AN ABSTRACT OF THE THESIS OF

Jarrod A. Nelson for the degree of Master of Science in Electrical and Computer Engineering presented on December 2, 2005.

Title: Dependency Speculation in Dynamic Simultaneous Multi-Threading

Abstract approved: _____

Ben Lee

The purpose of this thesis is to explore dependency speculation in Dynamic Simultaneous Multi-Threading (DSMT). DSMT is a microprocessor architecture which attempts to extract Thread Level Parallelism (TLP) from single-threaded programs at run-time. This is accomplished by running multiple iterations of program loops in parallel. The DSMT architecture was originally developed by Dr. Daniel Ortiz-Arroyo and Dr. Ben Lee at Oregon State University.

To extract TLP from loops successfully, inter-thread dependencies must be resolved by either speculation or stalling. To maximize performance both stalling and misspeculation must be minimized. To this end, two techniques are presented which attempt to improve stride speculation and dynamic inter-thread dependency resolution. To study these proposed changes, a detailed, cycle-accurate simulation environment for DSMT with extensive statistics gathering capabilities was developed. Results generated by the simulator not only show the performance of the proposed changes but also the capabilities of the new simulator.

Dependency Speculation in Dynamic Simultaneous Multi-Threading

by

Jarrod A. Nelson

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 2, 2005
Commencement June 2006

Master of Science thesis of Jarrod A. Nelson presented on December 2, 2005.

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.

Jarrod A. Nelson, Author

ACKNOWLEDGEMENTS

The author expresses sincere appreciation to the following individuals. Dr. Ben Lee, who inspired me to pursue graduate studies at Oregon State University. David Zier for tirelessly pursuing DSMTSim and the DTP project. And finally, my parents for their endless support.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Dependency Speculation in Dynamic Simultaneous Multi-Threading

1 Introduction

For decades processor performance has been improved by not only upping clock frequencies but also increasing the utilization of *Instruction Level Parallelism* (ILP). Techniques such as out-of-order execution, pipelining and branch prediction have pushed ILP to new heights. However, in recent years the improvements in ILP have been slowing and performance increases have been harder to attain. As a result, microprocessor architectures have begun to use *Thread Level Parallelism* (TLP) in an attempt to improve overall performance. There are already microprocessors on the market which allow multiple threads to run on a single processor [4]. Unfortunately, these techniques do not improve the performance of individual programs and can even hurt their performance [13]. To improve individual programs, other methods must be used.

In general, multithreading can be divided into two main types, multi-program, as mentioned above, and single program. Single program multithreading can be further divided into two types, *static*, or compiler techniques, and *dynamic*, or hardware techniques. Static Multithreading is performed prior to runtime and relies on the compiler to find and expose TLP [2,3]. Dynamic techniques are performed at runtime and rely on the hardware to identify threads which can yield TLP. Static techniques are often limited by factors such as pointer ambiguity and complex dynamic behavior. Dynamic techniques have the advantage of additional runtime information that is unavailable to the compiler.

To improve the performance of individual programs using TLP, *Dynamic Simultaneous MutliThreading* (DSMT) [1] has been proposed as a dynamic multithreading technique for individual programs. DSMT seeks to improve the performance of a single program by creating dynamic threads from loops within the program. This technique uses a standard SMT core with additional hardware to spawn and control dynamic threads.

One of the key limiting factors to DSMT's performance is loop dependencies which must be resolved to allow the execution of speculative iterations. This thesis explores these dependency issues and presents a novel method for ensuring their proper resolution. The technique uses a *Register Generation Table* (RGT) which can track when register values tend to be produced and allow contexts to speculate on whether or not it is safe to read a register value from a previous context. The RGT determines if and when a speculative context should be held in order to resolve a dependency.

This thesis is organized as follows: Section 2 discusses other dynamic multithreading architectures and research into critical path dependency chains and data value prediction. Section 3 provides an overview of the DSMT architecture. Section 4 covers the creation of the simulation environment used for this research. Section 5 discusses the proposed improvements to dependency speculation in DSMT. Section 6 presents and analyzes the results produced by enhancing dependency speculation. And finally, Section 7 proposes future work to improve Dependency speculation and DSMT in general.

2 Related Work

In recent years, several techniques have been proposed to allow for dynamic threading of single programs. One such architecture is DMT [5]. Much like DSMT, this architecture presents a hardware only solution where the compiler gives no assistance in recognizing dependency. DMT attempts to spawn threads based off function return values and alternate branch outcomes. Due to a large amount of misspeculation, large trace buffers are used to perform efficient recovery. These trace buffers are quite complex and present a very large challenge to practical implementation.

Another dynamic threading architecture is the Speculative Multithreaded Processor presented in [15]. As with DSMT, the SM architecture seeks to exploit loops to extract TLP from a single program. This design uses a Clustered Multi-Processor (CMP) as the basis for the architecture and it is focused on exploiting small or fine-grained threads. Unlike DSMT much of the work to break dependencies in loops is based on data value speculation [6]. An attempt is made to predict all the values fed into the thread as opposed to just stride values. This is done using trace prediction which takes into account the execution trace when predicting the final value for a needed register.

Another technique for identifying dependency chains can be seen in critical path work such as [7]. In this technique, several basic heuristics are used to mark possible critical instructions. If a marked instruction is committed then the corresponding entry in the Critical Path Prediction Buffer is incremented. If this count

exceeds a cut-off value then the next time it is encountered, the instruction will be flagged as critical. The rest of this work focuses on using this critical flag along with data value prediction to break the critical path and free up ILP. Work is also presented which shows how this flag can be used with clustered scheduling architectures to improve their performance.

3 Dynamic Simultaneous Multi-Threading

Dynamic Simultaneous Multi-Threading is a multi-threaded microprocessor architecture developed by Dr. Daniel Oritz-Arroyo and Prof. Ben Lee at Oregon State University [1]. It is designed to create TLP from a single program by running multiple loop iterations in parallel. Section 3.1 presents an architectural overview of DSMT. Section 3.2 reviews the original dependency speculation methodology.

3.1 Architectural Overview

The DSMT architecture is based around an SMT core as presented in [14]. To better utilize the available resources, threads are spawned from loops in a single program to exploit the SMT contexts. Figure 1 shows the basic block diagram of DSMT. The components needed in addition to the SMT core are shaded gray. These are the Thread Control and Initiation Unit (TCIU), the Scheduler and the Loop Detection Unit (LDU). In addition to the new components, it was necessary to enhance several existing components to support dynamic threading. This included inter-thread dependency resolution in the Dispatch, Utility Bits added to the Contexts and an enhanced ROB which can determine thread continuation.

The TCIU is responsible for maintaining the contexts and spawning, squashing and retiring all dynamic threads. The Loop Stride Prediction hardware is also located in the TCIU. When new contexts are spawned, the TCIU is responsible for setting up the new context, initializing the PC and providing the stride values. It also updates the tail pointer so that it indicates the most speculative thread. When a thread completes, the ROB will set the corresponding Join and Continue bits which tell the TCIU that a

thread is ready to be retired and whether or not the loop continues. The TCIU also holds the R and D anchor bits which indicate the past dependency behavior of the current loop.

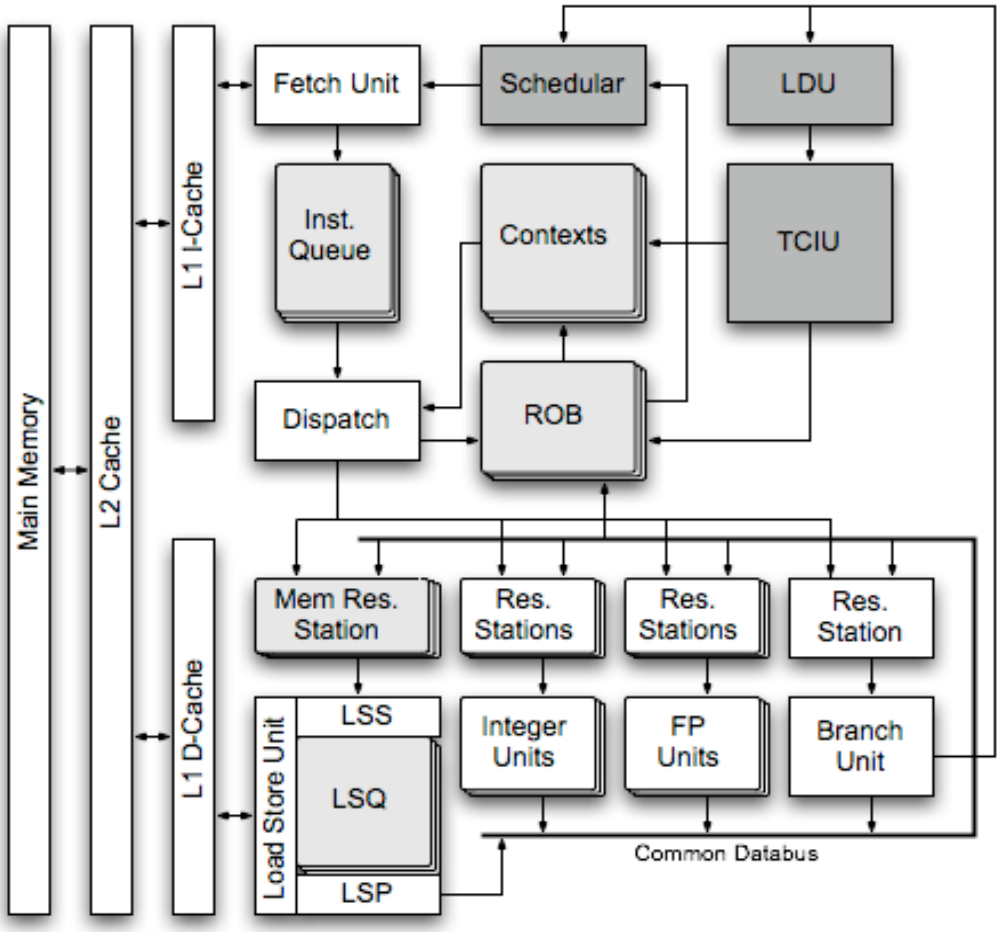

Figure 1: DSMT Architectural Diagram

The Scheduler is responsible for determining the fetch policy. When multiple contexts are in use, it determines which threads get fetching priority and provides the Fetch Unit with the PC address to use.

The LDU is responsible for identifying loops which may be multi-threaded. It keeps a table with all past loops encountered and their performance in normal and DSMT mode. Upon identifying a backwards branch the LDU notifies the TCIU of the loop address. The TCIU then puts the simulator into PreDSMT mode and the loop is executed twice more to determine its non-DSMT IPC, dependencies and stride values. Once these iterations are complete, the TCIU transitions the simulator into DSMT mode. The LDU will then monitor the loops DSMT IPC. If the DSMT performance it not an improvement, the loop will be marked as bad and the simulator will return to normal mode.

3.2 Dependency Resolution

To resolve dependencies between loop iterations, DSMT used a set of utility bits to identify dependent registers and track speculative register forwarding. In addition to the more typical valid and tag fields associated with superscalar execution, each register has an R, D, and L bit. The R-bit indicates that the register value was generated within the current iteration (or context). The D-bit is used to track inter-thread dependencies. The L-bit is set whenever a register is speculatively read from a previous context. These bits are used along with the D_anchor and R_anchor bits to determine current and past loop behavior to aid in resolving dependencies.

When a speculative context attempts to read a register value, it checks to determine if its R-bit is set. If the R-bit is not set then it must read the register from a previous context. The first step is to check the D_anchor bit to determine if this register has had an inter-thread dependency recently. If the D_anchor bit is clear then

the context simply searches back for a previous context whose R-bit is set for that register. If none is found then the value is read from the head context and it is assumed that the value was generated outside the loop. If an R-bit is found then this indicates a new dependency and the D-bit is set.

If the D_anchor bit is set then it is assumed there is an inter-thread dependency involving this register. If the previous context's R-bit is set then the value is read and again the D-bit is set. If the previous context's R-bit is clear then the current context is held and waits for the value to be generated. Unfortunately, dynamic behavior can mean that the value may or may not be produced by the proceeding context. Therefore, the context is only held for a short time. If the register value has not been produced then the context will search back (as above) for the first context with an R-bit set.

The L-bit is set anytime a register is loaded from a previous context. If any instruction from the previous contexts commits a value to that register, then misspeculation has occurred and the context is squashed. To ensure correct execution, all contexts after the misspeculated one must also be squashed. This can potentially mean all the speculative contexts have to be restarted and all results thrown out.

This method can be very sensitive to dynamic behavior. There is no mechanism for helping the dependency resolution determine if it is safe to attempt a speculative read. The context is simply held for an arbitrary length of time before second-level speculation is used. This thesis presents the Register Generation Table as a possible solution to this problem in Section 5.

4 Creating a Simulation Environment

Most current simulators are limited in detail and focused mostly on getting raw performance data such as overall IPC or functional unit utilization. In order to study DSMT in detail, a new simulation environment has been created which forms the basis of the DSMT architectural simulator (*DSMTSim*). In an effort to avoid creating an entirely new compiler, this environment was based on the PISA target used by SimpleScalar [7]. Instead, development focused on creating a modular environment which would allow the rapid development of detailed execution-based cycle-accurate simulators. This environment is called *NetSim* (Section 4.1). At Oregon State University, NetSim has been used to create an architecturally accurate superscalar simulator (Section 4.2) and an entirely new DSMT simulator (Section 4.3) [8].

4.1 NetSim Environment and NetSimBase

The NetSim environment consists of DLLs which contain the objects used to create simulators. These objects provide a range of components depending on the type of simulator and the architecture being simulated. By combining the objects provided by NetSim a new simulator can be created as a simple executable which references the DLLs. To ease development, NetSim is written using C# and the .NET Framework. Using an object-oriented programming language allows new simulators to be created based on the block diagram of the processor. Each component in the block diagram will have an object created for it which tracks its state and models its behavior.

NetSim is divided into three main parts. Each part is based on the previous and provides more advanced modules. All the core components are contained within

NetSimBase. NetSimBase provides the objects needed to create a simple functional simulator. These include components such as memory, registers, a system call handler and an instruction database. Figure 2 shows how these components are connected to create a basic functional simulator. Also contained in NetSimBase is *FastMod*. FastMod is a functional simulator which can be used to create execution traces or as a "fast forward" module for more advanced simulators.

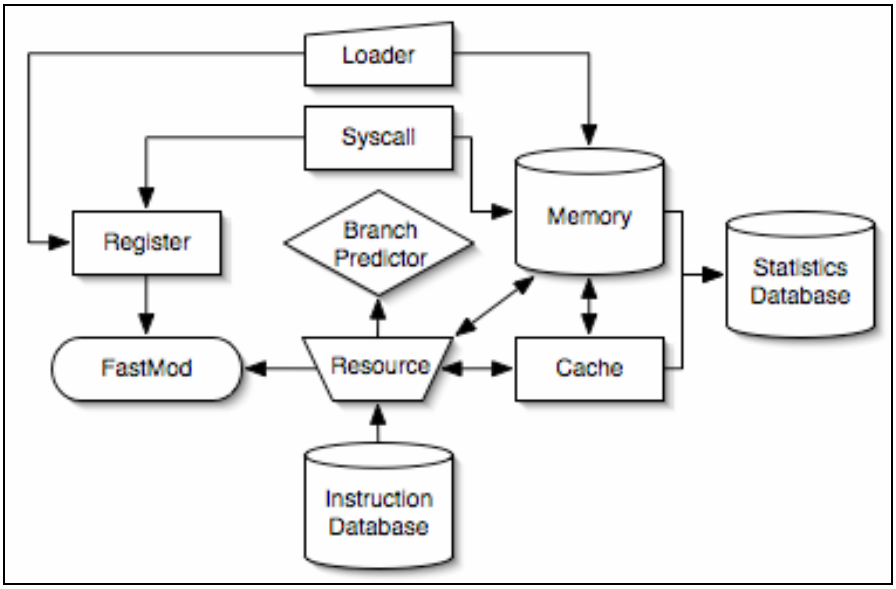

Figure 2: Structural Overview of NetSimBase

To facilitate retargeting, each instruction object contains an Execute() method which will calculate the result of the instruction based on the operands supplied by the simulator. This allows the ISA to be changed without completely rebuilding the simulator. Instead, only a new instruction database is needed. Functional Units are designed so that they just provide the operands to the Execute() method and then simulate the instruction delay before outputting the results. This also allows for the

ISA to be easily extended or modified and facilitates the creation of entire new types of functional units.

4.2 SuperSim and NetSimHP

Once the functional core of NetSim had been completed and tested, the project moved on to creating a cycle-accurate superscalar simulator (*SuperSim*). The NetSim environment was extended by adding *NetSimHP*. NetSimHP contains all the components necessary to create a cycle accurate simulator. These components are aware of simulated clock cycles. Each one contains an Update() and a Tick() method. The Update() method is used by the simulated module to calculate its next state and perform any inter-module communication. This communication is performed through method calls to the other modules involved. The Tick() method represents the actual clock transition. In this method each module will transition to the new state determined in the Update() method.

NetSimHP contains components which are simply wrappers for NetSimBase components with cycle accurate support and new components needed for a superscalar processor. These components not only form the basis for SuperSim but also for the SMT core of DSMTSim.

4.3 DSMTSim and NetSimMT

The most recent simulator created using NetSim is DSMTSim. This is a very detailed architectural simulator which serves as the primary research platform for DSMT. Each architectural component of DSMT is modeled as an individual object which tracks the state and models the behavior of that component. Rather than

relying on timing assumptions DSMTSim uses only delay values for memory and instruction execution. All other timing in the system is based on actual behavior. By avoiding predetermined timings for various operations, DSMTSim can provide real world analysis of the behavior of DSMT.

Each module gathers its own statistics based on the actual resource utilization. These statistics reflect not only the numbers but also the reasons behind them. An example is issue utilization. The dispatch unit tracks not only the number of instructions issued but also the reasons for any un-issued instructions. This additional data makes spotting architectural bottlenecks easy. The accurate architectural model used by DSMTSim means that the research data produced is more "real world". For example, there is no set branch penalty but instead the penalty depends on the pipeline depth and the exact state of the processor.

DSMTSim collects a wealth of statistics as it runs a simulation. Each component tracks its own performance and gathers statistics independently of other modules. When the simulation has finished the Statistics Database collects all the statistics from each component and dumps them to a file. As the statistics are gathered additional processing can be performed to calculate percentages and other information which can be useful to the user. Each statistic is printed to the file based on the specifications provided by the component. Figure 3 shows a small sample of the statistics produced by DSMTSim.

```
Simulation Statistics
-----------------------------------------
Cycle Count    72662 # Number of cycles executed
Instr Count   293892 # Number of instructions executed
IPC           4.0446 # Instructions per Cycle
CPI           0.2472 # Cycles per Instruction
LpsFound          17 # Num loops found
LpsFullDSMT    8 ( 47.059%) # Num loops in Full DSMT
LpsDSMT       34 (200.000%) # Num loops in DSMT
LpsPreDSMT    17 (100.000%) # Num loops in PreDSMT
LpsBadIPC                0 # Num loops with bad IPC
LpsSysCall               0 # Num loops with Syscall
NumBwrdBr      29556 # Num backwards branches from BrFU
NumUniqLps        12 # Number of unique loops in LoopTable
GoodLps        2 ( 16.667%) # Number of Good loops
BadIPCLps      4 ( 33.333%) # Number of Bad_IPC loops
BadNstLps      1 (  8.333%) # Number of Bad_Nested loops
BadSysLps      0 (  0.000%) # Number of Bad_Syscall loops
BadSize        0 (  0.000%) # Number of Bad_Size loops
BadPDSMT       0 (  0.000%) # Number of Bad_PreDSMT loops
PendLps        1 (  8.333%) # Number of Pending loops
PreLps         4 ( 33.333%) # Number of PreDSMT loops
UknLps         0 (  0.000%) # Number of Unknown loops
AvgCycPr      841.5833 # Average Cycles spent on a loop
AvgItrPrLp     70.0833 # Average Iterations spent on a loop
CycPrItr       12.0083 # Cycles / Iteration
CycPrItrD      11.9600 # Cycles / Iter in DSMT
MaxItr             750 # Maximum Iterations for a loop
MinItr               3 # Minimum Iterations for a loop
CtxSpawned         881 # Number of contexts spawned
CtxSquashed        122 # Number of contexts squashed
SpwnPerSqsh    13.8479 # Squashed per spawned percentage
SquashNormal  84 (  9.535%) # Squashed normal loop ending
SquashMpStd    0 (  0.000%) # Squashed misspredicted stride
SquashBadIPC  13 (  1.476%) # Squashed from bad DSMT IPC
SquashMSLoad   0 (  0.000%) # Squashed from misspec loads
SquashITDep   25 (  2.838%) # Squashed MS interthread Dep.
SquashSpTrap   0 (  0.000%) # Squashed speculative trap.
SquashBdInst   0 (  0.000%) # Squashed bad instruction
SquashIAddr    0 (  0.000%) # Squashed bad Instr. Addr.
SquashMisc     0 (  0.000%) # Squashed misc. reasons
...
```

Figure 3: A Sample Portion of DSMTSim Statistics

5 Enhancing Dependency Resolution

To enhance dependency resolution and speculation in DSMT two key changes are proposed. The first change is to the stride prediction method. The second is the introduction of new hardware to assist in determining when a register value is safe to read.

5.1 Stride Prediction

The original stride prediction method used by DSMT involved monitoring the instruction stream for add immediate instructions. These instructions add a predefined value to a register. When used in loops they often indicate a stride value which is consistent for every iteration. The key drawback of this method is the complexity of determining which add immediate instructions are strides and which are not. Hardware must include not only a method to monitor the instruction stream but also to determine which instructions indicate a stride for the current loop. When one considers nested loops the problem becomes even more complicated as the stride instructions for inner loops must be excluded. This methodology is unable to detect strides which are set by a register value.

To simplify this stride detection process, a new methodology has been introduced, which tracks register values directly to look for consistent strides between iterations. During PreDSMT mode, the register results are captured at the end of each iteration and their values are then compared to that of the last iteration. If the difference is consistent then the register may have a predictable stride value. The last

step to ensure that a stride exists is to check the D-bit for that register. If there is a true dependency between iterations then the stride prediction is marked as valid.

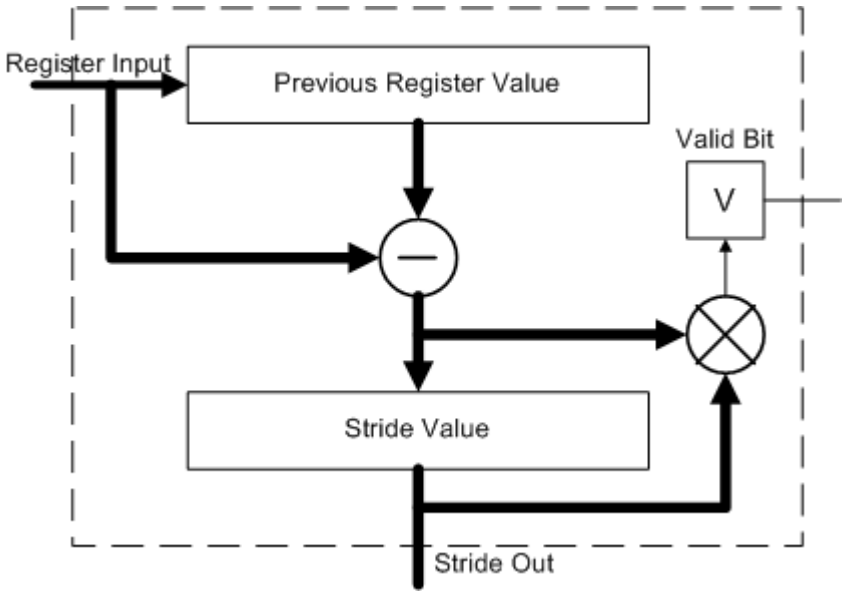

Figure 4: Stride Prediction Hardware

Figure 4 shows the new stride prediction hardware. In PreDSMT mode, the values for all integer registers are captured and compared to the last value. Whenever a new context is spawned, a multiple of the stride value is added to the last Previous Register Value and the result is then copied to the corresponding register. This primes the new thread with initial register values so that execution can begin immediately.

5.2 Register Generation Table

Predicting register dependencies in loops with complex dynamic behavior presents a major hurdle to the performance of DSMT. The original design did not track dynamic loop behavior and instead simply made a guess at whether a register was ready to read. If the register read was incorrect then the context and all the following

contexts were squashed and new contexts were spawned in their place. There are two main dependency issues which can cause squashing. The first is caused by dynamic loop behavior where a register is not produced every iteration. The second occurs when a register value is created more than once in a single loop iteration.

The *Register Generation Table* (RGT) seeks to deal with these issues by tracking each context's path through the loop and recording which registers were written too. The table is broken up such that each entry corresponds to 4 instructions. Entries are indexed using a portion of the current PC value for the context, as can be seen in Figure 5. The indexing is done as a direct map to simplify the process of using and updating the table.

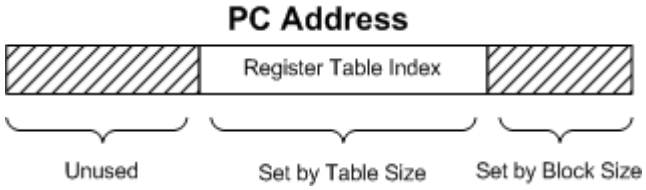

Figure 5: Register Generation Table Index Value

Each entry includes a *Prediction Counter* for each register and a Path Flag for each context. Figure 6 shows the Register Generation Table and the fields it contains. Figure 7 shows the block diagram for a Prediction Counter. The *Path Flags* are used to track which blocks each context has visited and the Prediction counters indicate the likelihood that the register will be generated in or after that block. This prediction is then used to determine whether a speculative context attempting to read the register from the current context needs to be held to avoid misspeculation.

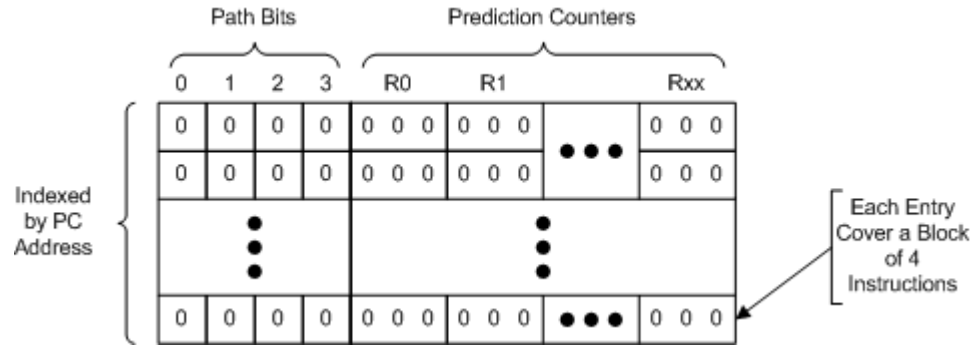

Figure 6: Register Generation Table

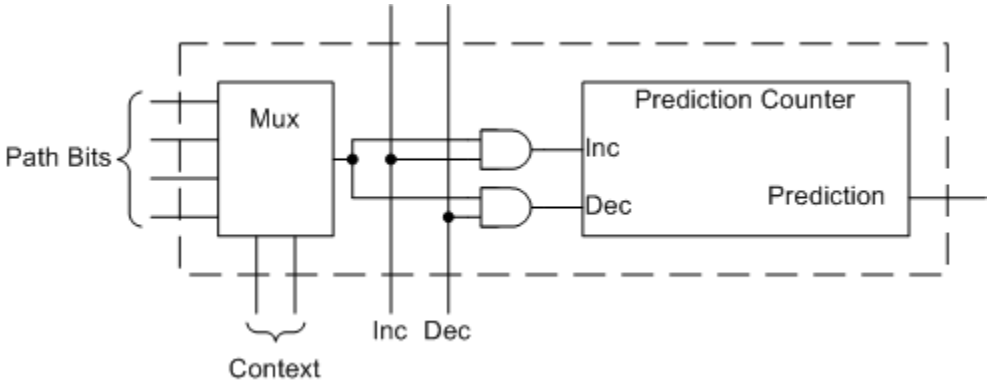

Figure 7: Prediction Counter Block Diagram

Updating the table is accomplished by setting the Path Flag for each block in which the current context has committed an instruction. When a register value is produced, each block with the corresponding Path Flag set will increment its prediction counter. This way each entry block contains information about the behavior of the instruction blocks which will follow it. Traditional methods of prediction often rely on a specific set of circumstances to occur before the prediction is updated as seen in branch predictors. For DSMT, the process of filling in one of these tables would mean a long runtime with mediocre performance in order to generate the dependency prediction data. But with Path Flags, large parts of the RGT

can be updated at the same time. This allows most entries in the RGT to be generated in just the 2 cycles of PreDSMT mode.

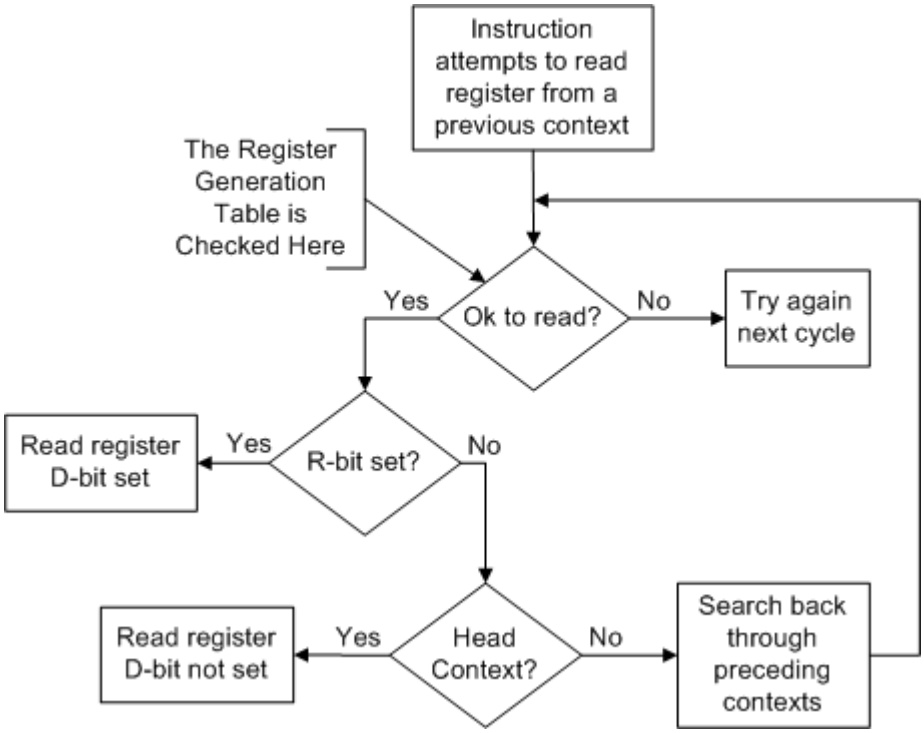

Figure 8: Flowchart for Modified Register Reading

To accommodate the RGT, the dependency resolution algorithm for DSMT has been modified as shown in Figure 8. When a context attempts to read from a previous context, it first checks the RGT to see if the register is safe to read. If not then it will wait and try again on the next cycle. If the register is safe to read from the preceding context, then the R-bit for that register is checked to determine if the value has been produced. If the value has been produced, then it is read and the D and L-bit are set to indicate an inter-thread dependency and a corresponding speculative register read.

If the R-bit is clear then the register has not been produced by that context and the speculation process must continue to previous threads. If the context being checked is the head then the register will be read anyway, the L-bit will be set but the D-bit is not. This indicates that the register may have been produced before the loop began. The speculative search continues to previous contexts until the register is read or an unsafe context is found.

This modification to the original dependency algorithm has two main advantages. First, if there is a high likelihood that a register will be produced by a given context, then the speculative context will wait for it to be produced. The second advantage is that if the proceeding context does not seem likely to produce the value then second level speculation can begin immediately. In the original algorithm, the speculative context would be held for a predetermined amount of time before attempting to proceed to second level speculation.

5.3 Dependency Examples

Figure 9 shows a simple loop from the matrix multiplication program with all inter-thread dependencies highlighted. At first the loop may seem to be hopeless but if the stride values are removed then only a single true dependency is left (Figure 10). This true dependency cannot be broken reliably as it is based off of two load instructions. To prevent contexts from being squashed each thread must wait for this dependent instruction to commit before proceeding. Anytime a context reads the value before the dependent instruction is completed, that context and all contexts after it will be squashed. With the traditional algorithm, the context would be stalled for a

set amount of time or until the register value is produced, whichever came first. Using the PC of the proceeding context, the new dependency algorithm can use the RGT to determine if the register can be safely read. And by checking every cycle, the register can be read as soon as the previous thread has advanced to a safe portion of the code. In trivial cases such as this example, both methods can resolve the dependency equally well. Stride prediction would be responsible any improvement seen. Only with more complex behavior will the RGT be able to show an improvement.

```
addu     r2, r8, r3
l.s      f2, 0(r4)
l.s      f0, 0(r2)
mul.s    f2, f2, f0
addiu    r3, r3, 120
add.s    f4, f4, f2
addiu    r5, r5, 1
addiu    r4, r4, 4
slti     r2, r5, 30
bne      r2, r0, -10
```

Figure 9: Code Segment with All Dependencies Highlighted

```
addu     r2, r8, r3
l.s      f2, 0(r4)
l.s      f0, 0(r2)
mul.s    f2, f2, f0
addiu    r3, r3, 120
add.s    f4, f4, f2
addiu    r5, r5, 1
addiu    r4, r4, 4
slti     r2, r5, 30
bne      r2, r0, -10
```

Figure 10: Code Segment with Only True Dependencies Highlighted

Upon entering PreDSMT mode, the RGT immediately begins gathering register generation data from each completed instruction. When a register value is committed both the Path Flag and the corresponding prediction counters get updated.

Figure 11 shows code from one of the loops in the SPEC2000 AMMP benchmark. The code has been broken up into 4 instruction blocks as it would be by the RGT. Instructions which produce true dependent register values are highlighted.

```
LOOP:
  lb      r3, 0(r16)
  sll     r2, r3, 1
----------------------------
  addu    r2, r2, r7
  lhu     r2, 0(r2)
  andi    r2, r2, 8
  beq     r2, zero, ENDCHECK
----------------------------
  c.lt.d  f6, f20           ;F6<F20?
  addiu   r5, zero, 1
  bc1f    ELSE              ;Branch
IF:
  addiu   r17, r17, 1       ;R17 A
----------------------------
  j       ENDIF
ELSE:
  mul.d   f2, f20, f4
  addi    r2, r3, -48
  mtc1    r2, f0
----------------------------
  cvt.d.w f0, f0
  add.d   f20, f2, f0       ;F20
ENDIF:
  beq     r4, zero, DONE
  addiu   r17, r17, -1      ;R17 B
----------------------------
  j       DONE
ENDCHECK:
  bne     r4, zero, OUT
  bne     r3, r6, OUT
  addiu   r4, zero, 1       ;R4
----------------------------
DONE:
  addiu   r16, r16, 1
  j       LOOP
OUT:
```

Figure 11: Loop Example from SPEC2000 AMMP

Table 1 shows the Path Flags when each register value gets produced. As each iteration proceeds through the loop, the Path Flags are set for each portion of code visited. When a register value is produced, all the blocks with their Path Flag set have their corresponding register prediction counter incremented. If the register is produced frequently then any blocks along the path will be considered unsafe.

Table 1: Path Flags for Each Dependency Case

R4	F20	R17 A	R17 B
1	1	1	1
1	1	1	1
0	1	1	1
0	1	0	?
0	1	0	1
1	0	0	0
0	0	0	0

Consider the case of R4. When this register is produced a large portion of the loop is skipped. As a result R4 is safe to read if the proceeding context has reached the 3rd block. R4 will not be produced by a context if that context is currently executing instructions from the 3rd, 4th, or 5th blocks. If R4 is produced frequently (i.e. more that once every 4th iteration), then the RGT will not allow R4 to be forwarded from a context which has not dispatched instructions past the 2nd block.

For R17, the RGT would consider the first 3 blocks unsafe if either or both cases were common. Blocks 4 and 5 would only be considered unsafe if case B were common. This would allow R17 to be read in block 4 if case B were uncommon. If case B was common, forwarding would be prevented until the 6th block.

6 Results and Analysis

To determine the performance of RGT, a range of loops were simulated and their results analyzed. These loops are from both a Spec2000 benchmark [10] and our own custom matrix multiplication program. These benchmarks present a variety of difficult and ideal loops for the RGT to be tested against. For the Spec Benchmarks we used the GCC compiled binaries provided by [11] and the reduced data sets created by [12].

Table 2: AMMP Loop IPCs

Loop Address	PreDSMT	Base DSMT	w/Reg Table	% Improvement
0x00415c28	1.5	0.5455	0.6667	22.22
0x00410c98	0.7778	0.5333	0.5833	9.38
0x00401238	0.8427	0.8414	0.8389	-0.30
0x00440470	0.8	0.6847	0.7677	12.12
0x00400b80	1.3714	1.0825	1.0657	-1.55
0x004010e8	0.8257	0.7563	0.8182	8.18
0x004530f0	3.4286	0.7619	1.0213	34.05
0x0044a468	0.8974	1.219	1.2234	0.36
0x004400f0	0.7368	0.6563	0.7	6.66
0x00441fb0	0.5	0.4286	0.4615	7.68
0x0044dd88	1.12	1	1.1212	12.12

Shown above in Table 2 are the DSMT IPC improvements for loops from the AMMP Spec2000 benchmark. In this case, by forcing speculative contexts to wait we got a significant improvement in performance. The loops from AMMP represent all loops which showed a change once the RGT was used. Many of the loops show a significant improvement in IPC, including several which had more than a 20% improvement. The results also showed a 7.2% reduction in the number of contexts squashed due to inter-thread dependencies. The performance of these loops is

improved significantly when using the RGT. The loop at address 0x00440470 was shown in the RGT example above. It is worth noting that the RGT produced a 12% increase in performance for this case.

Table 3: AMMP Loop Results

Category	Number of Loops
worse	2
No change	24
0-10%	5
10-20%	2
20+%	2
unknown	20

Table 3 shows a breakdown of the impact of the RGT on the loops in AMMP. The unknown loops are those which did not have enough iterations to ever enter DSMT mode. Of the loops which show no change most are very small and have almost no inter-thread dependencies. These loops are very similar to the example presented earlier. The two loops which actually presented worse performance were both loops which had a large amount of dynamic behavior. Unfortunately, both were only run in DSMT mode for just a few iterations so the IPC numbers could very likely have been impacted by other architectural issues. With such a small number of iterations a single cache miss could significantly impact the resulting IPC numbers.

Looking back at Table 2, one architectural limit to DSMT reveals itself. In most of the loops presented, the RGT granted a large improvement but this was still insufficient to outperform non-DSMT execution. The reason for this lies in the forwarding mechanism used. Consider the matrix code presented in Figure 10. The

result of that instruction in non-DSMT mode is forwarded to the next dependent instruction via the common data bus. This forwards the result from the functional unit to the reservation station of the waiting instruction. The dependent instruction can then begin execution on the very next cycle. While in DSMT mode, the result from one iteration must be committed to the register file before it can be forwarded. As a result, extra latency is introduced. In many cases this extra latency is not a problem as DSMT can simply fill the stall from other contexts or unrelated instructions. However, in many cases this will limit the maximum attainable IPC in DSMT mode and prevent the processor from extracting any TLP from that loop.

7 Future Work

Work still needs to been done to determine the best settings for the Register Generation Table. The increment and decrement values used in this thesis performed well but they may not be ideal. This also holds true for counter size and the cut-off value used in determining predictions. Further simulation results with a variety of settings for the table will be needed to determine the best values for each. It may also prove beneficial to be more aggressive with the more speculative contexts. If the tail context is squashed, no other contexts get squashed with it. This means there is a lower misspeculation penalty, which may mean a larger pay off for aggressive speculation.

The results presented in Section 6, highlighted an architectural inadequacy in DSMT. The forwarding within a single context is significantly faster than forwarding between contexts. Intra-context forwarding is also unaffected by unrelated stalled instructions. As a result true dependencies can make it impossible to match the non-DSMT performance of a loop, even with excellent dependency prediction. The forwarded value remains unavailable for use even after it is produced, while the speculative context waits for the instruction to commit. As a result, any inter-thread dependencies will hinder performance far more than intra-thread dependencies.

To combat the problem, changes need to be made to the inter-thread forwarding to allow it to use the much faster common data bus. By doing this speculative threads would be allowed to proceed much quicker after a dependency resolution. A value can be forwarded by using the context and tag already associated

with the instruction, but the problem is complicated by branch misspeculation. To ensure correct execution a method must be devised to track forwarding through the common data bus and allow for recovery after a branch misspeculation in the head thread.

8 Conclusion

The purpose of this thesis is to examine dependency speculation in DSMT and present the development of the simulator used for this research. The development of the NetSim environment has been crucial to the results produced in this thesis. The wealth of statistics has aided in identifying and modeling the key bottlenecks in the DSMT architecture. As a result of this, new methods of dependency speculation could be proposed and studied. The modular simulator allowed for the rapid development and integration of the Register Generation Table into the core design.

From the results presented in Section 6, there was a clear improvement in the number of threads squashed and the IPC of problem loops. Threads squashed due to stride misspeculations remain very low or nonexistent and those squashed due to dependency misspeculations have been reduced. The Register Generation Table has improved DSMT's dependency speculation and increased the performance of loops with difficult dependencies. There is still work to be done to allow DSMT to reach its full potential but predicting and speculating on inter-thread dependencies in dynamic loops is no longer the hurdle it used to be.

BIBLIOGRAPHY

[1] D. Ortiz-Arroyo and B. Lee, "Dynamic Simultaneous Multithreaded Architecture," *16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, August 13-15, Reno, Nevada, 2003.

[2] I. Park, B. Falsafi, T.N. Vijaykumar, "Implicitly-Multithreaded Processors," *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA 30)*, June 2003.

[3] J. G. Steffan, C. B. Cohan, A. Zhai, and T. C. Mowry, "A Scalable Approach to Thread-Level Speculation," *Intl. Symp. on Computer Architecture*, June 2000.

[4] Intel "Hyper-Threading Technology on the Intel Xeon Processor Family of Servers," *White Paper Intel Corp.*, 2002.

[5] H. Akkary, and M. A. Driscoll, "A Dynamic Multithreading Processor," *In Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 31)*, pages 226-236, December 1998.

[6] P. Marcuello, A. Gonzalez, and J. Tubella, "Speculative Multithreaded Processors," *Proceedings of the 12th International Conference on Supercomputing*, pp. 365-372, 1999.

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

[8] D. Burger, and T.M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Tech. Report 1342, Univ. of Wisconsin*, Madison, 1997.

[9] D. A. Zier, J. A. Nelson, and B. Lee, "NetSim: An Object-Oriented Architectural Simulator Suite," *The 2005 International Conference on Computer Design (CDES 05)*, Las Vegas, June 2005.

[10] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, July 2000.

[11] M. Postiff, et al., "The MIRV SimplerScalar/PISA Compiler," *University of Michigan EECS Department Tech. Report CSE-TR-421-00.* April 2000.

[12] A. J. KleinOsowski, and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, Volume 1, June 2002.

[13] S. Hily and A. Seznec, "Contention on the 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading," *Technical Report PI-1086*, *IRISA*, February 1997.

[14] J. Lo et al., "Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Mutlithreading," *ACM Transactions on Computer Systems*, Aug. 1997, pp. 322-354.

[15] P. Marcuello, and A.Gonzalez, "Clustered Speculative Multithreaded Processors," *Proceedings of the International Conference on Supercomputing* (ICS '99), 1999.