ABSTRACT

OZER, EMRE

Architectural and Compiler Issues for Tolerating Latencies in Horizontal Architectures

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

This dissertation presents a new architecture model named Weld for horizontal architectures such as VLIW and EPIC. Weld integrates speculative multithreading support into a VLIW/EPIC 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 more efficiently. Hardware contexts such as program counters and the fetch units are duplicated to support multithreading.

Also, a dual-thread Weld architecture is isolated and analyzed for cost/performance purposes within the general Weld architecture. The dual-thread Weld model supports one main thread and one speculative thread running simultaneously in a VLIW/EPIC processor with a register file and a fetch unit per thread. The cost/performance impact of the dual-thread Weld model, which includes analysis of migrating the disambiguation hardware to the compiler and the sensitivity analysis to the variation of branch misprediction and second-level cache miss penalties, is examined further.

ARCHITECTURAL AND COMPILER ISSUES FOR TOLERATING LATENCIES IN HORIZONTAL ARCHITECTURES

by

EMRE OZER

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

COMPUTER ENGINEERING

Raleigh

2001

APPROVED BY:

Dr. Thomas M. Conte Chair of Advisory Committee	Dr. Edward W. Davis
Dr. Wentai Liu	Dr. Eric Rotenberg

BIOGRAPHY

Emre Özer was born in Ankara, Turkey on October 18, 1971 and grew up in Istanbul. He graduated from Yıldız Technical University, Istanbul with Bachelor's and Master's degrees on Computer Science and Engineering in 1993 and 1996, respectively. He started to work as a teaching assistant in Computer Engineering Department of University of Istanbul in 1994. He was awarded with a scholarship for Ph.D. study abroad. He started his Ph.D. program in Electrical and Computer Engineering Department at North Carolina State University in 1996. He joined to StarCore in Atlanta in September 2001.

ACKNOWLEDGEMENTS

I want to thank my advisor Dr. Tom Conte for his patience, guidance and contributions. It was an honor to work in TINKER group with him. I also want to thank my committee members Dr. Eric Rotenberg, Dr. Edward Davis and Dr. Wentai Liu for their contributions to my work. Special thanks to all alumni and new TINKER group members: Sumedh Sathaye, Sanjeev Banerjia, Kishore Menezes, Roger Isaac, Bill Havanki, Sergei Larin, Matt Jennings, Chao-ying Fu, Mark Toburen, Kim Hazelwood and Huiyang Zhou.

I am very grateful to my family for their support, love and encouragement from thousand miles away from here. I want to individually thank to my dear mother Saadet Özer, my dear father Dursun Özer, my sister Elif Özer and finally my brother Sırrı Özer.

TABLE OF CONTENTS

1.	INTRODUCTION	11
1.1.	Background on VLIW-style architectures	11
1.2.	Motivation of The Dissertation	3
1.3.	Layout of The Dissertation	4
2.	THE GENERAL WELD ARCHITECTURE MODEL	6
2.1.	ISA Extension	6
2.2.	Thread Creation and Synchronization	7
2.3.	Operation Welder	9
2.4.	Thread Table	11
2.5.	Speculative Memory Operation Buffer (SMOB)	12
2.6.	Speculative Store Buffer (SSB)	15
2.7.	Instruction Cache	16
2.8.	Interrupt and Exception Handling	17
2.9.	Concluding Remarks	19
3.	COMPILER SUPPORT FOR THE WELD ARCHITECTURE	27
3.1.	Bork Insertion Algorithm	27
3.2.	Complexity Analysis	30
3.3.	Performance Evaluation of the General Weld	30
3.4.	Concluding Remarks	35
4.	DUAL-THREAD WELD MODEL	48
4.1.	The Dual-thread Weld Architecture	48

4.2. Speedup Results of the Dual-thread Weld 52
4.3. Variance of the Sizes of the SMOB and SSB 53
4.4. Maximum Treegion Overlapping 54
4.5. Concluding Remarks 55
5. DUAL-THREAD WELD WITHOUT MEMORY SPECULATION 64
5.1. Algorithm 64
5.2. Speedup Results 65
5.3. Variance of Branch Penalty 66
5.4. Variance of L2 Miss Latency 67
5.5. Welding versus No welding 68
5.6. Variance of Register File Copy Cycle Time 68
5.7. Variance of the Size of the SSB 69
5.8. The Performance Effects of Using a Machine Model without Universal Units 69
5.9. The Performance Effects of Using a Machine Model Only with Universal Units 70
5.10. Concluding Remarks 71
6. RELATED WORK 85
6.1. Multithreading in Horizontal Architectures 85
6.2. Multithreading in Dynamically Scheduled Architectures 86
7. CONCLUSION AND FUTURE WORK 93
REFERENCES 95

LIST OF TABLES

TABLE 1. SEMANTICS OF BORK INSTRUCTION 20

TABLE 2. PRACTICAL COMPLEXITY POLYNOMIAL FUNCTIONS 37

TABLE 3. THE PROPERTIES OF THE EXECUTION-DRIVEN SIMULATION ENVIRONMENT 38

TABLE 4. THE LATENCIES OF INSTRUCTIONS 38

TABLE 5. INSTRUCTION CACHE MISS RATES 45

TABLE 6. COMPARISON OF PREVIOUS MULTITHREADING TECHNIQUES WITH WELD 91

LIST OF FIGURES

FIGURE 1. UNPREDICTABLE RUN-TIME EVENTS 5

FIGURE 2. DISCRETE WINDOW SHIFTING 5

FIGURE 3. THE GENERAL WELD ARCHITECTURE 20

FIGURE 4. THE MECHANISM OF THREAD CREATION AND SYNCHRONIZATION IN WELD 21

FIGURE 5. AN EXAMPLE FOR SYNCHRONIZATION OF TWO THREADS WITH A TRUE PATH. 21

FIGURE 6. AN EXAMPLE FOR SYNCHRONIZATION OF TWO THREADS WITH A WRONG PATH. 22

FIGURE 7. THE GENERAL STRUCTURE OF THE OPERATION WELDER 23

FIGURE 8. AN EXAMPLE WITH TWO THREADS FOR 4-WIDE VLIW MACHINE 24

FIGURE 9. THE EXECUTION STEPS OF THE SAMPLE PROGRAM 24

FIGURE 10. THE STRUCTURE OF THE THREAD TABLE 25

FIGURE 11. THE STRUCTURE OF THE SPECULATIVE MEMORY OPERATION BUFFER 25

FIGURE 12. SPECULATIVE STORE BUFFER STRUCTURE (SSB) 26

FIGURE 13. AN ILLUSTRATIVE EXAMPLE ABOUT THE SCHEDULED PROGRAM ORDER AND A RELATIVELY PRECISE INTERRUPT. 26

FIGURE 14. BORK INSERTION ALGORITHM 36

FIGURE 15. AN EXAMPLE FOR THE BORK INSERTION ALGORITHM 36

FIGURE 16. THE FINAL CODE AFTER INSERTING THE BORK OPERATION 37

FIGURE 17. EXPERIMENTAL FRAMEWORK 39

FIGURE 18. SPEEDUP RESULTS 40

FIGURE 19. IPC RESULTS 41

FIGURE 20. PERCENTAGE OF WELDED MULTIOPS 42

FIGURE 21. PERCENTAGE OF VERTICAL MULTIOPS 43

FIGURE 22. SPEEDUPS WITH WELDING VS. NO WELDING 44

FIGURE 23. THE L2CACHE MISS RATIOS 46

FIGURE 24. DCACHE MISS RATIOS 46

FIGURE 25. BRANCH PREDICTION ACCURACY 47

FIGURE 26. THE CODE SIZE INCREASE (%) 47

FIGURE 27. THE DUAL-THREAD WELD ARCHITECTURE 56

FIGURE 28. THE DUAL-THREAD REGISTER FILE DESIGN 57

FIGURE 29. AN EXAMPLE SHOWING THE THREAD CREATION AND SYNCHRONIZATION IN THE DUAL-THREAD WELD. CASE 1: THE ACTUAL PATH IS THROUGH THREAD 2. 57

FIGURE 30. AN EXAMPLE SHOWING THE THREAD CREATION AND SYNCHRONIZATION IN THE DUAL-THREAD WELD. CASE 2: THE ACTUAL PATH IS NOT THROUGH THREAD 2 58

FIGURE 31. THE STRUCTURE OF THE SMOB AND SSB 58

FIGURE 32. SPEEDUP RESULTS 59

FIGURE 33. IPC RESULTS 59

FIGURE 34. SPEEDUP RESULTS OF VARIABLE NUMBER OF SMOB ENTRIES WITH FIXED 64-ENTRY SSB 60

FIGURE 35. IPC RESULTS OF VARIABLE NUMBER OF SMOB ENTRIES WITH FIXED 64-ENTRY SSB 60

FIGURE 36. SPEEDUP RESULTS OF VARIABLE NUMBER OF SSB ENTRIES WITH FIXED 256-ENTRY SMOB 61

FIGURE 37. IPC RESULTS OF VARIABLE NUMBER OF SSB ENTRIES WITH FIXED 256-ENTRY SMOB 61

FIGURE 38. THE BORK INSERTION ALGORITHM WITH A PERFECT VALUE PREDICTOR 62

FIGURE 39. THE SPEEDUP RESULTS OF THE DUAL-THREAD WELD WITH PERFECT VALUE PREDICTION (PVP) VS. WITHOUT PVP 62

FIGURE 40. IPC RESULTS OF THE DUAL-THREAD WELD WITH PERFECT VALUE PREDICTION (PVP) VS. WITHOUT PVP 63

FIGURE 41. THE ARCHITECTURE OF THE DUAL-THREAD WELD WITHOUT SMOB 71

FIGURE 42. THE BORK INSERTION ALGORITHM FOR THE DUAL-THREAD WELD WITHOUT MEMORY SPECULATION 72

FIGURE 43. AN EXAMPLE OF BORK INSERTION IN THE MODEL WITHOUT MEMORY SPECULATION 72

FIGURE 44. THE FINAL CODE AFTER BORK INSERTION IN THE MODEL WITHOUT MEMORY SPECULATION 73

FIGURE 45. THE SPEEDUP RESULTS OF THE DUAL-THREAD WELD WITH NO MEMORY SPECULATION (I.E. NOSMOB) 75

FIGURE 46. THE IPC RESULTS OF THE DUAL-THREAD WELD WITH NO MEMORY SPECULATION (I.E. NOSMOB) 76

FIGURE 47. SPEEDUP RESULTS WHEN THE BRANCH PENALTY IS VARIED FROM 5 TO 15. 77

FIGURE 48. SPEEDUP RESULTS FOR L2 MISS LATENCY VARIANCE 78

FIGURE 49. SPEEDUP WITH WELDING VERSUS WITHOUT WELDING 79

FIGURE 50. SPEEDUP WITH VARIANCE OF REGISTER FILE COPY CYCLES 80

FIGURE 51. SPEEDUP WITH VARIANCE OF THE NUMBER THE SSB ENTRIES 81

FIGURE 52. IPC WITH VARIANCE OF THE NUMBER THE SSB ENTRIES 81

FIGURE 53. SPEEDUP WITH WELDING VS. NONWELDING IN A MACHINE MODEL WITHOUT ANY UNIVERSAL UNITS 82

FIGURE 54. SPEEDUP RESULTS OF THE DUAL-THREAD WITH NO SMOB FOR A MACHINE MODEL WITH NO UNIVERSAL UNITS 82

FIGURE 55. IPC RESULTS THE DUAL-THREAD WITH NO SMOB FOR A MACHINE MODEL WITH NO UNIVERSAL UNITS 83

FIGURE 56. SPEEDUP WITH WELDING VS. NONWELDING IN A MACHINE MODEL WITH ALL UNIVERSAL UNITS 83

FIGURE 57. SPEEDUP RESULTS OF THE DUAL-THREAD WITH NO SMOB FOR A MACHINE MODEL WITH ALL UNIVERSAL UNITS 84

FIGURE 58 IPC RESULTS OF THE DUAL-THREAD WITH NO SMOB FOR A MACHINE MODEL WITH ALL UNIVERSAL UNITS 84

1. Introduction

The very first attempt of designing processors in horizontal style (VLIW or EPIC) started with Cydrome Cydra 5 [66][67] and Multiflow TRACE [71][72]. They did not become popular because their performances were not promising enough due to the lack of advanced parallelizing ILP compilers. Recently, improved compiler techniques and hardware simplicity has attracted many manufacturers to build microprocessors in a horizontal architecture. There are a few commercial and a large number of embedded VLIW/EPIC microprocessors. The commercial ones are IA-64 Itanium[78] and Transmeta Crusoe[77] and Sun MAJC[76] and the embedded ones are Texas Instruments TMS320C62 [68], Philips TriMedia TM1000 [69], Chromatic Mpact [70], Fujitsu FR500 [79] and StarCore SC100 [80].

1.1. Background on Horizontal architectures

Very Long Instruction Word (VLIW) architectures are statically scheduled architectures that contain multiple functional units (FUs). Multiple independent operations are sent to these FUs at each clock cycle. Independent operations that can be executed in parallel are determined at compile time. Similarly, Explicitly Parallel Instruction Computing (EPIC) is the VLIW architecture with some superscalar-like extra features. The common features in VLIW and EPIC are independent instructions packed into MultiOps by the compiler and no run-time dependency checking hardware. This effectively reduces the amount of hardware on chip and simplifies the implementation

and even leads to a faster clock cycle. Those are the features that deviate from the superscalar processors. However, there are some disadvantages of the statically scheduled architectures such as object-code compatibility and vulnerability of the schedule to the unpredictable run-time events. Object-code compatibility arises from two things: 1) operation latencies assumed by the compiler may not be correct for the next-generation processor. 2) The number of functional units may be different from generation to generation. The object-code compatibility issue and solution approaches can be found in[74][75]. The main topic of this dissertation is to attempt to eliminate the unpredictable run-time deficiencies of VLIW/EPIC architectures. The recently announced Hewlett-Packard, SGI and IBM servers that employ the Intel/HP Itanium EPIC processor for general-purpose and transaction workloads. In all of these applications, horizontal architectures (i.e. VLIW or EPIC) suffer from operation latencies that vary at run time. The processor might have to stall for a large number of cycles until the memory system returns the data required by the processor. 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. It has been used in multiple-context processors where different programs are interleaved into the pipeline on a cycle-by-cycle basis [13][14][15]. Speculative multithreading is also used to improve the single program performance by spawning threads (loop iterations or an acyclic piece of code). A new architecture model named Weld is introduced to support speculative multithreading for VLIW/EPIC processors. It sits on a VLIW/EPIC processor core and contains an additional hardware mechanism that can allow multiple threads from a single program and weld operations from different threads to utilize the empty issue slots.

1.2. Motivation of The Dissertation

There are two main goals that Weld tries to achieve: utilizing the processor resources during unpredictable run-time events and dynamically filling the issue slot holes that cannot be filled at compile time. Unpredictable events that cannot be detected at compile time such as a cache miss may stall a VLIW processor for numerous cycles. By issuing instructions from other threads within the same program in case of an unpredictable run-time event, Weld can overcome a crucial drawback of VLIW/EPIC architecture. The unpredictable run-time event problem is shown in Figure 1. The figure depicts a cache miss as an example. The window in the left-hand side shows all instructions scheduled by the compiler. The window in the right-hand side shows the run-time view of the same program. Instruction 3 misses in the cache and therefore the processor stalls until the value is retrieved from the lower-level memory. In the meantime, the variable number of cycles that is dependent on the lower-level memory access latency is lost. The Weld architecture tolerates those wasted cycles by issuing other active threads when the processor stalls for an unpredictable run-time event. Horizontal architectures are limited by discrete scheduling window (see Figure 2) where attempts are made to fill the schedule holes from a scheduling region at compile time. The compilers partition the program into several scheduling regions and each scheduling region is scheduled by applying different ILP optimization techniques such as speculation, predication and loop unrolling and etc. The compiler tries to pack operations into VLIW packets but may leave some empty schedule slots due to dependences between operations in the region. The obvious question that may arise is that why the

compiler does not merge some scheduling regions to get bigger regions and may increase the chance of a higher ILP. There are various reasons for a compiler not to do that. Firstly, the code size increases since a large amount of code should be duplicated in order to make regions bigger. Secondly, acyclic region formation does not include loops and has to stop going beyond the loops. The VLIW compiler cannot fill all schedule slots in every MultiOp [22] because the compiler does not migrate operations from different scheduling regions. A hardware mechanism called the *operation welder* is introduced to achieve our second goal. It merges operations from different threads in the issue buffer during each cycle to eliminate NOPs. Executing operations from different scheduling regions (i.e. threads) and filling the issue slots from these regions simultaneously at run time can increase resource utilization and performance. Integration of scheduling regions at compile-time might be a software alternative to hardware operation welder. However, this would increase the code size enormously since treegions need be tail-duplicated. Moreover, this can also slow down the scheduling time. The main motivation of this study is to increase the instruction level parallelism using compiler-directed threads (scheduling regions) while maintaining the hardware simplicity of VLIW architecture.

1.3. Layout of The Dissertation

The remainder of the dissertation is organized as follows. Chapter 2 introduces the General Weld Architecture. Chapter 3 provides the compiler framework for the Weld architecture and also presents performance results and evaluations. Chapter 4 proposes a dual-thread Weld Architecture model. Chapter 5 discusses the dual-thread Weld without

load-store speculation at the thread level. Chapter 6 gives the related work in multithreading techniques and provides an illustrative comparison of Weld with the other relevant schemes. Finally, Chapter 7 concludes the dissertation and discusses the future work.

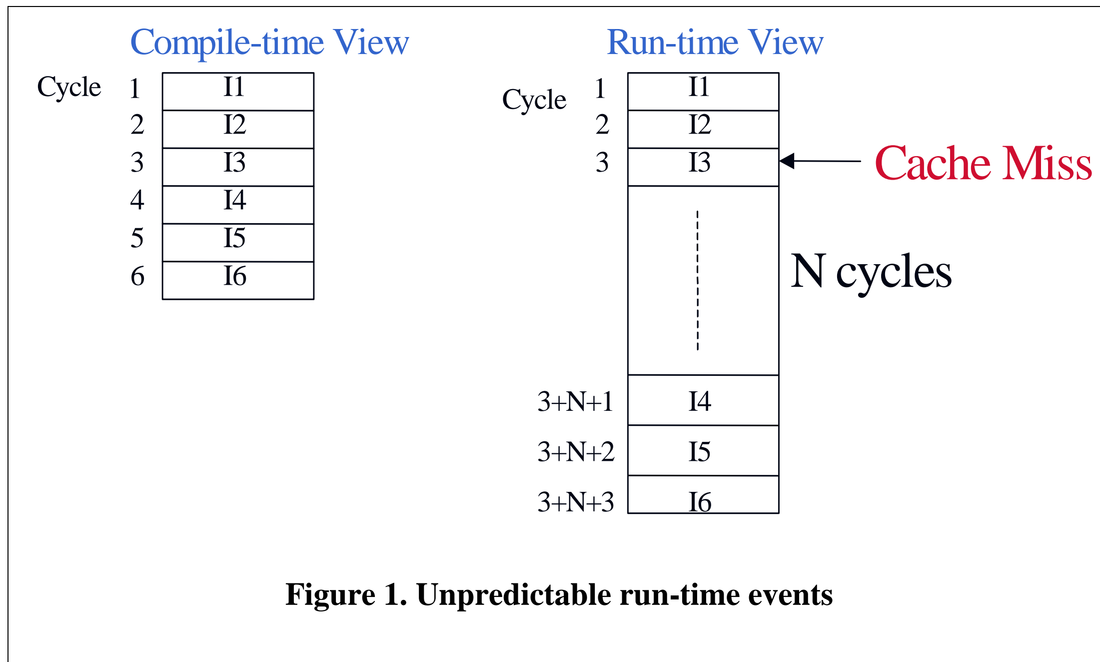


Figure 1. Unpredictable run-time events

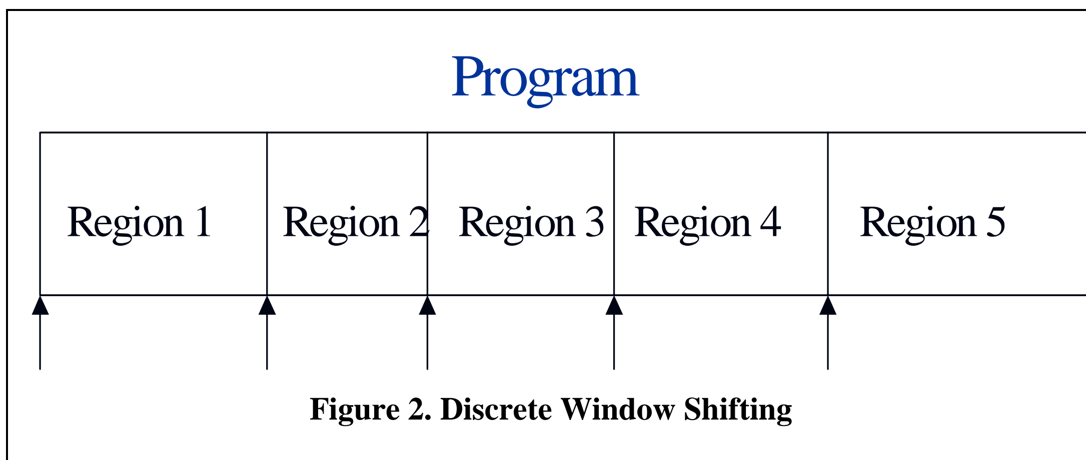


Figure 2. Discrete Window Shifting

2. The General Weld Architecture Model

Weld assumes that threads are generated from a single program by the compiler. There is a single main thread and several speculative threads but the main thread has the highest priority among all threads. The general Weld architecture is shown in Figure 3. Each thread has its own program counter, fetch unit and a register file while all threads share the branch predictor, instruction and data caches. Weld has a 5-stage pipeline. The fetch stage fetches MultiOps from the Icache, and the weld/decode stage welds and decodes them. The operation welder is integrated into the decode stage in the pipeline. This stage can also be pipelined but we assume a single combined operand read and weld stage in this study. The operand read stage reads operands into the buffer for each thread, and sends them to the functional units. The execute stage executes operations and finally the write-back stage writes the results into the register file and Dcache.

2.1. ISA Extension

A new instruction and some extension of the ISA are required to support multithreading in a VLIW or EPIC processor [57]. A new instruction is needed to spawn and synchronize threads: a branch and fork operation (**bork**) to spawn a thread. A bork spawns a new thread and creates a new context. Semantics of bork operation are given in Table 1.

A single bit must be added to each MultiOp for distinguishing between separable and inseparable MultiOps. In a separable MultiOp, operations from a single MultiOp can be issued at different cycles in time. On the other hand, in an inseparable MultiOp all the

operations have to be issued at the same time. The reason for this classification is that there might be anti or output dependencies between operations in a MultiOp. Splitting such operations from the whole MultiOp may disrupt the correct execution of the processor. Therefore, the compiler detects and marks such MultiOps as inseparable and the hardware sends this MultiOp as a whole at run time. Note that MultiOps in an EPIC architecture such as IA-64 [23] are always separable.

Also, a synchronization bit is added to each MultiOp in the ISA. This bit is set in the first MultiOp of each thread at compile-time to help synchronize threads at run-time. With the combination of an ISA bit and hardware support, threads can merge in a straightforward way.

2.2. Thread Creation and Synchronization

A bork instruction has a target address to start the speculative thread. After a bork is executed, a new hardware context (register file, fetch unit and a program counter) is assigned to the new thread if there is any available. If there is none, the bork behaves like a NOP. Then, the register file of the ancestor thread is copied into the register file of the descendant thread. The program counter of the descendant thread is initialized with the bork target address. Moreover, the target address is also stored in the Thread Table for thread synchronizations. Speculative threads can then spawn other speculative threads, and so on. A thread (main or speculative) can spawn only one thread i.e. the compiler guarantees that there is only one bork operation per executed path. If there is a stall on one thread due to a cache miss, the other threads can still continue to fetch and execute. Even if there is no stall in any thread, the decode/weld stage can still fill from multiple

threads by taking advantage of any empty fields in the MultiOps. Therefore, the Weld architecture can utilize both horizontal (operations from multiple threads in a cycle) and vertical threading (operations from only one thread in a cycle) simultaneously. Figure 4 illustrates the mechanism of thread creation and synchronization in Weld for three threads. In the very beginning, there is only one thread, which is the main thread. At the time of the first bork execution in the main thread, a new speculative thread is created by copying the main thread's register file into the speculative one's and writing the bork target address into the new program counter. From this point on, the main thread cannot spawn any other thread as complied by the Weld execution flow. However, the speculative thread can spawn another speculative thread if it executes a bork operation. In the illustration, this can be seen as the speculative thread 1 creates the speculative thread 2 upon executing a bork operation by dumping its register file contents into the speculative thread 2's register file and writing the bork target address into the new program counter. When the main thread merges with the speculative thread 1 as seen in the third column, the main thread dies, which means the register file is made available for the new speculative threads and the speculative thread 1 becomes the new main thread. Now, the new main thread merges with the speculative thread 2. Similarly, it dies and the speculative thread 2 becomes the new main thread.

An ancestor thread merges with its own descendant thread when the ancestor fetches the first VLIW instruction in the descendant thread. The fetch unit checks if the synchronization bit of the fetched VLIW is set. If set, the PC address of the instruction is compared with the address stored in the Thread Table. If they are the same, the descendant thread is correctly speculated and can be committed. If the addresses do not

match, then the descendant thread is incorrectly speculated and must be squashed. The ancestor thread dies and the descendant thread takes over in case of a commit. This mechanism provides inexpensive thread synchronization at run-time. In case of a squash, the descendant thread's register file is made available for new threads and the operations belonging to this thread are flushed. Also, the speculative stores in the Speculative Store Buffer (SSB) and the speculative loads in the Speculative Memory Operation Buffer are invalidated. Figure 5 shows an example for synchronization of two threads. Thread 0 (T0) spawns Thread 1 (T1) at address 100. The address 100 is also written in the Thread Table together with the thread id. If the control flow goes into T1, T0 fetches the first VLIW, which is the VLIW at address 100. The fetch unit for thread 0 checks if the synchronization bit is set. Then, the PC address 100 is compared with the address in the Thread Table, which is also 100. There is a match, therefore T1 can commit, which means T0 dies and T1 becomes the main thread. If the control flow goes into Thread 2, which means that T1 is misspeculated as shown in Figure 6. The PC address 400 is compared with the address 100 in the Thread Table. There is no match, so T1 is misspeculated and must be squashed.

2.3. Operation Welder

Weld/Decode stage in the pipeline takes MultiOps from each active thread, welds them together and decodes the welded MultiOp. The welding should be done before operand read stage because the number of bits in an operation to be routed before decode is much less than the number of bits in the operand read stage. Therefore, the operation welder is integrated into the decode pipeline stage. It can send an operation to any

functional unit as long as it is the correct resource to execute the operation. It consists of an array of multiplexers to forward the operation to the selected functional unit. Control logic sends control signals to the multiplexers to select the right input. Each thread has a buffer called the prepare-to-weld buffer to hold a MultiOp. At each cycle, a MultiOp is taken from the fetch buffer to weld/decode stage. 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 the input to the multiplexer control. Figure 7 shows the general hardware structure of an operation welder. Each functional unit has a multiplexer in front of it. Each multiplexer has an input connection from all operations in every thread assuming that all functional units are universal. Based on the priority of threads and availability of operations in the MultiOps, multiplexer control selects one of the inputs and sends it to the associated functional unit. The number of multiplexer inputs is determined by the number of threads times machine issue width. In the Weld architectural model, the main thread has the highest priority. Among the speculative threads, old speculative threads in time have higher priority than younger ones. Some operations in a MultiOp of a speculative thread might be sent to the functional units, the remaining operations stay in the prepare-to-weld buffer. In this case, the fetch from this thread is suspended by issuing a fetch stall line until the remaining operations in the buffer are issued. The fetch stall signal can be asserted by checking the empty/full bits of the remaining operations. The multiplexer control logic takes empty/full and separability bits of all operations in every thread and issues the necessary control signals for multiplexers. The number of multiplexer control signals for each

multiplexer is the logarithm in base 2 of the product of issue width and the number of threads the hardware can support.

An example in Figure 8 shows how operation welder works. There are two threads scheduled for 4-wide issue VLIW machine. Two ALU, one branch and one universal functional units are used in the architecture. The thread on the right is the speculative thread and it has a separability bit for each MultiOp. Figure 9 shows the execution steps of the sample program. At cycle 0, the first MultiOps from both threads are in the prepare-to-weld buffer. First, the whole MultiOp from A is forwarded into the issue buffer. Then a check is made if the separability bit of the MultiOp from B is set. Since it is set, the operations from B are separable. B1 is welded into the slot 4 of the issue buffer. The rest of operations stay in the prepare-to-weld buffer of B. At cycle 1, the second MultiOp from A is put in the buffer and forwarded into the issue buffer. B2 is welded into the slot number 3 of the issue buffer. At cycle 2, thread A puts the third MultiOp and thread B puts the second MultiOp in their prepare-to-weld buffers. Similarly, thread A forwards all operations into the issue buffer. However, there is no welding possible for thread B because its separability bit is 0. Therefore, all operations from this MultiOp must be sent to the issue buffer at the same time. Only the MultiOp from thread A is sent to the functional units. At cycle 3, the last MultiOp from thread A is put into the prepare-to-weld buffer and forwarded into the issue buffer. A check is made if all operations from B can be welded. Since there are three slots available in the issue buffer and they are the right resources, B3, B4 and B5 are all welded into the issue buffer.

2.4. Thread Table

The thread table (TT) keeps track of threads spawned and merged. There is an entry for each active thread. Each entry keeps thread id, register file id, the borked PC address. The thread id is a unique number that identifies each active thread. This unique id can be obtained from a time-stamp counter. Time stamp represents the order of threads in time. Each time a new thread is created, the counter is incremented by one and stamped into the thread. The time-stamp counter is reset to zero when there is only one active thread, which is the main thread. The register file id is the identifier of the register file assigned for the thread. Time stamp id is also attached to each operation in a thread to distinguish operations from different threads. When an operation completes, it searches TT to find the register file id by comparing the attached time stamp with time stamps in the table.

TT can be designed as a shift register as shown in Figure 10. When two threads merge, the ancestor thread dies and the descendant thread takes over. This involves deleting the ancestor thread's entry from TT. Before removing it, its borked PC is copied into the borked PC field of its own ancestor thread if there is any. Then, the descendant thread' s entry is shifted up and overwritten into the ancestor thread. The number of TT entries depends on the maximum number of threads that the Weld architecture implementation can allow.

2.5. Speculative Memory Operation Buffer (SMOB)

Load operations from speculative threads are kept in a buffer called the Speculative Memory Operation Buffer (SMOB). It uses a single shared fully associative buffer to resolve run-time memory violations. For simplicity, the SMOB is assumed to be a single buffer with multiple read and write ports shared by all threads. A SMOB entry

contains speculative load memory address, speculative thread's time stamp and a valid bit as shown in Figure 11. An outstanding speculative/nonspeculative store memory address is compared associatively in the SMOB for a conflict. A conflict can occur only when an outstanding store address matches a load address in a more speculative thread in time in the SMOB. The main thread store operations check all load operations in the SMOB. On the other hand, a speculative thread checks only the descendant speculative threads (i.e. by comparing time stamps larger than itself) for a conflict. Note that time stamps are assigned to threads in the ascending order from the time stamp counter. If there is a conflict, the speculative thread and all descendant speculative threads are squashed. The SMOB entry and the other entries with the same time stamp or bigger are invalidated in the SMOB. The main thread re-executes all instructions in the speculative thread where a thread misspeculation has occurred. Currently, there is no selective re-execution mechanism in the Weld architecture.

The mechanics of the SMOB is similar to Address Resolution Buffer (ARB) [31][32]. ARB is designed as a set-associative buffer divided into several banks where the associative comparison is made. Each bank is accessed with the memory address. Each row in a bank has a memory address field and many divided stages. Each stage corresponds to a memory operation that occurred in order they are sent to the instruction window, i.e. the first stage holds the earliest memory operation and the last stage holds the last memory operation. Each stage also has a load bit, a store bit and a value bit. The stages are considered as a circular queue with the head and tail pointers that show the current active window.

When a load is executed, the ARB bank is found by the load address and then all addresses in the bank are associatively compared with the outstanding load address for a possible match. If match, then an earlier store at the same address is searched in the active window. If there is an earlier store, then this store value can be forwarded to the processor. If no match, then an available row entry is assigned to this load. If there is no available entry, then a row entry is freed by squashing all instructions beyond the freed memory operation.

When a store is executed, the ARB bank is determined and all addresses in the bank are associatively compared with the outstanding store address for a possible match. If no match, a new entry is allocated for the store. If match, a check is made to find a succeeding load operation at the same address. If there is such an operation, the load operation and all operations after it are squashed.

In Memory Conflict Buffer (MCB)[62] technique, the compiler inserts a special operation named check that is inserted in the original location of the speculative load. As soon as it is executed, the MCB hardware is checked for a possible load/store conflict. MCB is a set-associative buffer that keeps track of stores and loads. A similar concept can be applied in Weld architecture by replacing the SMOB with the MCB. For each load in the speculative thread, a check instruction can be inserted into the beginning of the treegion at the thread merge time where the decision is made whether the speculative thread must be committed or squashed. There are two disadvantages of using this scheme. 1) The detection of a load/store conflict is delayed for several cycles as compared to the SMOB technique because the store operations can signal the load/store conflict as soon as they are executed in the SMOB. However, the check instruction

signals the conflicts in the MCB scheme. 2) A high number of check instructions must be inserted in the beginning of each treegion but the beginning of each treegion is most densely populated section of the treegion. This requires creating empty MultiOp or MultiOps for such instructions and therefore increasing the schedule length of the treegion.

In Weld architecture model, load-store conflicts and saving speculative stores until commit time are split into two different buffers: SMOB and SSB, respectively. The reason for this is to save buffer space by not allocating a store value field for a load operation. This would waste the expensive caching area in the processor. The load-store conflicts are resolved in the SMOB and the store forwarding on a load in the same thread is performed in the SSB.

Other run-time memory disambiguators can be found in [59][60][61][63].

2.6. Speculative Store Buffer (SSB)

Speculative store values are kept in the *speculative store buffer* (SSB) shared by all speculative threads. The main thread updates the data cache as soon as the store operations are executed. Speculative threads write store values in the SSB until the commit time. Each SSB entry contains a memory address, store value, a next link pointer, store write time and a valid bit as shown in Figure 12. Each thread has a head pointer and a tail pointer to the SSB that denote the first and last store operations from that thread. Those pointers are saved in a table called the *pointer box*. When a speculative store operation executes, a search is made to find an available entry in the SSB. If any, the entry is allocated and memory address and store value are all written into the entry. The

next link pointer of the previous store in the same thread pointed by the tail pointer is set to point to the current SSB entry. The time stamp of a store operation decides how to access the correct pointer set (i.e. the head and tail pointers). A speculative load operation in a thread can read the most recent store by reading the value with the latest store write time. When a store is written into the table, the store write time value is obtained from a counter that is incremented by one every time a new store is written. The counter is reset when there is no active speculative thread left. This can be performed by comparing the load address with the store addresses between the head and tail addresses in the SSB. Speculative stores are written into the data cache, in order, as soon as the speculative thread is verified as a correct speculation since stores in a thread are executed in order. Those entries are removed (i.e. at the time of a commit or a squash) from the SSB by visiting all links starting from the thread's head pointer to the tail pointer. If the SSB becomes full, all speculative threads stall. The main thread continues and frees the SSB entries by committing threads.

2.7. Instruction Cache

A shared instruction cache is used for all active threads. The instruction cache has multiple read ports to allow each thread to read a MultiOp per cycle. Each cache block is assumed to hold a MultiOp. The program in memory is compressed, that is the program is encoded such that it is NOP-free. When the program is loaded into the instruction cache, it is uncompressed by an expander [58]. When uncompressed, the MultiOps are expanded to include NOPs where necessary.

2.8. Interrupt and Exception Handling

In this section, interrupt handling for a VLIW processor is discussed systematically and new terms about the interrupt problem in a VLIW processor are defined.

In VLIW architecture, an interrupt is inherently imprecise with respect to the original program order because this order of operations is not known at run time. Only the order of MultiOps is known at run time. If an operation within a MultiOp causes an interrupt, the whole MultiOp can be said to be at the precise interrupt boundary. So the interrupt will be precise with respect to the scheduled program order (*i.e*. the order of MultiOps), although it could seem to be imprecise with respect to the original program order. The precise interrupt concept is redefined here for a VLIW processor. An interrupt is called ***relatively precise*** if it satisfies these conditions:

(1) All MultiOps that precede the interrupted MultiOp must have been completed and modified the processor state.

(2) All MultiOps that succeed the interrupted MultiOp should not have been completed and modified the processor state.

(3) The program counter must point to the interrupting MultiOp.

An interrupt is called ***relatively imprecise*** if it does not satisfy any of the conditions above. Then, the original precise interrupt definition should be termed as ***absolutely precise***. Similarly, an interrupt is ***absolutely imprecise*** if it does not satisfy any of the absolutely precise interrupt conditions. Figure 13 illustrates an example that explains the distinction between an absolutely precise interrupt in original program order and a relatively precise interrupt in the scheduled program order. It demonstrates a sequential machine with an absolutely precise interrupt at operation I2. On the other hand, the same

program is scheduled for a two-issue VLIW machine. In this case, the same interrupt occurs at the boundary of MOP4. The interrupt is precise relative to the order of MultiOps, but imprecise relative to the original program order because I4, I6, I3 and I7 of the previous MultiOps may have completed or even retired. In relatively precise interrupts, the interrupt handler should not modify any of source registers in the interrupted operation. Otherwise, subsequent operations with respect to the original program order that use the same source register would need to be re-issued and re-executed. The processor cannot reexecute those operations because they may have already updated the processor state.

An interrupt or an exception in the Weld architecture model has two different behaviors to external interrupts and internal interrupts (i.e. exceptions). In order to handle the external interrupts, the current context, which includes the main thread and speculative threads must be saved before switching to the interrupt handling routine. There are two ways of switching to the handler. The first one is to save all PC addresses of all active threads and save the register files, SMOB, SSB and thread table contents. Upon returning from the interrupt, the saved state can be restored and all threads can be re-activated. The disadvantage is the time to save all thread states and restore them. The second solution can be to save only the main thread's state and squash all speculative threads. This is much simpler way and does take less time to save and restore. However, the performance degrades since all speculative threads are squashed as soon as an interrupt occurs. As to exceptions, such as division-by-zero and an overflow, the handling depends on if the exception has occurred in a speculative thread or the main thread. If it occurred in a speculative thread, it should not be taken since the thread is still

speculative. Instead of squashing all speculative threads, the more speculative threads after the excepted one are squashed and the excepted thread stalls until the main thread progresses and takes the exception. The rest of the speculative threads (i.e. the ones before the excepted thread) continue to progress. If an exception occurs in the main thread, the exception is taken immediately and all speculative threads are squashed to guarantee the correctness. Particularly, if the exception handler modifies a live-out register, this modified register is not reflected on the speculative threads, therefore all speculative threads are thrown away from the processor.

2.9. Concluding Remarks

In this chapter, the general architecture of Weld is introduced. The components such as thread creation, thread synchronization, interrupt handling mechanisms and memory disambiguation required for multithreading support for a generic VLIW/EPIC processor are explained in detail. The architectural design of these components is given but their implementations may vary according to the systems they will be developed for.

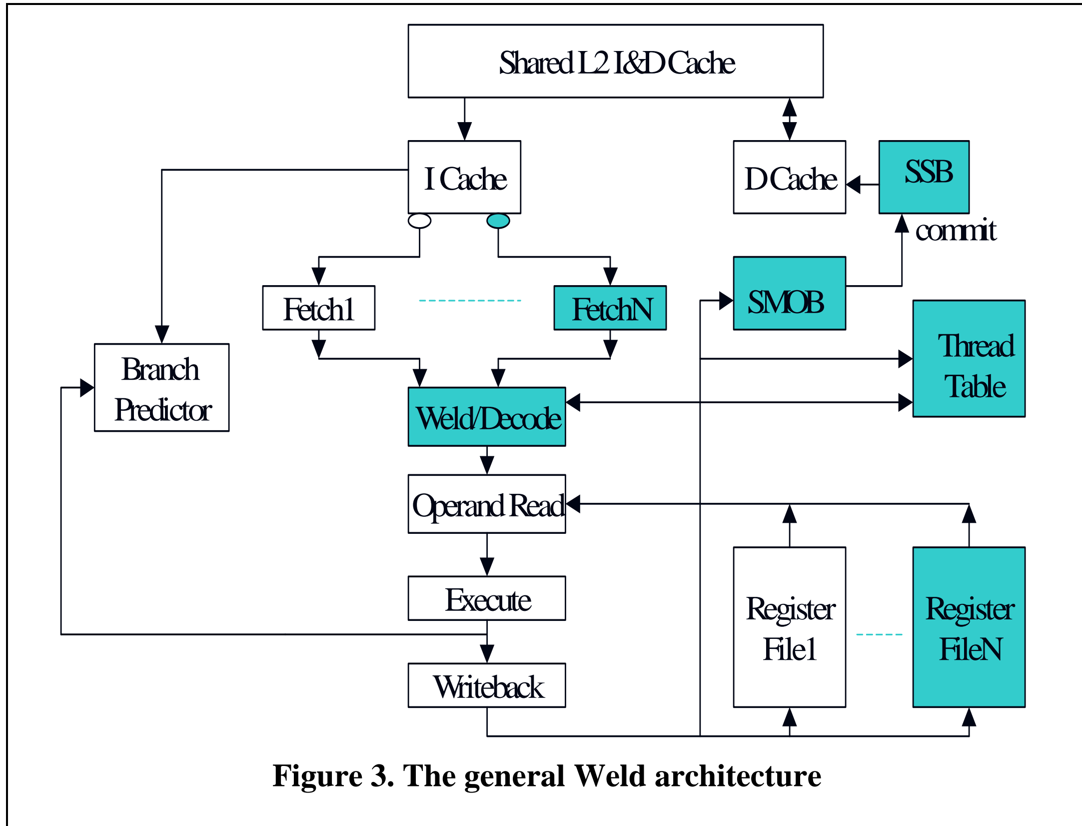


Figure 3. The general Weld architecture

Table 1. Semantics of Bork Instruction

Instruction	**Definition**
Bork Target Address	Target Address: PC address of the thread

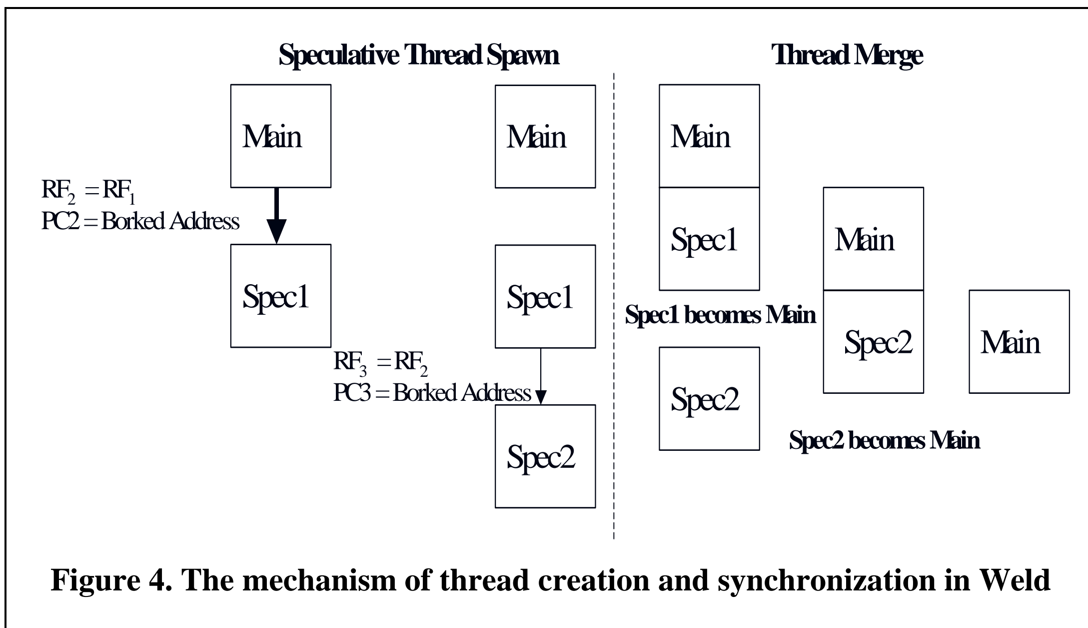


Figure 4. The mechanism of thread creation and synchronization in Weld

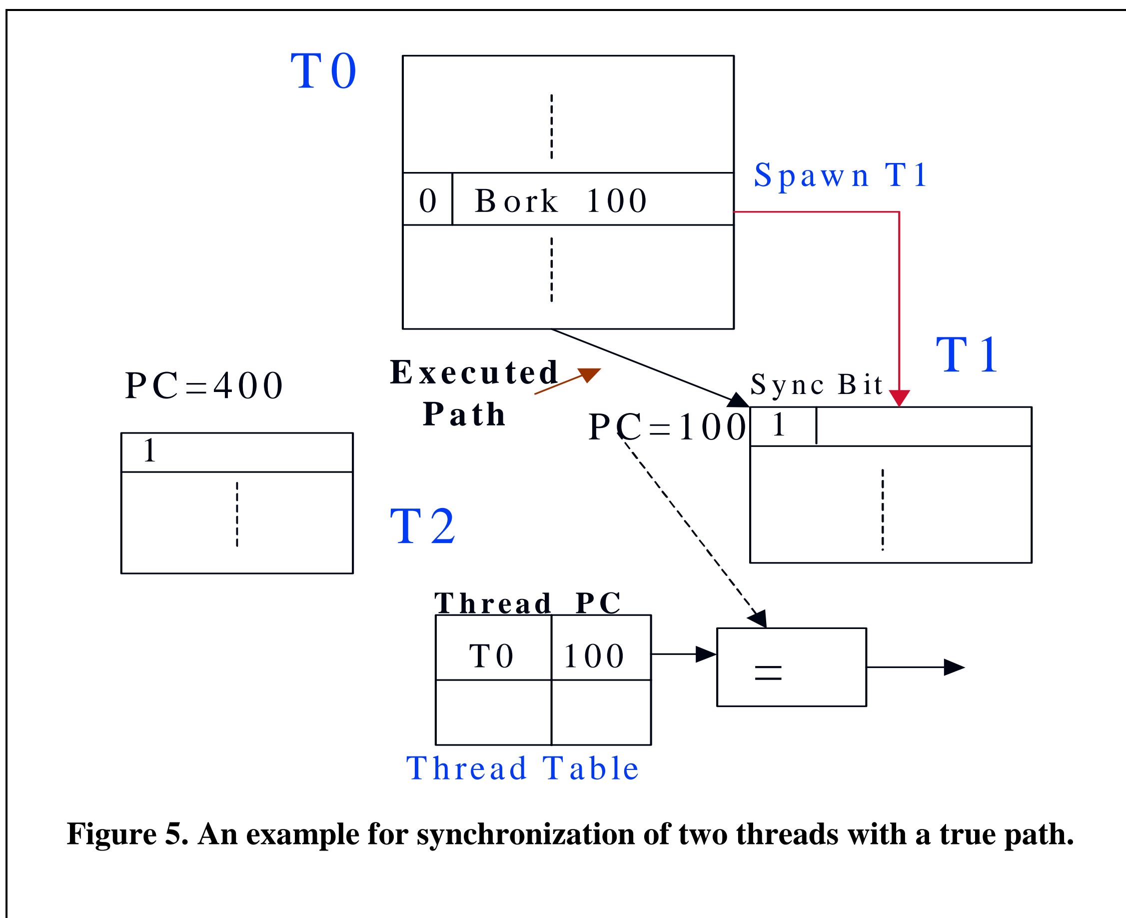


Figure 5. An example for synchronization of two threads with a true path.

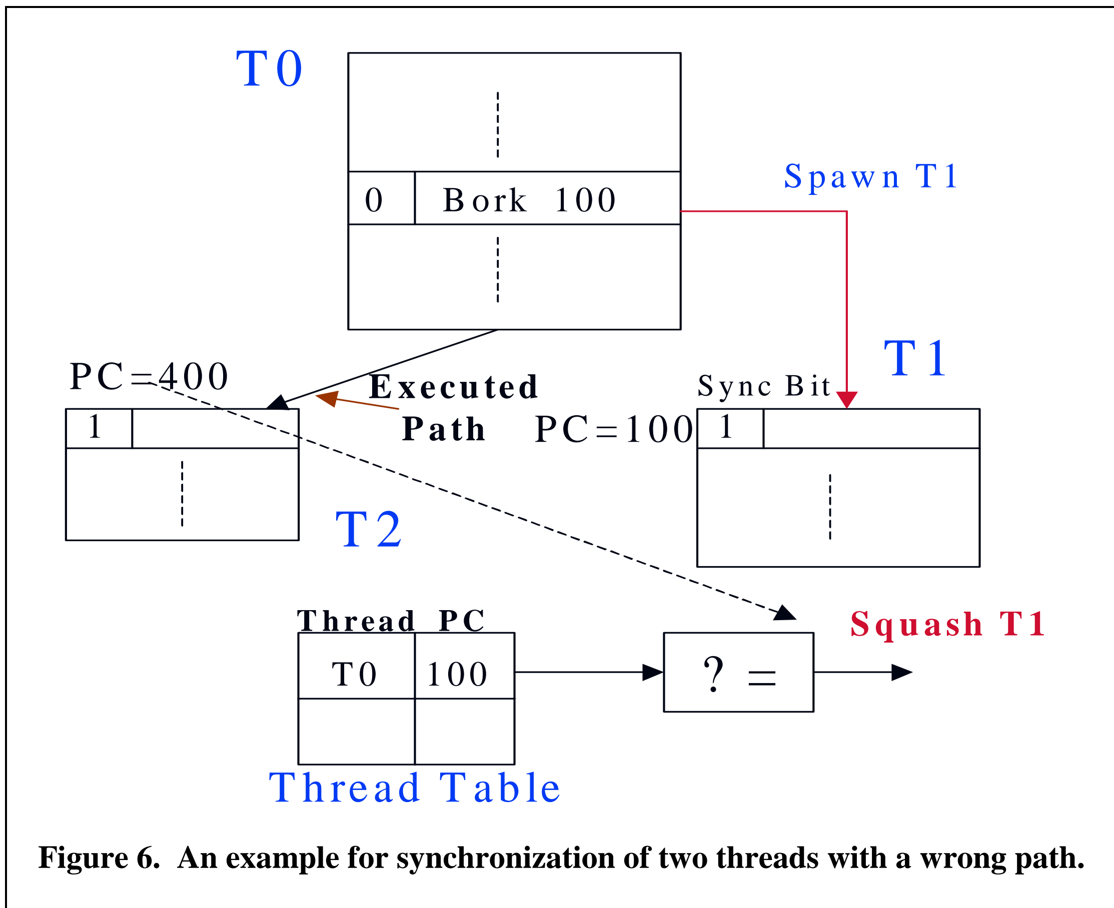
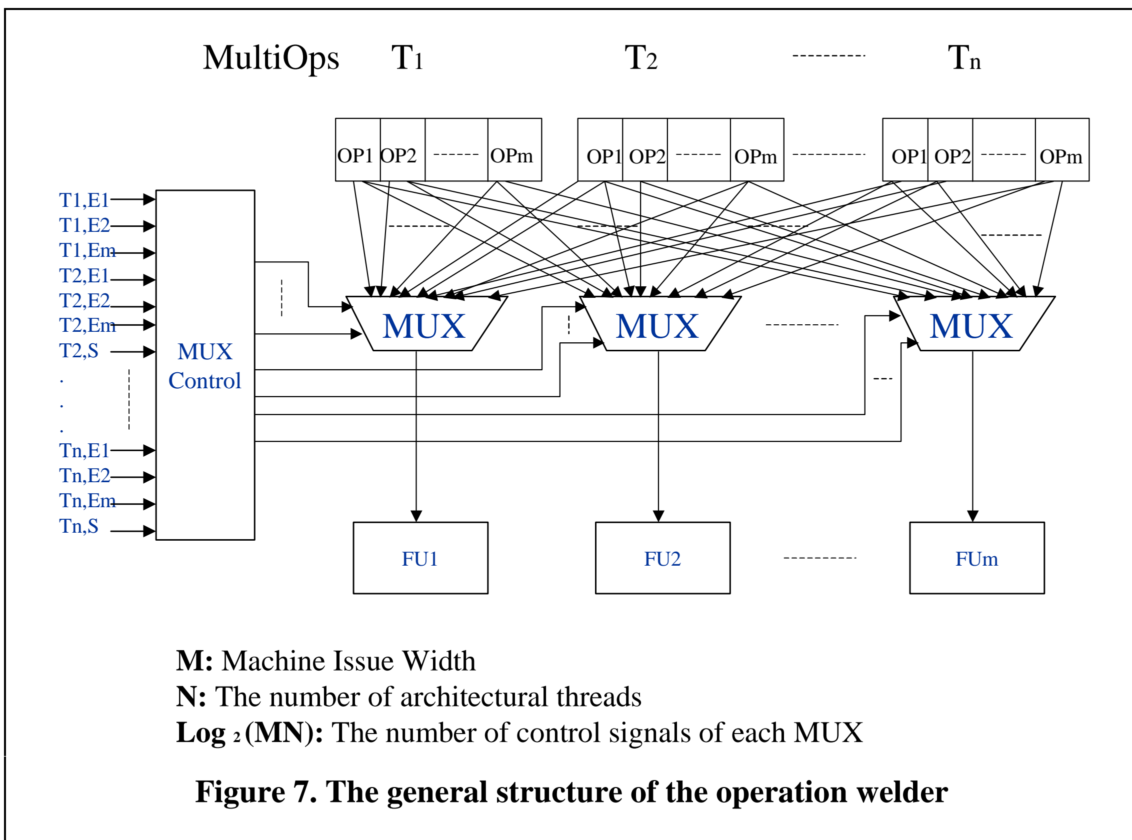


Figure 6. An example for synchronization of two threads with a wrong path.



M: Machine Issue Width
N: The number of architectural threads
Log $_2$(MN): The number of control signals of each MUX

Figure 7. The general structure of the operation welder

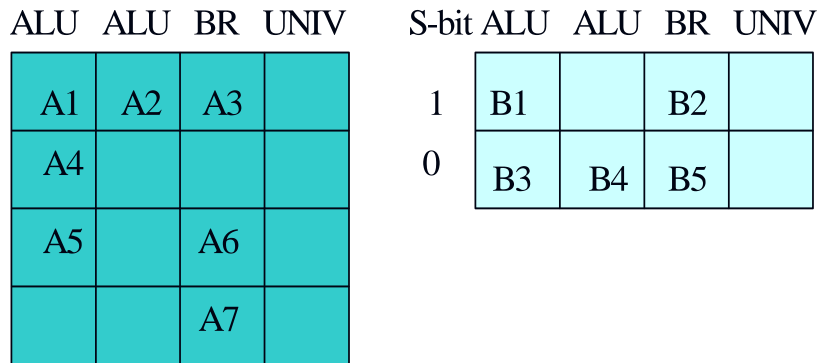


Figure 8. An example with two threads for 4-wide VLIW machine

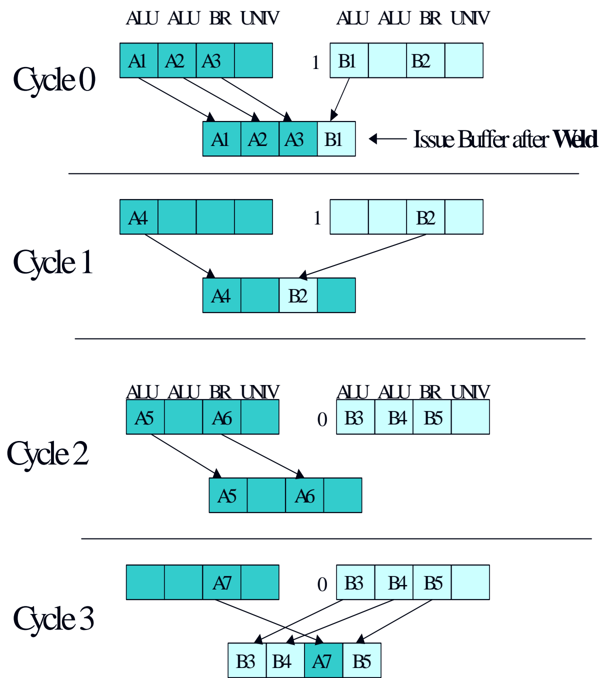


Figure 9. The execution steps of the sample program

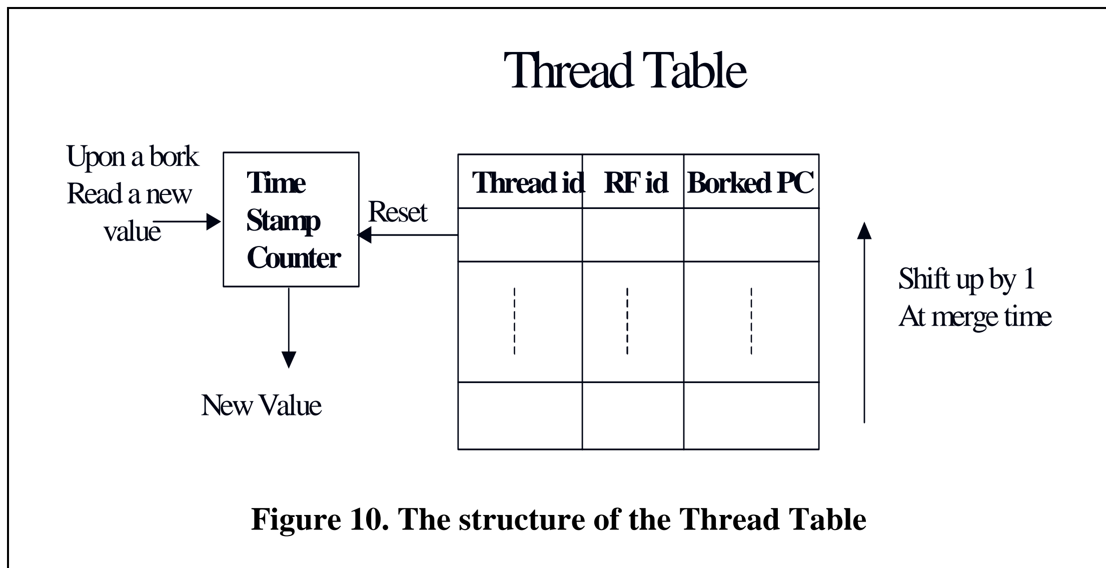


Figure 10. The structure of the Thread Table

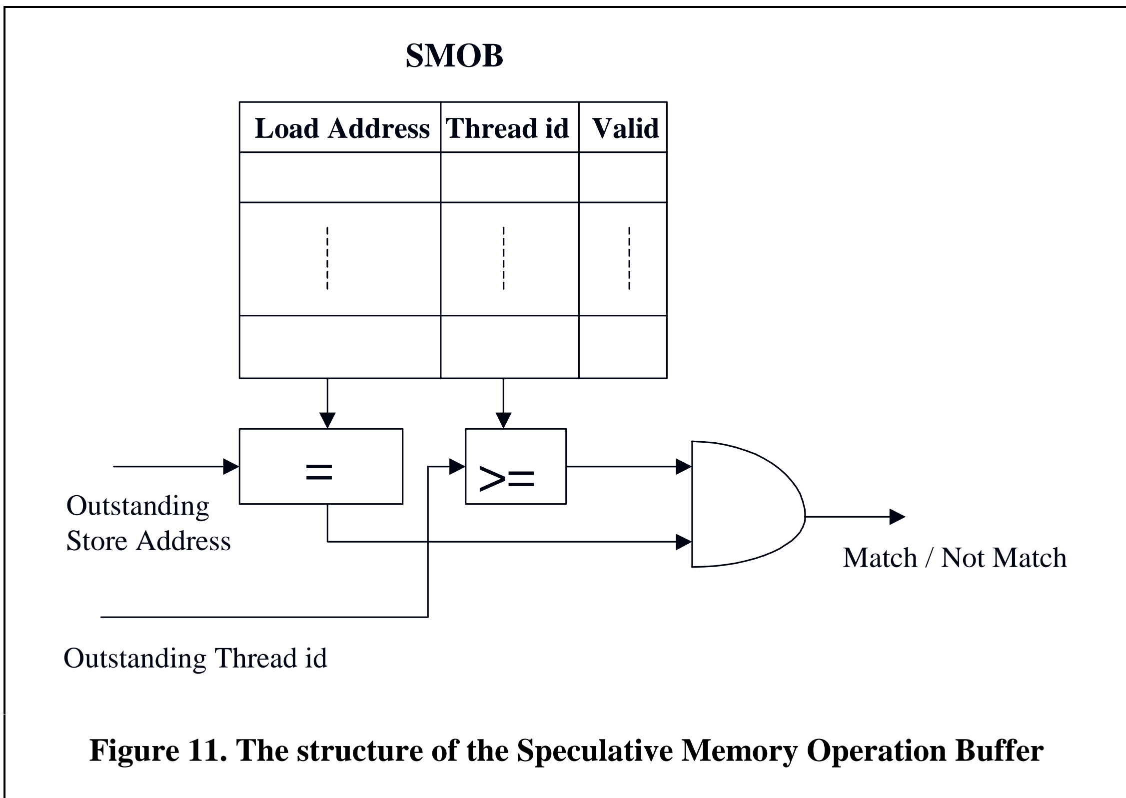


Figure 11. The structure of the Speculative Memory Operation Buffer

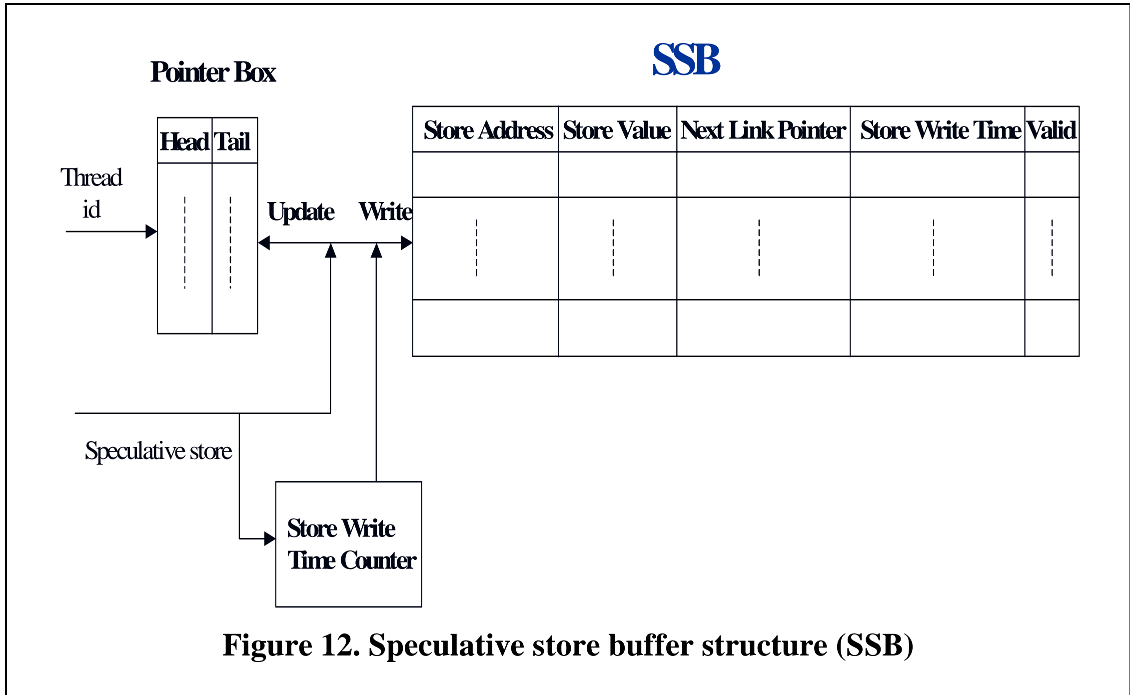


Figure 12. Speculative store buffer structure (SSB)

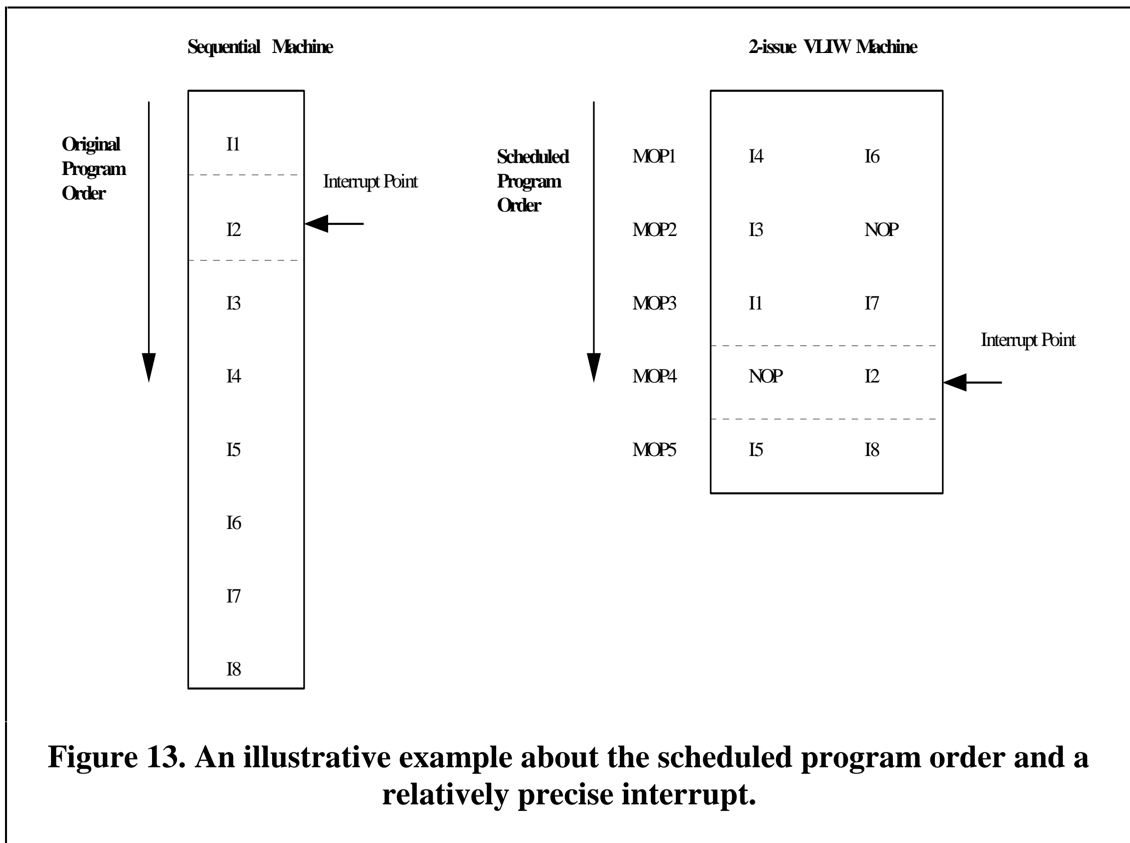


Figure 13. An illustrative example about the scheduled program order and a relatively precise interrupt.

3. Compiler Support for the Weld Architecture

The compiler support for the Weld architecture requires a separate compiler phase in the back-end compiler. The LEGO experimental compiler has region formation, schedule, register allocation phases. This is the order of phases we have used in this study. After forming regions, the code is scheduled for a specific machine model. Later, the physical registers are allocated for the code based on a procedure-based register allocation. An additional compiler stage named bork insertion is run through the register allocated code to insert bork operations in the code. A separate compiler stage is needed because the bork insertion cannot be performed at the same time with scheduling. The bork insertion requires all schedule times of the operations in some treegions under consideration be assigned. Therefore, the bork insertion is not merged into the schedule stage.

3.1. Bork Insertion Algorithm

The bork insertion algorithm is a compiler technique that determines the earliest schedule cycle to schedule a bork operation in the code to spawn a thread. The bork insertion algorithm is implemented within the LEGO Compiler[20]. The LEGO Compiler is an experimental compiler developed at North Carolina State University. It uses tree-shaped region (treegion) scheduling [21] as a global acyclic scheduler. Each node of a treegion is a basic block. Treegions are single entry, multiple exit regions and can be formed with or without profile information. The scheduler is capable of code motion and

instruction speculation over branches. Once treegions are selected, the scheduler schedules each treegion separately (i.e. no instruction migration between treegions) and therefore, treegions can have a high potential for being separate threads. After the program is scheduled, the register allocator is invoked and physical registers are assigned. Then, borks are inserted into the scheduled and register allocated code. Borks spawn speculative treegions and are inserted as early as possible in the code to fully overlap the treegions. True data dependencies between two treegions are considered when inserting borks. Load and store dependencies are resolved by the SMOB and therefore the compiler ignores these dependencies during the scheduling of borks. There is only one bork per path in each treegion, and each thread can spawn at most one speculative thread. Figure 14 shows the pseudo-code of the algorithm.

The algorithm scans through all treegions in the program. To consider the true data dependencies, it takes a treegion (*Tmain*) and computes the live-out set of operands from *Tmain* to each succeeding treegion, *Ts*. For each path in *Tmain*, if there are any register definitions of each live-out operand, the location and schedule time of those operands in that path are found. The completion times of all live-out operand definitions for the path are computed. The maximum completion time among all dependencies is found by taking the maximum of all completion times in the path. The earliest time to schedule a bork is determined by the maximum completion time. A schedule hole is sought to insert a bork into a cycle between the earliest cycle time and the last schedule time of the selected path in *Tmain*. The bork is inserted into the path in *Tmain*. The algorithm tries to insert a bork for every possible path in *Tmain* to *Ts*. When there is no path left, the next succeeding treegion is processed. This continues until all treegions in

the source file are visited. More than one bork can appear on a path since a bork is inserted for each path in a treegion after all paths are visited. Elimination is needed to reduce the number of borks to one for each path. To accomplish this, the earliest bork is kept and the later bork(s) in the schedule are removed. The elimination phase of redundant borks is performed after insertion of borks for each treegion. After elimination of borks, the first operation of the first MultiOp in the treegion Tmain is marked as the synchronization MultiOp by setting the synchronization bit.

Figure 15 shows an example of bork insertion algorithm. The figure shows the schedule of a piece of code before the insertion of borks. Treegion 1 enters into Treegion 2 with two exits. Each rectangle represents a basic block. Within a basic block, a row represents a MultiOp that contains two operations. Only the operations that are relevant are shown in the figures. X denotes an operation not under consideration and empty slots denote NOPs. Also, stime represents the scheduling time for each MultiOp. All operations are assumed to take one cycle and all functional units are universal. There are two paths entering into Treegion 2 from Treegion 1. On the left path, there is only one live-out (R1). The completion time of the operation that defines R1 is the sum of its stime and the operation latency, which is 4. Cycle 4 is the earliest cycle a possible bork can be scheduled on the left path. The algorithm starts searching from cycle 4 for an empty slot to insert a bork. An available slot is found at cycle 4 on slot 2. Then, a bork is scheduled in that slot. On the right path, R2 and R4 are the live-outs. The completion times of the operations that define R2 and R4 are computed similarly, which are 5 and 8 respectively. The maximum of 5 and 8 determines the earliest time for a bork, which is cycle 8. However, there is no cycle 8 on the right path. So, no bork is scheduled on this path. Next

step is to eliminate the redundant borks in the paths. Since there is no redundant borks in either path, no bork is eliminated. The final code is shown in Figure 16.

3.2. Complexity Analysis

The complexity of the algorithm is O (N) = N. N_s . E . K where N is the number of treegions, Ns is the number of succeeding treegions, E is the number of exits from a treegion and K is the number of cycles between the earliest and the last schedule time of a path. The worst-case complexity of the algorithm is O (N) = N^4 where N_s, E and K converges to N. However, the practical or empirical complexity of the algorithm ought to be a lot less than that assuming that N_s, E and K are a lot less than N.

The practical complexity can be computed in terms of N, the number of treegions. A counter is inserted in the innermost loop of the algorithm, which is the loop that goes from the earliest schedule time to the latest schedule time. For each procedure, a pair of the number of treegions in the procedure and its complexity counter is gathered in a benchmark. A polynomial function is fitted by interpolating these pairs in each benchmark program using the least squares method. The best polynomial function that can fit on these pairs is chosen using Matlab. The complexity polynomial functions are shown in Table 2. As seen in the table, all benchmarks have a complexity in the first order, which is O (N) = N. The practical complexity of the algorithm is proved to be a lot less than its theoretical one.

3.3. Performance Evaluation of the General Weld

The experimental framework used in this work is shown in Figure 17. A program in C is fed into the front-end compiler. The front-end compiler consists of an IMPACT compiler that generates an intermediate code called Lcode. ELCOR compiler from HP Labs. reads Lcode and generates another intermediate code called Rebel. LEGO compiler that is a back-end compiler reads Rebel code and performs optimizations and spits out optimized Rebel code as an output. Yula takes the optimized Rebel code and converts it back to C code. The optimized C code is compiled using gcc and an executable is generated. When the executable is run, a dynamic trace is produced and the simulator consumes this trace on the fly and simulates the Weld architecture. A simulator was written to simulate multiple threads running at the same time. The simulator reads instructions on the fly generated by running the program and simulates them. The simulator has 2-way set-associative data and instruction caches and 2-way set-associative second-level shared data and instruction cache. No hardware prefetching or a victim cache is used in all caches. The least recently used (LRU) policy is used for replacements of cache blocks. A shared 4-way 16KB pre-address scheme (PAS) branch predictor is used. Each history register holds 10 bits that corresponds to 10 branches. The parameters in the simulator can be found in Table 3. Since a trace-driven simulation has been used, only the correct path of the code can be accessed. For misspeculated threads, a constant penalty – for squashing threads and flushing them from the pipelines-- is added to the total execution time. The machine model used for the experiments is a 6-wide VLIW processor with 2 universal and 4 ALU/BR units, 128 integer and 128 floating-point registers. Floating-point operations consume and produce values in the floating-point register file. In the simulator, the floating-point operations are executed only in the

universal units. The register allocator allocates floating-point registers into the floating-point registers and adds a type (i.e. integer, floating-point, predicate etc.) for each register. The simulator considers these types for the registers during the simulation of the programs. Universal units can execute any type of instructions and ALU/BR units can execute only ALU and branch instructions. SPECInt95 benchmark suite is used for all runs. 100 million instructions were executed from training inputs (except 10 million instructions for gcc). The latencies of the operations used in the simulations are shown in Table 4. We assume that L1Icache and L1Dcache have multiple ports to read and write simultaneously (i.e. it is assumed that a thread has dedicated ports both for data and instruction caches). In the experiments, only the number of useful operations is considered. The incorrectly speculated operations that have been speculated by the compiler are not taken into consideration when calculating the actual IPCs. Multiple thread runs are compared to the baseline model with a single thread run of the same benchmark program. The same compiler optimizations are applied to produce codes for single thread and multiple threads models in the experiments.

Figure 18 shows speedup results for a model with Icache, L2Icache, Dcache, branch predictor, 128-entry SMOB and 64-entry SSB. As shown in the graph, the maximum speedup of 22% and 25% is attained with two and three threads over all benchmarks, respectively. With four threads and after, it is 26%. There is no significant speedup observed when the number of threads increases after four threads. This is because the amount of penalty cycles increases with the number of threads such as the SMOB and SSB fill up quickly and stall the speculative threads, the chances for the SMOB conflicts occur more often as the architecture keeps more active threads. This

effect can be observed in 129.compress, 130.li, 147.vortex, 126.gcc and 124.m88ksim. In 132.ijpeg and 134.perl, the same effect can be noticed. However, the overhead cycles are more dominating than other benchmarks. Therefore, the speedup drops in 132.ijpeg after two threads and in 134.perl after five threads. This is because the number of SMOB conflicts in two-thread model in jpeg and five-thread model in perl is much less than the number of SMOB conflicts in the models with higher number of threads in both benchmarks. Figure 19 depicts the IPC results of Weld with the base IPC results. Essentially, the IPC results represent the same sort of comparison with the percentage speedup results. The base IPC is 1.3 on the harmonic mean of all programs. The IPCs are 1.6 with two threads and stays around 1.7 after two threads with small increments.

Figure 20 depicts what percentage of the total issued MultiOps are welded with multiple threads. On the average, 25% of the total issued VLIWs come from welding with two threads and 30% with three threads and after. Figure 21 shows the percentage of the total issued MultiOps that come from a single thread, i.e. a vertical thread. The results are complementary of the welded percentages, which are 65% for two threads and 70% three threads and after. This result tells us that most of multithreading gain comes from vertical multithreading. The biggest factor for less number of welded VLIWs is the number of inseparable MultiOps in speculative threads. Figure 22 shows the direct contribution of welding to the performance. The figure compares the speedup of the Weld architecture using the operation welder with the same architecture model without using the operation welder. On the average, the welding improves the performance by 5% in two threads and 7% in three threads. The speedup stays in 8% in four threads and after. The most significant observation can be done in 132.ijpeg. It seems that welding slows

down the processor, however it is not true at all. The slowdown is caused by the number of SMOB conflict squashes as projected in the general Weld performance results discussion. As a conclusion, the high number of SMOB conflict squashes cancels out the advantages of using the operation welder in 132.ijpeg.

Table 5 shows the instruction cache miss ratios from one thread to six threads. The last row in the table gives the harmonic mean of the miss rates. As the number of threads increases, the instruction cache miss rate also improves. This is because the speculative threads that are squashed due to a SMOB conflict work like a prefetch engine for the main thread. The main thread re-fetches these MultiOps that have been brought into the cache by the speculative threads. Note that in case of a thread squash, the instruction cache is not flushed. The improvement can be observed for each benchmark and the harmonic mean of all programs. Figure 24 shows the data cache miss ratios. With similar reasoning as in the instruction cache miss ratio, the data cache misses decrease as the number of threads increase. The speculative threads behave like prefetch routines and help improve the data cache miss rate. Figure 23 shows the L2cache miss ratios. Increasing the number of threads does not have much effect on the miss ratio for the harmonic mean of the benchmarks. This can be observed in 129.compress, 130.li, 132.ijpeg and 099.go. In Figure 25, the branch prediction accuracy results are depicted for 16KB branch predictor using per address scheme (PAS). As the number of threads increases, the branch prediction accuracy drops slightly like 0.5% over the average of the benchmarks.

Figure 26 illustrates the dynamic code size increase due to annotated bork instructions. The number of dynamically executed bork operations is counted. The bork

operations that never execute in other paths are not included as part of the code increase. The increase in code size on average is 3.8% with two threads and 4.3% with six threads on the average, which are quite negligible numbers.

3.4. Concluding Remarks

As seen from the experimental results, 3-thread model is the best model in terms of performance. Supporting more than three threads is not a cost-effective solution. On the other hand, 2-thread model has equally sufficient performance with 2% less in speedup than the 3-thread model. However, the design complexity and simplicity in terms of hardware used are more feasible in 2-thread model than 3-thread model considering duplicating register files and fetch units three times and complicated associative comparison in the SMOB and SSB units. 2-thread model may have a simple thread synchronization mechanism and simple associative comparison logic in the SMOB and SSB. As a consequence, 2-thread model or dual-thread model will be explored more for further research.

```
foreach Treegion Tmain begin
  foreach succeeding Treegion Ts begin
   FindLiveOprds(LiveOprds[], Tmain)
    foreach path in Tmain begin     //starting from the entry basic block of Tmain to its exit
     into Ts
       DefLiveSet[] = Definitions of LiveOprds[] in path
       MaxCompletionTime = The maximum completion time of operations in DefLiveSet[]
       EarliestCycle = MaxCompletionTime
       for EarliestCycle to Last Schedule Time of path begin
            Find an available hole to schedule a BORK
       end
       if (a hole is found) begin
            Insert a BORK operation into this path
       end
       else Do not insert BORK on this path
    end
  end
end
DeleteRedundantBORKS per path in Tmain
```

Figure 14. Bork insertion algorithm

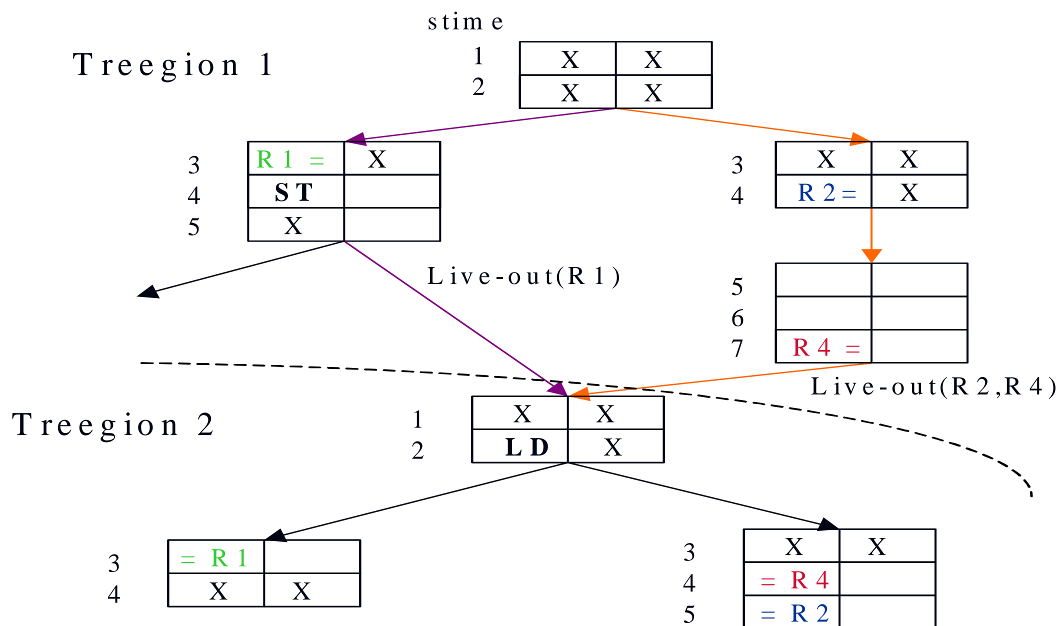


Figure 15. An example for the bork insertion algorithm

Figure 16. The final code after inserting the bork operation

Table 2. Practical Complexity Polynomial Functions

Benchmark	**Complexity**
129.compress	3.64N
130.li	5.22N
132.ijpeg	5.16N
134.perl	49.66N
147.vortex	16.08N
099.go	9.01N
126.gcc	20.85N
124.m88ksim	34.07N

Table 3. The properties of the execution-driven simulation environment

Simulator Properties
2-way set associative 32KB L1 instruction and 32KB data caches
512KB 2 way set associative L2 instruction and data cache
16KB shared PAS branch predictor
Variable Speculative Memory Operation Buffer (SMOB) Variable Speculative Store Buffer (SSB)
1-cycle L1 cache hit time, 10-cycle L2 hit time
Variable-cycle (30, 90) L2 miss time
Variable-cycle (5,10,15) branch penalty, SMOB squash penalty time and misspeculated bork squash time

Table 4. The latencies of instructions

Instruction	Latency
ALU	1 cycle
FP Add	1 cycle
FP Mul & Div	3 cycles
LD	2 cycles
ST	1 cycle
BR	1 cycle
Bork	1 cycle

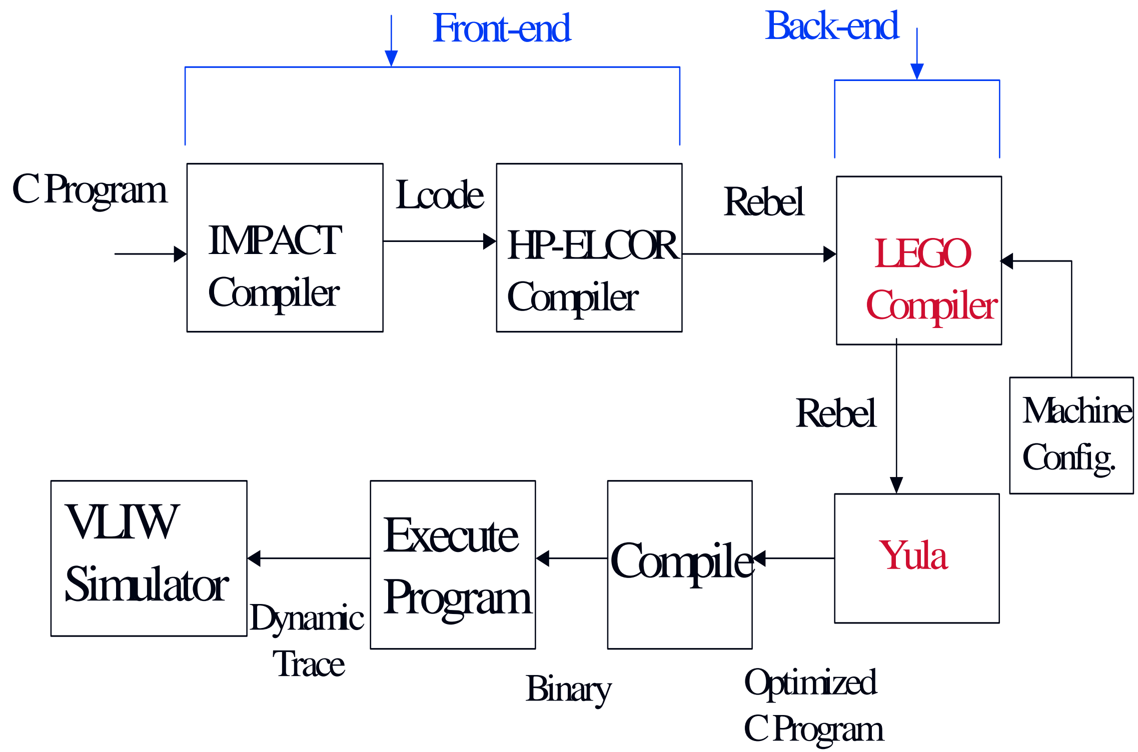


Figure 17. Experimental framework

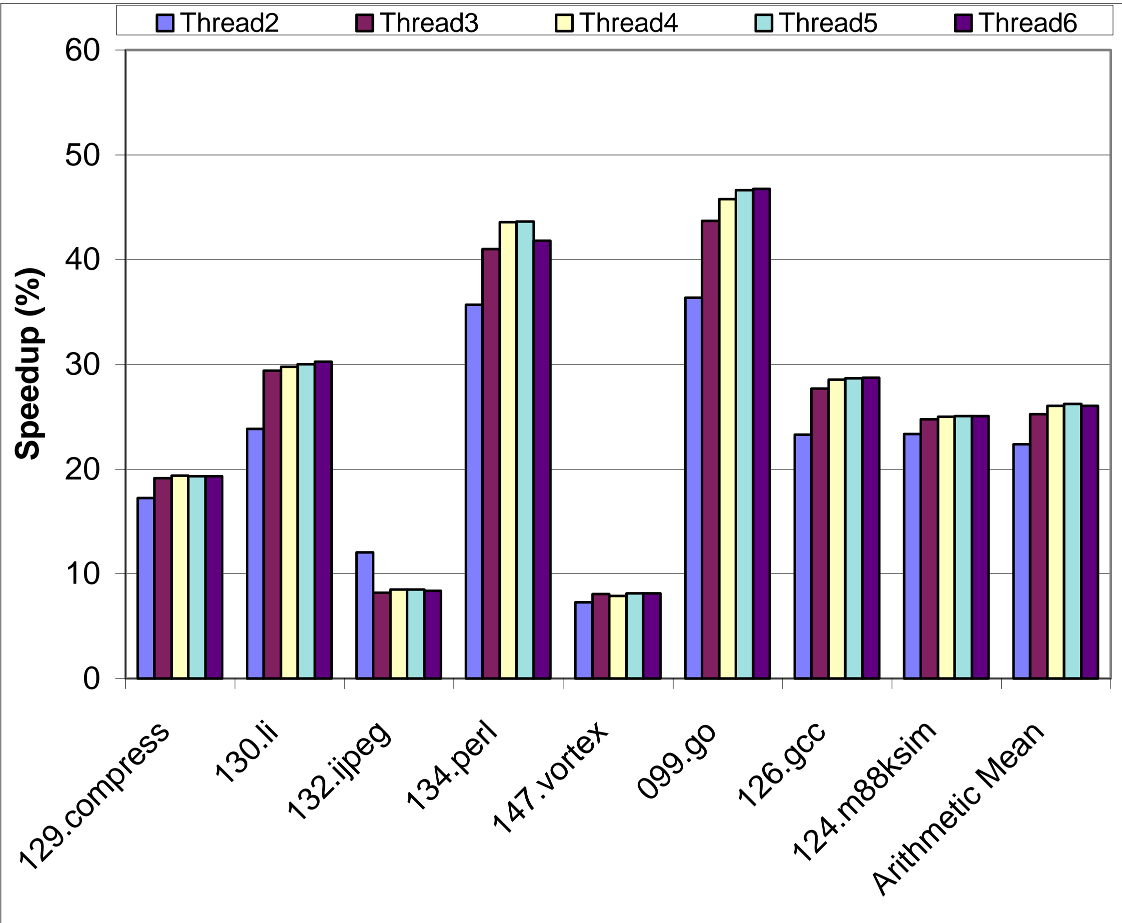


Figure 18. Speedup results

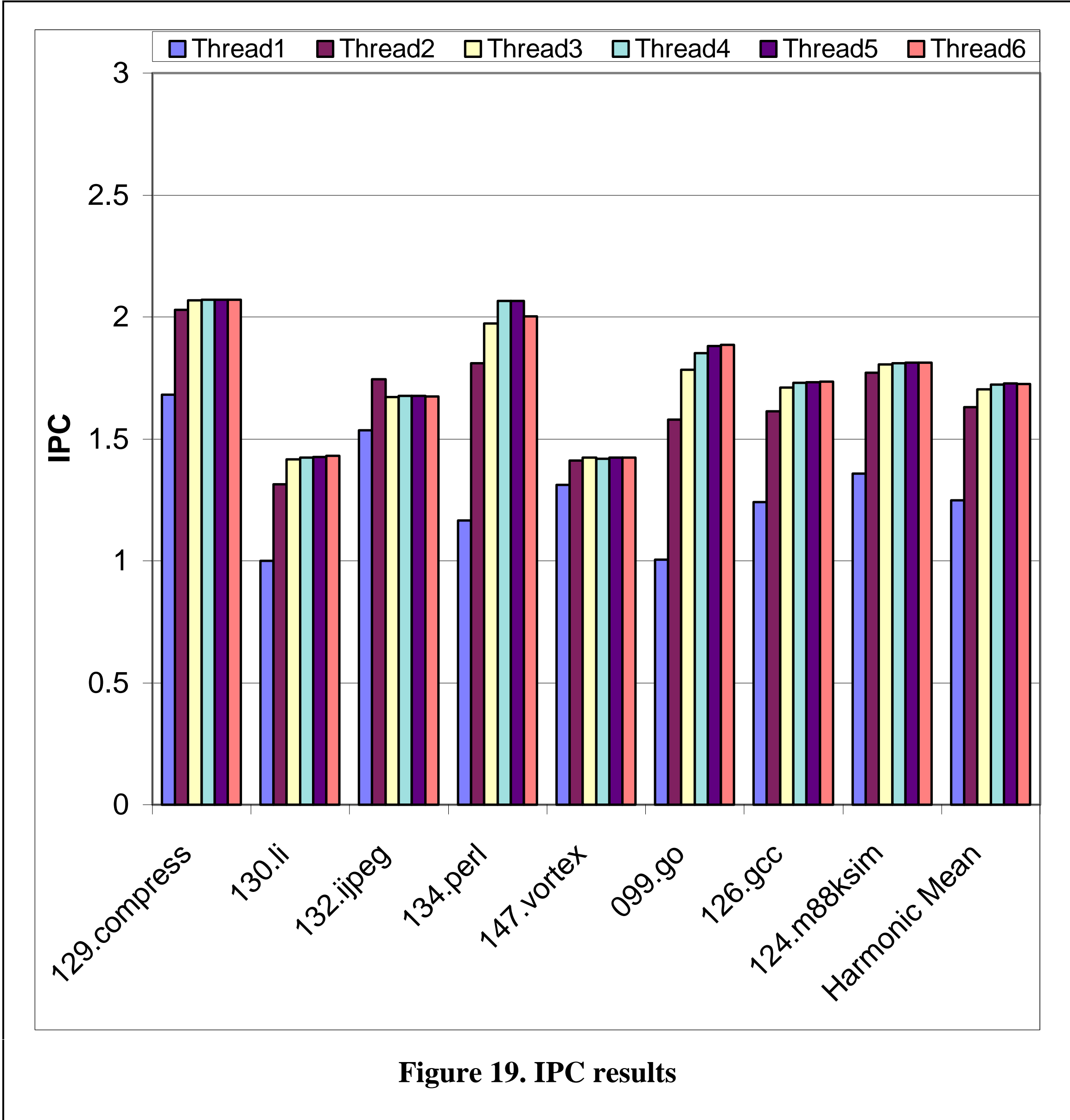


Figure 19. IPC results

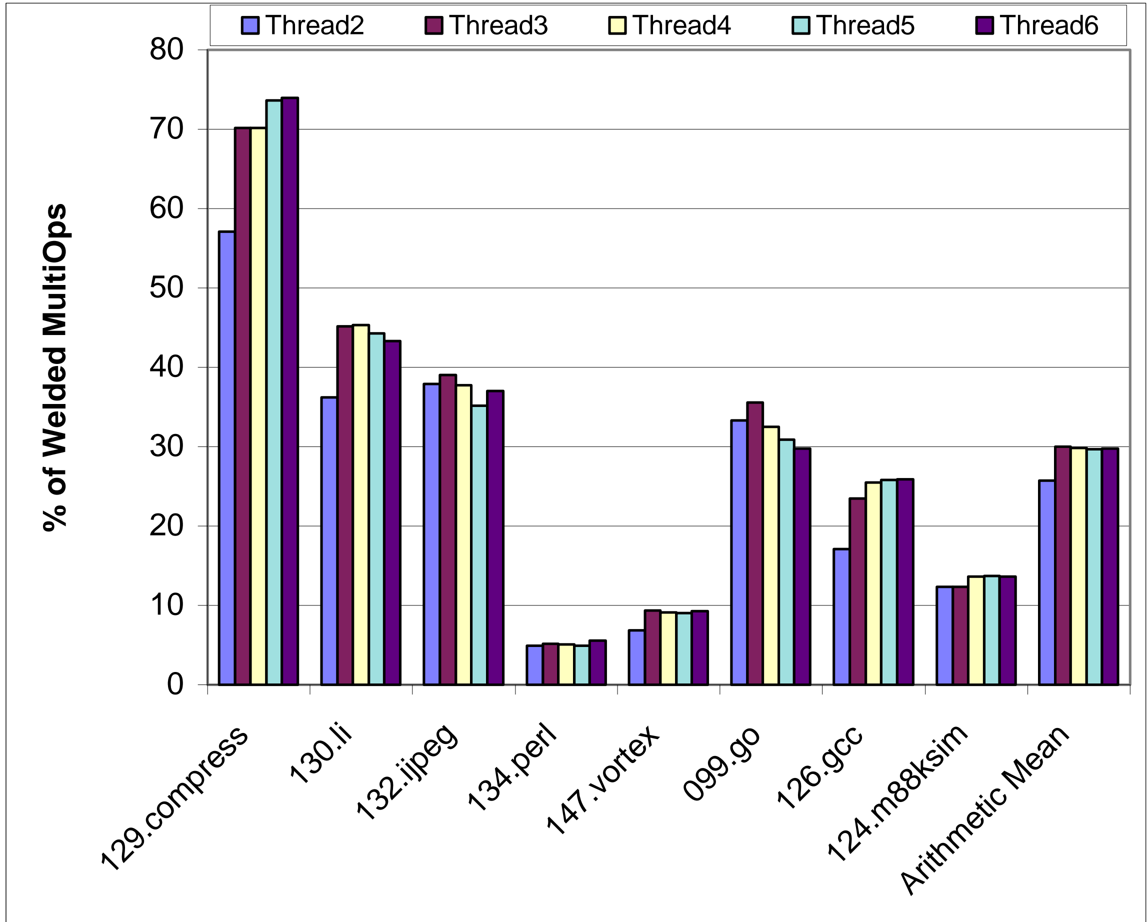


Figure 20. Percentage of Welded MultiOps

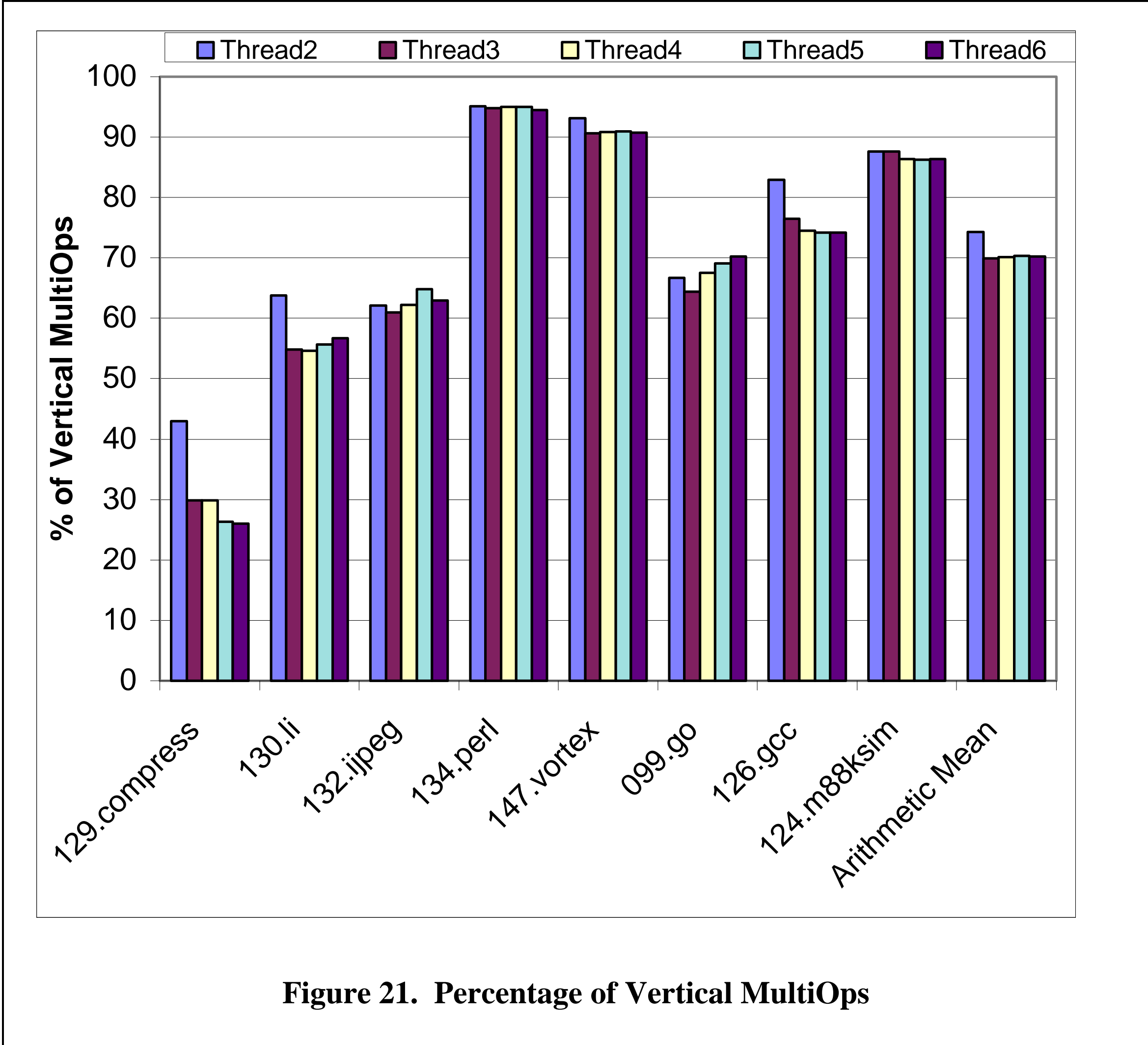


Figure 21. Percentage of Vertical MultiOps

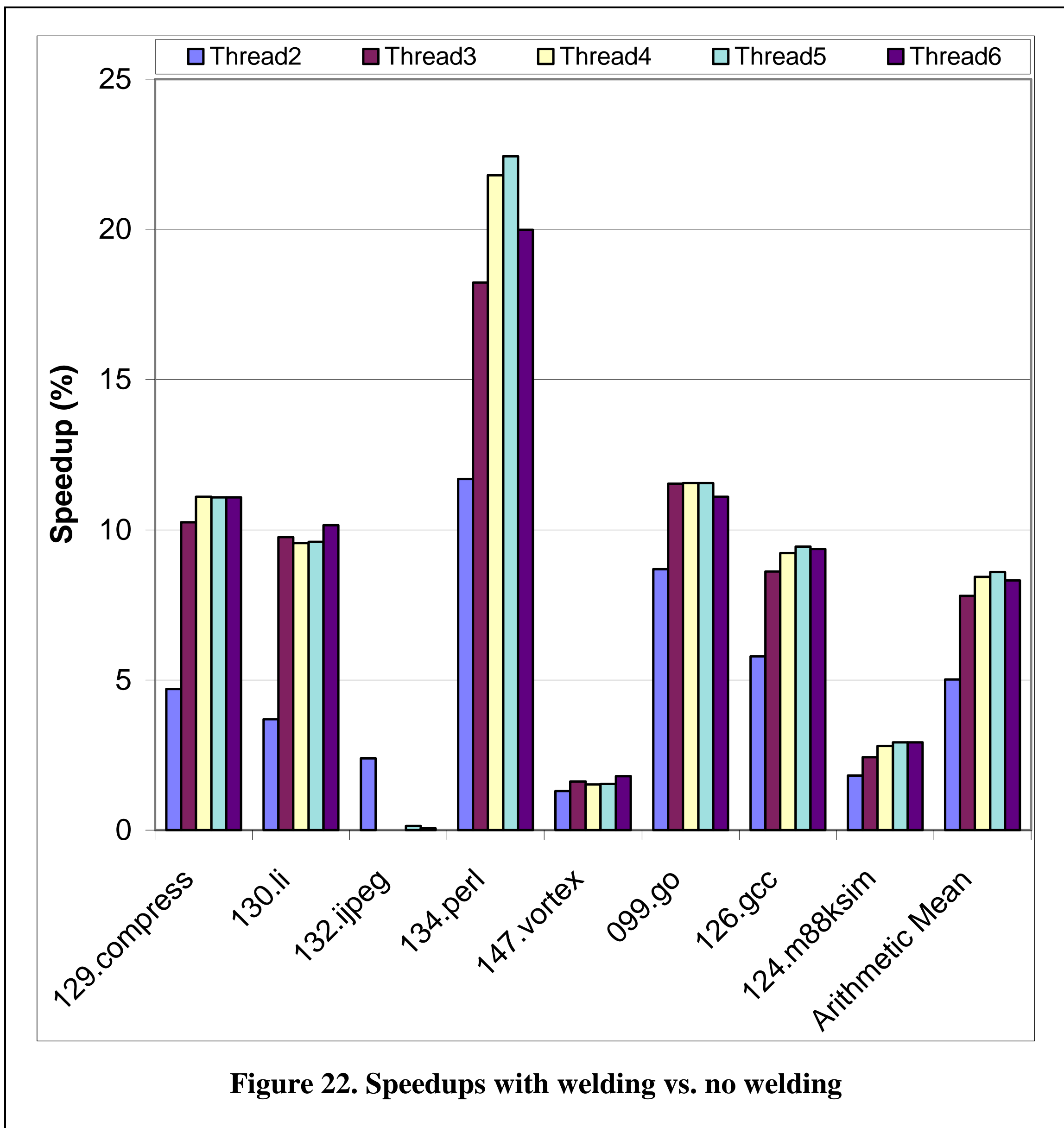


Figure 22. Speedups with welding vs. no welding

Table 5. Instruction cache miss rates

Benchmark	Thread1	Thread2	Thread3	Thread4	Thread5	Thread6
129.compress	0.002527	0.002094	0.001889	0.001841	0.001837	0.001836
130.li	0.001993	0.001703	0.001563	0.001461	0.001415	0.001405
132.ijpeg	0.004938	0.004675	0.004181	0.004011	0.003996	0.00399
134.perl	0.009385	0.009228	0.007962	0.008619	0.008618	0.007704
147.vortex	0.050772	0.03875	0.036181	0.03555	0.035576	0.035562
099.go	1.441241	1.335487	1.265958	1.231752	1.216377	1.211244
126.gcc	1.295577	1.090344	1.016939	0.991653	0.982973	0.980461
124.m88ksim	0.00239	0.00227	0.002208	0.002185	0.002179	0.002177
Harmonic Mean	**0.00486**	**0.004313**	**0.003968**	**0.003845**	**0.003798**	**0.003763**

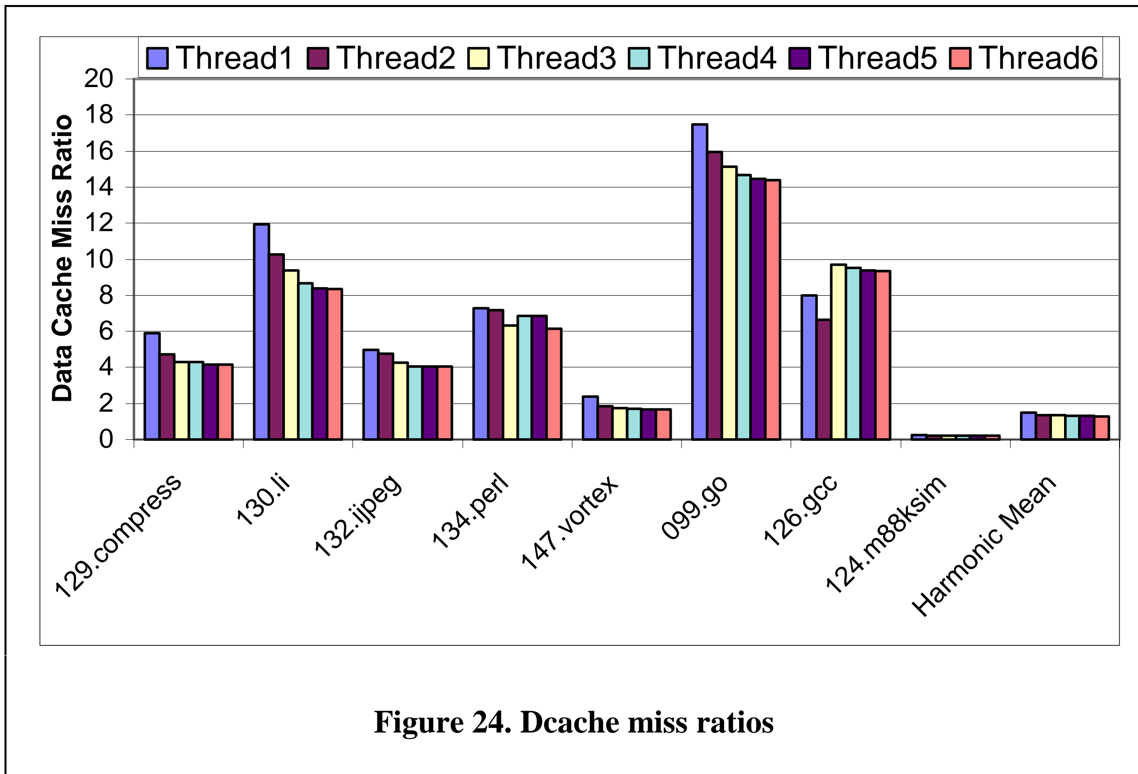


Figure 24. Dcache miss ratios

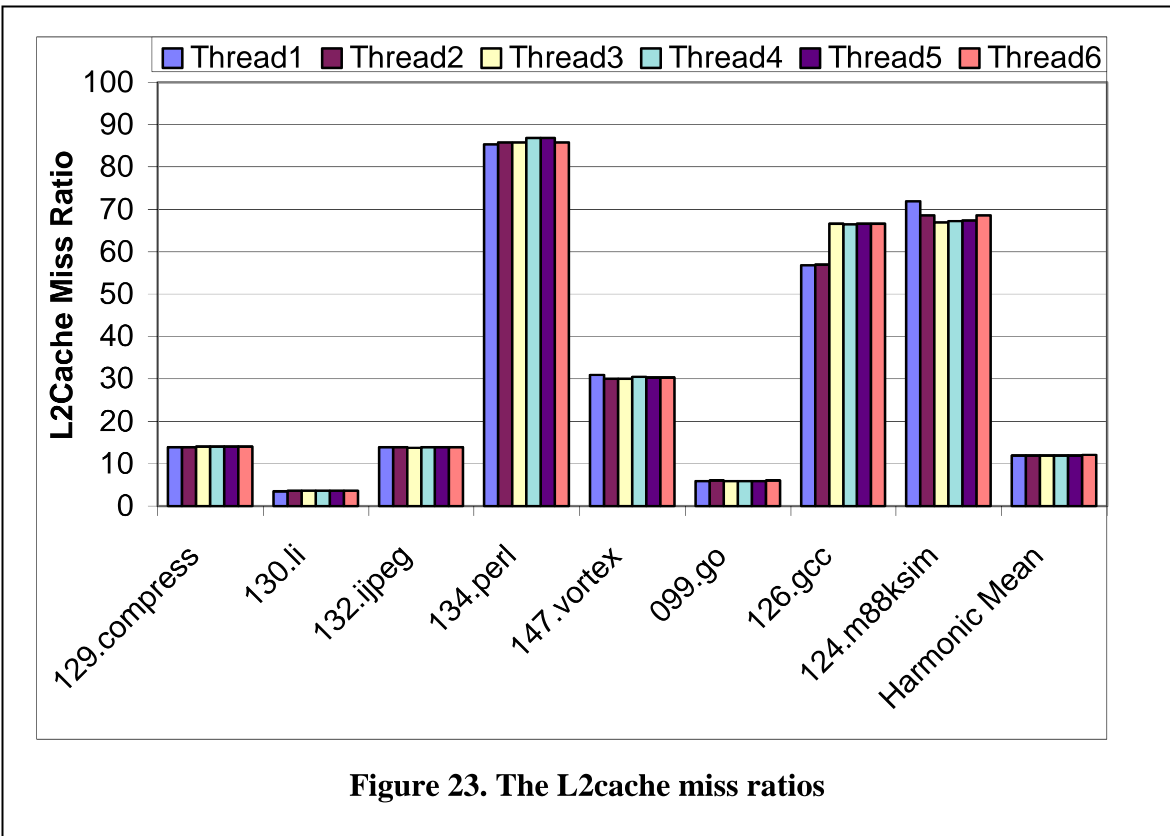


Figure 23. The L2cache miss ratios

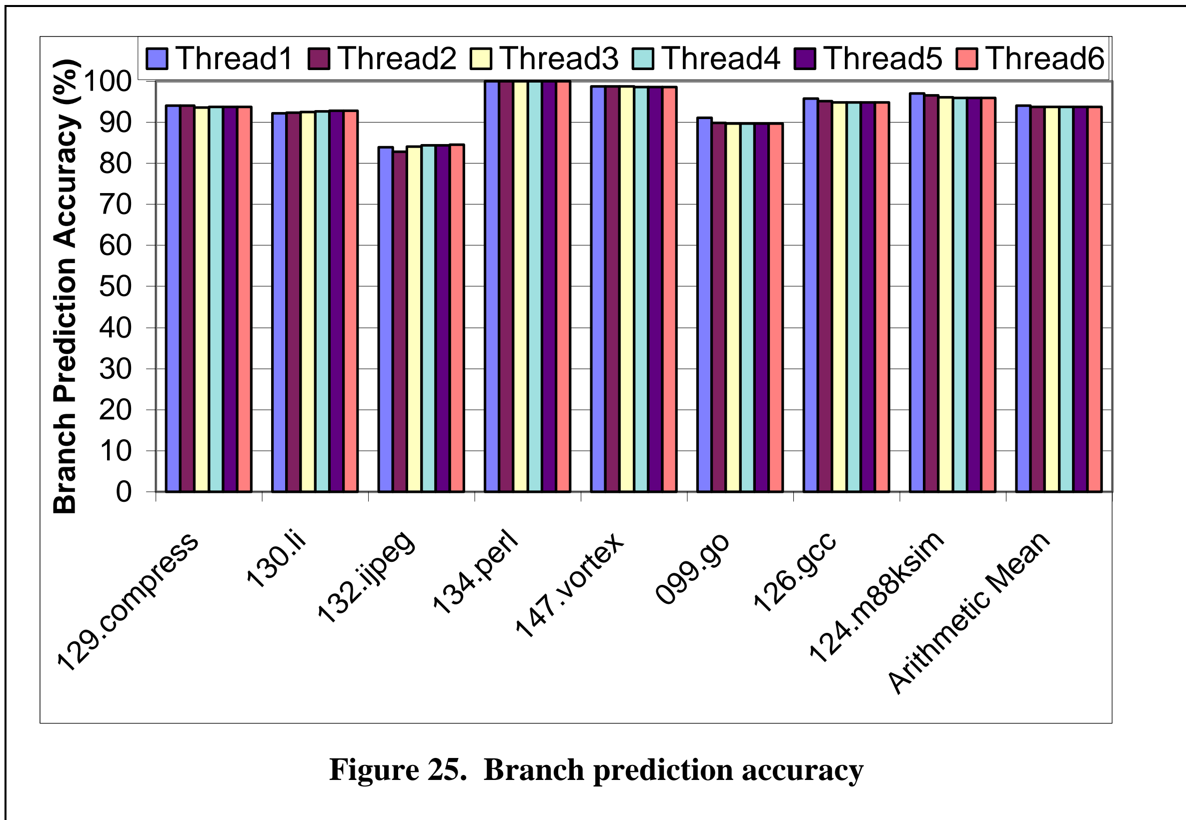


Figure 25. Branch prediction accuracy

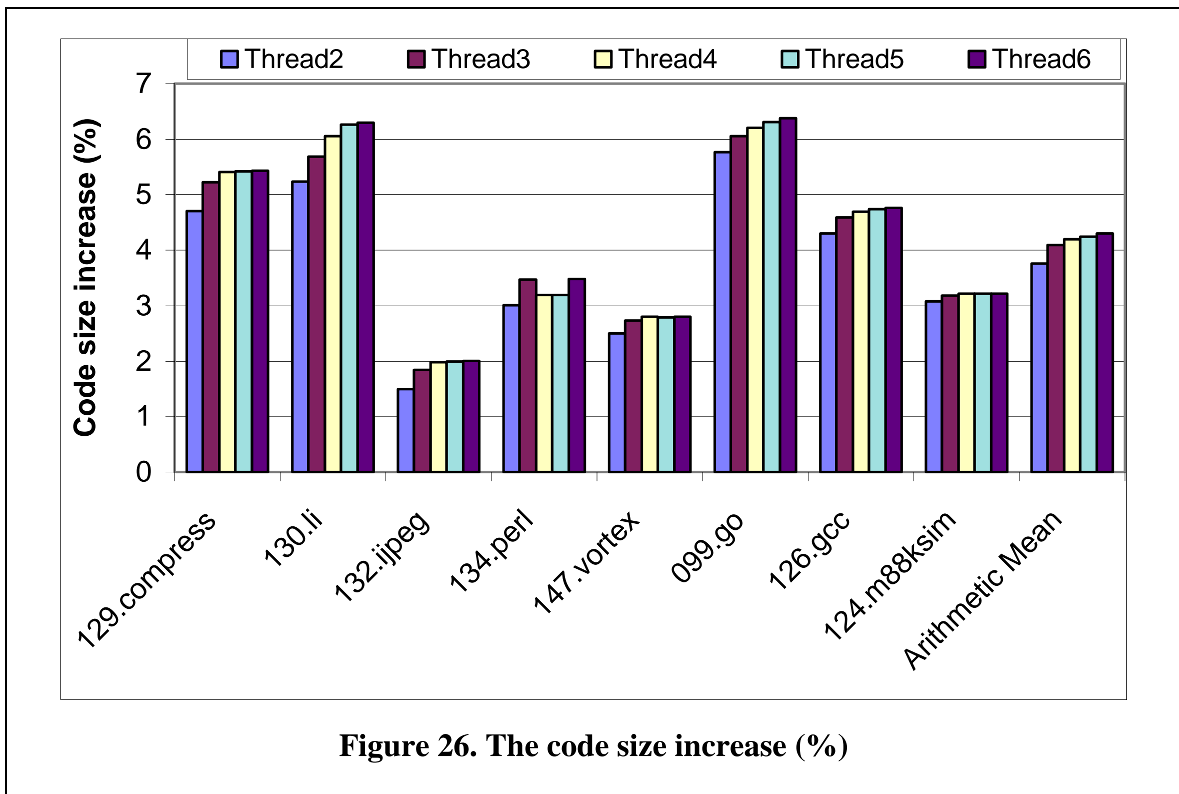


Figure 26. The code size increase (%)

4. Dual-thread Weld Model

This chapter covers the two-thread or the dual-thread model in more detail. It analyzes the performance change by varying several factors that can affect the architectural implementation. First, a variant of the general Weld architecture model is used in the dual-thread architecture model. It is less complex and more efficient than the general Weld in terms of hardware.

4.1. The Dual-thread Weld Architecture

In dual-thread Weld model, there is a main thread and a speculative thread. The main thread spawns the speculative thread and merges with it. The architecture is shown in Figure 27. Each thread has its own program counter, fetch unit and a register file while all threads share the branch predictor, instruction and data caches. The Icache has two read ports, one for each thread. The fetch stage fetches MultiOps[22] (VLIWs) from the Icache, and the weld/decode stage merges two MultiOps and decodes them. The weld in the weld/decode stage consists of an array of multiplexers that can forward any operation to any functional unit as long as it is the correct resource to execute the operation. It fills the schedule slots in the main thread that cannot be filled at compile time from the speculative thread at run time. The weld is designed in such as a way that the main thread's MultiOp has priority over the speculative thread's MultiOp, thus speculation never delays forward progress. The operations from the main MultiOp are always copied to the issue buffer. The welder inserts operations from the speculative MultiOp into the holes in the issue buffer left by the main MultiOp. The operand read stage reads operands

into the buffer for each thread, and sends them to the functional units. The execute stage executes operations and finally the write-back stage writes the results into the register file and Dcache.

There are two register files one for each thread in the dual-thread Weld architecture. Two register files can be designed such that a fast copy of all registers from one to another can be completed in one cycle. The copy is done at the time when a bork executes. There is no register transfer needed from the main thread to the speculative one after the thread creation point. The compiler guarantees that a speculative thread be created after all live-out operations in the main thread complete and write their results into its own register file. A possible dual-register file design can have duplicated and cross-connected memory bit cells. Each memory bit cell corresponds to a bit from each register file. Figure 28 shows a word consisting of two registers that is one from each register file. *A* and *B* represent the register file *A* and *B*, respectively and the subscript denotes the bit number. Two extra lines: Copy A-B and Copy B-A are added to control the direction of data bit copy. The data bit transfer is done between the read data bit line of one register file to the write data bit line of the other one through a pass transistor used as a switch. This kind of register file layout allows a faster copying of one register file to another. A similar design strategy can also be observed in Checkpoint Repair [73] technique. It is a recovery mechanism from branch predictions and exceptions. The processor state is saved at appropriate points or checkpoints. When a branch misprediction or an exception occurred, the processor is repaired by restoring the previously saved state. To restore register file, the back up register file cells are placed next to the main register file cells to maintain the fast mass copy.

When a bork instruction is executed at run time, the target address of the bork instruction is saved in the borked address field of Main Thread Register (MTR) that is a special register, the register file of the main threads dumps its contents to the other register file and sets the PC of the speculative thread to the borked address. MTR has a 1-bit register file identifier (RF Bit) and the borked PC address if it spawned a speculative thread. Since there are only two register files in the model, a 1-bit is sufficient to represent register file (0 denotes the register file A and 1 denotes the register file B). Only the main thread information is kept in the buffer because it is fairly easy to reach to information about a possible speculative thread. By toggling the register bit field in MTR, the register file for the speculative thread is known. In the dual-thread model, each operation is attached 1-bit register file number to route them to the correct register file. This is done by reading the MTR for both threads as explained above.

Thread merge is detected with the help of a synchronization bit added to each MultiOp in the ISA by the compiler. This bit is set in the first MultiOp of each thread at compile time. When a MultiOp (either from Fetch A or Fetch B) with the synchronization bit set is fetched from the cache, the synchronization bit detector in Figure 27 detects a possible thread merge point. Then, the borked PC field in MTR is compared with the PC value of the MultiOp (either from thread A or B) just fetched. If the addresses match, the speculative thread is correctly speculated. The main thread dies and the speculative thread becomes the main thread. This is achieved by flipping the register file bit and clearing the contents of the borked PC field in MTR. If the addresses do not match, the speculative thread is misspeculated and must be squashed. Also, the Borked PC field is cleared in MTR. Figure 29 shows an example of thread creation and synchronization in

the dual-thread Weld architecture. The example is prepared for 2-issue VLIW/EPIC processor. Thread 1 is the main thread and it has a bork operation in the second MultiOp. As soon as this bork in the MultiOp is executed, the processor creates Thread 2 (i.e. speculative thread) at address 100 by copying the register file A into the register file B. At the same time, it writes 100 into the PC B and the Borked PC field of the MTR. If the actual execution path is through Thread 2, the PC address (i.e. 100) of the first MultiOp detected by 1 in its synchronization bit is compared with the Borked PC in MTR, which is also 100. A match will tell the processor that Thread 2 is correctly speculated. In this case, Thread 1 dies and Thread 2 becomes the main thread by setting the RF bit in the MTR. The new main thread uses the register file B. The Borked PC field (shown as X in the figure) is cleared for the new speculative threads and the register file A is made available. On the other hand, if the actual execution path is not through Thread 2 as shown in Figure 30. Thread 2 is misspeculated and must be squashed. After squashing Thread 2's operations from the pipelines, Thread 1 remains as the main thread. However, the Borked PC field in the MTR is cleared for the new speculative threads. The RF bit remains the same because the main thread did not change.

Load operations' addresses from the speculative thread are saved in the SMOB until the merge time. A speculative load is always executed and the address of the load is kept in the SMOB. The SMOB is a fully associative buffer and contains two fields for each entry: a valid bit and the load address. The store addresses from the main thread always check the SMOB for a possible match by comparing the store address with the load addresses in the SMOB. If there is a match, a hazardous situation occurred because a speculative load completed before a store at the same address. In this case, the

speculative thread must be squashed from the processor. If there is no match until thread merge time, no hazardous situation occurred and the SMOB entries are cleared. In a similar way, speculative stores are from the speculative thread executed but are not allowed to modify the data cache. Instead, they are written into SSB. Each entry in the SSB contains a valid bit, store address and store value of a speculative store operation. The structure of the SSB is in FIFO style and not complex because there can be at most one active speculative thread. In case of a thread commit (i.e. a correct speculation), the speculative stores are written into the data cache in FIFO order. In case of a thread squash – this can be triggered from either the MTR or the SMOB -- all SSB entries are invalidated by flipping over the valid bits in each entry. The SMOB and SSB for the dual-thread Weld are shown in Figure 31.

4.2. Speedup Results of the Dual-thread Weld

In this section, the performance results of the dual-thread Weld model with 128-entry SMOB and 64-entry SSB are shown. The number of the SMOB and SSB entries is chosen randomly. However, they will be varied to measure the sensitivity of the number of entries on performance further in this section. Figure 32 shows the speedup results of the dual-thread Weld with 128-entry SMOB and 64-entry SSB compared to the base model with a single thread. The average speedup is 22% over all benchmarks in the last column. The highest speedups are obtained in 134.perl and 099.go. This is because not only they have high potential for treegion overlapping but also they have the least number of SMOB conflicts among the other benchmarks. On the other hand, 147.vortex has the

lowest performance improvement. Although 147.vortex has a decent treegion overlapping, it has a large number of SMOB conflicts. A high squash penalty degrades what is gained by the parallelism. Figure 33 depicts the IPC results of the dual-thread Weld with the base IPC results. Essentially, the IPC results represent the same sort of comparison with the percentage speedup results. From this on, only the percentage of the speedup results will be shown in this work.

4.3. Variance of the Sizes of the SMOB and SSB

The number of SMOB entries is chosen as 128 in the previous runs. However, this random choice should be justified by varying the number of entries. The number of SMOB entries is changed from 64 to 1024 entries by keeping a fixed 64-entry SSB as shown in Figure 34. As seen from the graph, the speedup stabilizes at 256 entries. After 256, no major change is observed. The mean speedup is now 23% with 256-entry SMOB and 64-entry SSB. The number of stalls due to the unavailability of SMOB entries stays the same after 256 entries, therefore increasing the number of SMOB entries beyond 256 performs equally well with 256 entries. The IPC results are shown in Figure 35.

A similar study is performed for the SSB entries. By varying the number of SSB entries from 32 to 512 entries, the best number of SSB entries is determined. In this case, the number of SMOB entries is fixed at 256 since this is the best value for the SMOB. Figure 36 shows the speedup results of variable number of SSB entries. As shown in the graph, the speedup does not increase beyond 128 entries. This is the best value that should be chosen as the number of SSB entries. The mean speedup is now 23.4% with 256-entry SMOB and 128-entry SSB. For similar reasons explained in the results with

varying SMOB entries, the number of stalls due to unavailability of the SSB does not change after 128 entries. The IPC results are shown in Figure 37.

4.4. Maximum Treegion Overlapping

In this section, a limit study is done as to overlapping treegions or threads in the earliest time possible. The time that a speculative treegion can be spawned is limited by the true register dependencies as explained in the bork insertion algorithm. If these true dependencies can be broken by using value speculation, it would be possible to insert a bork operation in the earliest cycle for each path. The earliest cycle for each path will be in the root basic block.

For this study, the compiler inserts bork operations in the code assuming that there is a perfect value predictor for the operations that have live-outs in the main treegion. The speculative treegion is assumed to read these register values from the perfect predictor. In this way, two treegions can be overlapped in the earliest possible cycle. The algorithm used in the compiler is shown in Figure 38. A search for a schedule slot in a path starts from the first cycle because all live-outs are assumed to be value-speculated perfectly. In this way, two treegions can be overlapped in the earliest possible point at run-time. The run-time simulation results will tell us if the maximum overlapping of treegions can really help improve the speedup significantly. The speedup results for the dual-thread Weld with 256-entry SMOB, 128-entry SSB with a perfect value prediction of register live-outs is compared to the same model without using any value prediction. The results are depicted in Figure 39. From the graph, we can see that the average speedup goes up by 1.3% with perfect value prediction. The earliest issuance of the speculative thread causes the SMOB conflicts to occur more often. This will add

more penalty cycles for squashing and throwing away the some useful MultiOps executed by the speculative thread. This is the main reason why a great performance improvement is not observed using perfect value prediction in the dual-thread Weld architecture model. Even in some benchmarks such as 129.compress, 134.perl, 099.go and 126.gcc, the performance with perfect value prediction is a little worse than the one without perfect value prediction. The IPC results are also shown in Figure 40.

The performance improvement would be really small when a realistic value predictor is used because it would incur some thread squashes due to value mispredictions.

4.5. Concluding Remarks

This chapter presented a dual-thread version of the general Weld architecture model. The dual-thread model uses less hardware as complied with the philosophy of the horizontal architectures. The experimental results showed that a speedup of close to 24% is possible. The speedup increase is limited by the high number of the SMOB conflicts between two threads. Eliminating the SMOB from the architecture may eliminate squashing due to memory speculation the speculative thread.

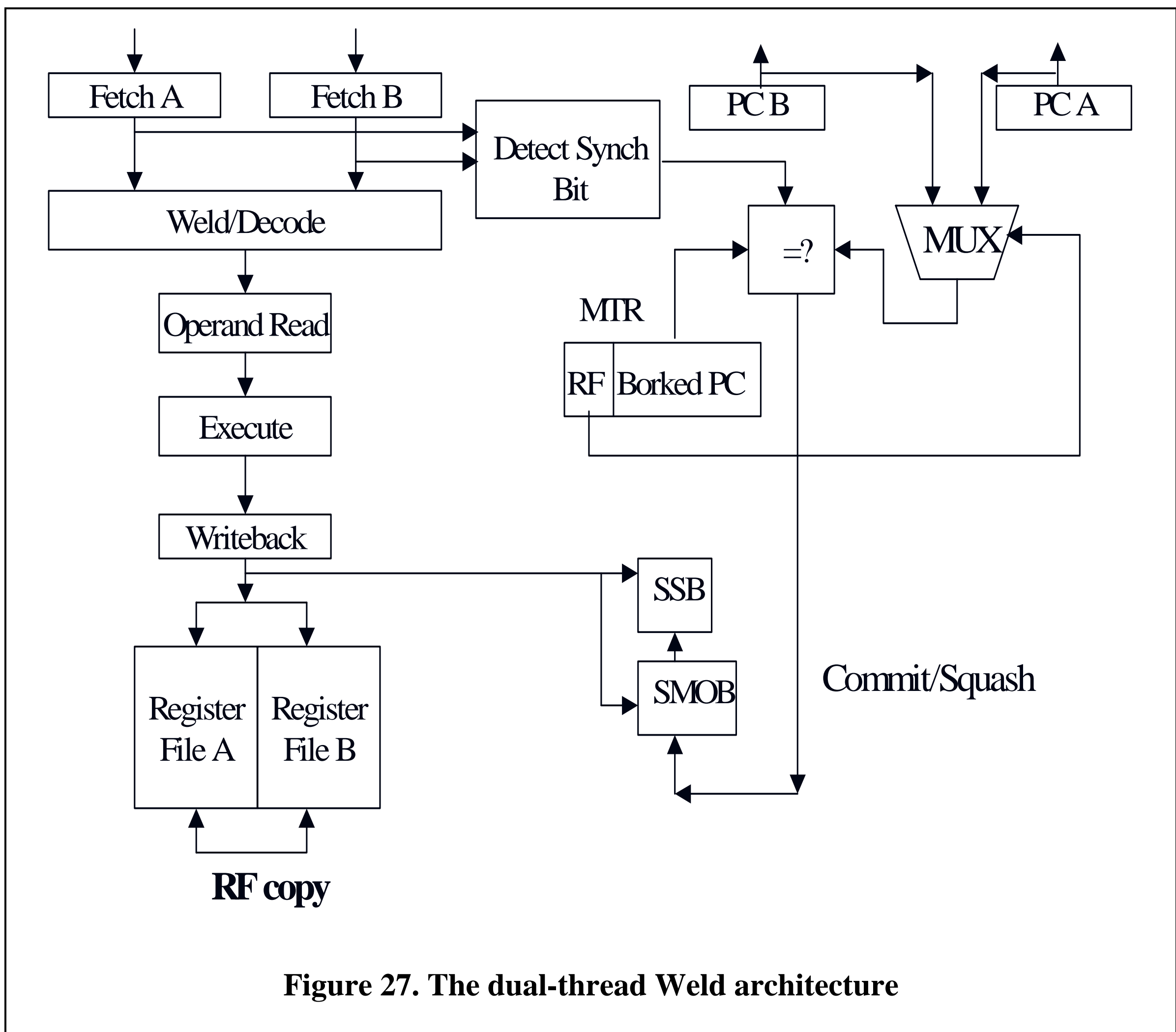


Figure 27. The dual-thread Weld architecture

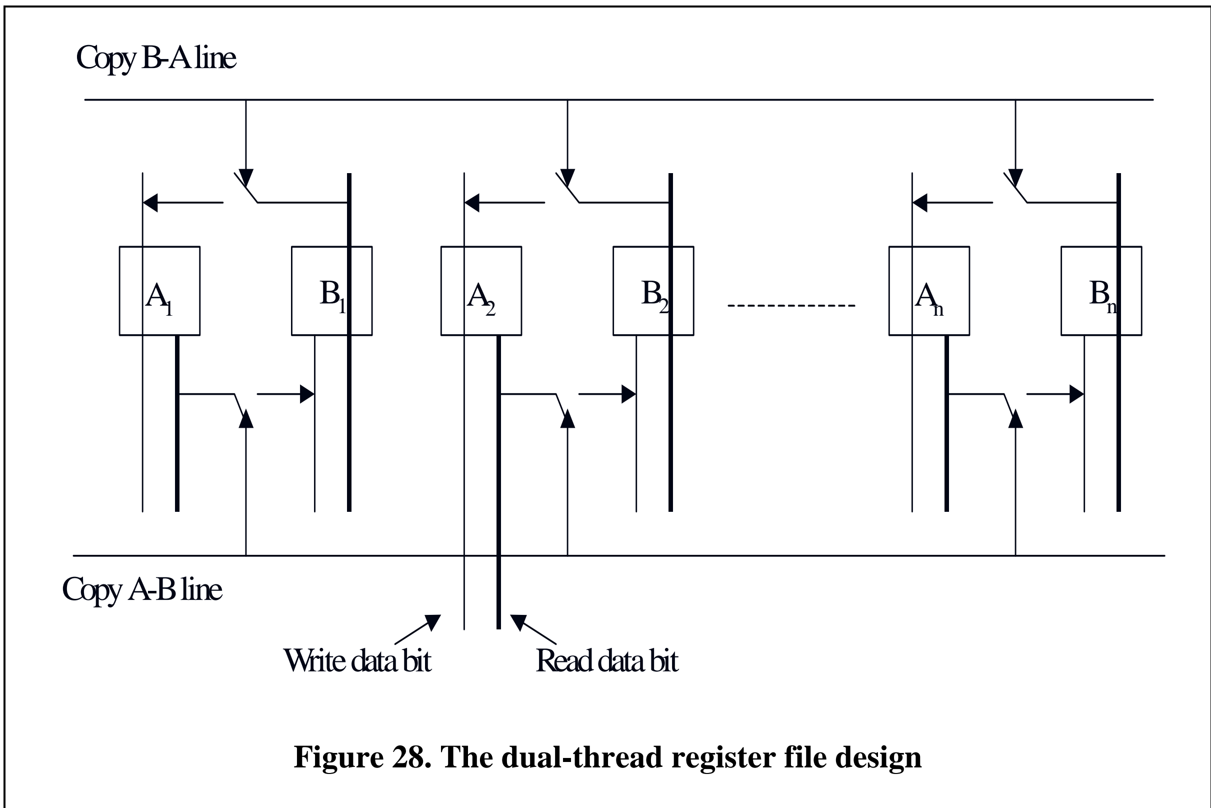


Figure 28. The dual-thread register file design

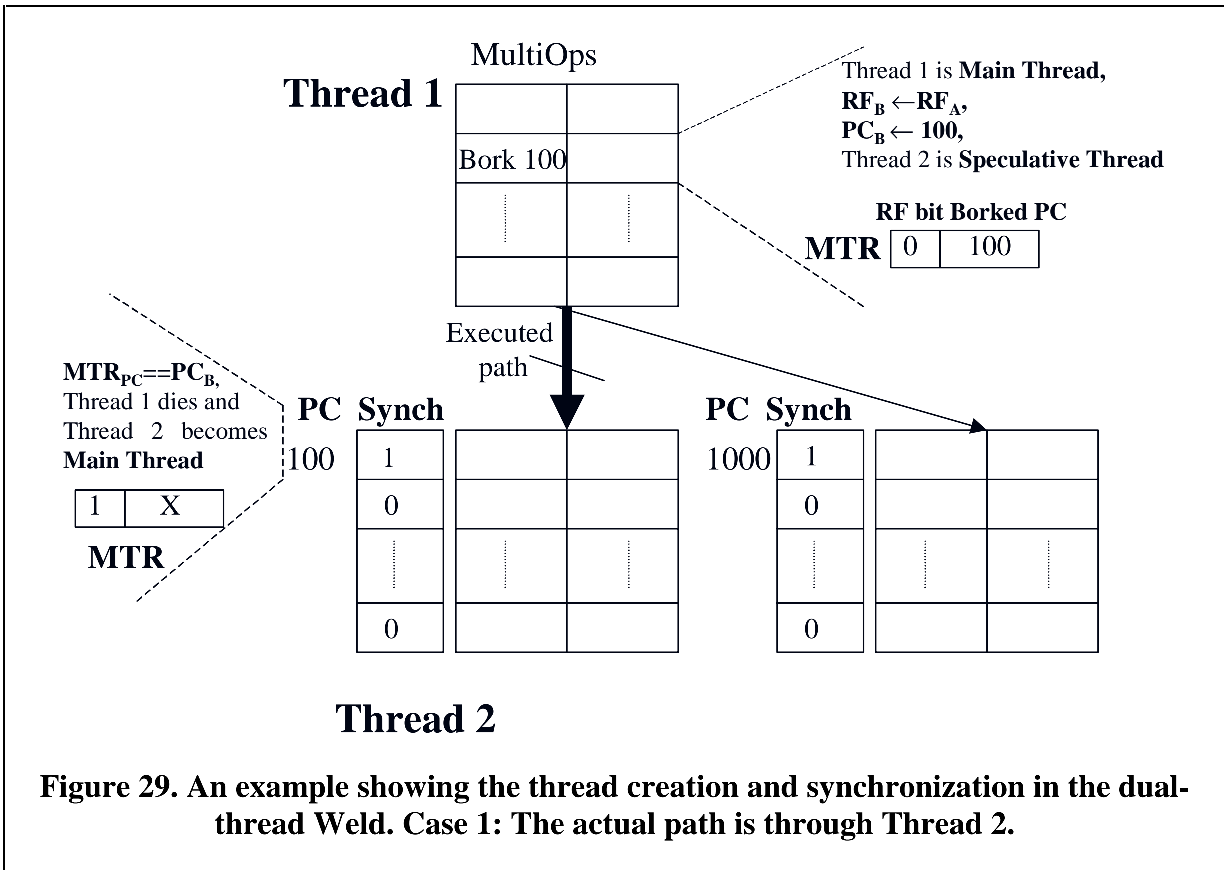


Figure 29. An example showing the thread creation and synchronization in the dual-thread Weld. Case 1: The actual path is through Thread 2.

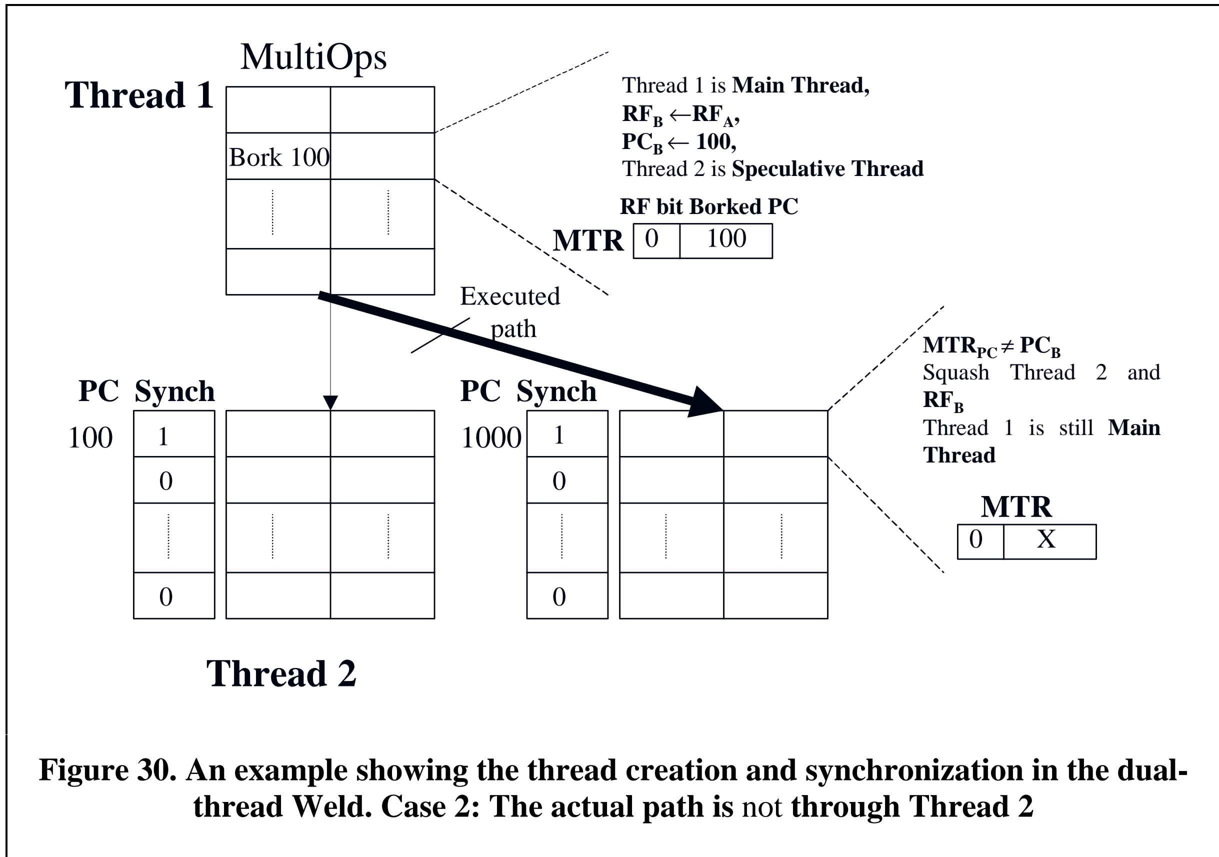


Figure 30. An example showing the thread creation and synchronization in the dual-thread Weld. Case 2: The actual path is not **through Thread 2**

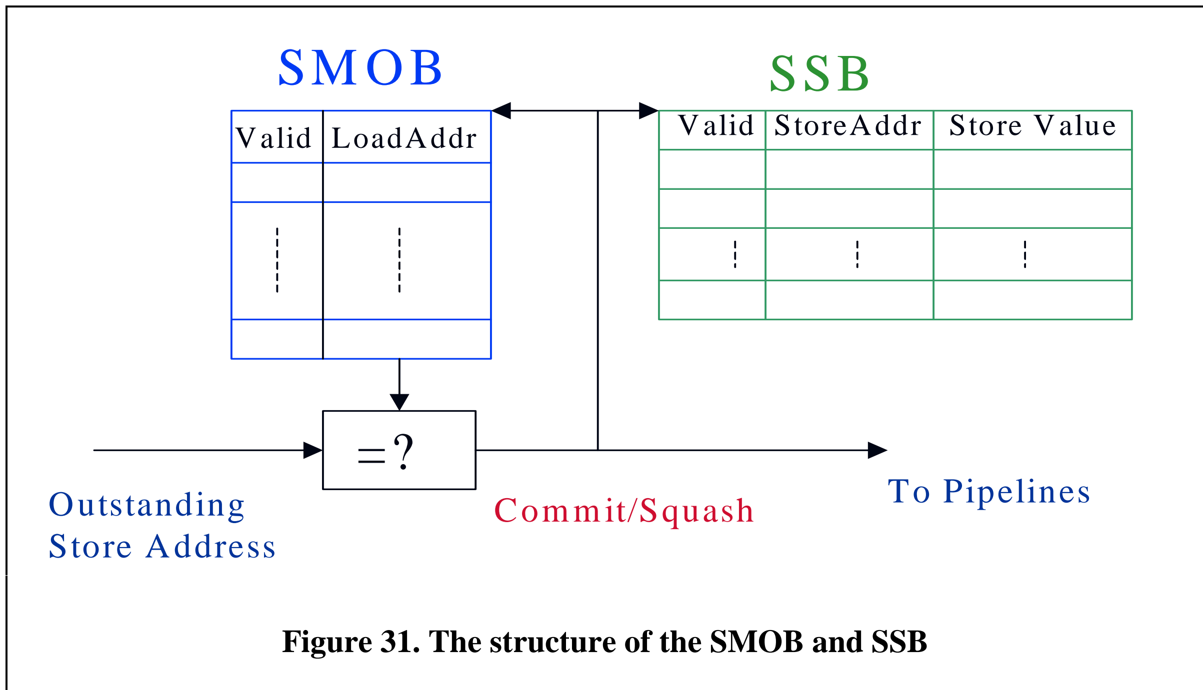


Figure 31. The structure of the SMOB and SSB

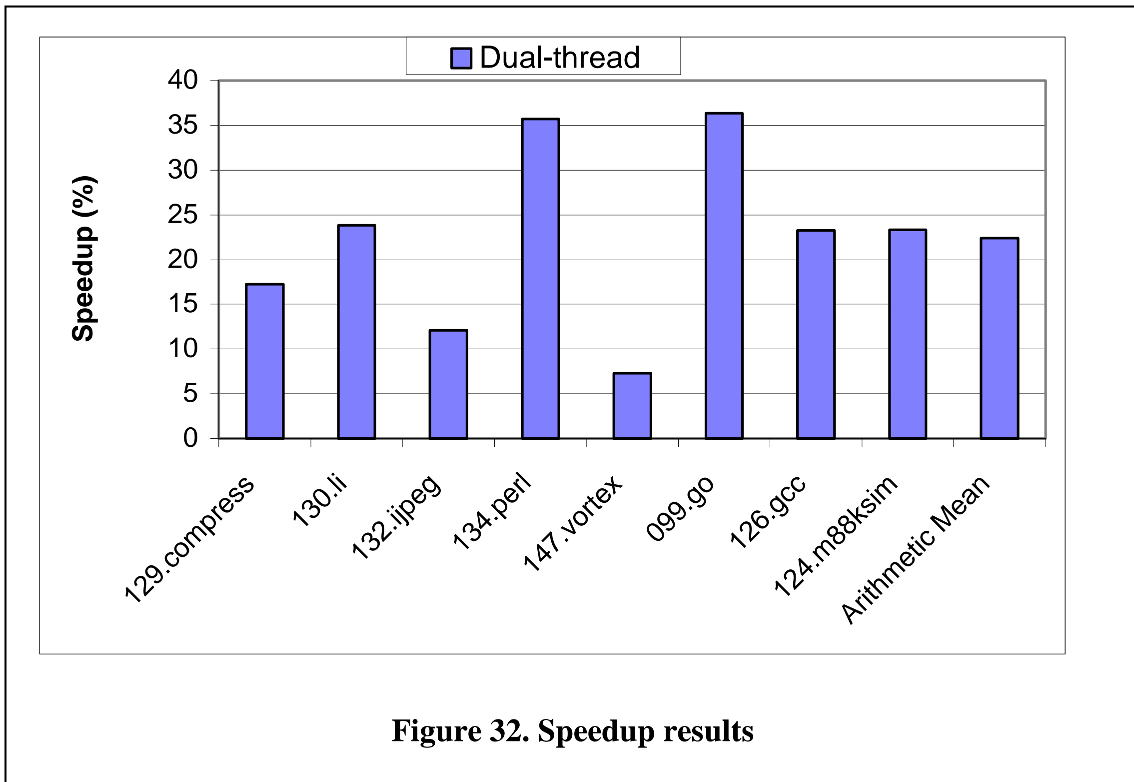


Figure 32. Speedup results

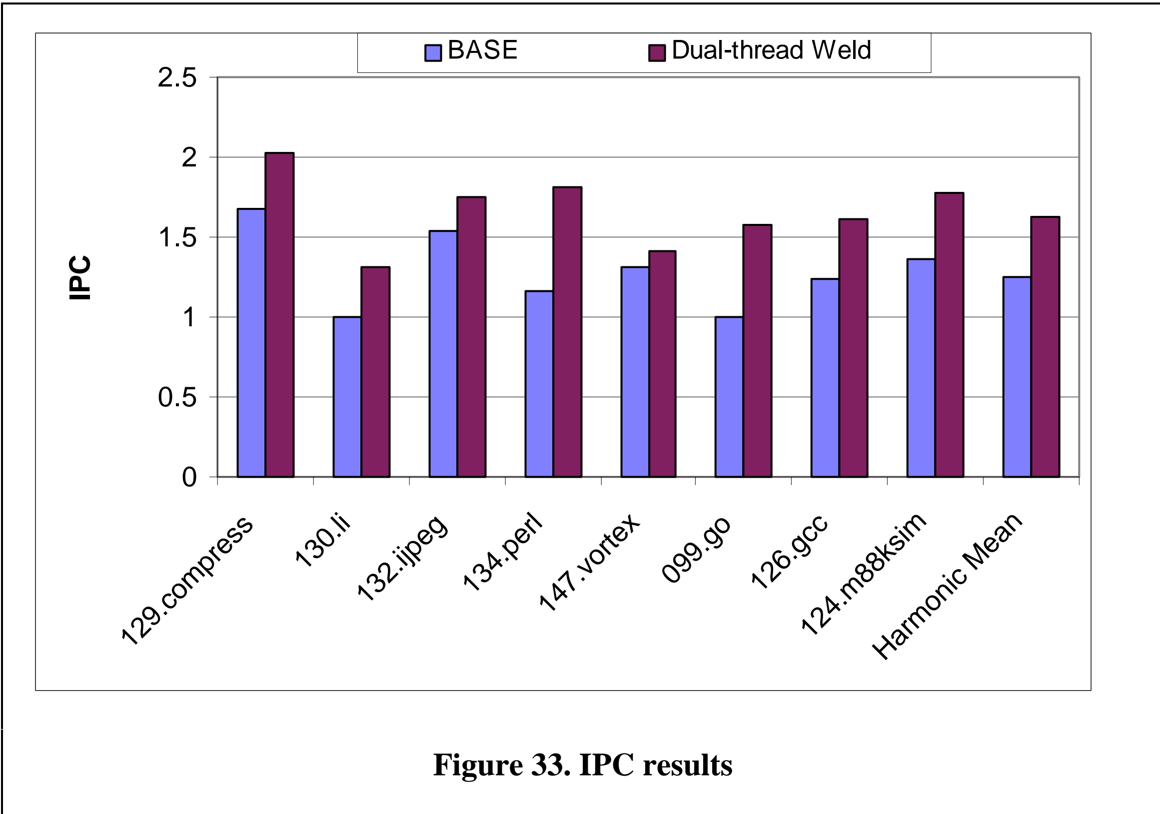


Figure 33. IPC results

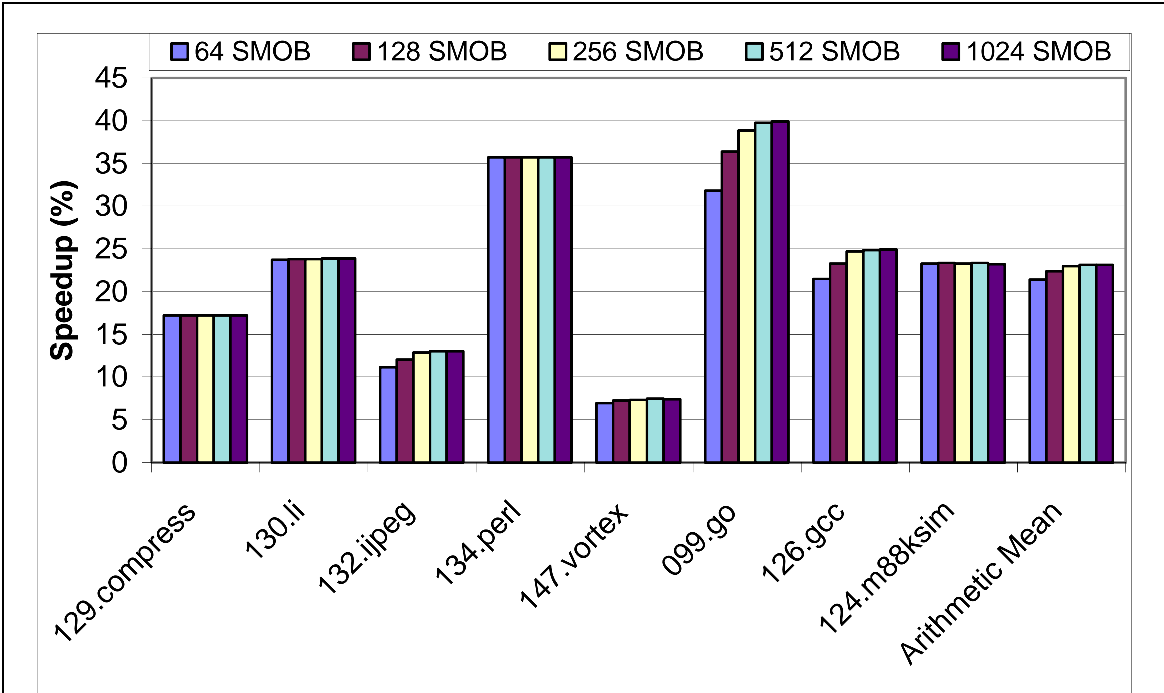


Figure 34. Speedup results of variable number of SMOB entries with fixed 64-entry SSB

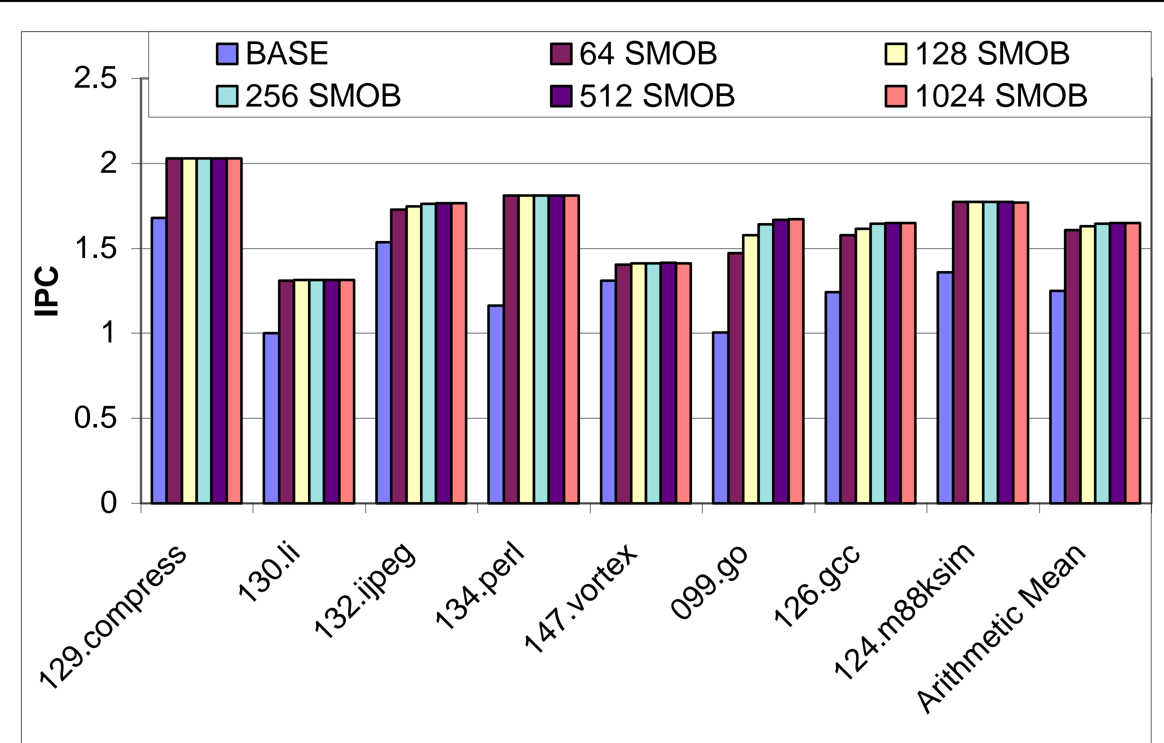


Figure 35. IPC results of variable number of SMOB entries with fixed 64-entry SSB

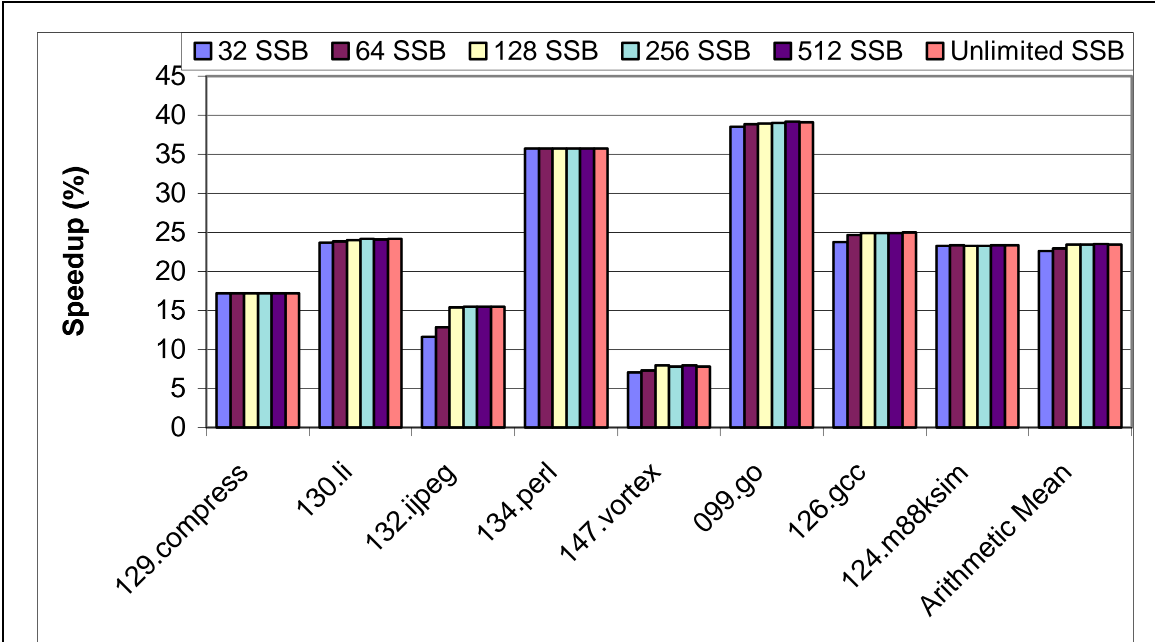


Figure 36. Speedup results of variable number of SSB entries with fixed 256-entry SMOB

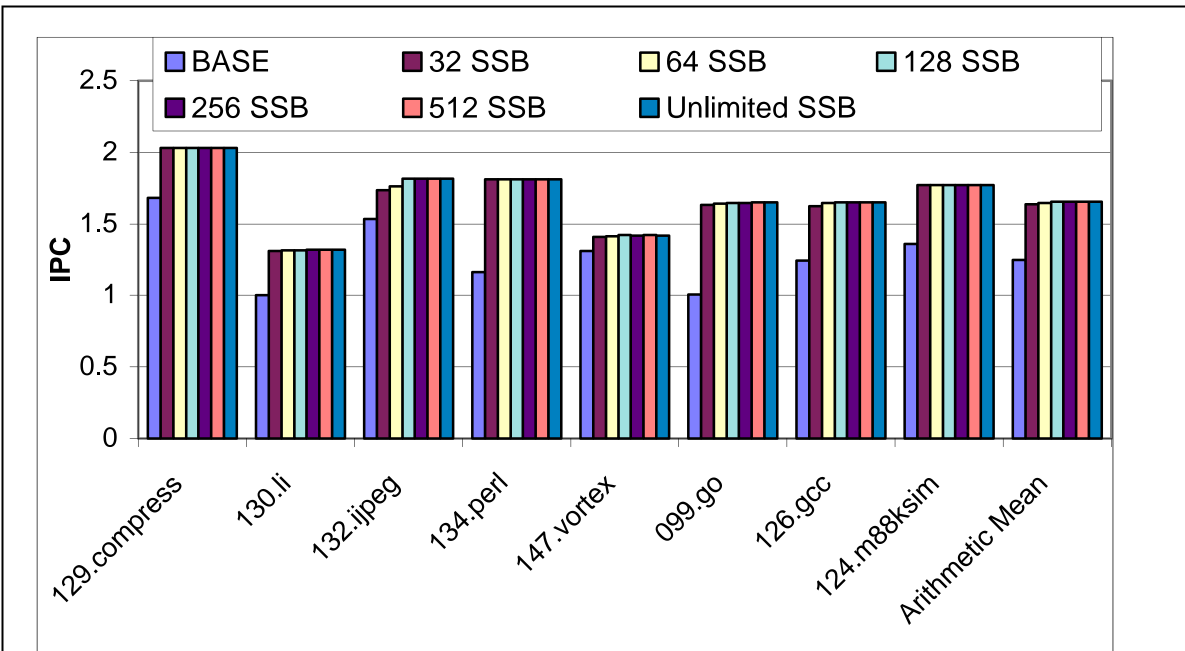


Figure 37. IPC results of variable number of SSB entries with fixed 256-entry SMOB

```
foreach Treegion Tmain begin
  foreach succeeding Treegion Ts begin
    foreach path in Tmain begin
        EarliestCycle = 1
        for EarliestCycle to Last Schedule Time of path begin
            Find an available hole to schedule a BORK
        end
        if  (a hole is found)  begin
            Insert a BORK operation into this path
        end
        else Do not insert BORK on this path
     end
  end
end
DeleteRedundantBORKS per path in Tmain
```

Figure 38. The bork insertion algorithm with a perfect value predictor

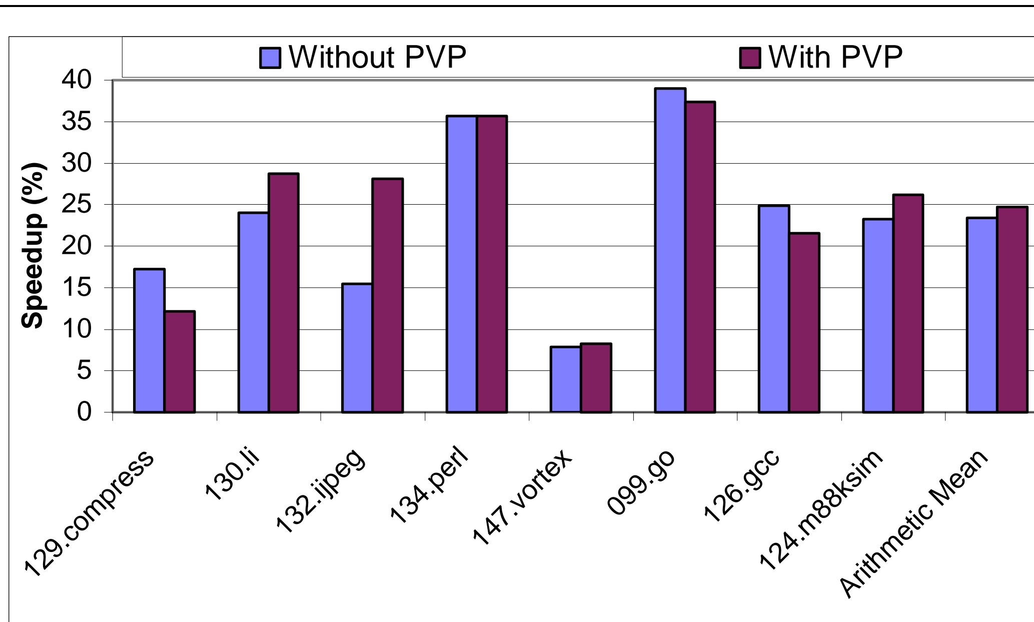


Figure 39. The speedup results of the dual-thread Weld with perfect value prediction (PVP) vs. without PVP

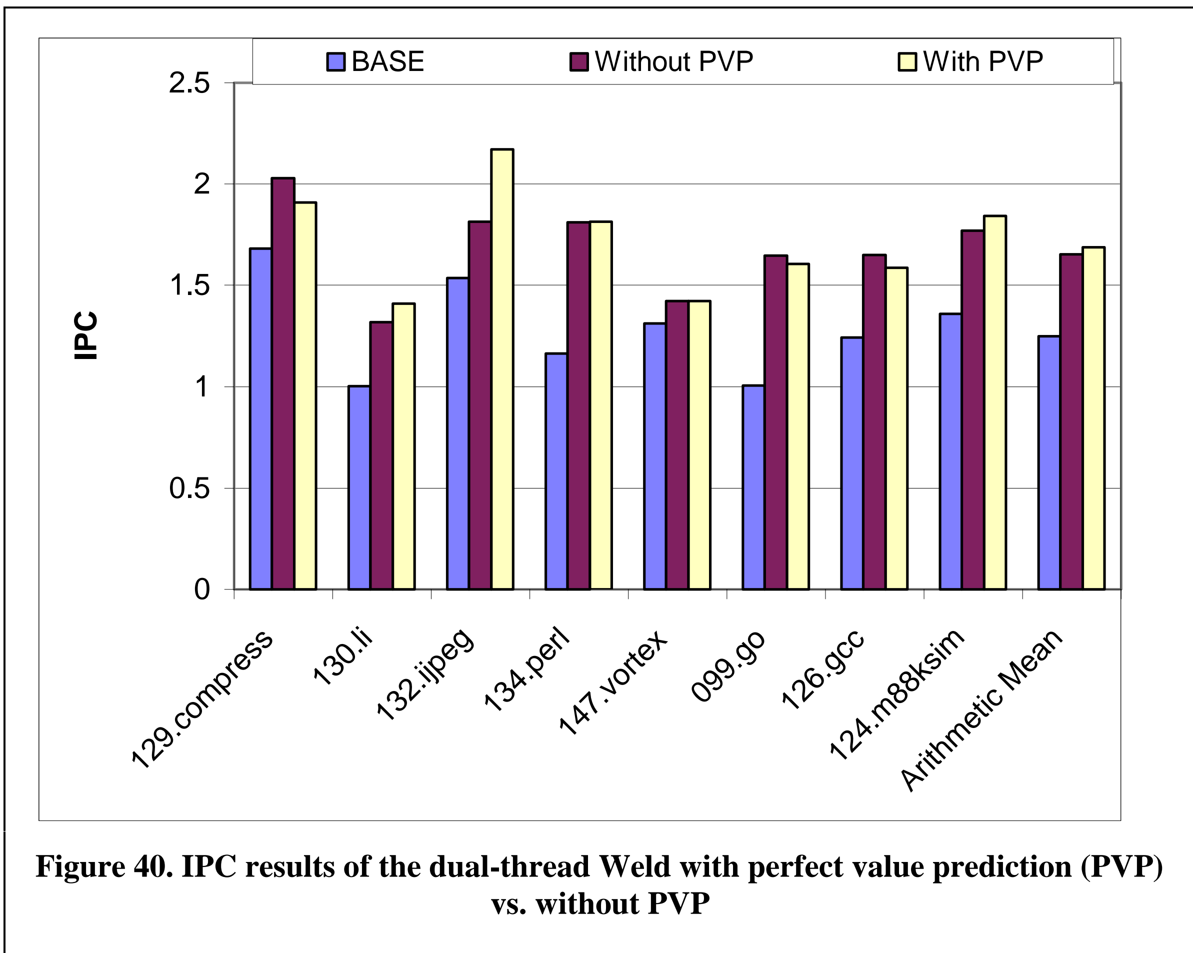


Figure 40. IPC results of the dual-thread Weld with perfect value prediction (PVP) vs. without PVP

5. Dual-thread Weld without Memory Speculation

This section introduces the dual-thread Weld model without using hardware memory speculation. So far, the speculative memory operation buffer (SMOB) resolves the conflicts caused by executing a speculative load before a store. In such a case, the speculative thread is completely squashed from the processor because there is no selective recovery hardware for the architecture model. Squashing requires throwing some useful work and flushing the pipelines, SMOB and SSB. This steals a great amount of useful cycles from the processor. This section analyzes the effects of eliminating the SMOB hardware by making the compiler not allow the speculative loads before the stores. Obviously, this would restrict the bork operations to be scheduled in the later schedule cycles since a speculative load in the speculative thread must come after all stores in the main thread. However, its run-time effect on the overall performance may be more effective than using the SMOB hardware because of the following reasons. First, all squash penalty cycles for the SMOB conflicts are eliminated. Secondly, all the useful work performed by the speculative thread until the load operation are not thrown away. Thirdly, the stall cycles relating to the fact that the SMOB becomes full can be eliminated as well. Intuition tells us that the performance can improve if the compiler guarantees the correctness of the order of the load and store operations. We also empirically show that this is the case. The architecture without the SMOB is shown in Figure 41.

5.1. Algorithm

The bork insertion algorithm in Figure 14 is modified to guarantee that when scheduling a bork operation per path, all store operations in the path complete before the loads in the speculative thread. The previous live-out restrictions also still apply in the algorithm. The modified algorithm is shown in Figure 42. The algorithm computes the schedule time of each store in the path under consideration. After computing schedule times, the maximum store completion time among these stores is calculated by adding the operation latencies. The earliest time a bork can schedule is the maximum of the maximum live-out completion time and the maximum store completion time if there is at least one load operation in the speculative treegion. If there is not even a single load operation, then the earliest time for a bork is determined by the maximum live-out completion time.

An example is shown about the bork insertion in the dual-thread model without memory speculation Figure 43. The example is the same as the example in Figure 15 in Chapter 3.1. However, we have to consider the latency of the store operation when computing the schedule time for the bork operation. For path 1, the store operation is scheduled at cycle 4 and its completion time is cycle 5. On the other hand, the maximum live-out completion time is at cycle 4. Now, the maximum of 5 and 4 is taken as the earliest cycle an empty slot can be searched to insert a bork. The bork is inserted at cycle 5 in the second slot as shown in Figure 44. For path 2, a bork cannot be scheduled since the live-out condition is violated with or without a store operation.

5.2. Speedup Results

The performance results of the dual-thread Weld with no SMOB and 128-entry SSB are shown in Figure 45 together with the results of the dual-thread Weld with 256-entry SMOB and 128-entry SSB. Both are presented with respect to the base model. As seen from the graph, all benchmarks experience a big performance boost. This increase is accounted for eliminating all squash penalty cycles and not squashing all useful executed MultiOps up to a load operation in the speculative thread, as it is the case in the model with the SMOB. The largest improvement occurred in 147.vortex because it encounters the largest number of SMOB conflicts among all benchmarks. Eliminating SMOB gets rid of all penalty cycles incurred by squashing. On the contrary, the smallest improvement in speedup can be observed in 134.perl. For the same reasons, 134.perl has smaller number of SMOB conflicts and therefore takes less advantage of no SMOB than other programs. 124.m88ksim is the only benchmark that loses performance without using the SMOB. This is because the number of the SMOB stalls is pretty small and not using the SMOB has no effect on performance. On the contrary, later issuing of borks causes slowdown in 124.m88ksim. The average speedup is now close 35% as compared to 23% with SMOB while the same number of SSB entries (i.e.128) is being used. Figure 46 shows the IPC results for the model. The average IPC approximates to 1.9 as compared to 1.6 in the dual-thread Weld with memory speculation.

5.3. Variance of Branch Penalty

The machine model has 5-stage pipeline that determines the branch penalty and also the thread squash penalty. However, the recent microprocessors have much deeper pipelines and therefore higher branch penalties. A study is needed to find out the

performance effects on the dual-thread Weld architecture when the branch penalty and therefore the thread squash penalty vary. Figure 47 shows the speedup results when the branch penalty (BP) is five, ten and fifteen. As seen in the graph, the speedup is not sensitive to the variance of the branch penalty or the number of pipeline stages. In spite of the longer pipeline stalls in the main thread due to the longer branch penalty, the speculative thread can still progress and compensate the idle cycles caused by the main thread. Even, in some programs such as 129.compress, 132.ijpeg, 147.vortex, 099.go and 126.gcc, the speedup improves as the branch latency increases.

5.4. Variance of L2 Miss Latency

Also, the latency of the second-level (L2) cache is varied. The default latency was 30 cycles and it was increased to 90 cycles to watch the effects of longer L2 miss latency on performance in the dual-thread Weld architecture. Figure 48 shows the results. Note that the branch penalty is taken as a constant of 5 cycles in both runs. On the average, the speedup increases 0.2% in spite of the L2 miss latency increase. This is attributed to the fact that the speculative thread takes advantage of longer stalls in the main thread and makes up for the stalled cycles. This effect can be observed in 130.li, 132.ijpeg, 134.perl, 147.vortex, 099.go and 124.m88ksim. In 129.compress and 126.gcc, the opposite is observed because the speculative thread cannot progress or not even be borked during these longer stalls in the main thread. Therefore, the speedup drops in both benchmarks from 30 cycles to 90 cycles.

5.5. Welding versus No welding

Figure 49 shows the speedup using welding versus not using it. The average speedup is 10% with operation welding. This speedup is much higher than the dual-weld model with SMOB because treegions are spawned much later in the model without the SMOB, that means that borks are located down in the code. The lower blocks of a treegion is much more sparse than the upper blocks. This inherently opens opportunities for welding more operations into the main thread's empty slots at run time.

5.6. Variance of Register File Copy Cycle Time

So far, we assumed a 1-cycle for register file copy with a mass transfer of registers in a register file to another. However, we also consider long latency register file transfer in case register file are implemented disjointly. Register transfer from disjoint register files may take more than one cycle. The transfer time is changed from 1 cycle to 10, 20, 30, 40, 50 and 60 cycles to see the effects on performance. Figure 50 shows the speedup results with 1, 10, 20, 30, 40, 50 and 60 cycle register file copy time. The speedup with 1-cycle register transfer time is known as 34%. From 1 cycle to 10 and 20 cycles, the speedup drops to 28% and 22%, respectively. There is a linear relationship between the speedup drop and increase in the register file copy cycle. At every 10-cycle increase in the copy time, the speedup drops by 6%. Until the copy time is 60 cycles, a positive speedup is possible. At 60 cycles and after, a slowdown on performance is observed.

In summary, the register file copy cycle is an important factor in designing the dual-thread weld processor. The performance of the processor is very sensitive to the copy time. A careful design of the register files would make the register file transfer fast and therefore not critical on performance.

5.7. Variance of the Size of the SSB

The size of the SSB can affect the performance because the processor has to stall the issue when the SSB becomes full. So far, we used a size of 128 entries for the SSB, which was inherited from the dual-thread Weld model with SMOB. Here, we measured the effects of the size of the SSB on performance. The sizes of 64, 128, 256 and 512 entries are experimented. Figure 51 shows the speedup results for these SSB sizes. The speedup is 32% with 64-entry SSB and goes up to 34% with 128 entries, 35.5% with 256 entries and 36.5% with 512 entries. The speedup increase rate drops by 0.5% as the size of the SSB doubles. Figure 52 shows the IPC results of the same SSB entries.

5.8. The Performance Effects of Using a Machine Model without Universal Units

The performance effects of welding on different machine models need to be measured. A machine model with no universal functional units is chosen to see the effects of the operation welder on a machine without any universal functional unit. The machine model has 4 ALU/BR units, one LD/ST and one FP units. This machine model also limits the compiler schedule because there is less flexibility to move the operations around without universal units. This would inherently create more empty slots in the

schedule but may increase chances of operation welding. Figure 53 shows the speedup of the machine model with welding versus the one without welding. The speedup is a little more than 11% with the operation welder. This would show us that the operation welder can be effectively utilized even with a more restricted machine models where the compiler due to the machine model is also restricted. Figure 54 and Figure 55 show the overall speedup and IPC results for this machine model.

5.9. The Performance Effects of Using a Machine Model Only with Universal Units

Here, a different machine model with all universal functional units is experimented for the operation welder. This machine model has six universal units that can execute any operation. The operation welder can weld an operation to any of these universal units. Intuitively, the slot utilization should be much better than a model with less universal units. However, the empty slots can be filled at compile time more effectively as well since the compiler has more flexibility to schedule operations in a machine with only universal units. Therefore, the effective slot utilization of an operation welder in such a machine model may not be significantly high. Figure 56 shows the speedup of the machine model with welding versus the one without welding. The speedup is at about 12% with the operation welder. This speedup is not significant compared to other machine models. This justifies our assumption that the compiler performs a better job on scheduling operations in a machine with all universal units and creates more compact schedules. Figure 57 and Figure 58 show the overall speedup and IPC results for this machine model.

5.10. Concluding Remarks

The memory speculative dual-thread Weld with disambiguation hardware causes a large number of thread squashes due to the memory data conflicts. This chapter developed a compiler technique that eliminates the disambiguation hardware and therefore thread squashes based on the load-store conflicts at thread level. The performance results showed that the dual-thread Weld without hardware memory disambiguation has a speedup of 35% compared to a single-threaded VLIW/EPIC processor. This provides not only a higher performance model but also much simpler hardware implementation with no special hardware disambiguators.

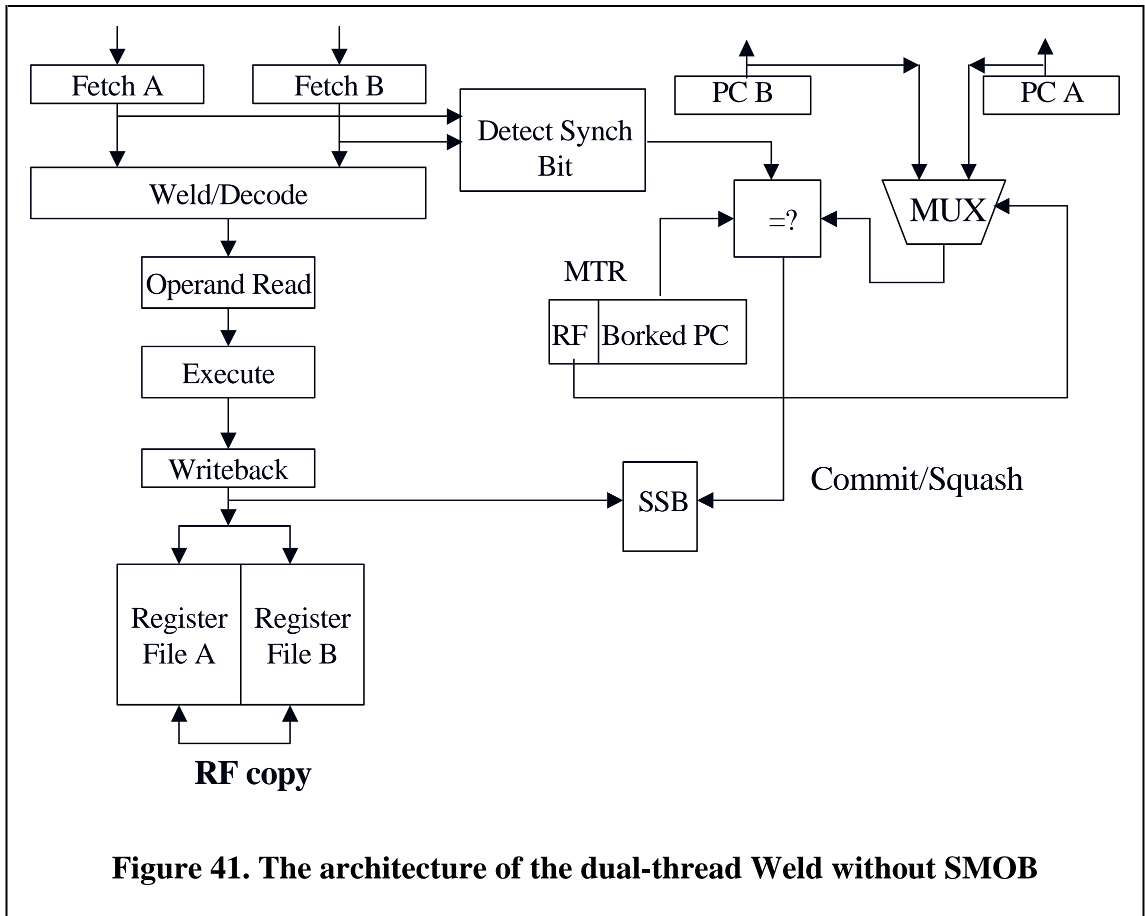


Figure 41. The architecture of the dual-thread Weld without SMOB

```
foreach Treegion Tmain begin
  foreach succeeding Treegion Ts begin
   FindLiveOprds(LiveOprds[], Tmain)
   foreach path in Tmain begin      //starting from the entry basic block of Tmain to its exit into Ts
        FindStoreOprds(StoreOprds[],path)
        DefLiveSet[] = Definitions of LiveOprds[] in path
        MaxLiveCompletionTime = The maximum live-out completion time in DefLiveSet[]
        MaxStoreCompletionTime = The maximum store completion time in StoreOprds[]
        if (a load Op in Ts) EarliestCycle = Max(MaxLiveCompletionTime, MaxStoreCompletionTime)
        else EarliestCycle = MaxLiveCompletionTime
        for EarliestCycle to Last Schedule Time of path begin
             Find an available hole to schedule a BORK
        end
        if (a hole is found) begin
             Insert a BORK operation into this path
        end
        else Do not insert BORK on this path
    end
  end
end
DeleteRedundantBORKS per path in Tmain
```

Figure 42. The bork insertion algorithm for the dual-thread Weld without memory speculation

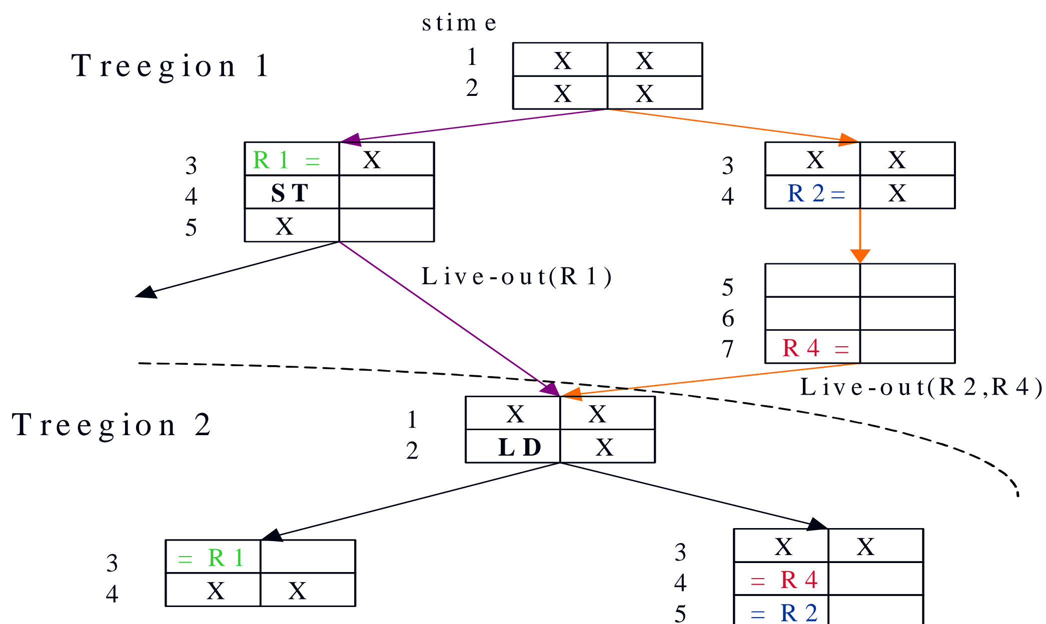


Figure 43. An example of bork insertion in the model without memory speculation

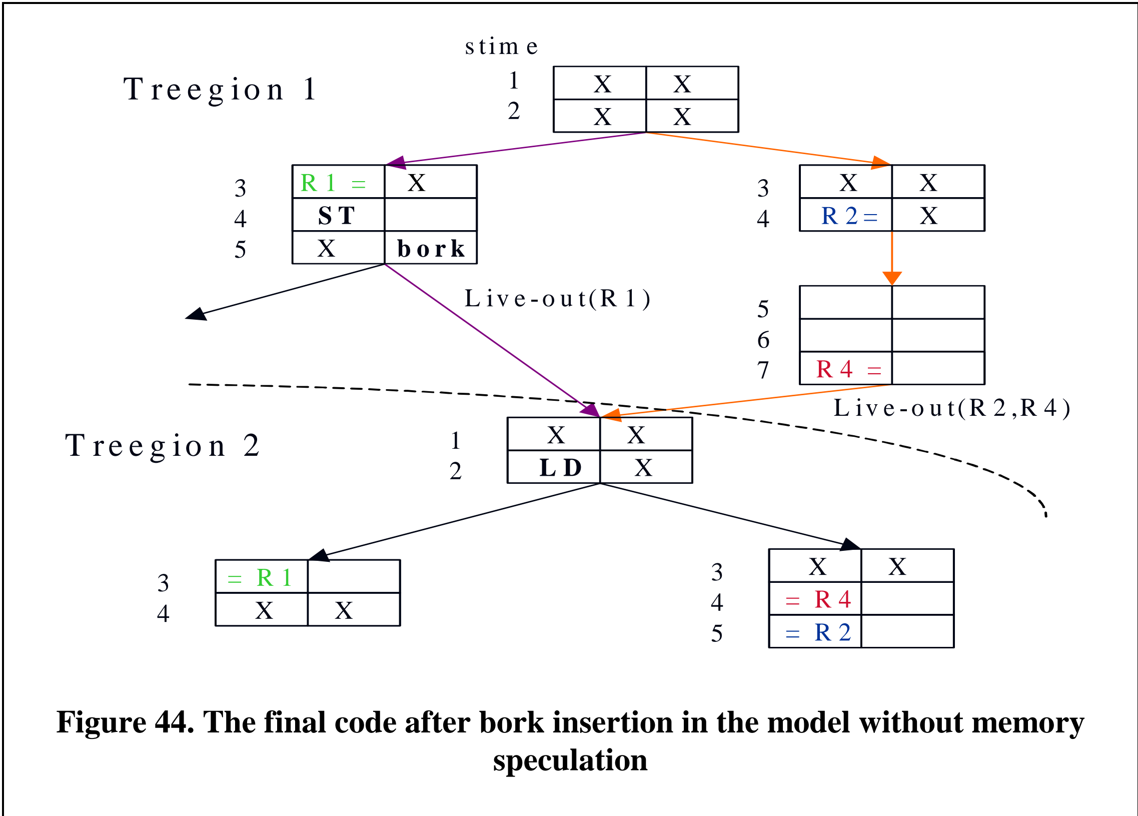


Figure 44. The final code after bork insertion in the model without memory speculation

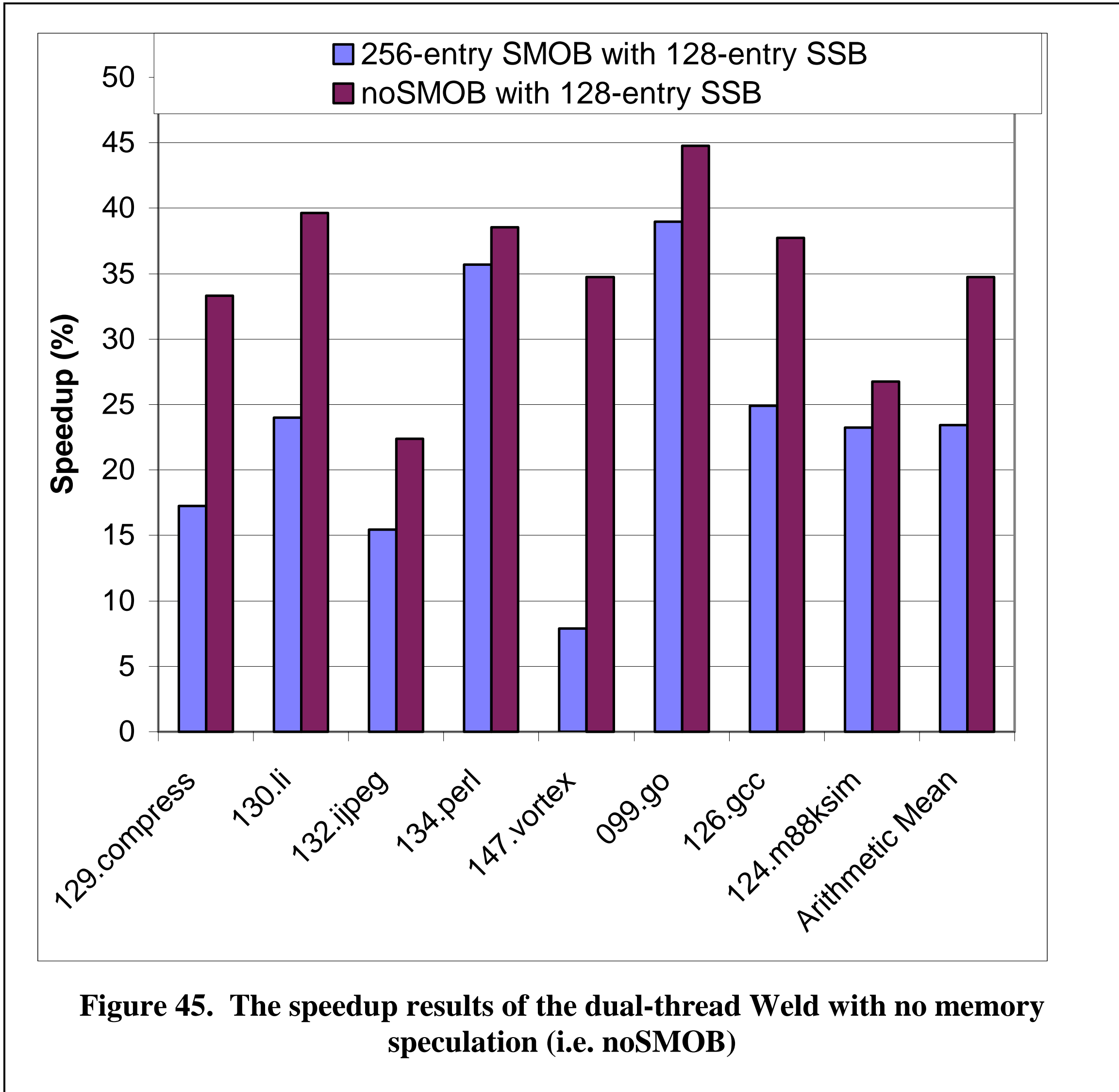


Figure 45. The speedup results of the dual-thread Weld with no memory speculation (i.e. noSMOB)

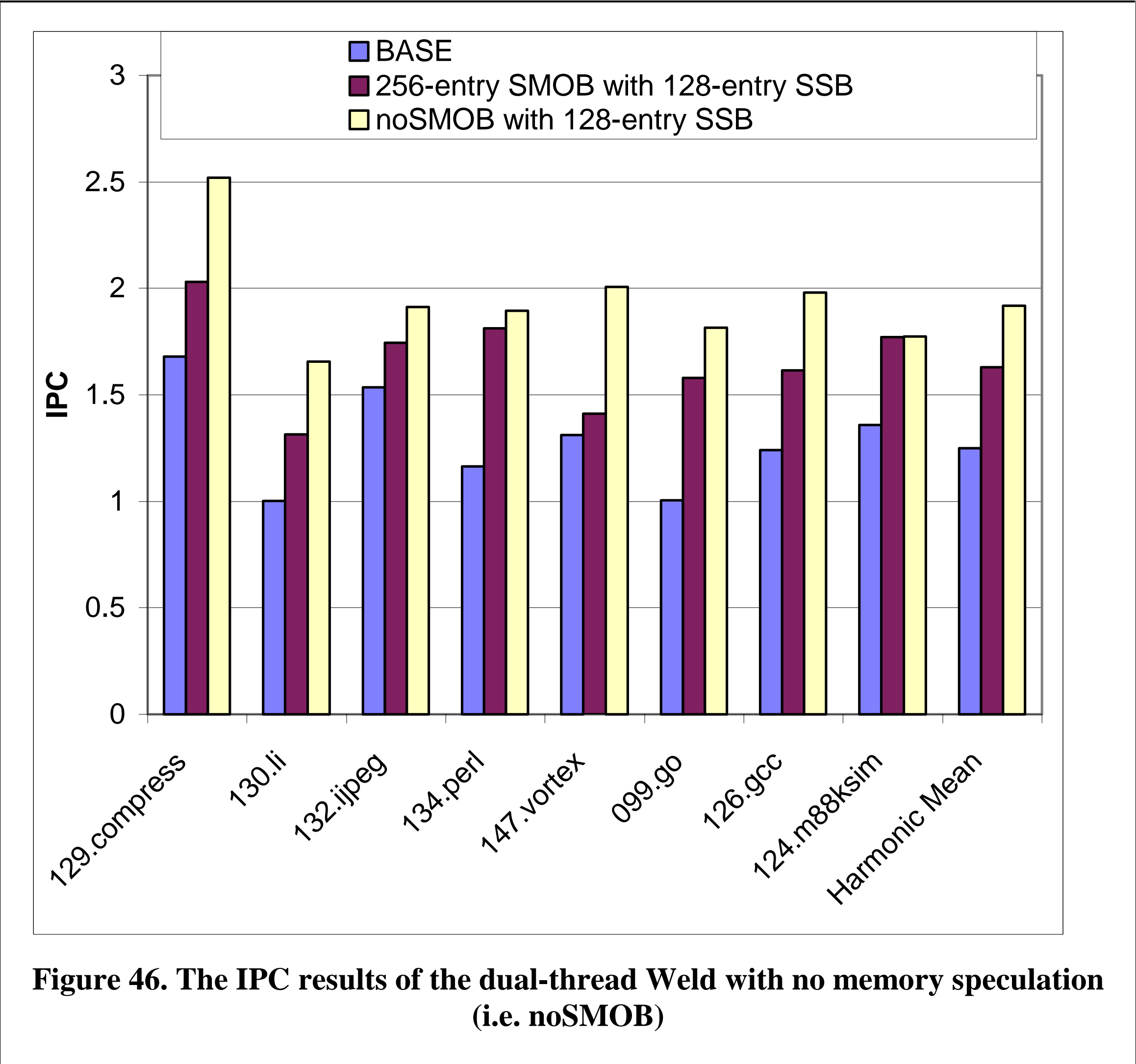


Figure 46. The IPC results of the dual-thread Weld with no memory speculation (i.e. noSMOB)

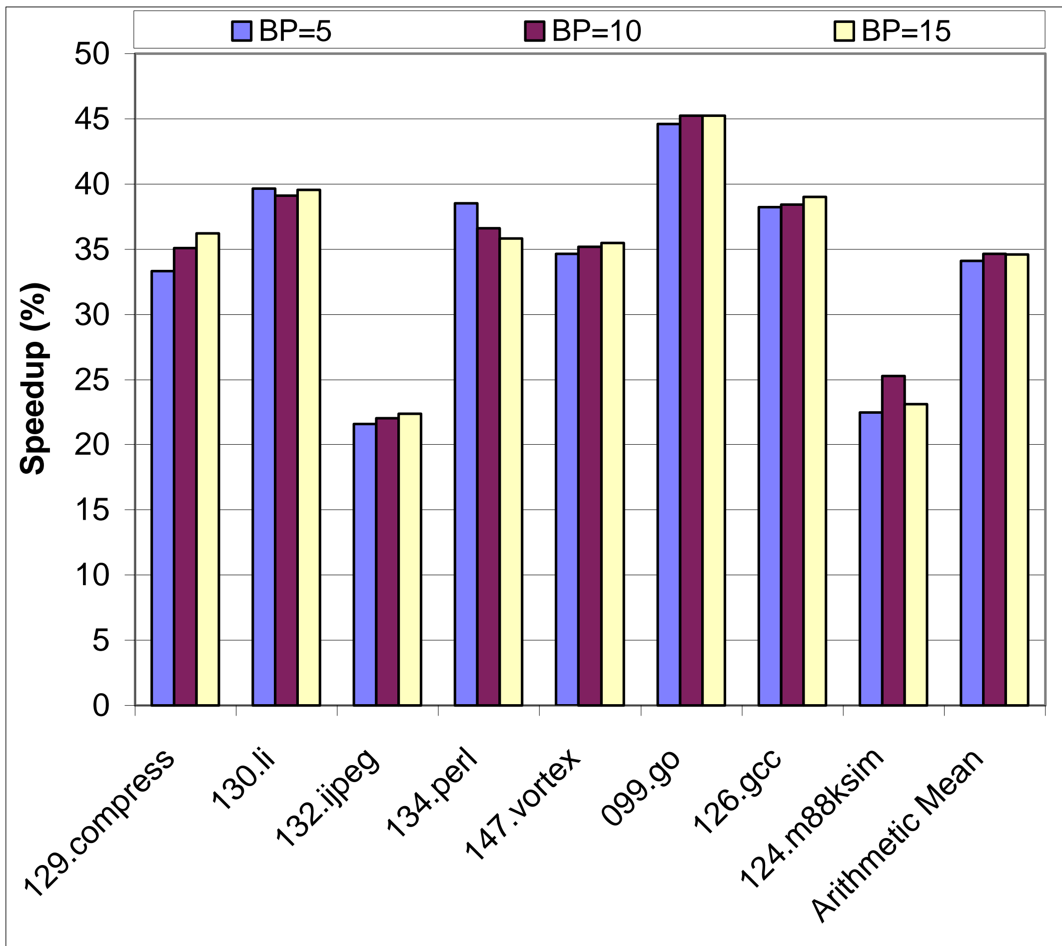


Figure 47. Speedup results when the branch penalty is varied from 5 to 15.

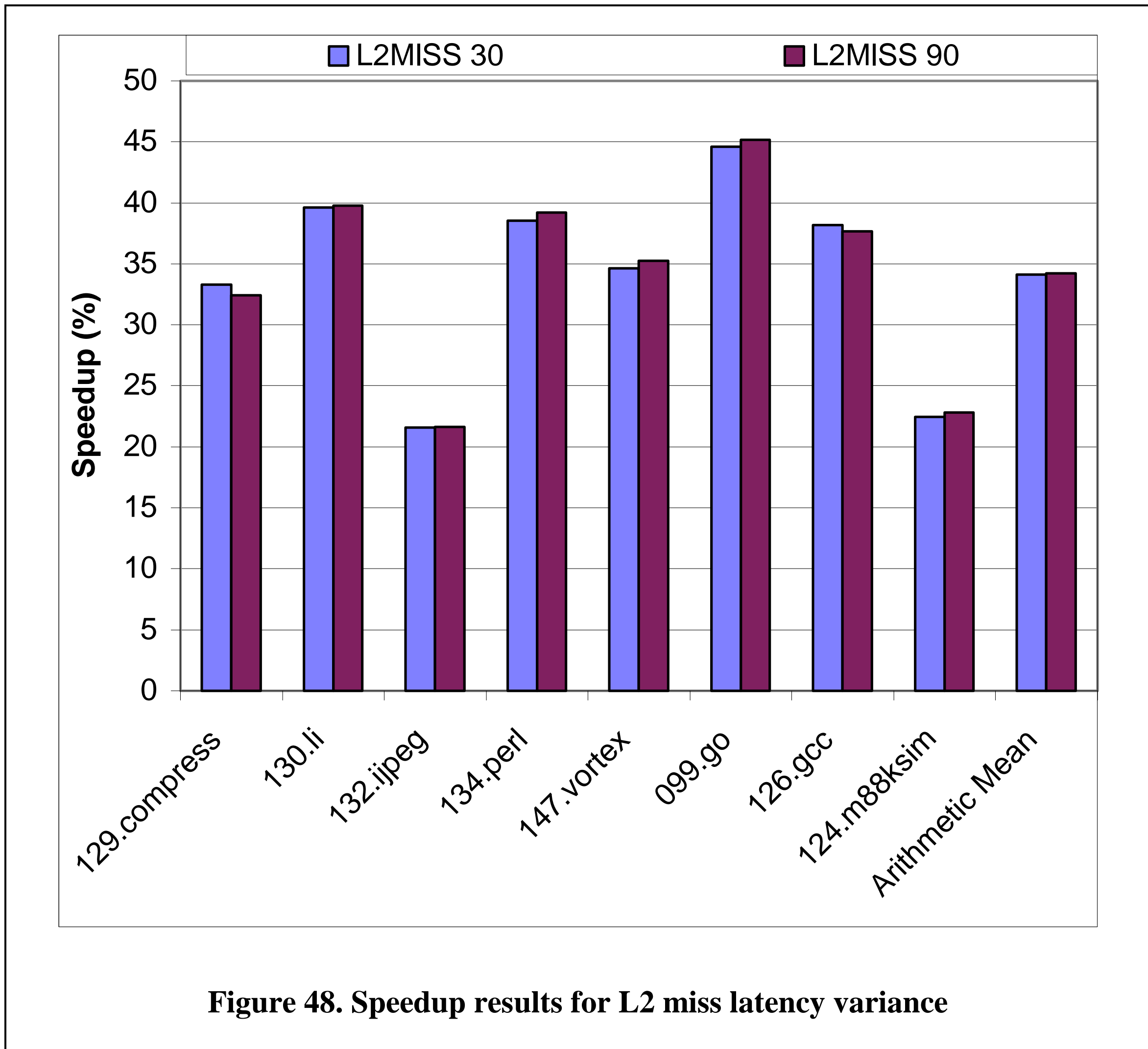


Figure 48. Speedup results for L2 miss latency variance

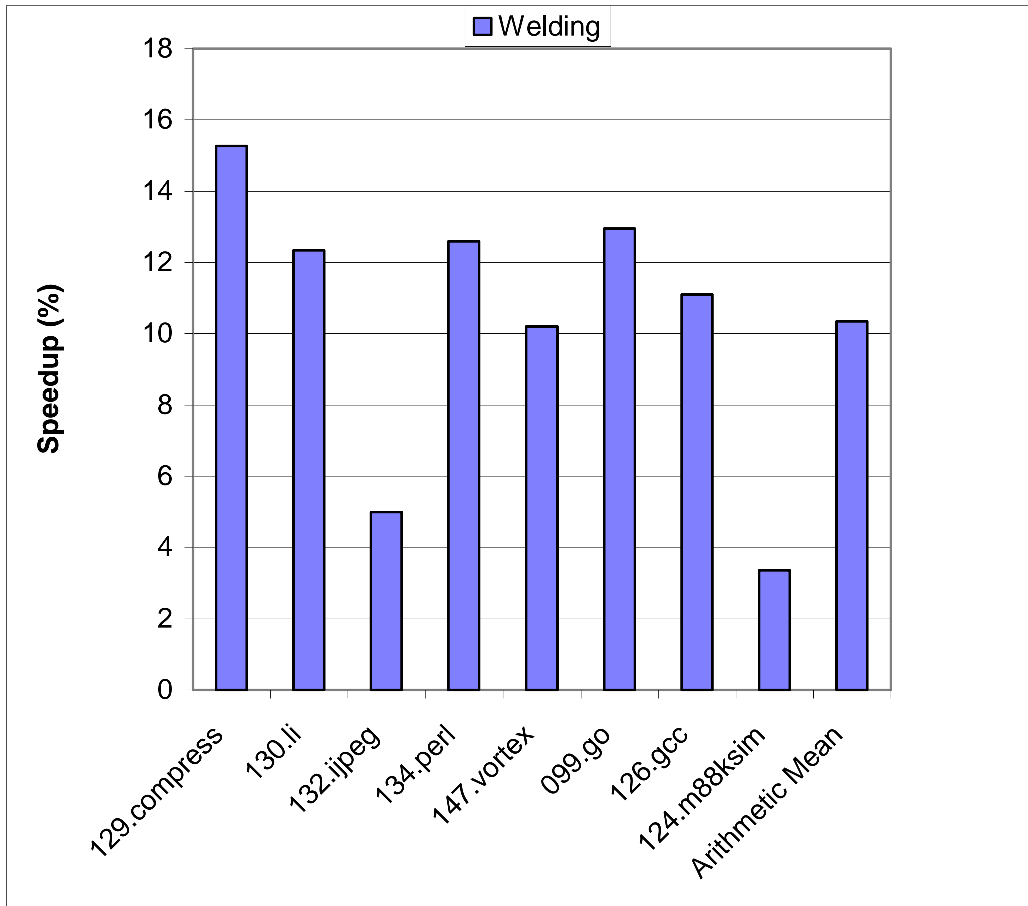


Figure 49. Speedup with welding versus without welding

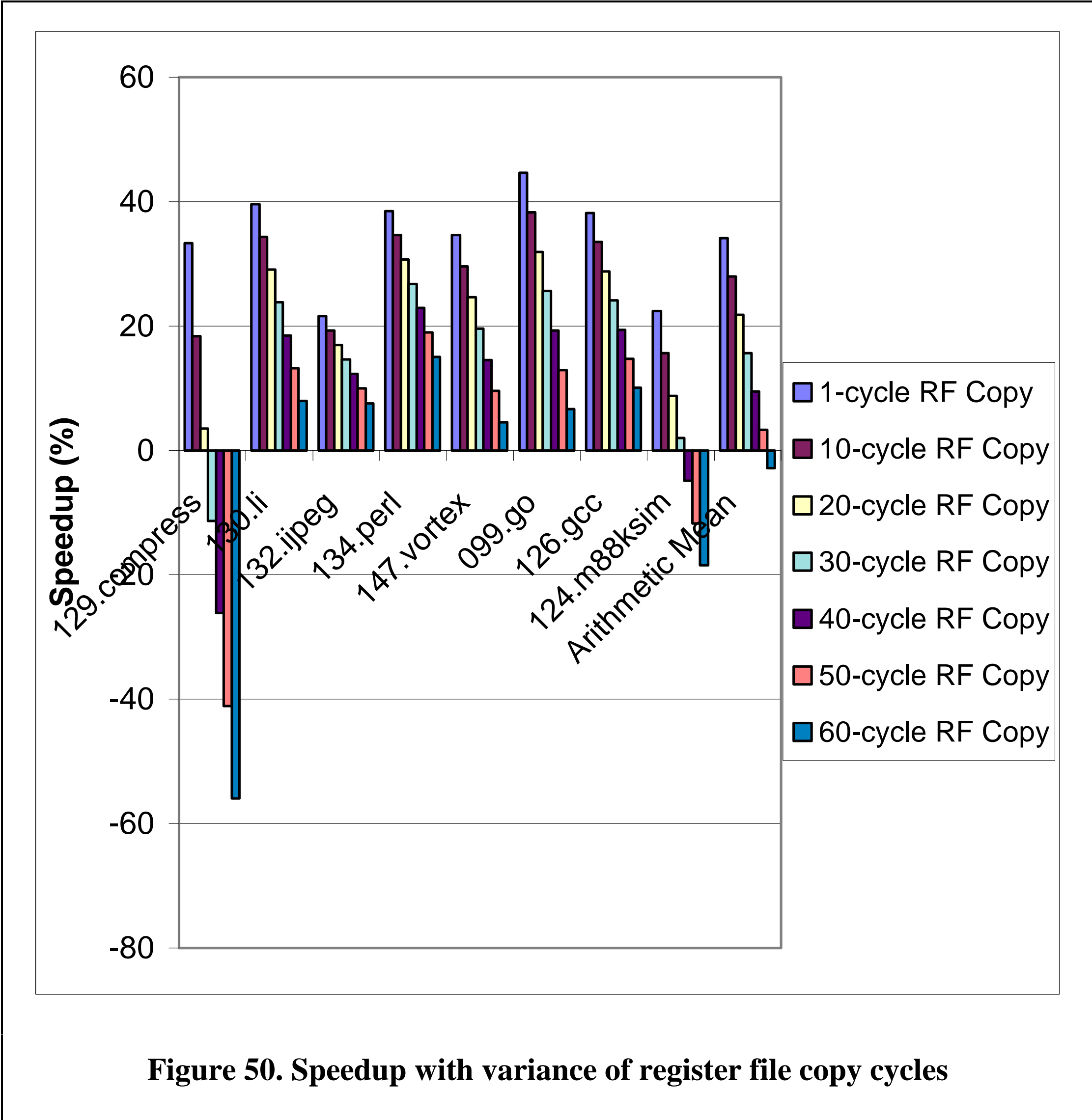


Figure 50. Speedup with variance of register file copy cycles

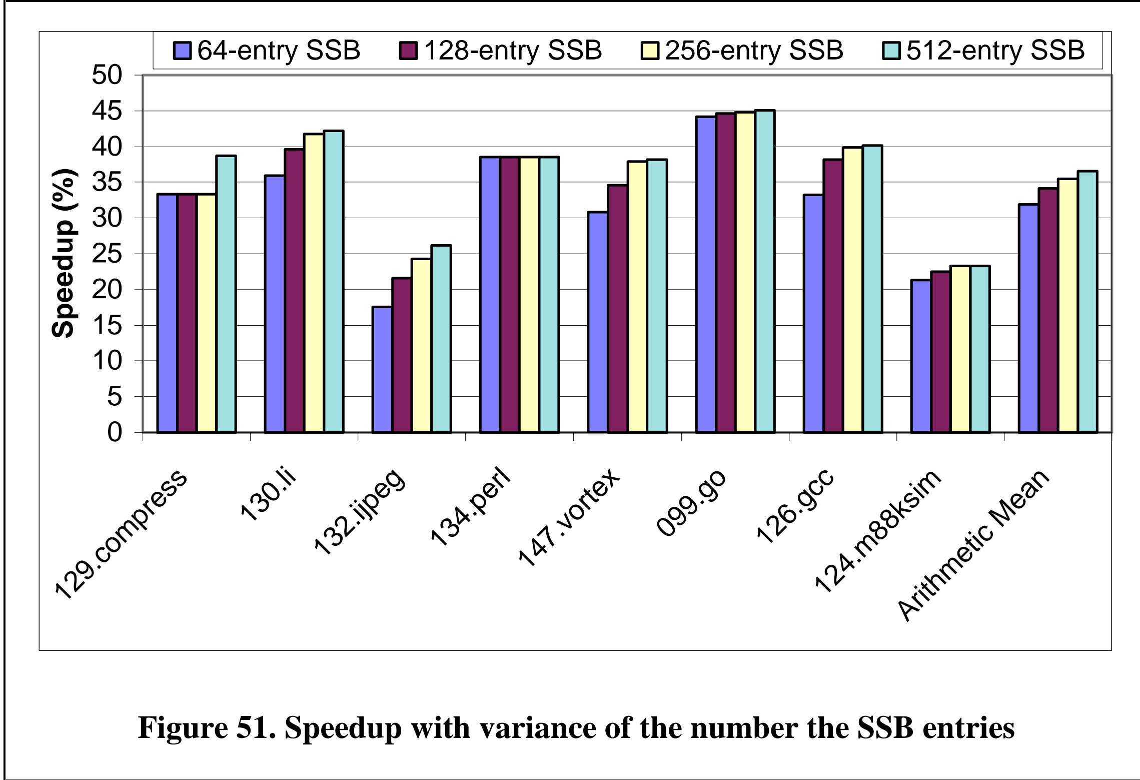


Figure 51. Speedup with variance of the number the SSB entries

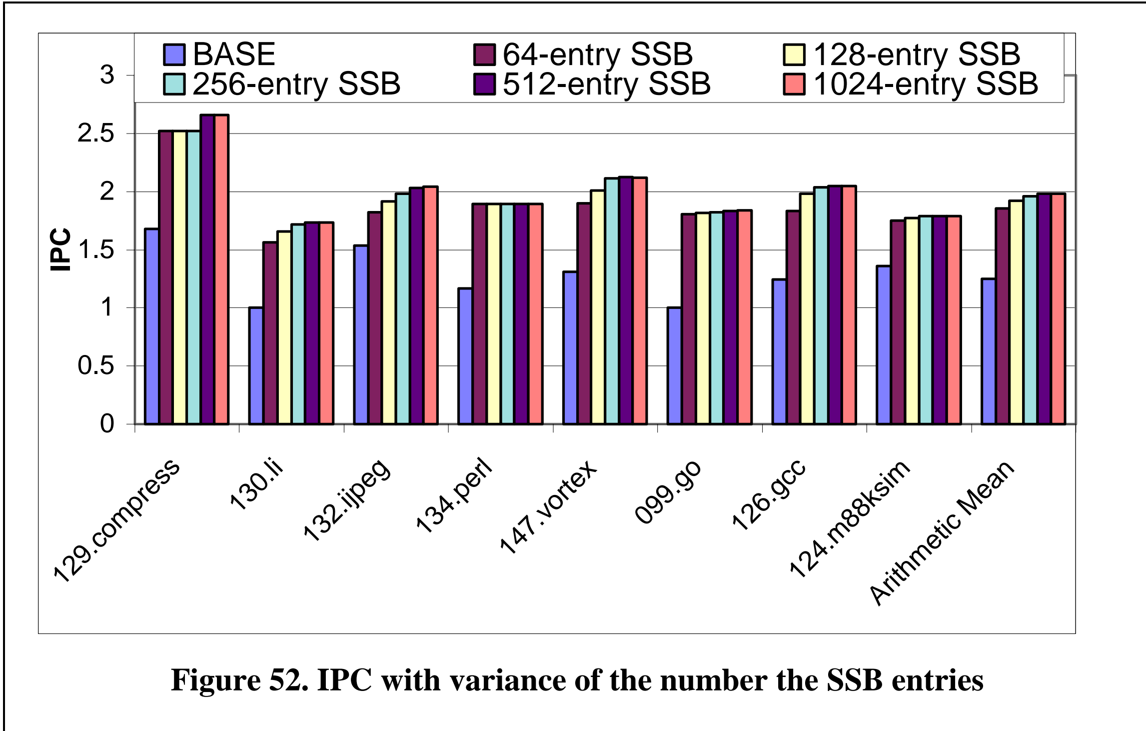


Figure 52. IPC with variance of the number the SSB entries

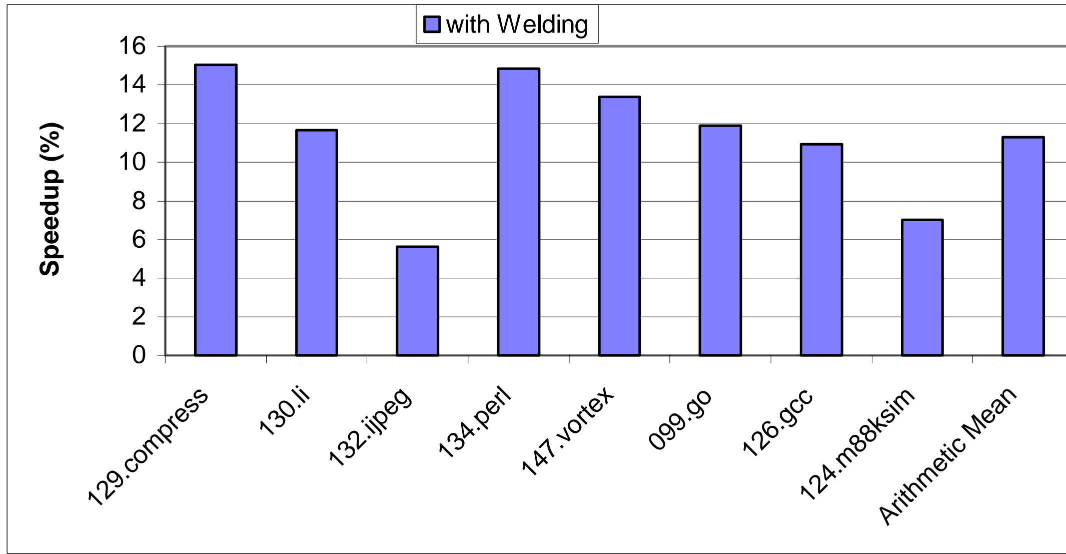


Figure 53. Speedup with welding vs. nonwelding in a machine model without any universal units

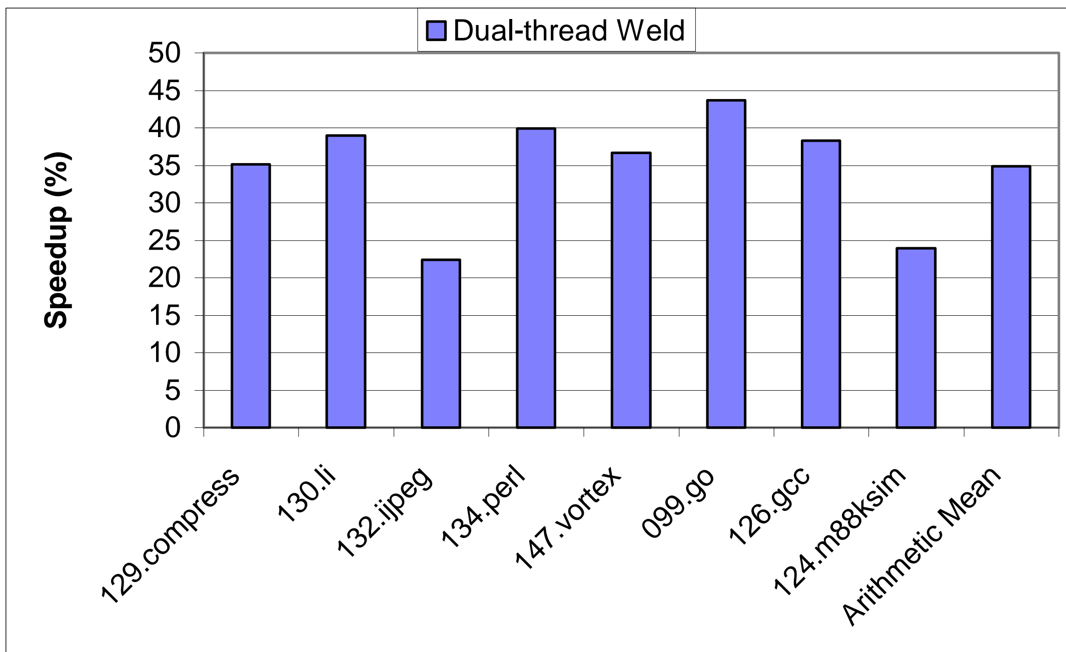


Figure 54. Speedup results of the dual-thread with no SMOB for a machine model with no universal units

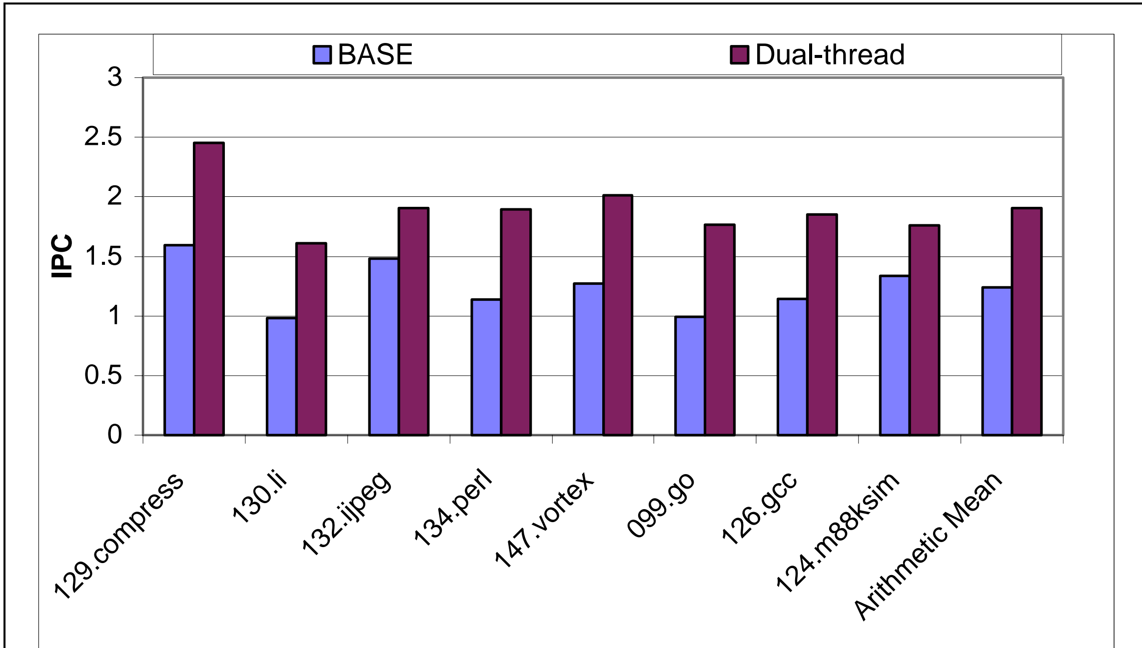


Figure 55. IPC results the dual-thread with no SMOB for a machine model with no universal units

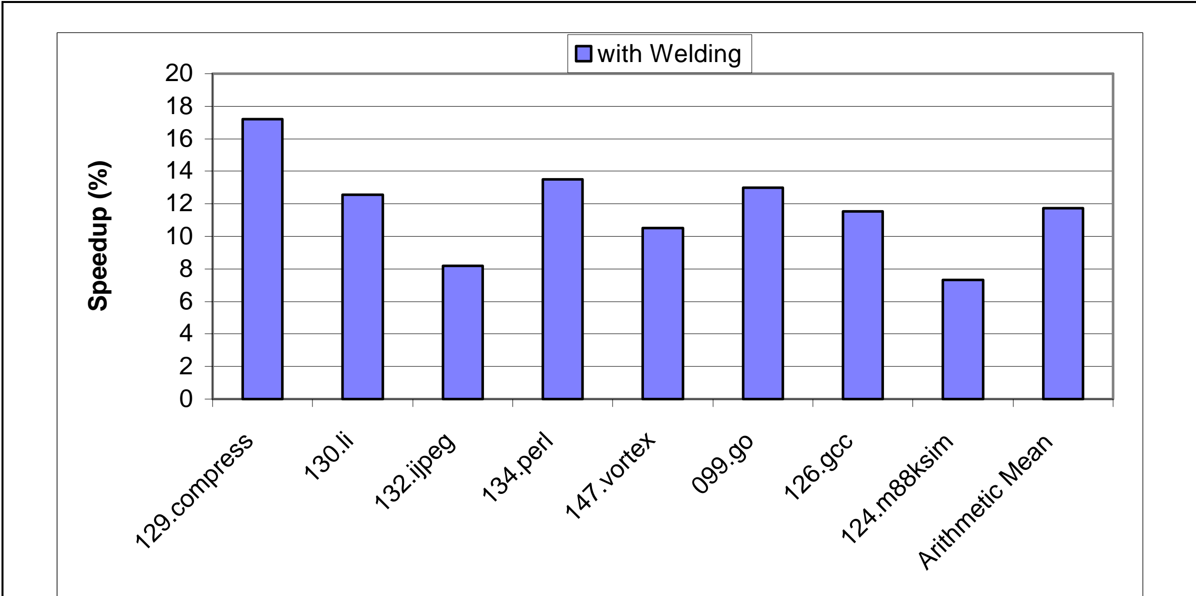


Figure 56. Speedup with welding vs. nonwelding in a machine model with all universal units

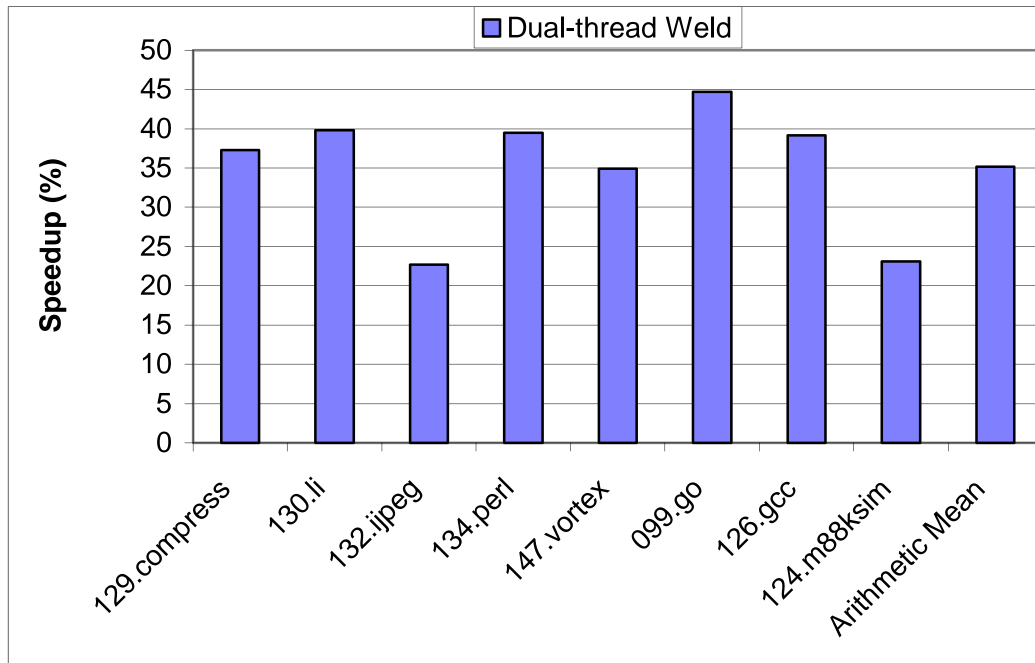


Figure 57. Speedup results of the dual-thread with no SMOB for a machine model with all universal units

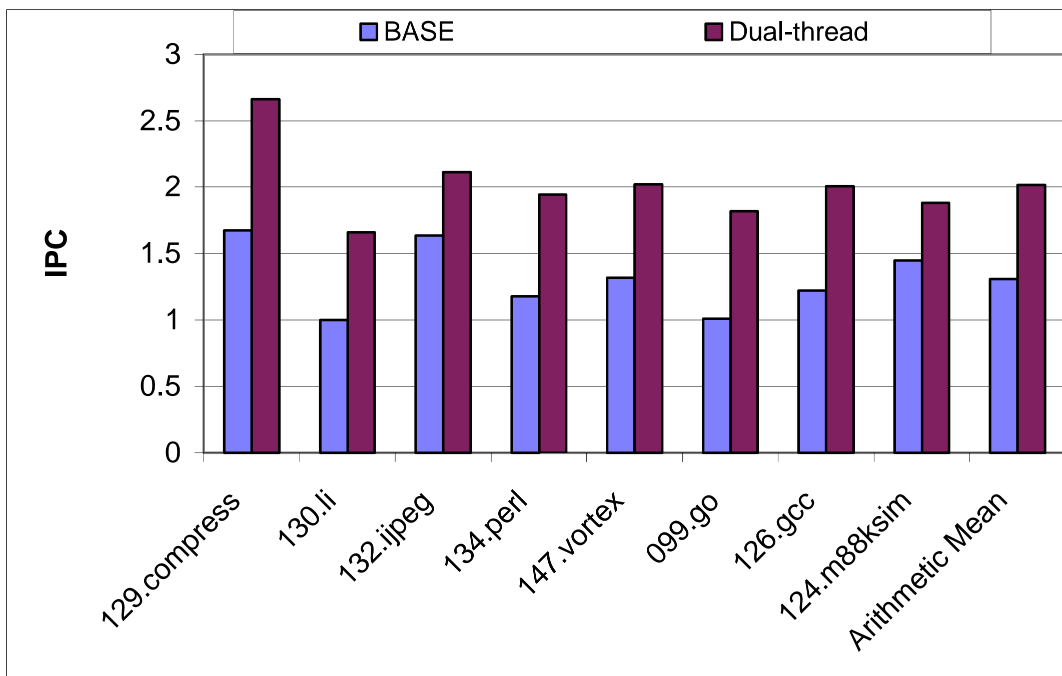


Figure 58 IPC results of the dual-thread with no SMOB for a machine model with all universal units

6. Related Work

There are three primary works of interest on the topic of multithreading for horizontal architectures. Most of the previous multithreading studies are developed for dynamically scheduled processors. A brief introduction and comparison with Weld of all schemes are given in the following two sections.

6.1. Multithreading in Horizontal Architectures

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. If a NOP exists in the issue slot I of thread 0, then the issue slot I of thread 1 is checked. This goes on until a non-NOP operation can be found. The effective issue slot utilization is not comparable to the operation welder. In the simulations, the different combinations of benchmarks are used as multiple threads instead of using multiple threads from a single program.

In Processor Coupling [10][11], several threads are scheduled statically and interleaved into clusters at run time. A cluster consists of a set of functional units that share a register file. Operations from different threads compete for a functional unit within a cluster. Interleaving used here is very similar to the dynamic interleaver technique by Prasadh et al. Interleaving does not occur across all issue slots, i.e. if there

are two functional units with the same type, there is no operation migration from one unit to another. The issue slot utilization is much less than the operation welder. The compiler inserts explicit fork and forall operations to partition code into several parallel threads. On the other hand, the Weld architecture allows operation migration at run time and speculative threads to be spawned.

XIMD [12] 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. However, the Weld architecture merges threads at run time taking advantage of dynamic events.

6.2. Multithreading in Dynamically Scheduled Architectures

Oplinger et al. [49] shows that loop-level parallelism (cyclic code) is not sufficient enough to extract thread-level parallelism. They found that threads at procedure boundaries (acyclic code) can offer an opportunity for thread-level parallelism as good as loops.

SPSM (Single-program Speculative Multithreading) [1] speculatively spawns multiple paths in a single program and simultaneously execute those paths or threads on a superscalar core. The thread speculation is performed by inserting fork and suspend

instructions into the code. There is a main thread that can spawn many speculative threads (whereas speculative threads can also spawn speculative threads in Weld). When the main thread merges with a speculative thread, the speculative thread's state merges with the main thread. At this point, the speculative thread dies and the main thread continues. Both the main and speculative threads can execute in parallel. SPSM is, however, for dynamic (superscalar) architectures.

Dynamic Multithreading Processor (DMT) [2] is simultaneous multithreading on a superscalar core with threads created by the hardware from a single program. Each thread has its own program counter, rename tables, trace buffer, and load and store queues. All threads share the same register file, Icache, Dcache and branch predictor. Value prediction is used to predict the live-out values from one thread to another. Threads are spawned by the hardware at loop and procedure boundaries. Fully associative load and store buffers are employed for memory disambiguation. DMT is proposed for dynamically scheduled processors.

MultiScalar Processors [3] 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. In each processing unit, there are two register files [4], one is the past register set that contains register values sent by the predecessor tasks. The other one is the present register task that is the working or current register set for the unit. Two different copies of register set are needed to facilitate the maintenance in case of a task commit or squash. In Weld, once threads are created, there is no data flow between register files.

Superthreaded Architecture [5] 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. On the other hand, Weld can generate multiple speculative threads from acyclic and potentially from cyclic code and also caches and functional units are not separate but all shared.

Clustered Speculative Multithreaded Processors [6] are proposed to utilize the idle machine resources by spawning and running speculative threads from a single application. Threads are spawned by hardware and execute concurrently without any support from the compiler on a superscalar processor core. Loop iterations from SPECInt95 benchmark suite are used as threads and detected at run time whereas the compiler in Weld generates threads for a statically scheduled processor and welds operations at run time.

TME (Threaded Multiple Path Execution) [7] executes multiple alternate paths on a Simultaneous Multithreading (SMT) [8][33][34][35] superscalar processor. It uses free hardware contexts to assign paths of conditional branches. Speculative loads are allowed. The threads are created at compile time and executed in the Weld architecture.

Simultaneous Subordinate Microthreading (SSMT) [29] provides microthreads that can run simultaneously with the main thread to improve the performance of a single program. Those microthreads are stored in RAM as a microcode inside the processor. The main thread and microthreads run on a superscalar core simultaneously. Such microthreads can be branch prediction, prefetching and cache management. The compiler

inserts a spawn instruction where a particular optimization in the program needs to be performed. Similarly, assisted execution [51] creates nanothreads that can run simultaneously with the main thread at trap time. Nanothreads they used in the experiments are data prefetching handlers that are forked at data cache misses. Another use of Multithreading in a superscalar core has been performed for Exception Handling[27]. Only TLB misses are considered in this study. TLB miss handlers are implemented as separate threads. When a TLB miss is detected, the exception handler is spawned and executed. At the same time, the main thread continues to fetch instructions instead of squashing all instruction following the excepting one. When the handler completes, the main thread starts executing the excepting instruction. A similar idea of executing exception handler with the main thread is also used in M-Machine [26]. Another technique called Relational Profiling 99 provides service threads to collect profile information about the application program that runs simultaneously with the service threads.

Slipstream processors [38][39] execute two instances of a program on a chip-multiprocessor (CMP) [40][41] or a simultaneous multithreaded processor (SMT) to improve performance and provide fault tolerance. Similarly, DataScalar [56] architectures are proposed to reduce memory overheads by executing the same program on multiple processors. Each processor has on-chip memory. Some data are distributed and some are replicated so that no off-chip request is needed.

Multipath execution processors[48][42][43][44][47] explore execution of instructions from both paths a branch to reduce the branch misprediction penalty and potentially increase performance and utilization of the processor resources.

Steffan [45][46] et al. presents thread-level speculation technique for a CMP system. Essentially, the compiler creates threads speculatively and the hardware detects memory and register use violations. They extended the invalidation-based cache coherence protocol to detect violations caused by speculative memory operations. Each processing unit has a private cache and a shared second-level caches. Speculative state is kept in the first-level caches instead of a special hardware. Data forwarding between threads is performed using the memory, instead of a forwarding hardware. Similarly, Hydra CMP [52][53] presents data speculation support for a CMP processor using cache coherence protocol. Krishnan et al. [54][55] presents a speculative multithreading for CMP processors. Register transfers are performed through a bus connected to each processor. Synchronization between processors is done by the scoreboard registers. On the other hand, memory dependence violations are detected by the memory disambiguation table incorporated into L2 cache.

Data-driven Multithreading [50] pre-executes some long latency instructions that might cause branch predictions or data cache load misses as separate threads. Those threads run simultaneously with the main thread. When the main thread merges with those instructions, it does not re-execute them if they are correctly speculated.

Balasubramonian et al. [64] describes double-thread superscalar architecture model. When the primary thread stalls, the future thread starts executing speculatively. The future thread does not modify the processor state but helps data loads prefetch and the branches resolve for the primary thread.

Luk et al. [65] proposed a software-based pre-execution technique that tolerates long-latency memory operations in SMT processors. The software spawns a pre-

execution thread that executes and brings the data into cache. This reduces the data cache misses for the main thread. However, the pre-executed thread does not merge with the main thread. Its results are discarded when the thread stops. The pre-executed thread is created and killed by special instructions in the compiler.

There are other multithreaded architectures such as multiple-context processors [13][14][15][16][17][28], concurrent multithreading [18][19][37] and multithreaded processor architecture [25][30]. These in part or fully rely on dynamic instruction scheduling.

Table 6. Comparison of previous multithreading techniques with Weld

Technique	Scheduling	Cyclic/Acyclic	Thread Gen.	Spec. Thread
SPSM	Dynamic	Both	Static	Yes
DMT	Dynamic	Both	Dynamic	Yes
MultiScalar	Dynamic	Both	Static	Yes
Superthreaded Architecture	Dynamic	Cyclic	Static	No
CSMP[1]	Dynamic	Cyclic	Dynamic	Yes
TME	Dynamic	Acyclic	Dynamic	Yes
Multi-threaded RISC	Static	NA[2]	NA	No
Processor Coupling	Static	Both	Static	No
XIMD	Static	Both	Static	No
Weld	**Static**	**Acyclic[3]**	**Static**	**Yes**

A comparison of previous multithreading techniques with Weld is shown in Table 6. The first column shows the names of multithreading techniques with Weld in the last row. The comparative criterions are in the other columns as follows. The second column represents whether the processing element used in the particular multithreading technique

[1] Clustered Speculative Multithreaded Processor
[2] Different benchmarks are executed as separate threads.
[3] Potentially for cyclic code as well.

is a dynamically scheduled or statically scheduled processor. The third column shows whether threads are generated for cyclic or acyclic code. The fourth column is whether threads are generated at run time (dynamic) or at compile time (static). The last column shows whether threads can be spawned speculatively or not. A speculative thread is a thread spawned by predicting its control or value dependencies.

7. Conclusion and Future Work

In this work, a new paradigm, Welding Architecture (Weld), is proposed for VLIW processors. The Weld architecture combines the ISA, compiler and hardware for multithreading support. Threads, which are acyclic treegions in the control graph, are created by the compiler with the help of bork instructions. At run time, threads are welded to fill in the holes by special hardware called the operation welder. Thread synchronization and bookkeeping are performed by the hardware. The preliminary results showed that two thread or dual-thread model performs as well as the models with more than two threads with less amount of hardware.

In the second part of the dissertation, we focus on the dual-thread Weld architecture model. A various hardware parameters are modified such as speculative memory operation (SMOB) and speculative store buffer (SSB) sizes, branch and second-level cache miss penalties and register file copy penalty. Also, a perfect value predictor is applied to measure how the performance changes when the bork operations are scheduled in the earliest cycle possible. The performance results are not promising if a perfect value predictor is used to value-speculate all live-out operations between two threads. The performance improvement would be negligible if a practical value predictor is used because it would incur some thread squashes due to value mispredictions. An almost software-only approach is used to investigate the effects of eliminating the hardware SMOB. Instead of using the SMOB, the compiler algorithm is modified such that all load-store conflicts between the threads are guaranteed not to occur at run time. We concluded that the performance increases when the hardware SMOB is eliminated

because all SMOB-related squashes that can consume a great deal of useful cycles in the processor are eliminated as well.

A possible architectural/compiler future work would be the investigation of eliminating the hardware SSB and guaranteeing at compile time that no store-store conflict would occur at run time. The performance may reflect a minor slowdown without the SSB while saving a precious chip area, particularly in the embedded processors.

Another possible future work is to insert bork operations while taking the profile information into consideration. The algorithm inserts a bork operation on highly executed paths instead of all paths. This would make the algorithm more robust and efficient in terms of code size and reduce the wasted cycles due to bork mispredictions.

References

[1] 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.

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

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

[4] S. E. Breach, T.N. Vijaykumar and G. S. Sohi, "The Anatomy of the Register File in a Multiscalar Processor", in *Proc. 27tht Ann. Int'l Symp. Microarchitecture,* San Jose, CA, Dec. 1994.

[5] 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.

[6] P. Marcuello and A. González, " Clustered Speculative Multithreaded Processors," in *Proc. of the Int'l Conf. on Supercomputing*, Rhodes, Greece, June 1999.

[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] S. W. Keckler and W. J. Dally, “Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism,” in *Proc. 19th Ann. Int’l Symp. Computer Architecture,* Australia, May 1992.

[11] M. Fillo, S. W. Keckler, W. J. Dally, N.P. Carter, A. Chang, Y. Gurevich and W.S. Lee, ”The M-Machine Multicomputer,” in *Proc.28th Ann. Int’l Symp. Microarchitecture*, Ann Arbor, MI, Dec. 1995.

[12] 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.

[13] W.Weber and A. Gupta, “Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results,” in *Proc. 16th Ann. Int’l Symp. Computer Architecture*, Jerusalem Israel, May 1989.

[14] J. Laudon, A. Gupta and M. Horowitz, “Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations” in *Proc. 6th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1994.

[15] A. Agarwal, B.-H. Lim, D. Kranz and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," in *Proc. 17th Ann. Int'l Symp. Computer Architecture*, Seattle, WA, May 1990.

[16] B. J. Smith, "A Pipelined Shared Resource MIMD Computer," in *Proc. of the Int'l Conf. on Parallel Processing*, 1978.

[17] M. K. Farrens and A. Pleszkun, "Strategies for Achieving Improved Processor Throughput," in *Proc. 18th Ann. Int'l Symp. Computer Architecture*, Toronto, Canada, May 1991.

[18] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," in *Proc. 19th Ann. Int'l Symp. Computer Architecture*, Gold Coast, Australia, May 1992.

[19] G. E. Daddis and H. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors," in *Proc. of the Int'l Conf. on Parallel Processing*, Aug. 1991.

[20] W. A. Havanki, "Treegion Scheduling for VLIW Processors", *Master's Thesis*, Dept. of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911, July 1997.

[21] W. A. Havanki, S. Banerjia and T. M. Conte, "Treegion Scheduling for Wide-issue Processors", in *Proc. 4th Int'l Symp. High Performance Computer Architecture*, Las Vegas NV, Feb. 1998.

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

[23] Intel Corporation, "IA-64 Application Developer's Architecture Guide", May 1999.

[24] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", in *IEEE Trans. on Computers*, Vol. 37, NO. 5, May 1988.

[25] R. Govindarajan S. S. Nemawarkar and P. LeNir, "Design and Evaluation of a Multithreaded Architecture", in *Proc 1st on High-Performance Computer Architecture*, Raleigh, NC, Jan. 1995.

[26] S. W. Keckler, A. Chang, W. S. Lee, S. Chatterjee and W. J. Dally, "Concurrent Event Handling through Multithreading", *IEEE Trans. on Computers*, Vol. 48, No. 9, September 1999.

[27] C. B. Zilles, J. S. Emer and G. S. Sohi, "The Use of Multithreading for Exception Handling", in *Proc. 32nd Ann. Int'l Symp. Microarchitecture*, Haifa, Israel, Nov. 1999.

[28] J. Laudon, A. Gupta and M. Horowitz, "Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors", Technical Report, CSL-TR-92-523, Computer Systems Lab., Stanford University, May 1992.

[29] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt and Y. N. Patt, "Simultaneous Subordinate Microthreading (SSMT), in *Proc. 26th Ann. Int'l Symp. Computer Architecture*, Atlanta, GA, May 1999.

[30] M. Loikkanen and N. Bagherzadeh, "A Fine-Grain Multithreading Superscalar Architecture", in *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques' 96*, October 1996.

[31] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-grain Parallelism", *Proc. 19th Ann. Int'l Symp. Computer Architecture*, Gold Coast, Australia, May 1992.

[32] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References", in *IEEE Trans. on Computers*, May 1996.

[33] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J.L. Lo and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", *Proc. 23rd Ann. Int'l Symp. Computer Architecture*, Philadelphia, PA, May 1996.

[34] J. L. Lo, S. J. Eggers, H. M. Levy, Sujay S. Parekh and D. M. Tullsen, "Tuning Compiler Optimizations for Simultaneous Multithreading", *Proc. 30th Ann. Int'l Symp. Microarchitecture*, Raleigh, NC, Dec. 1997.

[35] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm and D. M. Tullsen, "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading", in *ACM Transactions on Computer Systems*, August 1997.

[36] T. Heil and J. E. Smith, "Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines", *Proc. 33rd Ann. Int'l Symp. Microarchitecture*, Monterey, CA, Dec. 2000.

[37] W. Yamamoto and M. Nemirovsky, "Increasing Superscalar Performance through Multistreaming", *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques*. (Cyprus). June 1995.

[38] K. Sundaramoorthy, Z. Purser and E. Rotenberg, “Slipstream Processors: Improving both Performance and Fault Tolerance”, *Proc. 9th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[39] Z. Purser, K. Sundaramoorthy and E. Rotenberg, “A Study of Slipstream Processors”, *Proc. 33rd Ann. Int’l Symp. Microarchitecture*, Monterey, CA, Dec. 2000.

[40] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson and K. Chang, “The Case for a Single-Chip Multiprocessor”, *Proc. 7th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Oct.. 1996.

[41] L. Hammond, B. A. Nayfeh and K. Olukotun, “A Single-Chip Multiprocessor”, *IEEE Computer*, Sep. 1997.

[42] S. Wallace, D. M. Tullsen and B. Calder, “Instruction Recycling on a Multiple-Path Processor”, *Proc 5th on High-Performance Computer Architecture*, Orlando, FL, Jan., 1999.

[43] A. Klauser, A. Paithankar and D. Grunwald, “Selective Eager Execution on the PolyPath Architecture”, in *Proc. 25th Ann. Int’l Symp. Computer Architecture*, Barcelona, Spain, June 1998.

[44] A. Klauser and D. Grunwald, “Instruction Fetch Mechanisms for Multipath Execution Processors”, *Proc. 32nd Ann. Int’l Symp. Microarchitecture*, Haifa, Israel, Nov. 1999.

[45] J. G. Steffan and T. C. Mowry, “The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization”, *Proc 5th on High-Performance Computer Architecture*, Las Vegas, Nevada, Feb., 1998.

[46] J. G. Steffan, C. B. Colohan, A. Zhai and T. C. Mowry, “A Scalable Approach to Thread-Level Speculation”, in *Proc. 27th Ann. Int’l Symp. Computer Architecture*, Vancouver, Canada, June 2000.

[47] P. S. Ahuja, K. Skadron, M. Martonosi and D. W. Clark, “Multipath Execution: Opportunities and Limits”, in *Proc. of the Int’l Conf. on Supercomputing*, Melbourne, Australia, July 1998.

[48] A. K. Uht and V. Sindagi, “Disjoint Eager Execution: An Optimal Form of Speculative Execution”, in *Proc. 28th Ann. Int’l Symp. Microarchitecture*, Ann Arbor, MI, Dec. 1995.

[49] J. T. Oplinger, D. L. Heine and M. S. Lam, “In Search of Speculative Thread-Level Parallelism”, *Proc. Int’l Conf. Parallel Architecture and Compilation Techniques*, Newport Beach, CA, Oct., 1999.

[50] A. Roth and G. S. Sohi, “Speculative Data-Driven Multithreading”, *Proc 6th on High-Performance Computer Architecture*, Toulouse, France, Jan, 2000.

[51] M. Dubois and Y. H. Song, ”Assisted Execution”, CENG Technical Report 98-25, Department of Electrical Engineering, University of Southern California, Los Angeles, CA, Oct. 1998.

[52] L. Hammond, M. Willey and K. Olukotun, “Data Speculation for a Chip Multiprocessor”, *Proc. 8th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.

[53] K. Olukotun, L. Hammond and M. Willey, “Improving the Performance of Speculatively Parallel Applications on the Hydra CMP”, *Proc. of the Int’l Conf. on Supercomputing*, Rhodes, Greece, June 1999.

[54] V. Krishnan and J. Torrellas, ”Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor”, in *Proc. of the Int’l Conf. on Supercomputing*, Melbourne, Australia, July 1998.

[55] V. Krishnan and J. Torrellas, “A Chip-Multiprocessor Architecture with Speculative Multithreading”, in *IEEE Trans. on Computers*, Sep., 1999.

[56] D. Burger, S. Kaxiras and J. R. Goodman, “DataScalar Architectures”, in *Proc. 24th Ann. Int’l Symp. Computer Architecture*, Denver, CO, June 1997.

[57] M. S. Schlansker and B. R. Rau, “EPIC: Explicitly Parallel Instruction Computing”, *IEEE Computer*, Feb. 2000.

[58] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes and S. W. Sathaye, “Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings”, in *Proc. 29th Ann. Int’l Symp. Microarchitecture*, Paris, France, Dec. 1996.

[59] S. Gopal, T. N. Vijaykumar, J. E. Smith and G. S. Sohi, “Speculative Versioning Cache”, *Proc 5th on High-Performance Computer Architecture*, Las Vegas, Nevada, Feb., 1998.

[60] A. S. Huang, G. Slavenburg and J. P. Shen, “Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation”, *Proc. 21st Ann. Int’l Symp. Computer Architecture*, Chicago, IL, April 1994.

[61] M. Moudgill and J. H. Moreno, "Run-time Detection and Recovery from Incorrectly Reordered Memory Operations", *Research Report*, IBM Research Division, T. J. Watson Research Center, N. Y., 1997.

[62] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, W. W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer", " in *Proc. 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1994.

[63] A. Nicolau, "Run-time Disambiguation: Coping with Statically Unpredictable Dependencies", *IEEE Trans. on Computers*, Vol.38, No. 5, May 1989.

[64] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically Allocating Processor Resources between Nearby and Distant ILP", in *Proc. 28th Ann. Int'l Symp. Computer Architecture,* Goteborg, Sweden, June 2001.

[65] C. –K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", *Proc. 28th Ann. Int'l Symp. Computer Architecture,* Goteborg, Sweden, June 2001.

[66] G. R. Beck, D.W.L. Yen, T.L. Anderson," The Cydra 5 Minisupercomputer: Architecture and Implementation," *The Journal of Supercomputing*, 1993.

[67] B.R. Rau, D.W.L. Yen, W. Yen, and R. A. Towle," The Cydra 5 Departmental Supercomputer," *IEEE Computer*, January 1989.

[68] *TMS320C62XX CPU and Instruction Set Reference Guide*, Texas Instruments, July 1997.

[69] *TM 1000 Preliminary Data Book*, Philips Electronics North America Corporation, 1997.

[70] P. Kalapathy, “Hardware/Software Interactions on the Mpact,” *IEEE Micro*, Mar./Apr. 1997.

[71] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Popworth, and P.K. Rodman, " A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. on Comp.* vol.37, No.8, August 1988.

[72] P. G. Lowney, S. M. Freundenberger, T. J. Kaezes, W.D. Lichtenstein, R. P. Nix, J.S. O'Donnell, and J. C. Ruttenberg, " The Multiflow Trace Scheduling Compiler", *The Journal of Supercomputing*, 7, 1993.

[73] W. W. Hwu and Y. N. Patt, “Checkpoint Repair for High-Performance Out-of-Order Execution Machines”, *IEEE Trans. on Computers,* Vol. C-36, NO. 12, Dec. 1987.

[74] K. Ebcioglu and E. R. Altman, “DAISY: Dynamic Compilation for 100% Architectural Compatibility”, Research Report RC 20538, IBM Research, August 1996.

[75] T. M. Conte and S. W. Sathaye, “Dynamic Rescheduling: A Technique for Object-code Compatibility in VLIW architecture”, in *Proc. 28th Ann. Int’l Symp. Microarchitecture*, Ann Arbor Michigan, Nov. 1995.

[76] M. Tremblay, “A Microprocessor Architecture for The New Millennium,” *Hot Chips 11,* Palo Alto, CA, August 1999.

[77] Crusoe™, http://www.transmeta.com.

[78] Itanium, “Intel Itanium Processor at 800MHZ and 733MHZ Data Sheet”, May 2001.

[79] T. Sukemura, “FR500 VLIW-architecture High-performance Embedded Microprocessor”, *FUJITSU Sci. Tech. J., Vol 36, 1,* June 2000.

[80] StarCore SC100, http://www.starcore-dsp.com.