Weld: A Multithreading Technique Towards Latency-tolerant VLIW Processors

Emre Özer, Thomas M. Conte and Saurabh Sharma
{eozer,conte,ssharma2}.eos.ncsu.edu

Department of Electrical and Computer Engineering,
North Carolina State University, Raleigh, N.C., 27695.

Abstract. This paper presents a new architecture model, named *Weld*, for VLIW processors. Weld integrates multithreading support into a VLIW processor to hide run-time latency effects that cannot be determined by the compiler. It does this through a novel 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 fetch units are duplicated to support multithreading. The experimental results show that the Weld architecture attains a maximum of 27% speedup as compared to a single-threaded VLIW architecture.

1. Introduction

Variable memory latencies in VLIW processors are one of the most crucial problems that VLIW processors face. A VLIW processor might have to stall for a large number of cycles until the memory system can return 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, and exceptions. It has also been used to improve single program performance by spawning threads within a program, such as loop iterations or acyclic pieces of code. Weld is a new architecture model that uses compiler support for multithreading within a single program to combat variable latencies due to unpredictable run-time events in VLIW architectures.

There are two main goals that Weld aims to achieve: 1) better utilization of processor resources during unpredictable run-time events and 2) dynamic filling of issue slots that cannot be filled at compile time. Unpredictable events that cannot be detected at compile time, such as cache misses, may stall a VLIW processor for numerous cycles. The Weld architecture tolerates those wasted cycles by issuing operations from other active threads when the processor stalls for an unpredictable run-time event within the main thread. VLIW processors are also limited by the fact that they use a discrete scheduling window. VLIW compilers partition the program into several scheduling regions and each scheduling region is scheduled by applying different *instruction-level parallelism* (ILP) optimization techniques such as speculation, predication, loop unrolling, etc. The VLIW compiler cannot fill all schedule slots in every MultiOp [12][1] because the compiler cannot migrate operations from different scheduling regions. A hardware mechanism called the *operation welder* is introduced in this work to achieve our second goal. It merges operations

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

from different threads in the issue buffer during each cycle to eliminate empty issue slots, or NOPs. A scheduling region is a potential thread in our model. Executing operations from different scheduling regions and filling the issue slots from these regions simultaneously at run time can increase resource utilization and performance. The main objective of this study is to increase ILP using compiler-directed threads (scheduling regions) in a multithreaded VLIW architecture.

The remainder of this paper is organized as follows. Section 2 introduces the general Weld architecture. Section 3 explains the *bork insertion algorithm*, which defines how the compiler spawns new threads. Section 4 presents performance results and analysis. Section 5 discusses the related work in multithreading techniques. Finally, Section 6 concludes the paper and discusses future work.

2. The General Weld Architecture Model

The Weld model assumes that threads are generated from a single program by the compiler. During program execution there is a single main thread and potentially several speculative threads, but the main thread has the highest priority among all threads. The general Weld architecture is shown in Figure 1. Each thread has its own program counter, fetch unit and register file while all threads share the branch predictor and instruction and data caches. Weld consists of a basic 5-stage VLIW pipeline. The fetch stage fetches MultiOps from the Icache, and the decode/weld stage decodes and welds them together. The operation welder is

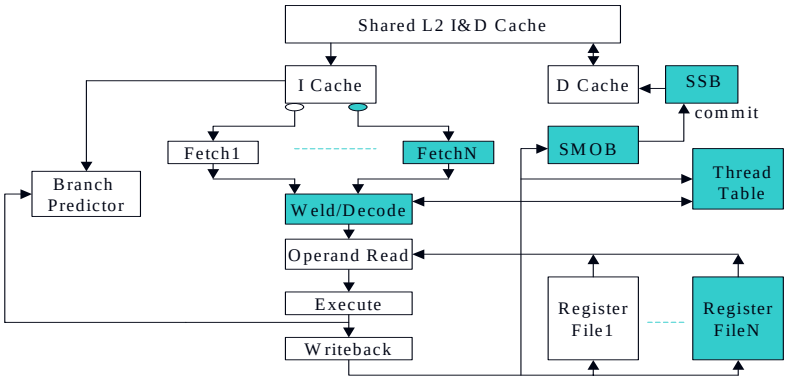

Figure 1. The general Weld architecture

integrated into the decode stage in the pipeline. 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 the write-back stage writes the results into the register file and Dcache.

2.1. ISA Extension

A new instruction and some extensions to the ISA are required to support multithreading in a VLIW. A new instruction is needed to spawn threads. A branch and fork operation, or *bork*, is introduced to spawn new threads and create new hardware contexts for those threads. It has a target address, which is the address of the new, speculative thread.

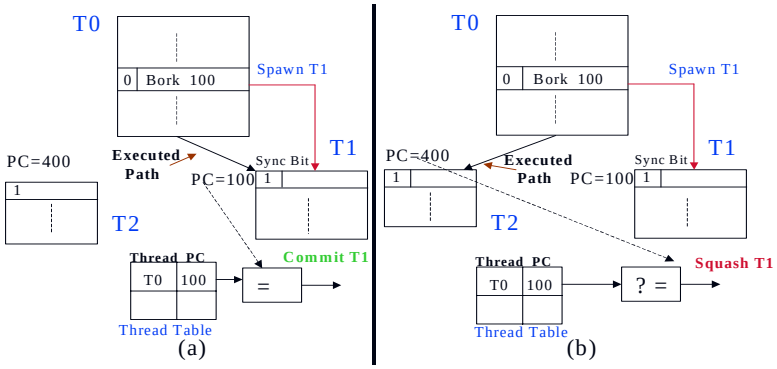

Figure 2. An example of synchronization of two threads

Two extra bits are added to each MultiOp in the ISA: *separability* and *synchronization* bits. A separability bit is necessary for each MultiOp to distinguish between separable and inseparable MultiOps. In a separable MultiOp, the individual operations that form the MultiOp can be issued across multiple cycles without violating dependencies between those operations. 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 program. The synchronization bit is added to each MultiOp in the ISA and is set in the first MultiOp of each thread at compile-time to help synchronize threads at run-time. With the combination of ISA and hardware support, threads can merge in a straightforward way as explained in Section 2.2.

2.2. Thread Creation and Synchronization

When a bork is executed, a new hardware context (register file, fetch unit and program counter) is assigned to the new thread if those resources are available. If there is not an available hardware context for the new thread, the bork behaves like a NOP. Once the new hardware context has been created, the register file of the ancestor thread is copied into the register file of the descendant thread, and the program counter of the descendant thread is initialized with the target address of the bork. Moreover, the target address is also stored in the Thread Table for thread synchronization. Speculative threads can then spawn other speculative threads, and so on. However, a thread (main or speculative) may spawn only one thread, i.e. the compiler guarantees that there is only one bork operation per executed path in order to reduce the complexity of thread synchronization. If there is a stall in 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 fill from multiple threads by taking advantage of any empty fields in the MultiOps.

An ancestor thread merges with its own descendant thread when the ancestor fetches the first MultiOp instruction in the descendant thread. The fetch unit checks if the synchronization bit of the fetched MultiOp 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. The ancestor thread dies and the descendant thread takes over in case of a commit. If the

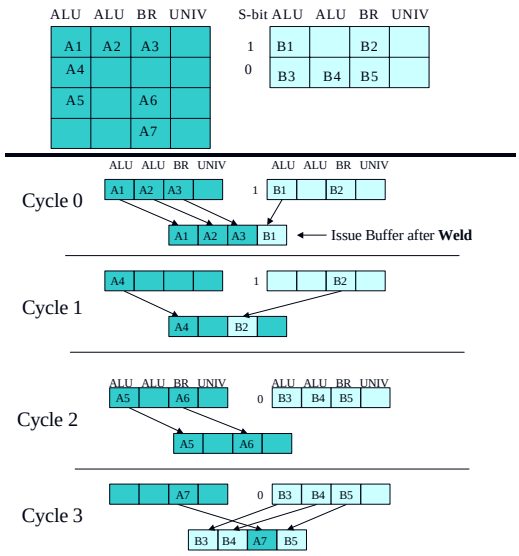

Figure 3. The example for the operation welder and its cycle steps

addresses do not match, then the descendant thread is incorrectly speculated. In this case, the descendant thread and its own descendant threads must be squashed. Also, the speculative stores in the Speculative Store Buffer (SSB) and the speculative loads in the Speculative Memory Operation Buffer (SMOB) are invalidated. Figure 2a shows an example of synchronization of two threads. Thread 0 (T0) spawns Thread 1 (T1) at address 100. 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 MultiOp, which is the MultiOp at address 100. The fetch unit for T0 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, then T1 was misspeculated, as shown in Figure 2b. The PC address 400 is compared with the address 100 in the Thread Table. There is no match, so T1 must be squashed.

2.3. Operation Welder

The 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 the 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. Each thread has a buffer called the *prepare-to-weld buffer* to hold a MultiOp. At each cycle, a MultiOp is sent from the fetch buffer to the weld/decode stage. Each operation has an empty/full bit that states whether an operation exists in the slot, and each MultiOp also has a separability bit. The operation welder consists of an array of multiplexers to forward the operations to the functional units. Control logic takes the empty/full and separability bits and sends control signals to the multiplexers. Each functional unit has a multiplexer in front of it. In the Weld architectural model, the main thread has the highest priority. Among the speculative threads, older threads have higher priority than younger ones. All operations in the main thread are always forwarded to the issue buffer. The empty slots in the issue buffer are filled from operations in the speculative threads. Some operations in a MultiOp of a speculative thread may be sent to the functional units while the remaining operations stay in the prepare-to-weld buffer. In this case, the fetch from this thread is stalled until the remaining operations in the buffer are issued.

The example in Figure 3 demonstrates how the operation welder works. There are two threads scheduled for a 4-issue VLIW machine. Two ALU units, one branch unit and one universal functional unit are used in the example architecture. The thread on the right shows the speculative thread and the separability bit for each of its MultiOps. 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 slot 4 of the issue buffer. The remaining operations stay in the prepare-to-weld buffer of B. At cycle 1, the second MultiOp from A is forwarded into the issue buffer. B2 is welded into slot number 3 of the issue buffer. At cycle 2, thread A puts its third MultiOp and thread B puts its second MultiOp into their respective prepare-to-weld buffers. Again, thread A forwards all operations into the issue buffer. However, there can be no welding with 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 that have been spawned and merged. There is an entry for each active thread. Each entry keeps a thread id, register file id, and 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. The

time stamp represents the age of the threads. Each time a new thread is created, the counter is incremented by one and stamped into the thread and written into the TT. The time-stamp counter must be wide enough so as not to overflow. Otherwise, the time order of the threads would be disrupted. The time-stamp counter is reset to zero when there is only one active thread, which is the main thread. If the counter overflows while there are active threads, all of the speculative threads are squashed and the counter is reset to zero. The register file id is the identifier of the register file assigned to the thread. The time-stamp id is also attached to each operation in a thread to distinguish operations from different threads. When an operation completes, it searches the TT to find the proper register file id by comparing the attached time stamp with time stamps in the table. The TT can be designed as a shift register. When two threads merge, the ancestor thread dies and the descendant thread takes over. This involves deleting the ancestor thread's entry from the 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.

2.5. Speculative Memory Operation Buffer (SMOB) and Speculative Store Buffer (SSB)

Load operations from speculative threads are kept in a buffer called the *speculative memory operation buffer* (SMOB). The operating principle of the SMOB is similar to the ARB [14][15]. However, it is different from the ARB in the sense that only load operations are kept in the SMOB. It uses a single shared fully associative buffer to resolve run-time load/store violations. A SMOB entry contains the speculative load memory address, the speculative thread's time stamp, and a valid bit. An outstanding speculative/nonspeculative store memory address together with the time stamp 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) for a conflict. 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 larger are invalidated in the SMOB. The main thread re-executes all instructions in the speculative thread where a thread misspeculation has occurred.

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 store operations are executed. Speculative threads write store values in the SSB until commit time. Each SSB entry contains a memory address, store value, a next link pointer, store write time and a valid bit. 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*. The time stamp (i.e. thread id) of a store operation decides how to access the correct pointer set (i.e. the head and tail pointers). When a speculative store operation executes, a search is made to find an available entry in the SSB. If one exists, the entry is allocated and the 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. 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. 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 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.

3. Bork Insertion Algorithm

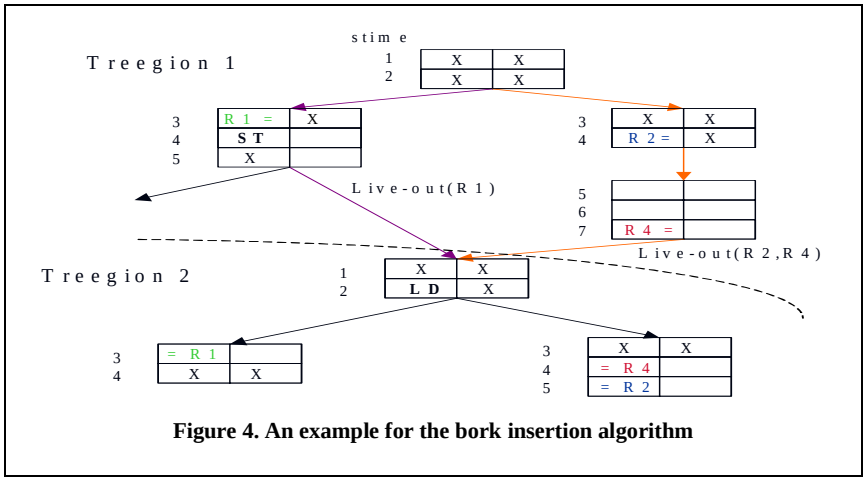

Figure 4. An example for the bork insertion algorithm

The bork insertion algorithm determines the earliest schedule cycle that the compiler can schedule a bork operation in the code and spawn a new thread. The bork insertion algorithm was implemented within the LEGO experimental compiler[10], which uses treegion scheduling [11] as a global, acyclic scheduling algorithm. 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 above dependent branches. Load and store operations are not allowed to be speculated during scheduling, but other operations can be hoisted up to any basic block within the treegion. Once treegions are formed, the scheduler schedules each treegion separately. After the program is scheduled, register allocation is performed 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 in an attempt to fully overlap treegions. True data dependencies between two treegions are considered when inserting borks. There is only one bork per path allowed in each treegion, enforcing the rule that each thread may spawn at most one speculative thread.

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 later bork(s) in the schedule are removed. The elimination phase of redundant borks is performed after insertion of borks for each treegion. Figure 4 shows an example of bork insertion. 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 are denoted by 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. An available slot is found at cycle 4 in slot 2. 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.

4. Performance Evaluation

A trace-driven simulator was written to simulate multiple threads running at the same time. 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, 128-entry SMOB and 64-entry SSB, 1-cycle L1 cache hit time, 10-cycle L2 hit time, 30-cycle L2 miss time, 5-cycle thread squash penalty time are assumed in the simulations. ALU, BR, BORK, ST and floating-point ADD take one cycle. LD takes two cycles and finally floating-point multiplication and division take three cycles. 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. Universal units can execute any type of instructions and ALU/BR units can execute only ALU and branch instructions. The SPECint95 benchmark suite is used for all runs. 100 million

instructions were executed using the training inputs. 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 thread models in the experiments.

Figure 5 shows speedup results for Weld models consisting of one to six threads. As shown in the graph, an average speedup of 23% is attained with two threads over all benchmarks. With three threads, it is 26%. After three threads, the speedup stabilizes at 27%. There is little change in speedup as the number of threads increases beyond three threads. This is because the number of penalty cycles increases with the number of threads. As the number of threads increases, the SMOB and SSB fill up more quickly and stall the speculative threads. Also, the chances for SMOB conflicts increase 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, the two-thread model gives the best performance. As the number of threads increases beyond two threads, the stalls due to SMOB conflicts take over and cancel out the benefit gained from multithreading. The same effect can be observed in 134.perl where the six-thread model is worse than the four- and five-thread models. This is because the number of SMOB conflicts in the four- and five-thread models is much less than the number of conflicts in the six-thread model. Therefore, the five-thread model can go ahead in time and finish earlier.

Figure 6 depicts speedup results of a model that uses the operation welder versus one that does not. Not using the welder implies that a speculative thread can only issue when the parent stalls. With two threads, the operation welder gives a 5% speedup over all the benchmarks. With three threads and beyond, the speedup is 8.5%. An increase in the number of threads increases the chances of welding operations from many speculative threads. However, in 132.ijpeg, the opposite effect can be observed because the number of SMOB stalls beyond two threads is so high that the benefit from welding is hidden behind the SMOB squashes.

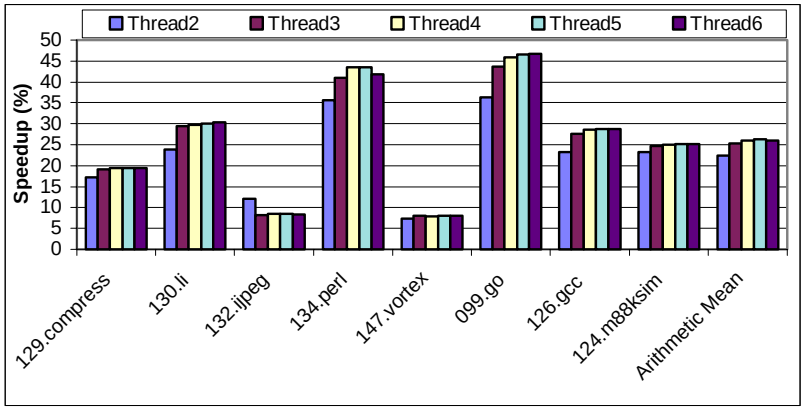

Figure 5. Speedup results

As seen from the experimental results, the models with three or more threads perform equally well. On the other hand, the two-thread model has performance of

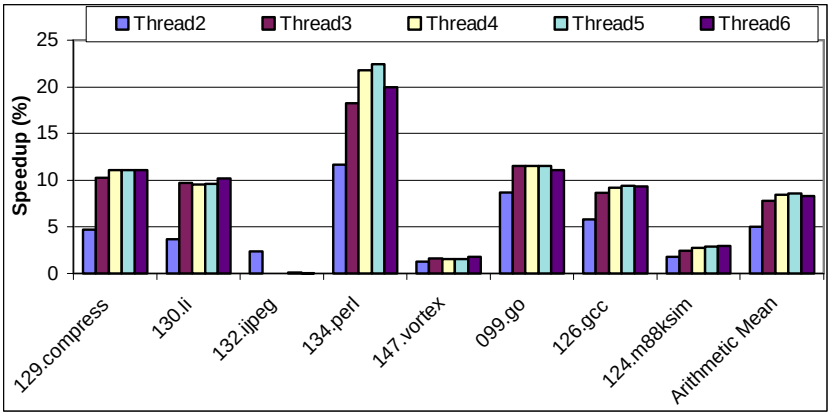

Figure 6. Speedup results of a model with welder vs. without welder

only 3-4% less in speedup than the other thread models, but with a much simpler register file organization, thread synchronization mechanism and less complicated SMOB and SSB.

5. Related Work

SPSM (Single-program Speculative Multithreading) [1] speculatively spawns multiple paths in a single program and simultaneously executes those paths or threads on a superscalar core. In SPSM 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. SPSM is, however, for dynamic (superscalar) architectures.

Dynamic Multithreading Processors (DMT) [2] provide 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. 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. In Weld, threads share the functional units and caches. Each MultiScalar 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. As in SPSM and DMT, MultiScalar is proposed for dynamically scheduled processors.

TME (Threaded Multiple Path Execution) [4] executes multiple alternate paths on a Simultaneous Multithreading (SMT) [5] superscalar processor. It uses free hardware contexts to assign paths of conditional branches. Speculative loads are allowed. In contrast, threads are created at compile time in Weld.

Prasadh [6] et al. proposed 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. Weld has neither of these limitations. Also, threads are the different benchmarks or programs in their experiments unlike multithreading from a single program in Weld.

In Processor Coupling [7][8], 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 does not occur across all issue slots. The compiler inserts explicit *fork* and *forall* operations to partition code into several parallel threads. On the other hand, Weld allows operation migration at run time and speculative threads to be spawned.

XIMD [9] is a VLIW-like architecture that has multiple functional units and a large global register file similar to VLIW. 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, Weld merges threads at run time by taking advantage of dynamic events.

6. Conclusion and Future Work

In this paper, a new architecture model called *Weld* is proposed for VLIW processors. Weld exploits the ISA, compiler, and hardware to provide multithreading support. The compiler, through the insertion of bork instructions, creates multiple threads from a single program, which are acyclic regions of the control graph. At run time, threads are welded to fill in the holes by special hardware called the *operation welder*. The experimental results show that a maximum of 27% speedup using Weld is possible as compared to a single-threaded VLIW processor.

We will focus on the dual-thread Weld model for further research. The issues such as the sizes of SMOB and SSB on performance, variable memory latencies, and higher branch penalty in deeper pipelines will be studied for the dual-thread model. We are also working on a compiler model for Weld without the SMOB in order to avoid squashes due to load/store conflicts at the thread level.

Acknowledgements

This research was supported by generous hardware and cash donations from Sun Microsystems and Intel Corporation.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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