ABSTRACT

Sharma, Saurabh

Weld for Itanium Processor

(Under the direction of Dr. Thomas M. Conte)

This dissertation extends a **WELD** for Itanium processors. Emre Özer presented WELD architecture in his Ph.D. thesis. WELD integrates multithreading support into an Itanium processor to hide run-time latency effects that cannot be determined by the compiler. Also, it proposes a hardware technique called *operation welding* that merges operations from different threads to utilize the hardware resources. Hardware contexts such as program counters and the fetch units are duplicated to support for multithreading. The experimental results show that Dual-thread WELD attains a maximum of 11% speedup as compared to single-threaded Itanium architecture while still maintaining the hardware simplicity of the EPIC architecture.

WELD FOR ITANIUM PROCESSOR

by

Saurabh Sharma

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Electrical and Computer Engineering

Raleigh

November 2002

Approved By:

______________________________ ______________________________

Dr. Eric Rotenberg Dr. Alexander Dean

____

Dr. Thomas M. Conte, Chair of Advisory Committee

Biography

Saurabh Sharma was born in Gwalior, India on March 16, 1976. He graduated from Jiwaji University, Gwalior with Bachelor's on Electronics Engineering in 1999. He started his Master's program in Electrical and Computer Engineering in North Carolina State University in 2000.

Acknowledgements

I want to thank my advisor Dr. Tom Conte for his patience and guidance. It is an honor working with him in the Tinker Research group. I also want to thank my committee members Dr. Eric Rotenberg and Alexander Dean for their contributions in my work. Special thanks to all Tinker group. I am very grateful to my family for their support, love and encouragement. I want to individually thank to my dear mother, Rekha Sharma and my dear father, Subhash Sharma.

TABLE OF CONTENTS

LIST OF TABLE……………………………………………………………v

LIST OF FIGURE…………………………………………………………..vi

Chapter 1 Introduction……………………………………………………1

1.1 Goals of Weld…………………………………………………….2

1.2 Layout of the Dissertation………………………………………..3

Chapter 2 Related Works…………………………………………………4

Chapter 3 Dual-thread WELD Model……………………………………7

3.1 Dual-thread WELD Architecture…………………………………7

3.2 Main Thread Register…………………………………………….9

3.3 Speculative Memory Operation Buffer……………………..9

3.4 Speculative Store Buffer…………………………………………10

3.5 Operation Welder…………………………………………...10

Chapter 4 Compiler Support Needed for Dual-thread WELD…………12

4.1 Flow of the Compiler Support……………………………………13

Chapter 5 Performance Evaluation……………………………………..15

5.1 Reducing Branch Misprediction by using two BHRs…….…24

Chapter 6 Conclusion and Future Work………………………………..27

Chapter 7 Bibliography…………………………………………………30

LIST OF TABLES

Table 1 – Details of Research Itanium Simulator……………………….17

Table 2 - Branch Misprediction Rate of Dual-Thread and Triple-thread Weld with respect to the baseline model…………………………………….…………22

Table 3 - Load Misprediction Rate per Thousand Instructions………….24

Table 4 - Branch Misprediction Rate of Dual-Thread Weld with one BHR with respect to two BHR model…………………………………………25

LIST OF FIGURES

Figure 1 - Architecture Model of Dual-thread WELD…………………….8

Figure 2 - Flow of the Compiler Support…………………………………13

Figure 3 - Performance of Dual-Thread Weld and OOO

over base line model…………………………………………19

Figure 4 - Percentages of Horizontal to Vertical Slots Welded…………21

Figure 5 - Speedup of Dual-thread and Triple-thread WELD

over a baseline model……………………………………...21

Figure 6 - Status of BHR in Dual-Thread Weld…………………………23

Figure 7 - Example of main thread BHR and the speculative thread BHR

in Dual-Thread Weld……...…………………………………26

Chapter 1 Introduction

Multithreading is a technique that has been used to tolerate long latency instructions or run-time events such as cache misses, branch mispredictions or exceptions. WELD is a new architecture paradigm that was proposed by Emre Özer in his Ph.D. thesis [1]. It sits on a VLIW/EPIC processor core and contains additional hardware mechanisms that can spawn multiple threads from a single program and weld operations from different threads to utilize the empty issue slots.

In this thesis, we extend the WELD technique for Itaniums to improve the single thread performance. This multi-threading technique utilizes the idle hardware context to execute speculative thread on behalf of the non-speculative thread. We explore the use of Dual-thread and Triple-thread welding. We also explore a novel technique for enhancing the branch predictor performance of the WELD.

1.1 Goals of Weld

WELD utilizes the processor resources during unpredictable run time events. Secondly, it dynamically fills the issue slot holes that cannot be filled at compile time. The compiler cannot detect the unpredictable events like cache miss or branch misprediction.

WELD can overcome this drawback by filling vertical holes by instructions from the speculated threads. Second problem associated with the in order VLIW/EPIC machine is discrete scheduling window where attempts are made to fill the schedule holes from a scheduling region at compile time. The compiler cannot fill all schedule slots in every MultiOp [2][1] or group because the compiler does not migrate operations from different scheduling regions. Introducing a hardware mechanism called *operation welder* solves this problem. It merges operations from different threads in the issue buffer.

---

[1] A MultiOp is a group of instructions that can be potentially executed in parallel.

1.2 Layout of The Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 discusses the Related Work with this thesis. In Chapter 3 we will talk about the Dual-thread WELD architecture and the Chapter 4 provides Compiler framework needed for this architecture. Chapter 5 presents performance evaluation. Chapter 6 concludes the dissertation and discusses the future work.

Chapter 2 Related Works

SPSM (Single-program Speculative Multithreading [3] speculatively spawns multiple paths in a single program and simultaneously execute those paths or threads on a superscalar core.

Dynamic Multithreading Processor (DMT) [4] is simultaneous multithreading on a superscalar core with threads created by the hardware from a single program.

MultiScalar Processors [5] consist of several processing units that have their own register file, Icache and functional units. Each processing unit is assigned a task, which is a contiguous region of the dynamic instruction sequence. Tasks are created statically by partitioning the control flow of the program. Loads are performed speculatively.

Superthreaded Architecture [6] has a set of thread units that contains functional units, a communication unit and a memory buffer. The thread unit has a dynamic scheduling core that can fetch and execute instructions simultaneously. Load speculation is not allowed in the Superthreaded architecture. The compiler generates threads from cyclic code. A thread can be one or many loop iterations.

TME (Threaded Multiple Path Execution) [7] executes multiple alternate paths on a Simultaneous Multithreading (SMT) [8] superscalar processor. It uses free hardware contexts to assign paths of conditional branches. Speculative loads are allowed.

Prasadh [9] et al. proposes a multithreading technique in a statically scheduled RISC processor. Statically scheduled VLIWs from different threads are interleaved dynamically to utilize NOPs. If a NOP is encountered in a VLIW at run time, it is filled with an operation from another thread through a dynamic interleaver. The dynamic interleaver does not interleave instructions across all issue slots and therefore there is one-to-one mapping between functional units.

XIMD [10] is a VLIW-like architecture that is targeted at exploiting fine-grained parallelism but also medium and coarse-grained parallelism. The architecture has multiple functional units and a large global register file similar to VLIW. On the other hand, each functional unit has an instruction sequencer to fetch instructions. A program is partitioned into several threads by the compiler or a partitioning tool. The XIMD compiler takes each thread and schedules it separately. Those separately scheduled threads are merged statically to decrease static code density or to optimize for execution time.

DDMT [11] is based on OOO processor in which speculative thread execute on ideal hardware thread context s to prefetch for future memory access and predict future branches.

Slipstream Processors [12] in which a non-speculated version of the program runs alongside a shortened, speculative version. Outcomes of certain operations in speculative version are passed to the non-speculative version, providing the speedup if speculative version was correct.

This is work is unique in that it's targeting the Itanium processor. Also it presents a technique with very low hardware cost for thread spawning.

Chapter 3 Dual-thread WELD Model

In this chapter we will discuss the architectural implementation of the Dual-thread WELD, as proposed in [1], and the compiler support needed for the architecture. It is less complex and more efficient than general WELD architecture in terms of hardware.

3.1 The Dual-thread WELD Architecture

In Dual-thread WELD architecture, there is a non-speculative thread and a speculative thread. Non-speculative thread spawns the speculative thread with the help of the compiler. Each thread has it's own program counter, fetch engine and register file. However, they share the same branch predictor and caches. The Icache has two read ports one for each thread. The fetch stage fetches either group for the non-speculative thread or one group each from the non-speculative and the speculative thread. The architecture model is shown in the Figure 1. The weld/decode stage merges the two groups from the threads and decodes them. The welder in the weld/decode stage consists of an array of multiplexer that can forward any operation to any functional unit. It is assumed that sufficient functional units exist to guarantee any two issued bundles can be executed in parallel without functional unit contention. The welder is designed in such a way that the

non-speculative thread's group has the highest priority. Thus the speculative thread never delays the non-speculative thread. The non-speculative thread operations are always copied to the issue buffer and the welder than merges the operations of the speculative thread to fill the nops. The operand read stage reads the registers; sends them to the execution unit that executes the operations and finally the write back stage updates the register file and the Dcache.

Figure 1. Architecture Model of Dual-thread WELD

3.2 Main Thread Register

The main thread register (MTR) keeps track of the thread spawned and merged. It holds a bit for the main or non-speculated thread register file and field for the Bork program counter (PC). The PC of the first group of the speculated thread is stored in the Bork PC field. If the speculated thread is committed, the RF bit is flipped and the Bork PC field is cleared. If the speculated thread is squashed only the Bork PC field is cleared.

3.3 Speculative Memory Operation Buffer

Speculative memory loads are kept in a buffer called the Speculative Memory Operation Buffer (SMOB), as proposed by [1]. The SMOB holds a valid bit and a field for the load address. An outstanding store memory address is compared associatively in the SMOB for a conflict. A conflict can occur only when an outstanding load address in the SMOB of the speculated thread matches a store address of the non-speculated thread. If there is a conflict, the speculative thread is squashed. At the time of squash all of the SMOB entries are invalidated. If the SMOB becomes full, speculative thread stalls. The main thread continues and frees the SMOB entries by committing the thread.

3.4 Speculative Store Buffer

Speculative memory stores are kept in a buffer called the Speculative Store Buffer (SSB), as proposed by [1]. The SSB holds a valid bit, field for the store address and a field for the store value. Speculative thread write store values until the commit time takes place. Speculative stores are written in the Dcache, in order, as soon as the speculative thread is verified as committed. At the time of squash all of the SSB entries are invalidated. If the SSB becomes full, speculative thread stalls. The main thread continues and frees the SSB entries by committing the thread.

3.5 Operation Welder

The operation welder takes groups whose operands are already read from each thread. It then welds and sends them to the functional units. It consists of a data switch (i.e. an array of multiplexers) and a control element that controls the forwarding of operations. Each thread has a buffer called the prepare-to-issue buffer. If some of operations from a group of speculative thread are issued and some stay in the buffer, then fetch from this thread is stalled. Each operation has an empty/full bit that states whether an operation exists in the slot and also each MultiOp has a separability bit. The empty/full bits and the separability bit are sent to the control box.

First, the welder forwards the main thread's group. Then, the empty slots are filled from speculative threads' group, if groups are separable. If only some of the operations in a group of a speculative thread are sent to the functional units, the remaining operations stay in the prepare-to-issue buffer. Unless the entire group is sent to the functional units, no succeeding fetch from the speculated thread takes place. The operation welder can send an operation to any functional unit as long as it is the correct resource to execute the operation.

Chapter 4 Compiler Support Needed for Dual-thread WELD

The ISA is extended by adding a new instruction and introducing some new bits to the encoding. These extensions required to support multithreading in Itanium architecture. The instruction branch and fork operation called a Bork instruction is added in the ISA for spawning new threads. We also need seperability and synchronization bit. For the first group of the treegion the compiler sets the synchronization bit. The compiler checks for the anti-dependency in the group and sets up the seperability bit if there is none. It also forms the treegions that are used as the threads in our model. These treegions are single entry multiple exit regions that can be formed with or without profile data information. The compiler inserts the Bork operation as described in [1]. In the case of more than one of the Borks present along the path compiler removes the Bork depending upon the edge profiling data.

4.1 Flow of the Compiler Support

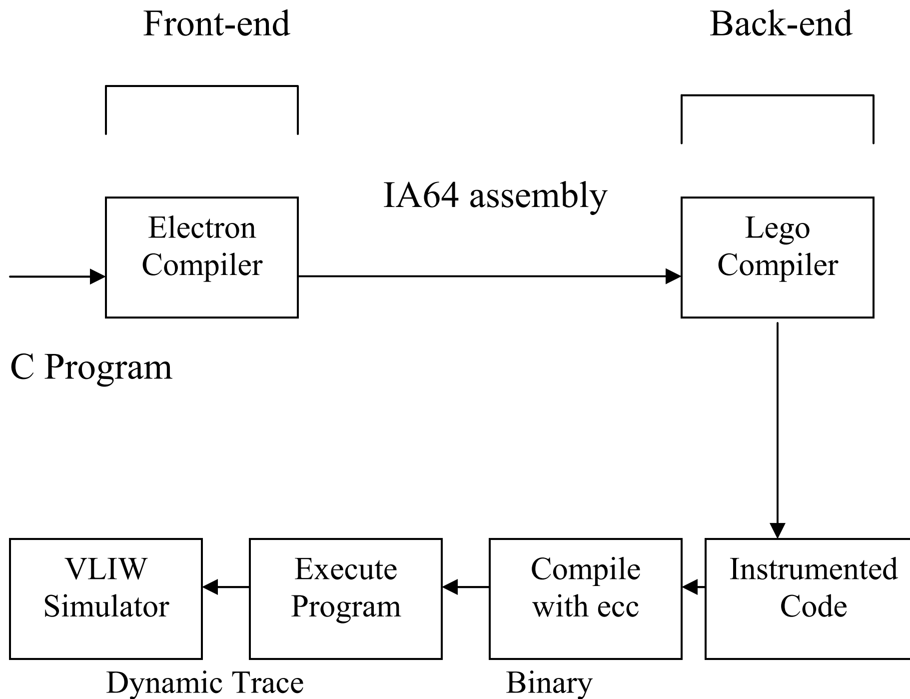

Figure 2. Flow of the Compiler Support

All the benchmarks used here are compiled with Intel Electron Compiler for Itanium architecture. It incorporates state-of-the-art compilation techniques i.e. software pipelining, predication, control and data speculation. The LEGO compiler parses the assembly files of the benchmark generated by the Electron compiler. The LEGO compiler sets seperability and synchronization bit and it also forms treegions. Moreover the LEGO compiler inserts Bork instructions as described in [1] for spawning the

speculative threads. Finally a database of all the instructions present inside the workload or the benchmark is made.

LEGO compiler also inserts instrumented code in the parsed assembly files of the workload. This is done to generate the traces required for the VLIW simulator. This instrumented code is parsed-out in the form of Itanium assembly. These files are than compiled with the help of the electron compiler. The executable formed produce traces that are fed into the VLIW pipeline simulator.

Chapter 5 Performance Evaluation

A trace-driven simulator was written to simulate two threads running at the same time. The details about the simulator can be found in Table1.The machine model used for the experiments is a research multi-threaded processor implementing Itanium instruction set architecture. Processors in the Itanium family fetch instructions in form of bundles. Each bundle is comprised of three independent instructions that the compiler has grouped together. The modeled processor has a maximum fetch bandwidth of two groups per cycle. Sufficient functional units exist to guarantee that any two issued bundles are executed in parallel without functional unit contention.

Three benchmarks from SPEC CPU2000 benchmark suite were used for all runs. 100 million instructions were executed from the reference inputs by skipping 500 million warm up instructions from each benchmark. Intel's electron compiler was used as a front-end for our experimental IA64 compiler that parse-in the IA64 assembly. Compiler then forms natural treegions without changing the schedule of the electron compiler.
We assume that Icache and Dcache have multiple ports to read and write simultaneously.

In the experiments, only the number of useful operations is considered. The incorrectly speculated operations are not taken into

consideration when calculating the IPCs. Dual and Triple thread runs are compared to the baseline model with a single thread run of the same benchmark program.

The results of the Dual-thread Weld architecture were than compared with the OOO execution core. The OOO execution employs the use of register renamer and reservation stations to dynamically schedule the in-flight instructions for execution.

The results of the Dual-thread Weld architecture were also compared with the Triple-thread WELD. For Triple-thread WELD three separate hardware context was used and a Thread Table (TT) was used instead of the MTR.

Simulator Properties
Fetch: 2 bundles or 1 group from 1 thread or 2 groups from two threads.
Branch Predictor: 16K Gshare, 16K entry 8-way assoc BTB
Register Files: Per thread register files, 128 IntReg, 128 FpReg, 64PredReg and 8 BtReg
L1:(separate I and D) 16K 4-way associative, 64B, 1 cycle latency
L2: (shared) 256K 8-way assoiative, 128B, 14 cycle latency
L3: (shared) 2048K 16-way associative, 128B, 30 cycle latency
Memory Latency: 200 cycles
64-entry SSB and 128-entry Fully-assoc SMOB
5-cycle SMOB squash penalty time
1-cycle register file copy time

Table 1. Details of the research Itanium simulator

Figure 3 shows the speedup of Weld and OOO over the in-order model. OOO processor can achieve impressive speedup over the baseline in-order processor, with an average speedup of 18%.

The Dual-thread Weld processor actually performs slightly better than the baseline in-order processor, with an average speedup of 6%. The explanation lies in the fact that Weld uses treegions as threads. The treegions considered here are the natural treegions and they were formed without changing the schedule of the electron compiler. The natural treegions on an average has a size of 20-25 instructions and as a result very low overlapping

of treegions takes place. For the compute-intensive benchmarks, though incurring a large amount of L1 misses rarely misses in L2. So the timeliness of filling the vertical slots through Welding is offset by the overhead incurred by spawning and squashing of thread. Furthermore in our Bork insertion algorithm we introduced a condition that insertion is possible only if we could insert 5 cycles ahead of the last instruction along the path in the treegion. Since the size of the natural treegion is small the number of Bork inserted is also small. So the size of natural treegion account for the Weld's low performance compared to OOO.

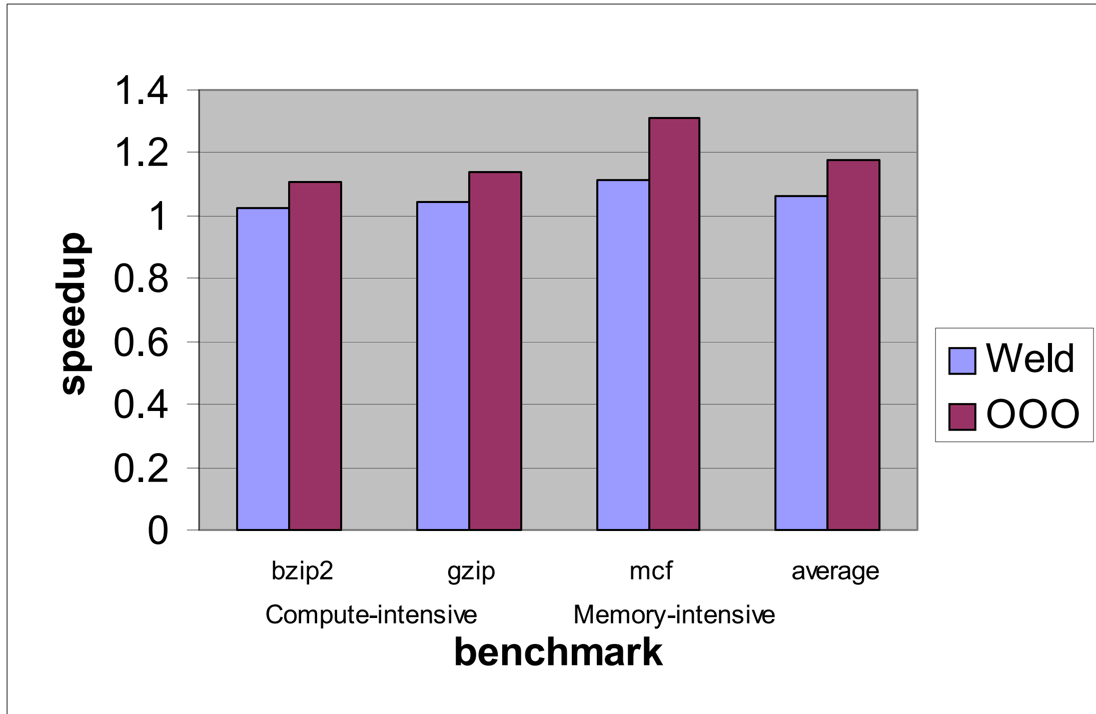

Figure 3. Performance of Dual-Thread Weld and OOO over base line model

In Figure 4 the percentages of vertical and horizontal slots filled with the welding for Dual-Thread WELD is shown. As expected the number of welding or the filling of the empty horizontal slot is much more for the compute-intensive benchmarks. The reason is that these benchmarks rarely miss in L2; as a result there is a less opportunity of filling the vertical slots. However for the memory-intensive benchmark like mcf, the majority of the gain is through the filling of the empty vertical slots. This is because the mcf has a high cache miss rate due to capacity misses.

In Figure 5 the speedup of the Dual-thread and Triple-thread Weld over the baseline model is shown. As shown, there is no major change in the speedup results between the Dual-thread WELD and the Triple-thread WELD for compute-intensive benchmarks. The reasons are that the opportunity for running three threads simultaneously was very low. However, in the case of mcf the speedup for Triple-thread Weld is around 15.1%. The reasons are that the mcf has high L1 and L2 cache misses. As a result speculative thread was able to find the opportunity to spawn another speculative thread while filling up the vertical nops.

Table 2 shows the branch misprediction rate for all the three benchmarks with Dual-thread and Triple-thread Weld. Here we could see that the miss-prediction rate is higher for the Dual-thread and Triple-thread WELD with respect to the baseline model. The reason is that the branches predicted in the speculated thread corrupt the Branch History Register (BHR) and finally the gshare table. The example of this scenario is shown in Figure 6.

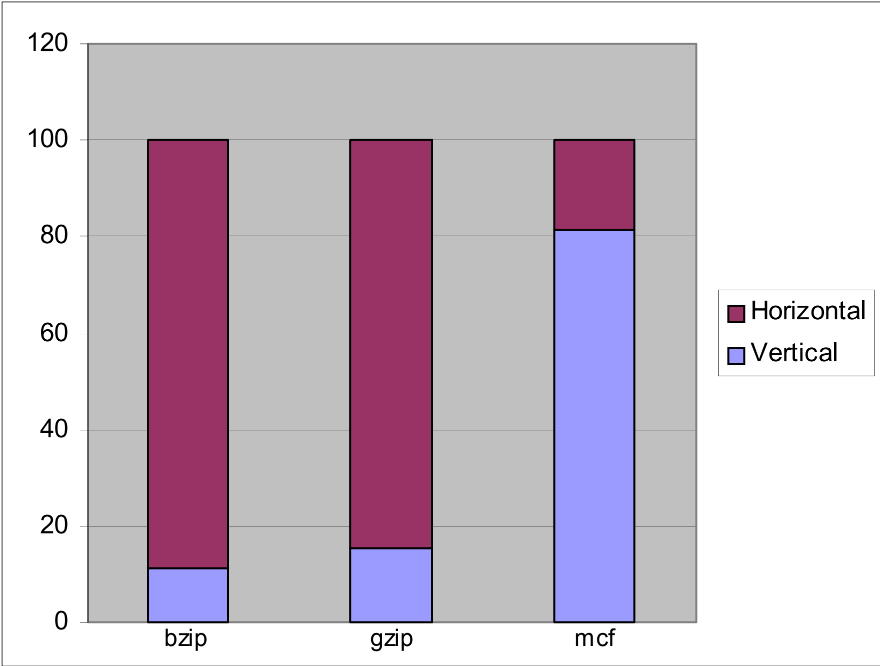

Figure 4. Percentages of Horizontal to Vertical Slots Welded

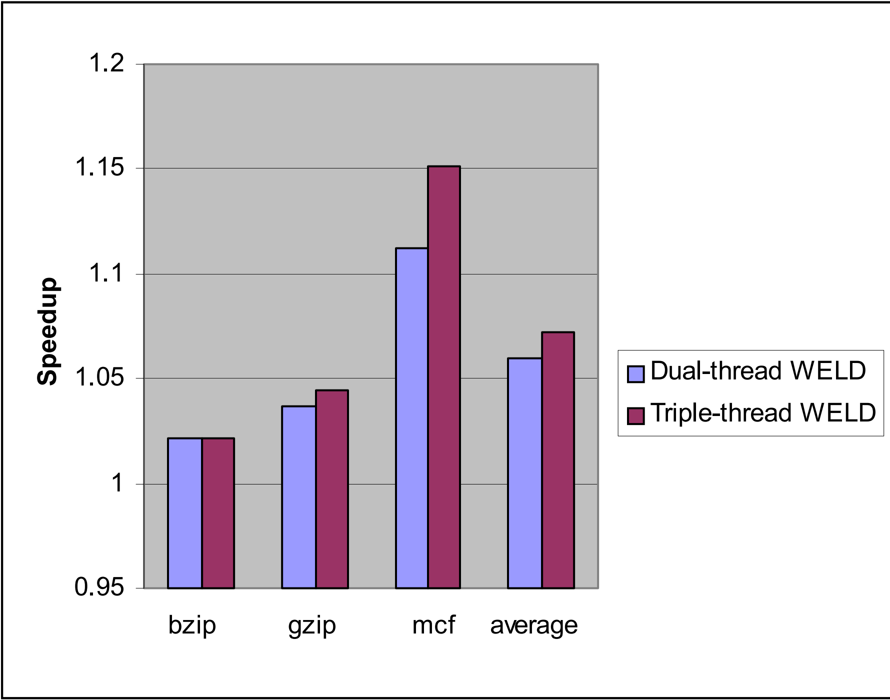

Figure 5. Speedup of Dual-thread and Triple-thread WELD over a baseline model

Branch Misprediction			
Benchmarks	Thread1	Thread2	Thread3
bzip2	0.027935	0.028352	0.028353
gzip	0.074710	0.095369	0.107453
mcf	0.013932	0.032920	0.044012

Table 2. Branch Misprediction Rate of Dual-Thread and Triple-thread Weld with respect to the baseline model

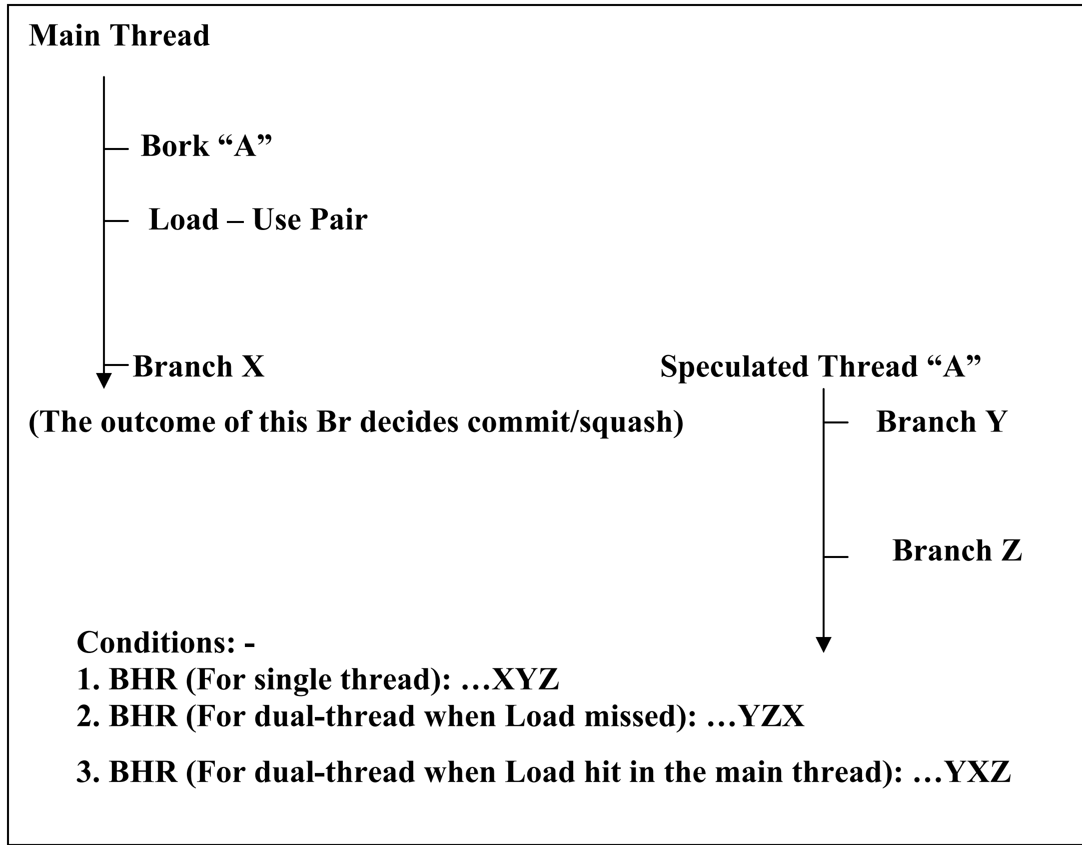

Figure 6. Status of BHR in Dual-Thread Weld

As we can see in the Figure 5 the status of BHR in Dual-thread WELD is shown. A Bork instruction present in the main thread spawns a speculated thread. If the load-use pair, as shown, doesn't stall the main thread and the speculated thread is always committed, the BHR will be as shown in condition 2. If this always happens the branch predictor will learn this pattern and the miss-prediction rate will be comparable to that of base model. If either the load-use pair stalls the pipeline or the speculated thread

is squashed the BHR will be corrupted. If the latter case happens frequently the misprediction rate will be greater than the misprediction rate of the baseline model.

Table 3 shows that the load misprediction rate per 1000 instructions. These results are calculated by multiplying 1000 into the number of misses in the L1 Cache and dividing the result by the total number of instructions simulated. In our case the total number of instructions simulated are 100 million.

MisPrediction Rate Per 1000 instructions			
Benchmark	Thread1	Thread2	Thread3
bzip	5.86	5.83	5.83
gzip	9.05	8.98	8.98
mcf	53.36	46.75	45.93

Table 3. Load Misprediction Rate per Thousand Instructions

5.1 Reducing Branch Misprediction by using two BHR

In this section we are proposing 2 Branch History Register (BHR), one for the non-speculative thread and the other for the speculative thread.

At the time of spawning, the non-speculative thread BHR is copied to the speculative thread BHR. At the time of commit, pop the new bits stored in the speculative thread BHR to the non-speculative thread BHR. In this manner the non-speculative BHR will remain same as the base model BHR but it is possible that the gshare table is corrupted. Figure 7 shows this scenario.

Table 4 compares the branch misprediction rates for Dual-thread WELD with two BHRs with respect to the Dual-thread WELD with only one BHR.

Branch Misprediction		
Benchmarks	Thread2	Thread2
bzip	0.028352	0.028772
gzip	0.095369	0.083463
mcf	0.032920	0.028547

Table 4. Branch Misprediction Rate of Dual-Thread Weld with one BHR with respect to two BHR model

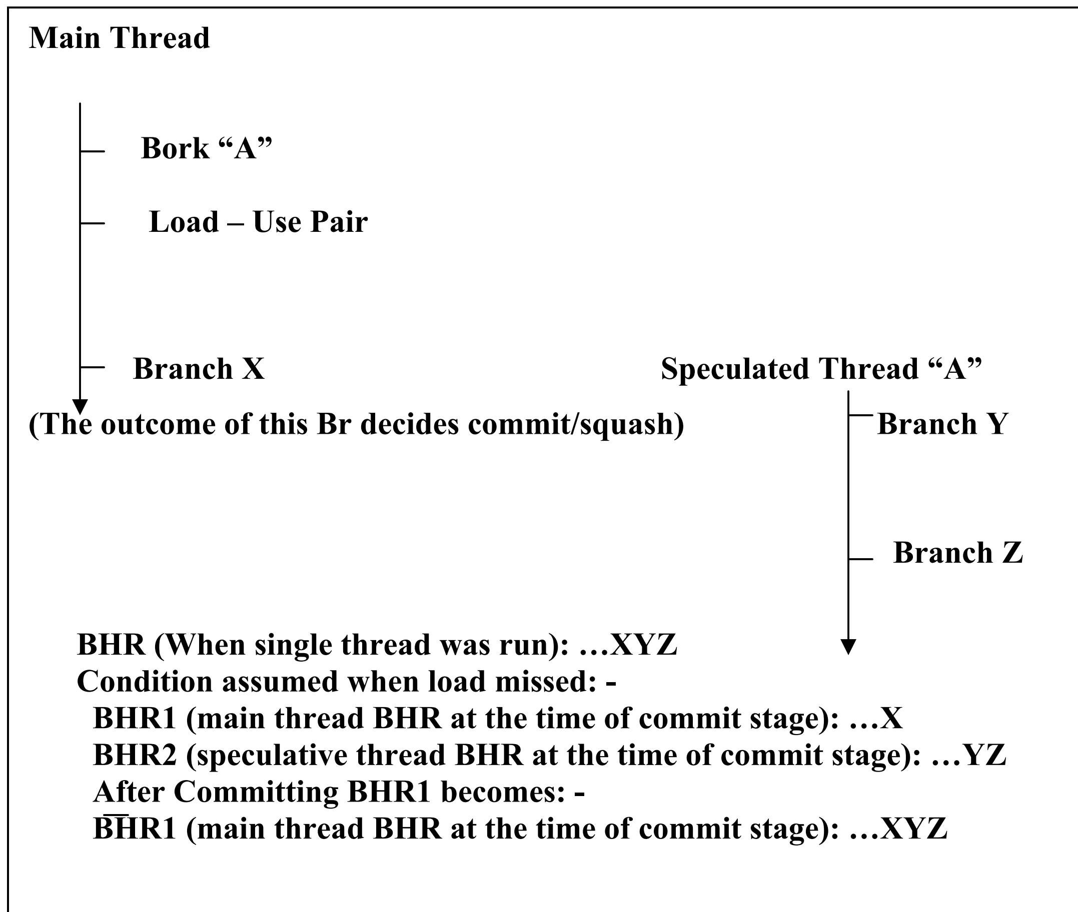

Figure 7. Example of main thread BHR and the speculative thread BHR in Dual-Thread Weld

Chapter 6 Conclusions and Future Work

In this thesis Weld is extended for Itanium processors. The Weld architecture combines the ISA, compiler and hardware for multithreading support. The compiler with the help of one new operation creates threads. This operation is Bork. At run time, threads are welded to fill in the holes by special hardware called the operation welder. The experimental results show a maximum of 10% speedup using a Dual-thread and 15% speedup for Triple-thread WELD processor compared to a single-threaded Itanium processor while maintaining the hardware simplicity of the EPIC architecture.

The future work continues towards enhancing performance using various performance boosting techniques. Weld's power was not harnessed to it's fullest extent by using natural treegions of smaller sizes. In our opinion we should make use of the scheduler of our own compiler on the assembly files generated by the electron compiler. Our experimental EPIC compiler schedules the treegion with aggressive speculation. It hoists operations from the bottom basic blocks in each treegion to the top of the treegion in order to fill the empty schedule slots. We could even use the tail-duplication mechanism to create large size of treegions. Large size of treegion will give us more opportunity to insert Bork in a schedule slot,

which will be much higher than the last basic block exit operation along the path of the treegion. This technique could boost the performance by much aggressive treegion-treegion overlapping.

We could even make the use of load profiling while inserting Bork. According to the observation made by [13], commonly 10 or fewer static loads cause 80% of the total L1 data cache misses. We could enforce a condition to always spawn a thread for pre-fetching these important loads. Furthermore we could be more aggressive in pre-fetching and filling up the vertical and horizontal holes by considering loop iterations as thread for Weld processor. We are also very interested in comparing the performance gain for using the pre-computation threads only for pre-fetching with respect to the performance gain attained from bypassing the results of the pre-computation thread to the non-speculated thread.

There is one more interesting issue; we could change the semantics of the Bork to give us the information of number of branches following the Bork instruction before the commit takes place. So at the time of spawning the non-speculative thread BHR will be copied to the speculative thread BHR. Moreover, the speculative thread BHR will be pushed by the number of branches stated by the Bork instruction. Finally at the time of commit, pop the new bits (corresponding to the branches executed in speculative

thread) stored into the speculative thread BHR onto the non-speculative thread BHR. In this manner we will be able to restore the BHR and the gshare table.

Chapter 7 Bibliography

[1] E. Ozer, "Architectural and Compiler Issues For Tolerating Latencies in Horizontal Architectures," Ph.D. thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, 2001.

[2] B. R. Rau, "Dynamically Scheduled VLIW Processors," Proc. 26th Ann. Int'l Symp. Microarchitecture, Dec 1993.

[3] P. K. Dubey, K. O'Brien, K. M. O'Brien and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading," in *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques*. (Cyprus). June 1995.

[4] H. Akkary and M. A. Driscoll, "A Dynamic Multithreading Processor," in Proc. 31st Ann. Int'l Symp. on Microarchitecture. Nov. 1998.

[5] G. S. Sohi, S. E. Breach and T. N. Vijaykumar, " Multiscalar Processors," in Proc.22nd Ann. Int'l Symp. Computer Architecture. (Italy). May 1995.

[6] J. –T. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control

Speculation," in Proc. Int'l Conf. Parallel Architecture and Compilation Techniques. (Boston). Oct. 1996.

[7] S. Wallace, B. Calder and D. M. Tullsen, "Threaded Multiple Path Execution," in Proc. 25th Ann. Int'l Symp. Computer Architecture, Barcelona, Spain, June 1998.

[8] D. M. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," in Proc. 22nd Ann. Int'l Symp. Computer Architecture, Italy, May 1995.

[9] G. Prasadh and C. Wu, "A Benchmark Evaluation of a Multithreaded RISC Processor Architecture," in Proc. Of Int'l Conf. on Parallel Processing, Aug. 1991.

[10] A. Wolfe and J.P. Shen," A Variable Instruction Stream Extension to the VLIW Architecture," in Proc. 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, ACM Press, Apr. 1991.

[11] A. Roth and G. Sohi. Speculative data-driven multithreading. In Seventh International Symposium on High Performance Computer Architecture, Jan. 2001.

[12] K. Sundaramoorthy, Z. Purser and E. Rotenberg. Slipstream Processors: Improving both performance and fault tolerance. In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems.

[13] S. G. Abraham and B. R. Rau. Predicting Load Latency using Cache Profiling. In Helwett Packard Lab, Technical Report HPL-94-110, Dec. 1994