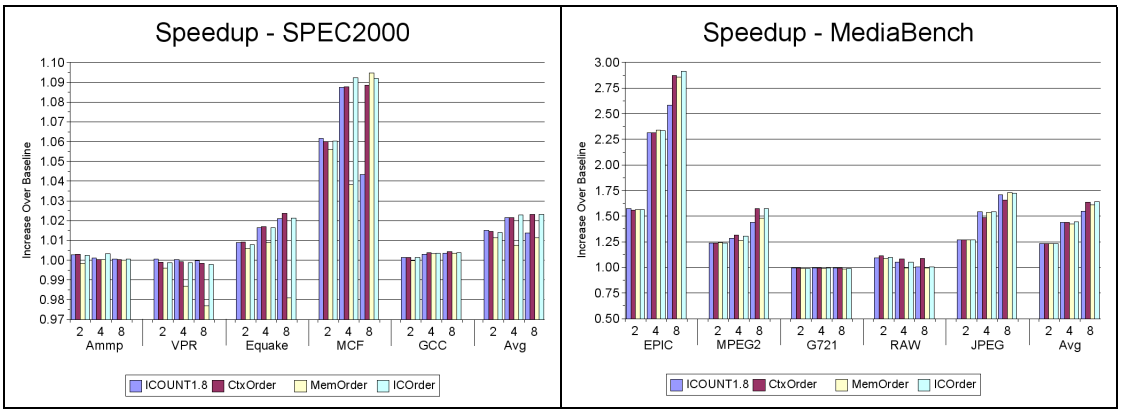
Fetch Width inst./cycle	Issue Width inst./cycle	Inst. Queue size/ctx	L/S Queue size/ctx	ROB size/ctx	L1 Cache Inst./Data	L2 Cache	Shared BTB
8	4 per ctx	32	256	64	32KB/32KB 4-way 1 port/4 ports miss 8 cycles	Unified 128KB 8-way miss 60 cycles	512 fully assoc. 2 bits

Table 3: DSMT Functional units configuration.

	Int ALU	Int Mult.	Int Div.	FP Add	FP Mult.	FP Div.	Load/Store
# of FU	3	3	3	2	2	2	4
# of Stages	1	4	8	4	7	16	1

7.1 Fetch Block Grouping Results

The first simulations was used to simulate FBG with the three selection algorithms previously described and to compare the results against DSMT using ICOUNT1.8. All simulations used a single-ported instruction cache capable of fetching eight



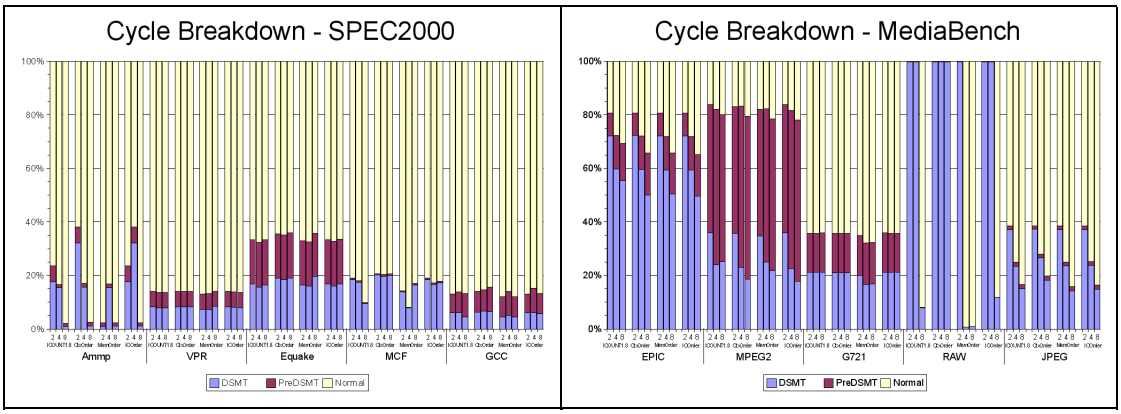
(a) (b)

Figure 10: Speedup for ICOUNT1.8 and FBG for (a) SPEC2000 and (b) MediaBench benchmarks.

instructions per cycle. The overall increase in system performance when using ICOUNT1.8 and FBG with two, four, and eight active threads is shown in Figure 10. Overall, FBG performs on par or better than ICOUNT1.8, with many benchmarks showing significant increases using FBG. The first trend the graph shows is how the SPEC2000 benchmarks respond differently than the benchmarks from MediaBench. While the average MediaBench program sees a speedup of just over 1.5 with eight threads, the average SPEC2000 program increase a little over 1.02. This can be attributed to the nature of the programs in the two benchmark suites. SPEC2000 benchmarks have more irregular data structures and control flow, thus reducing the number of loops that DSMT can multithread. On the other hand, MediaBench programs use algorithms that are heavily loop based, such as EPIC and MPEG2, which are based on matrix operations.

Another trend highlighted by Figure 10 is how the benchmarks perform differently depending on the selection algorithm used. While none of the selection algorithms have a clear advantage over the other algorithms, some benchmarks respond better to one algorithm over the others. For example, the RAW benchmark from MediaBench performs better when the CtxOrder selection algorithm is used, increasing the overall performance significantly beyond ICOUNT1.8 performance.

One trend shown in Figure 10 is the poor performance of the MemOrder selection

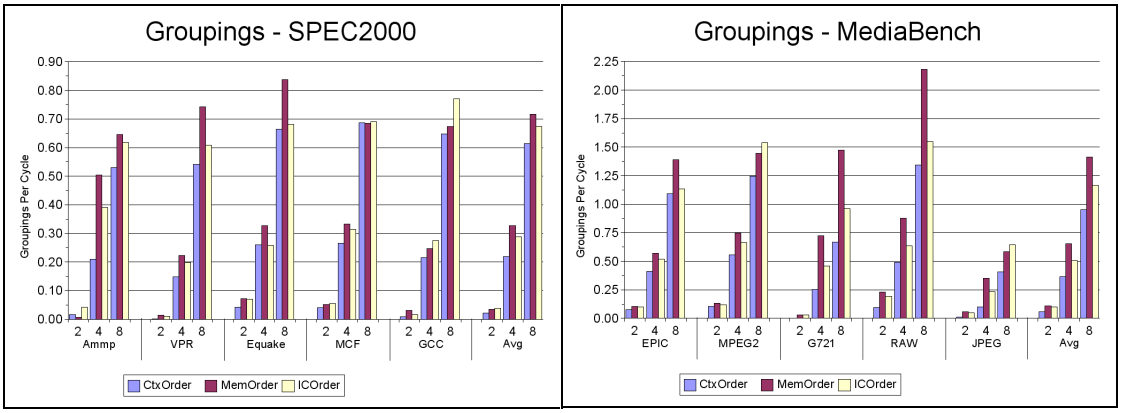


(a) (b)

Figure 11: Cycle breakdown for ICOUNT1.8 and FBG for (a) SPEC2000 and (b) MediaBench benchmarks.

algorithm, especially for SPEC2000 benchmarks. While the performance of CtxOrder and ICOrder are on par or better than ICOUNT1.8, MemOrder usually results in a much lower increase in system performance. In the VPR, Equake, and MCF benchmarks, MemOrder produces the lowest increase in system performance for the FBG selection algorithms and performs worse than ICOUNT1.8. The reason for the lack-luster performance can be explained by the threads that MemOrder chooses. Since MemOrder selects the instruction fetch request that accesses the cache block with the earliest address in program memory, the speculative threads will be serviced more often than the non-speculative thread. Additionally, whenever a speculative thread is squashed and restarted due to a misspeculation, it will monopolize the entire fetch bandwidth until it has caught up with the other threads. This slows down the entire system when in DSMT mode, making the loops less likely to be labeled Good by the LDU. On average, more cycles are spent outside of DSMT mode when MemOrder is used, due to its reduced performance over the other two selection algorithms, which is illustrated in Figure 11(a) for benchmarks from SPEC2000 and Figure 11(b) for MediaBench benchmarks.

The strength of FBG is in generating groupings. The average number of groupings per cycle each selection algorithm makes while in DSMT mode is illustrated in Figure 12. The Figure shows an interesting phenomenon. While MemOrder produces less

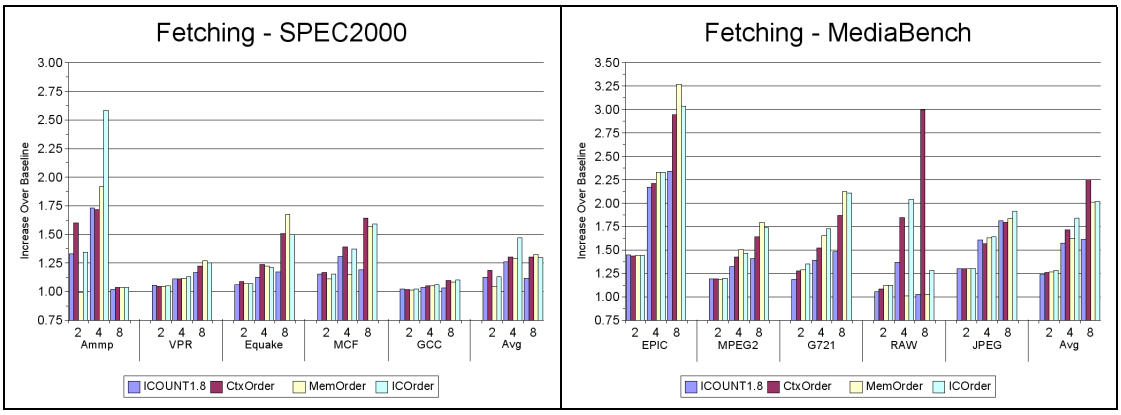


(a) (b)

Figure 12: Average groupings per cycle in DSMT mode for (a) SPEC2000 and (b) MediaBench benchmarks.

overall increase in system performance, it does produce on average more groupings per cycle than the two other selection algorithms. In the case of the RAW benchmark using eight active threads, it produces almost double the number of groupings per cycle than CtxOrder, even though CtxOrder has the best system performance for this configuration. This increase in groupings per cycle can be explained by the low number of cycles that are spent in DSMT mode by MemOrder during the program execution. In the RAW benchmark with eight threads, the MemOrder simulation spends roughly 68,000 cycles in DSMT mode, while CtxOrder and ICOrder spend over 7,000,000 and 900,00 cycles, respectively. The more time spent in DSMT mode, the more often single thread fetches will occur, driving down the average number of groupings for CtxOrder and ICOrder. Also, the longer the system is operating in DSMT mode, the longer FBG has to increase system performance. While MemOrder has a higher average groupings per cycle, it does not spend enough time in DSMT mode to significantly influence system performance, unlike CtxOrder.

The increase in fetch performance over the baseline configuration is graphed in Figure 13. Both the SPEC2000 and MediaBench benchmark groups see increases in fetch performance using FBG for all three selection algorithms, with MediaBench having a higher average increase. Comparing the increase in the average number of instructions fetched per cycle with ICOUNT1.8, FBG is the better performer in most

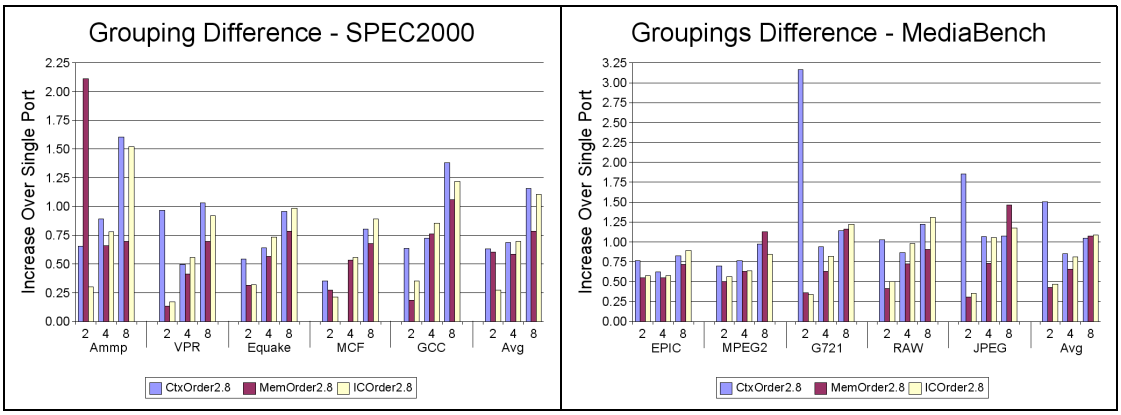


(a) (b)

Figure 13: Fetch increase for ICOUNT1.8 and FBG for (a) SPEC2000 and (b) MediaBench benchmarks.

of the configurations. Comparing the average groupings per cycle with the increase in fetch performance, we can see there is no direct translation between high groupings and high fetch performance. The RAW benchmark is a good example of this, as it has the most average groupings per cycle when there are eight threads and using MemOrder, but the fetch performance is highest when using CtxOrder for the same configuration. Once again, the disparity in performance can be explained by taking into consideration the amount of time spent in DSMT mode. Unless a significant amount of execution time is spent in DSMT mode, FBG cannot perform groupings and increase fetch performance. On average, the MediaBench benchmarks see more increase due to the higher percentage of their execution time spent in DSMT mode than the SPEC2000 benchmarks, seeing 1.25 to 2.25 times more instructions fetched per cycle than the baseline simulations.

When the number of ports on the instruction cache are increased to two, FBG can fetch two cache blocks per cycle as described in Section 4. The overall system performance changes only slight though, from 0.93 to 1.13 times the performance using a single-ported instruction cache for the three selection algorithms and ICOUNT2.8. Instruction fetch performance increases similarly, with the CtxOrder2.8 and ICOrder2.8 algorithms increasing more than ICOUNT2.8 for SPEC2000 benchmarks. However, the fetch performance for FBG for MediaBench benchmarks

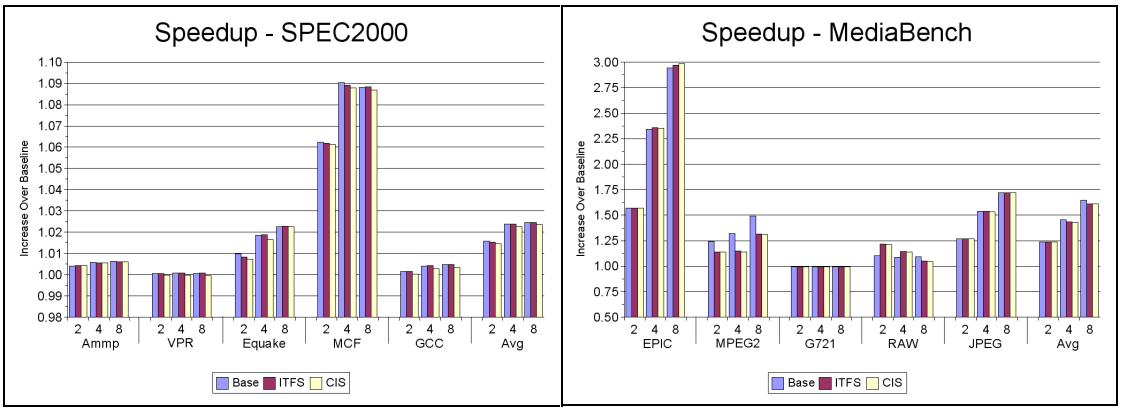


(a) (b)

Figure 14: Change in groupings per cycle when using a dual-ported instruction cache for (a) SPEC2000 and (b) MediaBench benchmarks

shows slightly less increase than ICOUNT2.8. The reason for this can be explained by Figure 14, which shows the ratio of groupings per cycle for the dual-port instruction cache over groupings per cycle for the single-port instruction cache. As the Figure shows, most benchmarks forms less groupings per cycle in DSMT mode for all three selection algorithms. Since the strength of FBG is in the number of groupings it can make per cycle, the drop in groupings shown in Figure 14 explains the lack of overall performance.

When FBG uses the additional port on the instruction cache to fetch an additional cache block, the second request is more likely to service only a single thread. This can reduce the average number of groupings per cycle and therefore fetch and overall performance. For example, if there are three threads requesting instructions be fetched during a cycle, and two of them are requesting the same cache block, FBG will form one grouped request and one ungrouped request. However, if a single-ported instruction cache was used, there is the chance that the ungrouped request would be deferred to a later cycle. This allows it to be considered for groupings with other fetch requests later, thus increasing the average number of groupings per cycle and instruction fetch performance. Figure 14 supports this theory since there is a drop in the groupings per cycle compared to single-ported instruction cache performance.



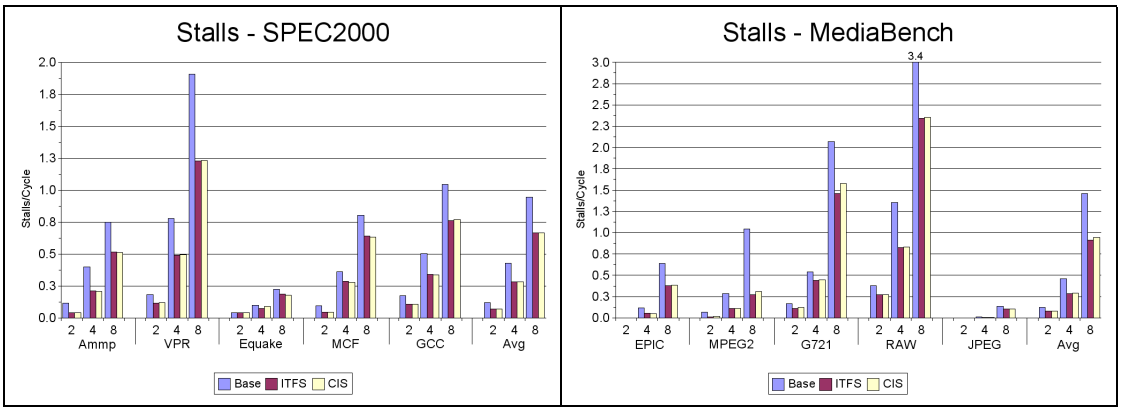
(a) (b)

Figure 15: System speedup for the base system, new inter-thread forwarding system, and CIS for (a) SPEC2000 and (b) MediaBench benchmarks.

7.2 CIS and ITFS Results

In this Section, the performance of ITFS and CIS will be presented and compared to a baseline configuration. To remove any potential bottlenecks from the instruction fetching stage, the number of ports on the instruction cache was increased to eight, each able to fetch eight instructions per cycle. When eight ports are used, it does not matter which selection algorithm is used for FBG. First, the benchmarks were run using the current inter-thread forwarding system in DSMT to produce the baseline configuration. Second, ITFS was simulated with the CIS system disabled. This allowed the performance of the two systems to be isolated. Finally, both ITFS and CIS were simulated together.

The overall increase in system performance for the three simulations is shown in Figure 15. The first trend that appears is the relatively similar performance of the three configurations for the majority of the SPEC2000 benchmarks. Both ITFS and CIS do, however, produce slightly less overall increase in performance for some of the benchmarks, such as VPR and Equake, when compared to the baseline configuration. In other benchmarks, such as GCC, ITFS is on par with the baseline, but CIS produces less performance. In fact, CIS has slightly less average performance than the baseline



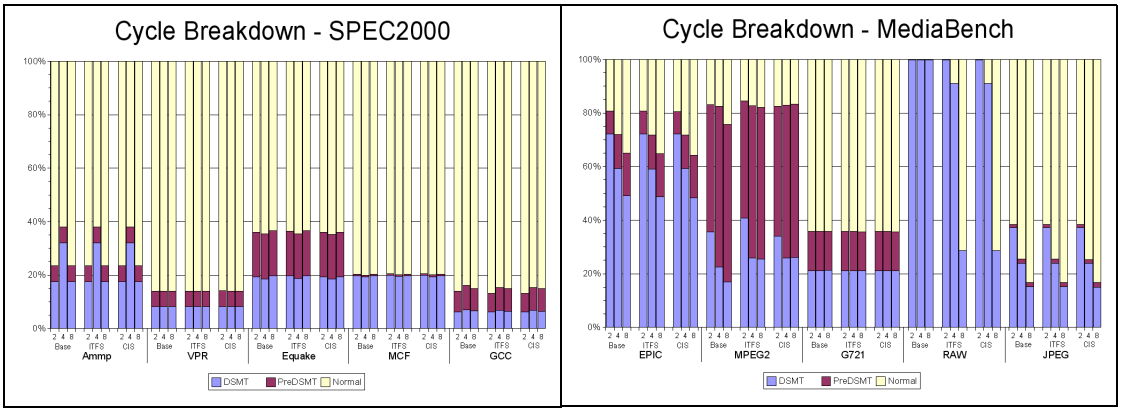
(a) (b)

Figure 16: Average number of stalls per cycle in DSMT mode for (a) SPEC2000 and (b) MediaBench benchmarks.

and ITFS configurations. This shows that there is a weakness in the CIS system. While CIS may not perform well in the SPEC2000 benchmarks, it does produce increased overall performance in some MediaBench programs as illustrated in Figure 15(b). Benchmarks such as EPIC and RAW have increased performance when using both ITFS and CIS when two and four threads are active. However, the performance of ITFS and CIS for RAW with eight threads is less than the baseline configuration.

Both ITFS and CIS attempt to reduce thread stalls by speeding up the forwarding of inter-thread dependencies. One way to measure this effect is to compute the average number of stalls from inter-thread dependencies that occur per cycle while in DSMT mode. This is shown in Figure 16. As the Figure shows, both CIS and ITFS have less stalls per cycle than the current inter-thread forwarding system in the baseline configuration. However, when the time spent in Normal, PreDSMT, and DSMT modes in Figure 17 is considered, the reason the performance for ITFS and CIS is not better becomes apparent. In the case of the RAW benchmark in Figure 17(b), at eight threads ITFS and CIS execute the majority of the instructions in Normal mode. This reduces the number of cycles CIS and ITFS have available to improve the overall performance of the system.

While this explanation works for the RAW benchmark, it does not explain the reduced performance of MPEG2 when using ITFS and CIS, which spends slightly



(a) (b)

Figure 17: Cycle breakdown for base configuration, new inter-thread forwarding system, and CIS for (a) SPEC2000 and (b) MediaBench benchmarks.

more time in DSMT mode than the baseline configuration. However, comparing the number of thread squashes due to misspeculated inter-thread dependencies presents a explanation. The number of thread squashes from misspeculated inter-thread dependencies for the current inter-thread forwarding system is roughly 300K and 800K for four and eight threads, respectively, in the RAW benchmark. For ITFS and CIS, these numbers rise to over 2 million. Thread squashes are expensive as one or more speculative thread must be flushed, re-initialized, and have execution restarted from the beginning of the loop.

The increase in misspeculated inter-thread dependency related thread squashes can be directly attributed to ITFS and CIS. Both systems cause inter-thread dependencies to be read earlier than the current inter-thread forwarding system in the baseline configuration. Increasing the speed values can be forwarded between threads increases the execution of the speculative threads, thus increasing the occurrence of inter-thread forwards per cycle. As the number of forwards increases, the chance of values being read from the incorrect thread or at the incorrect time also increases, leading to the increase in inter-thread dependency related thread squashes.

For the SPEC2000 benchmarks, the amount of time spent in DSMT does not vary a great deal across the current inter-thread forwarding system, ITFS, and CIS as shown in Figure 17(a). However, on average the number of thread squashes from inter-thread

dependency misspeculations does increase from 1% to just over 15% for ITFS and CIS over the baseline configuration. This increase in thread squashes accounts for the slight drop in overall system performance for the SPEC2000 benchmarks.

8 Future Work

While several selection algorithms were tested for FBG in this thesis, all possible algorithms were not explored. There may be other selection algorithms that can increase system performance beyond those that were tested. Similarly, the performance of FBG when using a dual-ported instruction cache could be increased with alternative selection algorithms. Future work on FBG should focus on testing new selection algorithms and determining which algorithm would provide the best performance when the instruction cache uses more than one port.

Future work on CIS should focus on determining the best values for the intra-thread critical path prediction, as this thesis simply used the settings presented in the paper by Tune *et al.* [21]. Since the number of thread squashes due to misspeculated inter-thread dependencies increases when using ITFS and CIS, future work should also focus on methods for either increasing the accuracy of the forwarding system or decreasing the costs of a thread squash, or both. If the accuracy of the forwarding system can be increased, CIS and ITFS should show higher performance than what they currently generate.

9 Conclusions

In this thesis three new ideas were explored for increasing the performance of a dynamic multithreaded processor that generates threads from loop iterations. The new instruction scheduling and fetching method FBG was simulated with three different selection algorithms, both single- and dual-ported instruction caches, and compared with ICOUNT1.8 and ICOUNT2.8. A new inter-thread forwarding system was explored which allows values to be forwarded from the ROB and common data bus to decrease stall time on inter-thread dependencies. Finally, the critical path CIS system further attempts to decrease inter-thread dependency stall time by prioritizing instructions on the intra-thread and inter-thread critical paths.

From the simulation results presented in Section 7, the FBG technique produces performance equal to and better than the current ICOUNT instruction fetching policy for both single-ported and dual-ported instruction caches. Several aspects of instruction fetching in a dynamic multithreaded processor became apparent during the simulation study. First, there is the interplay between which threads are selected for instruction fetching and how often the system operates in DSMT mode. The CtxOrder and ICOrder selection algorithms give priority to the non-speculative thread and threads with highest instruction throughput, respectively, which greatly increases the performance of the system. This feeds back into the evaluation of the performance of loops when multithreaded, causing the system to spend more time in DSMT mode and allowing FBG to further increase fetch performance. The converse is also true as MemOrder demonstrated. By prioritizing the speculative threads during instruction fetching, the non-speculative thread becomes starved for instructions. Some loops that performed well with the other selection algorithms do not perform well using MemOrder, causing the loops to be viewed as bad thread sources and thus causing the system to spend more time in Normal mode. For example, there is a single loop in the RAW benchmark from MediaBench that produces positive results when threaded in DSMT. Using MemOrder the loop has poor threaded performance, and thus the system remains in Normal mode for the majority of the execution time of the

benchmark.

The overall performance of the selection algorithms when using a dual-ported instruction cache was only slightly different than their performance when using a single-ported instruction cache. Since two cache blocks can be fetched per cycle, the number of groupings should have increased for FBG. However, the number of fetch requests servicing only a single thread increased, thus reducing the average number of groupings made per cycle in DSMT mode. Allowing more single thread requests to be serviced via the second port on the instruction cache reduced the grouping performance of FBG. If future work on FBG can identify new selection algorithms or make other modifications to reduce these occurrences, the performance of FBG can be pushed higher. However, using the current FBG selection algorithms with a dual-ported instruction cache has the advantage of keeping the cache simpler than using ICOUNT2.8. The cache bank conflicts when using FBG is lower than using ICOUNT2.8 and does not require additional hardware in the instruction cache to manage cache block conflicts.

Performance of the new inter-thread forwarding system and CIS are more mixed than the performance of FBG. While there are several benchmarks that show increased overall system performance when using either CIS or ITFS, the average performance of the two systems is slightly less than the current inter-thread forwarding system used in DSMT.

While exploring the cause for the decreased system performance, the amount of time the system spends in DSMT mode and the increase in thread squashes from misspeculated inter-thread dependencies were identified as the sources. Increasing the speed of inter-thread forwards leads to more inter-thread dependencies being read incorrectly, either from the incorrect source or at the incorrect time, thus increasing the occurrence of expensive thread squashes. However, even with the increase in thread squashes, CIS and ITFS succeed in reducing the occurrence of threads stall on inter-thread dependencies.

Both ITFS and CIS produced fewer stalls per cycle for the SPEC2000 and

MediaBench benchmarks, with CIS reducing the number of stalls for MediaBench programs more than ITFS. This is due to the nature of the threads produced from the loops found in MediaBench programs. Whereas SPEC2000 loops are shorter and generate more inter-thread dependencies when threaded, loops from MediaBench programs are more independent and thus respond better to decreasing the stall time on the fewer inter-thread dependencies they produce.

Bibliography

[1] J. Lo et al., "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading", pp. 322-254, Aug. 1997.

[2] Intel, " Hyper-Threading Technology on the Intel Xeon Processor Family for Servers",*Techn. documentation*, Intel Corp., 2002.

[3] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *23rd Annual International Symposium on Computer Architecture*, May 1996.

[4] H. Akkary and M. Driscoll, "A Dynamic Multithreading Processor," *31st International Symposium on Microarchitecture*, December 1998.

[5] P. Marcuello, A. Gonzalez, "Speculative Multithreaded Processors," *Proc. 12th International Conference on Supercomputing*, pp. 365-372, 1999.

[6] V. Krishnan and J. Torrelas, "Sequential Binaries on a Clustered Multithreaded Architecture with Speculation Support," *International Conference on Supercomputers*, 1998.

[7] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *22nd Annual International Symposium on Computer Architectures*, June 1995.

[8] D. Ortiz-Arroyo and B. Lee, "Dynamic Simultaneous Multithreading Architecture," *16th International Conference on Parallel and Distributed Computing Systems*, August 2003.

[9] G. Reinman et al., "Optimizations Enabled by a Decoupled Front-End Architecture," IEEE Transactions on Computers, 50(4):338-355, April 2001.

[10] G. Reinman et al., "A Scalable Front-End Architecture for Fast Instruction Delivery," *Proc. 26th International Symposium on Computer Architecture*, May 1999.

[11] E. Rotenberg, S. Bennette, and J. Smith, "Tace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 24-34, December 1996.

[12] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," *Proc. of the 30th International Symposium on Microarchitectures*, December 1997.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, Q1 2001.

[14] A. Falcon, A. Ramirez, and M. Valero, "A Low-Complexity, High-Performance Fetch Unit for Simultaneous Multithreading Processors," *Proc. of the 10th International Symposium on High Performance Computer Architecture (HPCA)*, February 2004.

[15] J. Moure, R. Garcia, and E. Luque, "Improving Single-Thread Fetch Performance on a Multithreaded Processor," *Euromicro Symposium on Digital Systems Design*, 2001.

[16] M. Mudawar, "Scalable Cache Memory Design for Large-Scale SMT Architectures," *Proc. of the 3rd Workshop on Memory Performance Issues: In Conjunction With the 31st International Symposium on Computer Architectur*, 2004.

[17] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[18] S. Wallace, B. Calder, and D.M. Tullsen, "Threaded Multiple Path Execution," *Proc. of the 25th International Symposium on Computer Architecture*, pp. 238-249, 1998.

[19] D. Tullsen and B. Calder, " Computing Along the Critical Path",*Techn. documentation*, University of California, October 1998.

[20] A. Zhai, C. Colohan, J. Steffan, and T. Mowry, "Compiler Optimization of Scalar Value Communication Between Speculative Threads," *Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[21] E. Tune, D. Liang, D.M. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," *Proc. of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[22] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-Path Prediction," *Proc. of the 28th International Symposium on Computer Architecture*, 2001.

[23] G. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Procs. of the 34th International Symposium on Microarchitecture*, pp. 318-327, April 1991.

[24] David A. Zier, Jarrod A. Nelsen, and Ben Lee, "NetSim: An Object-Oriented Architectural Simulator Suite," *The 2005 Int'l Conference on Computer Design*, June 2005.

[25] C. Lee, M. Potkonjak, and W. Mangione-Smith, " MediaBench: A tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. of the 30th International Symposium on Microarchitecture*, pp. 330-335, December 1997.