High-Performance and Low-Cost Dual-Thread VLIW Processor Using Weld Architecture Paradigm

Emre Özer and Thomas M. Conte, *Fellow*, *IEEE*

Abstract—This paper presents a cost-effective and high-performance dual-thread VLIW processor model. The dual-thread VLIW processor model is a low-cost subset of the Weld architecture paradigm. It supports one main thread and one speculative thread running simultaneously in a VLIW processor with a register file and a fetch unit per thread along with memory disambiguation hardware for speculative load and store operations. This paper analyzes the performance impact of the dual-thread VLIW processor, which includes analysis of migrating disambiguation hardware for speculative load operations to the compiler and of the sensitivity of the model to the variation of branch misprediction, second-level cache miss penalties, and register file copy time. Up to 34 percent improvement in performance can be attained using the dual-thread VLIW processor when compared to a single-threaded VLIW processor model.

Index Terms—Multithreaded processors, VLIW architectures, modeling of computer architecture.

1 Introduction

VLIW architectures emerged in both general-purpose and embedded/DSP processor markets such as the *Intel/HP Itanium* [23], *Lx* VLIW architecture [28] from *HP/STMicroelectronics*, *Sun Microsystems MAJC* [21], *Transmeta Crusoe* [22], the *Texas Instruments* 320C6x family [26], and *TriMedia TM-1* [27], *Fujitsu FR500* [24], and *Star*Core SC140* [25]. However, as in most instruction-level parallel (ILP) architectures, but perhaps to a greater degree, the performance of VLIW architectures suffers from penalties resulting from unpredictable runtime events.

The general *Weld* architectural model is proposed as a statically scheduled, horizontal architecture that supports the execution of multiple, simultaneously active threads from a single program [18]. It runs a single application with a single OS context and compiler-inserted thread creation with three primary goals in mind: 1) to provide tolerance for unpredictable runtime events such as cache misses and branch mispredictions, 2) to dynamically fill issue slots left empty by the compiler in order to increase ILP, and 3) to maintain the hardware simplicity inherent to VLIW architectures while meeting the previous objectives. It differs from previous multithreading techniques [6], [8] developed for VLIW architectures in three ways. First, it uses scheduling regions as the thread entity to enable the full overlap of two regions at runtime. The compiler uses control-flow and liveness analysis to determine when to spawn speculative threads, which results in more efficient use of issue slots across region boundaries. Second, threads are control-speculative, but we explore whether or not they also need to be memory data-speculative. Third, once a speculative thread is created, it does not communicate register values with the primary thread. The compiler guarantees that no interthread register transfer occurs after the thread split point. Leveraging the compiler in this way reduces hardware support requirements.

The design of the general *Weld* architecture involves changes to the ISA, compiler support for effective generation of multiple threads, and additional hardware resources such as multiple register files, buffers for speculative load and store instructions, a thread synchronization hardware and an operation welder—a hardware structure that dynamically fills issue slots from multiple threads—to enable multithreading. The primary change in the ISA is the introduction of a new instruction, called a *branch-and-fork*, or *bork*, instruction for spawning speculative threads. The compiler inserts *bork* instructions along individual control-flow paths within scheduling regions. The *borks* spawn speculative scheduling regions as speculative threads at the safest point and without the need to hardware dependence checking. In addition to the introduction of the *bork* instruction, two new bits (*synchronization* and *separability* bits) must be added to the VLIW ISA.

The VLIW processors using the *Weld* paradigm supporting an arbitrary number of threads are excessively hardware intensive [18]. The cost analysis of the *Weld* architecture with varying thread support showed that supporting as few as two threads was sufficient, but at a better cost/performance. In this paper, we focus on the dual-thread *Weld* model because it has the potential of simplifying the design of speculative load and store buffers, thread synchronization hardware, register files, and the

- *E. Özer is with ARM Ltd., 110 Fulbourn Rd., Cambridge, CB1 9NJ, UK. E-mail: emre.ozer@arm.com.*
- *T.M. Conte is with the Department of Electrical and Computer Engineering, North Carolina State University, Box 7911, Raleigh, NC. E-mail: conte@ncsu.edu.*

Manuscript received 7 Dec. 2004; revised 11 Feb. 2005; accepted 4 Mar. 2005; published online 20 Oct. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0300-1204.

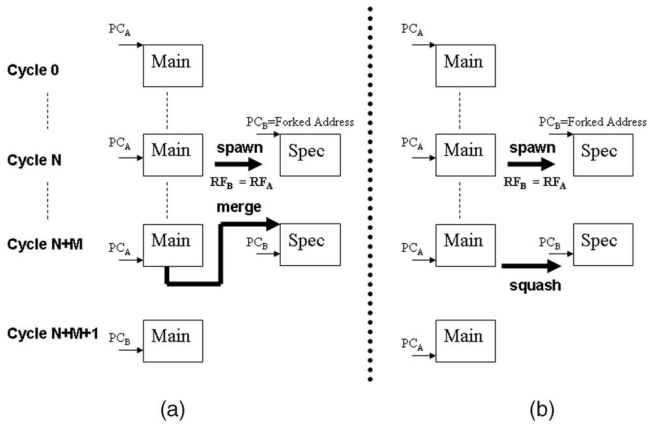

Fig. 1. High-level picture of the dual-thread Weld speculative multithreading model.

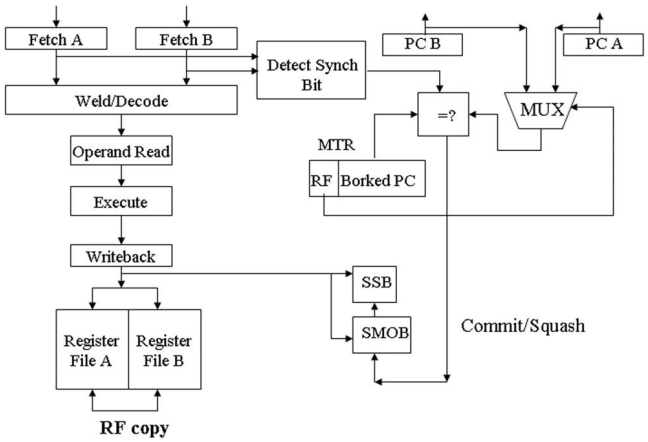

Fig. 2. Dual-thread *Weld* microarchitecture.

operation welder without sacrificing significant performance. Further, we propose a lower cost model of the dual-thread *Weld* model that does not use memory disambiguation hardware for speculative loads.

The remainder of the paper is organized as follows: Section 2 analyzes the microarchitectural and compilation issues of the dual-thread VLIW processor model using the *Weld* architecture. Section 3 presents the experimental framework and initial performance results, and then investigates the viability of a dual-thread VLIW processor without memory disambiguation hardware for speculative loads and its impact on performance. Section 4 discusses the related work. Finally, Section 5 concludes the paper.

2 Dual-Thread VLIW Processor Model

The dual-thread *Weld* model is a two-thread VLIW processor model that employs speculative multithreading from the same application in a single processor core. Speculative threads are selected and marked at compile time and spawned at runtime. One thread is always the main or nonspeculative thread and the other is the speculative thread. The main thread spawns the speculative thread by executing a *bork* instruction, continues its execution, and merges with the speculative thread by reaching to the start address of the speculative thread.

Fig. 1 shows a high-level picture of the dual-thread *Weld* speculative multithreading model. The main thread spawns a speculative thread somewhere in the middle of its code by executing a *bork* instruction at cycle N. Executing the bork initiates *en masse* copy of the register file of the main thread into the register file assigned to the speculative thread. At the same time, the *bork*ed address is written into the program counter $\mathrm{PC_B}$ of the speculative thread. In cycle $N + M$, the main thread reaches the point where it either squashes the speculative thread or makes it the main thread. The main thread merges with the speculative one when its next fetch address is the same as the beginning address of the speculative thread as shown in Fig. 1a (i.e., correct speculation). At this point, the main thread dies and the speculative thread becomes the new main thread. In the next cycle, the execution continues from the address pointed by $\mathrm{PC_B}$. On the other hand, a thread misspeculation occurs if the next fetch address does not match the starting address of the speculative thread as shown in Fig. 1b. In this case, the speculative thread is squashed, and the main thread continues its execution at the address pointed by its $\mathrm{PC_A}$ in the next cycle.

2.1 Microarchitecture

The microarchitecture of the dual-thread Weld is shown in Fig. 2. Each thread has its own program counter, fetch unit, and register file, while both threads share the branch predictor, instruction, and data caches. The instruction cache needs two read ports to service two different simultaneous cache requests, i.e., one for each thread. The fetch stage fetches MultiOps[1] [11] from the Icache, and the weld/decode stage merges two MultiOps and decodes them. The dual-thread operation welder is designed in such a way that the main thread's MultiOp has priority over the speculative thread's MultiOp, thus speculation never delays forward progress.

The dual-thread operation welder is shown in Fig. 3. The operations from the main thread MultiOp are always forwarded to the functional units. Then, the dual-thread operation welder inserts speculative operations into the holes left by the main thread MultiOp. The *synchronization* bit helps the main thread detect whether thread speculation is correct or not, and the *separability* bit tells that it is safe to separate the individual operations within a MultiOp and issue them in different cycles. The crossbar control takes operation slot *m empty*[2] bits and 1 *separability* bit from each thread and interleaves speculative operations into the issue register through the *Welding Crossbar*. 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* model. The dual register files can be designed to provide a fast copy of all registers from one to another, preferably in a single cycle. A potential register file

1. A MultiOp is a group of instructions that can be potentially executed in parallel.
2. Empty bit for each operation slot in a MultiOp is reset if there is an operation in the slot.

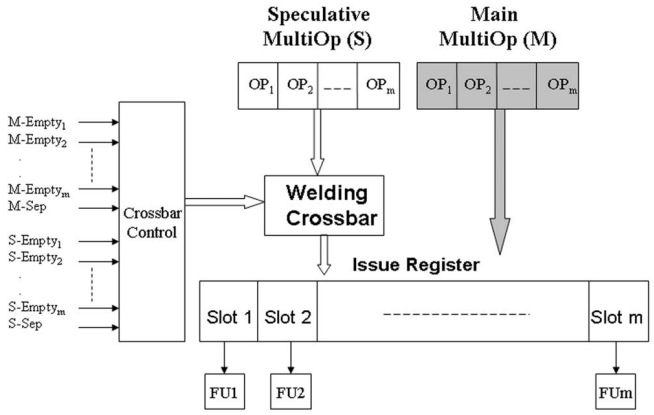

Fig. 3. Dual-thread operation welder.

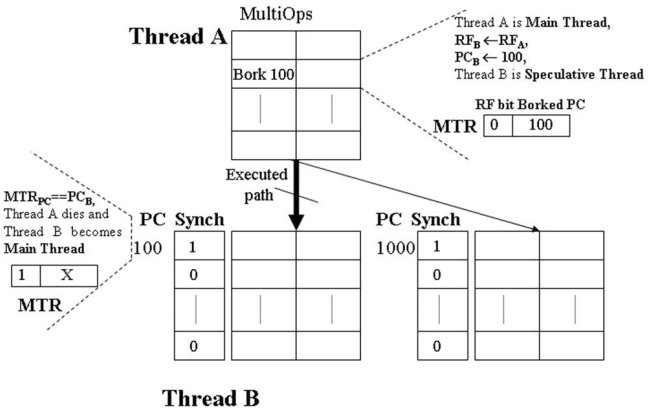

Fig. 5. Example showing thread creation and synchronization in the dual-thread *Weld*. Case 1: The actual path through Thread B.

design is shown in Fig. 4, which is similar to the *Checkpoint-repair* scheme [19]. The design has duplicated and cross-connected memory bit cells. Each memory bit cell corresponds to a bit from each register file. In the figure, A and B are the register files A and B, 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. The copy is performed at the time when a *bork* operation 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 only after all live-out operations in the main thread complete and write their results into its own register file.

When a *bork* instruction is executed at runtime, the target address of the *bork* instruction is saved in the *bork*ed address field of the *Main Thread Register* (MTR). The register file of the main thread dumps its contents to the other register file and sets the PC of the speculative thread to the *bork*ed address. The MTR has a 1-bit *Register File* (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). By toggling the *RF* bit in the MTR, the register file for the speculative thread can be determined. Each operation is attached with a *RF* bit to route results to the correct register file. This is done by reading the *RF* bit in the MTR for each thread before executing the MultiOps.

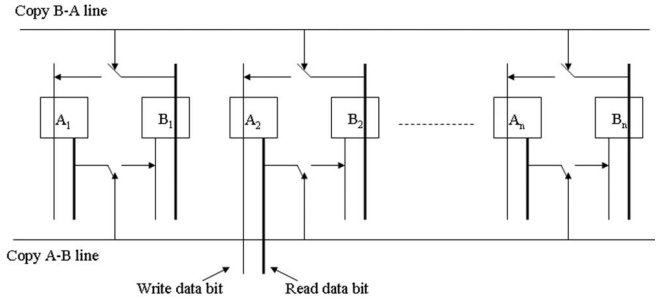

Fig. 4. Dual register file design with fast register copy.

Thread merge is detected with the help of a *synchronization bit* that is 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) in the speculative thread with the *synchronization bit* set is fetched from the cache, the *synchronization bit detector* detects a potential thread merge point. Then, the *borked PC* field in the MTR is compared with the PC value of the MultiOp. 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 *RF* bit and clearing the contents of the *borked PC* field in the MTR. If the addresses do not match, the speculative thread is mispeculated and must be squashed. Also, the *borked PC* field is cleared in the MTR.

Fig. 5 shows an example of thread creation and synchronization in the dual-thread *Weld* architecture. *Thread A* 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 the *Thread B* (i.e., a 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 register of the Thread B and into the *Borked PC* field of the MTR. If the actual execution path is through the *Thread B*, the PC address (i.e., 100) of the first MultiOp, which is detected by its synchronization bit, is compared with the *Borked PC* in the MTR. The addresses are the same and, therefore, the *Thread B* is correctly speculated. In this case, the *Thread A* dies and the *Thread B* becomes the main thread by flipping the *RF* bit in the MTR. Now, the register file *B* becomes the new main thread's register file. The *Borked PC* field in the MTR (shown as X in the figure) is cleared and the register file *A* is available for a new speculative thread.

On the other hand, if the actual execution path is not through the *Thread B* as shown in Fig. 6, the address 1,000 in the PC register does not match with the address 100 in the MTR. Therefore, the *Thread B* is mispeculated and must be squashed. After squashing *Thread B*'s operations from the pipelines, the *Thread A* remains as the main thread. However,

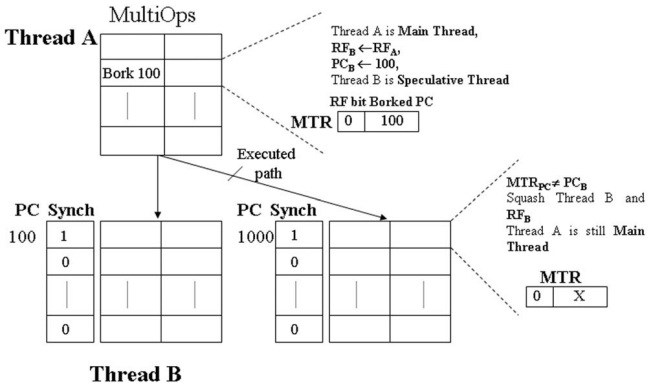

Fig. 6. Example showing thread creation and synchronization in the dual-thread *Weld*. Case 2: The actual path is not through Thread B.

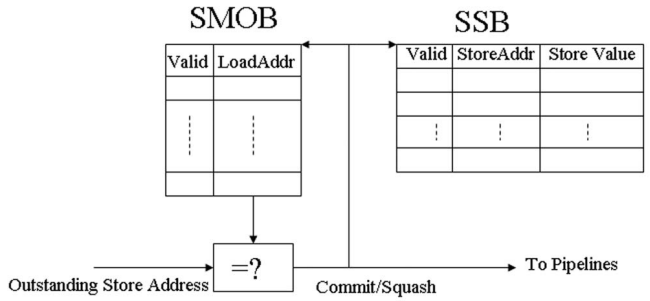

Fig. 7. SMOB and SSB structures.

the *Borked PC* field in the MTR is cleared for a new speculative thread. The *RF* bit remains the same because the main thread does not change.

Load operations' addresses from the speculative thread are saved in a buffer called the Speculative Memory Operation Buffer (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. Its mechanics is similar to ARB [20] in *MultiScalar* Processors. 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 from the speculative thread are executed, but are not allowed to modify the data cache. Instead, they are written into a special buffer called the *Speculative Store Buffer* (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. Also, a speculative load operation can access the SSB to retrieve data written by earlier speculative stores. In case of a thread merge (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 Fig. 7.

The microarchitectural implementation details of exception handling and recovery for the dual-thread Weld model is beyond the scope of this paper. Nevertheless, if an exception occurs in the main thread, the exception should be taken immediately and the speculative thread should be squashed to guarantee the correctness. This is because, if the exception handler modifies a live-out register, the modified register value is not visible to the speculative thread, so the speculative thread may read the wrong register value from its register file. On the other hand, if the exception occurs in the speculative thread, then the exception must be deferred until the speculative thread is known to be correctly speculated. The exception should be immediately taken if the speculation turns out to be correct.

2.2 Compiler Support

In this study, we used the LEGO compiler [9] as our backend. The LEGO compiler is an experimental VLIW compiler developed at North Carolina State University. The backend compiler has region formation, instruction scheduling, and register allocation phases. After forming regions, the code is scheduled for a specific machine model. Later, the physical registers are allocated using a global register allocator. An additional compiler phase, named *bork* insertion, is run through the register allocated code to insert *bork* operations in the code. A separate phase is needed because the *bork* insertion has to know the schedule times of the operations in each procedure before inserting *borks*.

The scheduling regions are called *treegions* [9], [10] in the LEGO compiler, which are single entry, multiple exit regions and can be formed with or without profile information. Each node of a treegion is a basic block and a treegion can have multiple paths in it. The compiler treats a treegion as a potential thread, and the bork insertion compiler stage inserts a *bork* operation in each path to spawn the next treegion in each exit. A bork operation is inserted in a path after operations whose live-outs reach from the path to the next treegion have all completed. This guarantees that all registers are ready before the register file copy. After inserting a *bork* operation in every path, some paths may have more than one *bork* operation since they may share some basic blocks. For each path, the earliest *bork* is kept and the rest is removed from the path. The *bork* algorithm is shown in Fig. 8. The algorithm creates both control and memory data speculative threads. A treegion or a thread is control speculative in the sense that the control can flow through some other paths and might spawn another treegion. It is memory data speculative that speculative loads in the speculative thread are allowed to complete before the stores in the main thread.

3 Performance Results of Dual-Thread Weld

This section discusses the performance analysis of the dual-thread *Weld* architecture model. We first examine the performance of the proposed architecture assuming hardware handling of memory speculative operations using the

```
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
```

Fig. 8. *Bork* insertion algorithm.

SMOB and, in that context, we study the performance impact of varying the size of the SMOB and the SSB. This study is followed by an analysis of a modified dual-thread model in which the SMOB is removed and the compiler guarantees proper load-store ordering. We conclude the performance analysis of the dual-thread *Weld* architecture model with a study of the effects of branch misprediction, second-level cache miss penalties, and dual-register file copy times on performance.

3.1 Experimental Framework

The experimental framework used for the performance analysis of the dual-thread *Weld* is shown in Fig. 9. The frontend consists of the *HP ELCOR* [32] and the University of Illinois at Urbana-Champaign *IMPACT* [31] compilers. The output of the front end is the *Rebel* [32] intermediate representation that is consumed by the *LEGO* compiler to perform the *bork* insertion algorithm. The *LEGO* compiler also performs various traditional optimizations, instruction scheduling, *bork* insertion, and generates optimized *Rebel* code, which is translated back to C as an optimized code. The optimized code is then compiled using *gcc*, and the resulting executable is run to generate dynamic VLIW traces that are consumed on the fly by the *dual-Weld* architecture simulator. The details of the *dual-Weld* simulator and the latencies of the operations used in the simulations are given in Table 1. The simulator does not model virtual memory structures such as the TLB, page tables, etc. The machine model used for our experiments is a six-issue VLIW processor with two universal functional units that can execute any type of instructions and four ALU/BR units that can execute only ALU and branch instructions. Each register file contains 128 integer and 128 floating-point registers. The SPECint95 benchmark suite is used for all runs, and 100 million instructions from each benchmark were executed using training input sets. Dual-thread runs are compared to a base model consisting of single thread run of the same program. Single-thread and dual-thread versions of each program are generated using exactly the same compiler optimizations. Detailed statistics about treegions or threads such as the average and maximum number of basic blocks and instructions per benchmark, etc., can be found in *Havanki's Master's Thesis* [9].

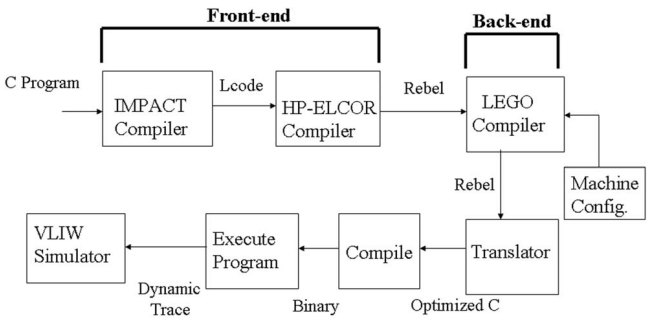

Fig. 9. Experimental framework.

3.2 Observations and Initial Results

Fig. 10 shows the distribution of the types of MultiOps (i.e., welded and nonwelded) that are issued to the functional units. A welded MultiOp is the one that contains operations from both threads, where a nonwelded MultiOp contains operations from either thread. The percentage of the welded MultiOps is much less than that of the nonwelded ones with an average of 17 percent versus 83 percent. These results suggest that much of the improvement in performance of the dual-thread Weld model comes from vertical multi-threading.

Fig. 11 shows the percentage improvement in performance of the dual-thread *Weld* model with a 128-entry SMOB and a 64-entry SSB over the single-threaded base model. From the figure, we see a mean of 22 percent improvement in performance across all benchmarks. The improvement by the dual-thread *Weld* model is primarily due to treegion overlapping, which increases issue slot utilization and conflicts in the SMOB. The largest improvements are seen in *134.perl* and *099.go* because both have high potential for treegion overlap, but more importantly, incur the least number of SMOB conflicts among all of the

TABLE 1
Properties of the Execution-Driven Simulation Environment

Simulator Properties	
2-way set associative split 32KB L1 instruction and 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	
30-cycle L2 miss time and variable	
5-cycle (Branch penalty, SMOB squash penalty time, mispeculated *bork* squash time) and variable	
1-cycle Register File copy time and variable	
ALU, BR, *BORK*, ST, FP ADD 1 cycle	
LD	2 cycles
FP MUL & DIV	3 cycles

benchmarks. Conversely, *147.vortex* shows the lowest performance improvement because it has a significantly larger number of SMOB conflicts than the remaining benchmarks, despite its similarly high treegion overlap potential. In general, those benchmarks with high SMOB conflict rates incur significant thread-squash penalties, which diminish the gains provided by the increased parallelism due to treegion overlapping.

3.3 Effects of SMOB and SSB Sizing

The number of SMOB and SSB entries was set at 128 and 64, respectively, in our initial experiments. However, considering the impact of SMOB conflicts on performance seen in the previous section, it is important that we examine the effects of the SMOB and SSB sizes on overall performance. In order to isolate the effect of SMOB size on performance, we vary the number of SMOB entries from 64 to 1,024 using a fixed 64-entry SSB. The results are shown in Fig. 12a.

As seen from the graph, the arithmetic mean stabilizes at 256 entries, and only *099.go* shows a minimal change in performance improvement beyond 256 entries. The number of stalls due to the unavailability of SMOB entries stays constant after 256 entries, therefore increasing the number of SMOB entries beyond 256 performs equally well with 256 entries. A similar study is performed to isolate the effect of SSB size on performance. In this study, we vary the number of SSB entries from 32 to 512 entries using a 256-entry SMOB. The results are shown in Fig. 12b. This graph shows that the mean does not increase beyond 128 entries. 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.

3.4 Dual-Thread Weld without Memory Speculation

Fig. 13 shows the performance results of the dual-thread *Weld* processor model with a perfect SMOB, which perfectly speculates all load operations. This figure shows how much performance is actually lost due to thread squashes caused by the SMOB. An average of 36 percent improvement in performance across all benchmarks can be attained in the dual-thread *Weld* model with a perfect SMOB. This is only about 24 percent for the same model with 256-entry SMOB. In this section, we investigate the possibility of closing this performance gap by using the dual-thread *Weld* model without the SMOB. We introduce a modified dual-thread *Weld* model in which there is no memory disambiguation hardware support for speculative loads (i.e., the SMOB) and the compiler is responsible for guaranteeing that proper

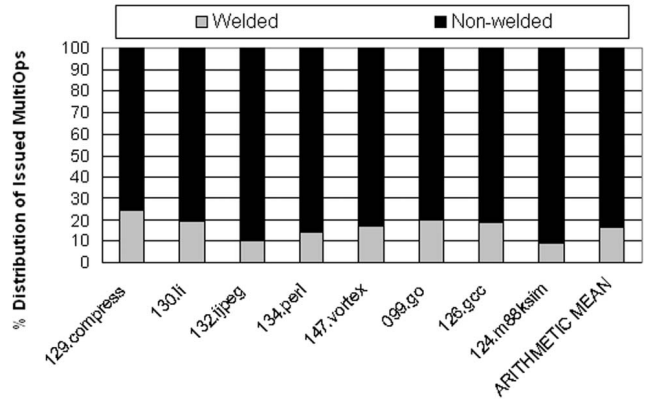

Fig. 10. Distribution of issued MultiOps.

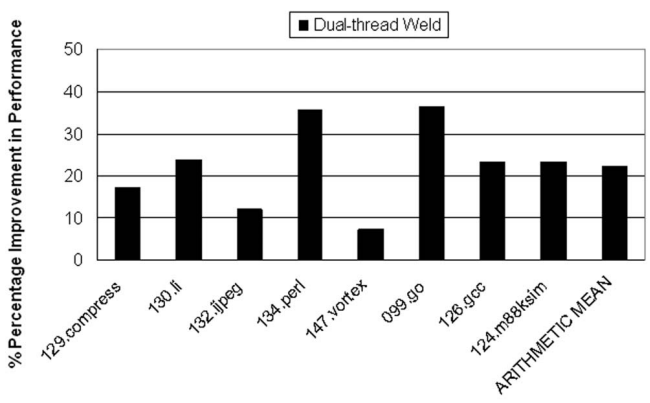

Fig. 11. Performance results of the dual-thread *Weld*.

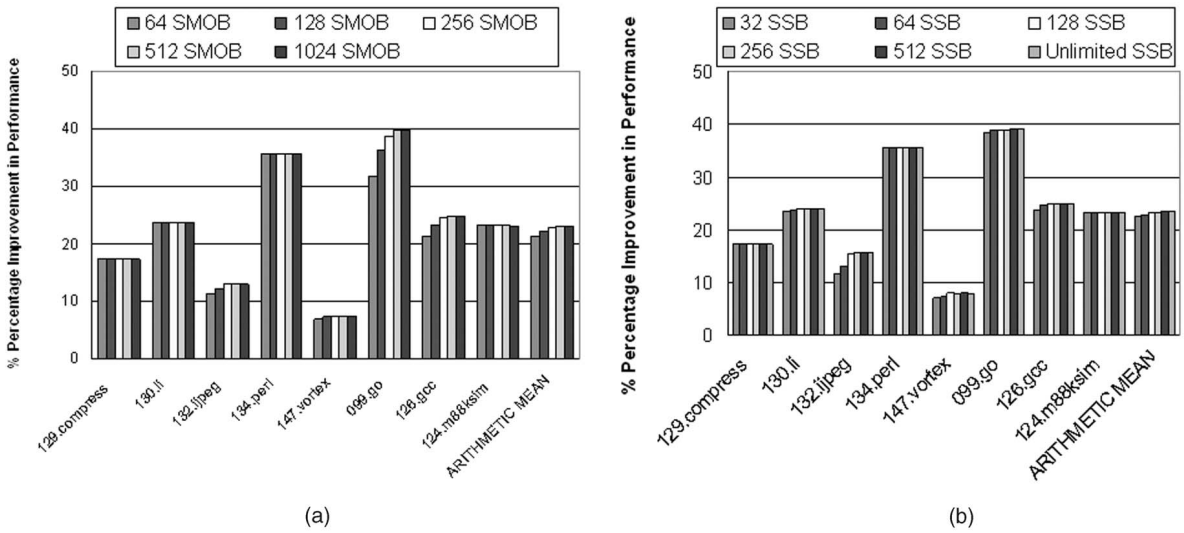

Fig. 12. Performance results of (a) the variable number of SMOB entries with fixed 64-entry SSB and (b) the variable number of SSB entries with fixed 256-entry SMOB.

load-store ordering is enforced across thread boundaries. In the base dual-thread *Weld* model, the SMOB resolves conflicts caused by executing speculative loads before their corresponding stores. However, in cases where conflicts occur, the speculative thread is completely squashed because there is no selective recovery hardware. Squashing results in throwing away useful work due to the flushing of the pipelines, SMOB, and SSB. This process steals a great number of useful cycles from the processor. The goal of the revised model is to reduce the squash penalty incurred by the hardware approach by disallowing speculative loads.

By enforcing load-store ordering, the compiler becomes more restricted in making its *bork* insertion and scheduling decisions. In this model, *bork* operations must be scheduled after all stores in the main thread to ensure that loads from the speculative thread cannot execute speculatively. Despite its restrictive nature in terms of *bork* scheduling, this approach may potentially outperform the dual-thread *Weld* model with memory speculation for the following reasons: 1) All squash penalty cycles resulting from SMOB conflicts are eliminated. 2) All useful work performed by the speculative thread prior to the violating load operation is not thrown away.

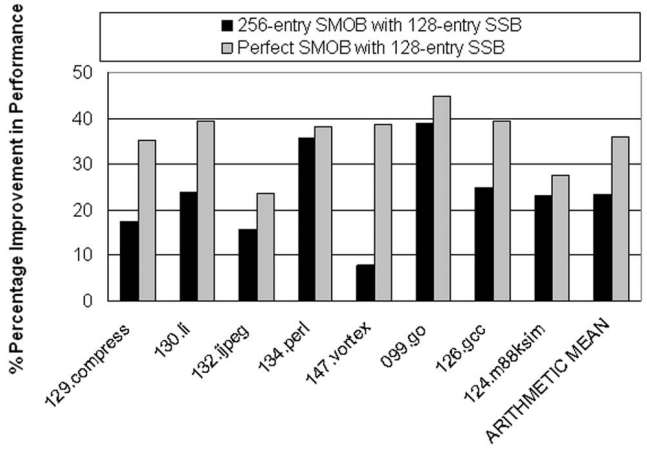

Fig. 13. Performance results with a perfect SMOB.

In the absence of a SMOB, the only source of thread mispeculation is branch mispredictions in the main thread. If the speculative thread is correctly speculated even though the main thread has some branch mispredictions, this means that some level of control-dependence can be maintained in the dual-thread Weld model. Table 2 shows the percentage of correctly speculated threads even though the main thread encounters branch mispredictions. If the rate of such threads is high, then significant control independence is exploited. An average of 66 percent of speculative threads can be correctly speculated, even though the main thread has some branch mispredictions in it across all benchmarks.

The *bork* insertion algorithm is modified to guarantee that, when scheduling a *bork* operation on a given path, all store operations on that path complete before any and all loads in the speculative thread. The previous live-out restrictions still apply. The modified algorithm is shown in Fig. 14. The algorithm computes the schedule time of each store in the path under consideration. After computing schedule times, the maximum store completion time is calculated by adding the operation latencies. The earliest time a *bork* can schedule is the maximum of the greatest

TABLE 2
Percentage of Correctly Speculated Threads, Even if the Main Thread has Branch Mispredictions

Benchmark	% Correctly Speculated Threads
129.compress	63
130.li	72
132.ijpeg	80
134.perl	97
147.vortex	47
099.go	45
126.gcc	37
124.m88ksim	87
AVERAGE	**66**

```
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
```

Fig. 14. Modified *bork* insertion algorithm for the dual-thread *Weld* without the SMOB.

live-out completion time and the greatest store completion time, assuming there is at least one load operation in the speculative treegion. If there are no loads in the speculative treegion, then the earliest allowable *bork* schedule time is determined by the maximum live-out completion time.

We study the performance of the nonmemory-speculative model using a 128-entry SSB, and the results are shown in Fig. 15 along with the results of the dual-thread *Weld* with a 256-entry SMOB and 128-entry SSB. Both sets of numbers are presented with respect to the base model. As seen from the graph, all benchmarks experience a significant boost in performance with the nonmemory-speculative dual-thread model. The largest improvement occurred in *147.vortex* because, as mentioned previously, it encounters the largest number of SMOB conflicts among all benchmarks. Similarly, the smallest improvement is observed in *134.perl* because it incurs the smallest number of SMOB conflicts among all benchmarks. The mean for the dual-thread *Weld* architecture with no SMOB and memory speculation is about 34 percent. Recall that the mean percentage improvement in performance for the dual-thread *Weld* architecture with perfect SMOB was 36 percent. Two percent point difference accounts for the loss in performance due to late schedule and therefore late execution of *bork* operations to spawn speculative threads.

3.5 Effects of Branch Misprediction

The microarchitecture of our simulated *Weld* processor is composed of a simple five-stage pipeline, which determines the branch-misprediction penalty as well as the thread-squash penalty. However, modern microprocessors have much deeper pipelines and, therefore, higher branch misprediction penalties. Therefore, in this section, we examine the performance effects of varying the branch misprediction penalty on the dual-thread *Weld* architecture. Fig. 16 shows the percentage improvement in performance of the nonmemory-speculative, dual-thread *Weld* model with a 128-entry SSB when the branch misprediction penalty (BP) is set to 5, 10, 15, and 25 cycles. As seen in the graph, the percentage improvement is not significantly sensitive to the variance of the branch penalty. In spite of longer pipeline stalls in the main thread due to longer branch penalties, the speculative thread can still progress using the idle cycles left by the main thread. In several programs, *129.compress*, *132.ijpeg*, *147.vortex*, *099.go*, and

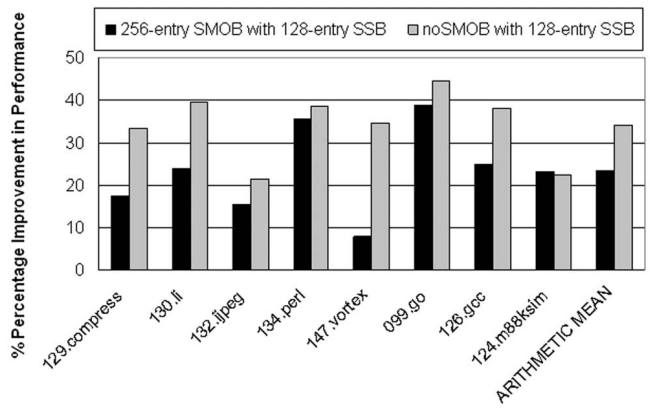

Fig. 15. Performance results of the dual-thread *Weld* with no memory speculation (i.e., no SMOB).

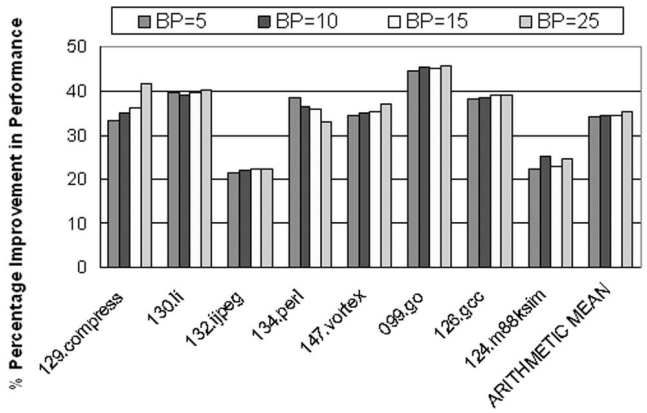

Fig. 16. Performance results when the branch penalty (BP) is varied from 5 to 25.

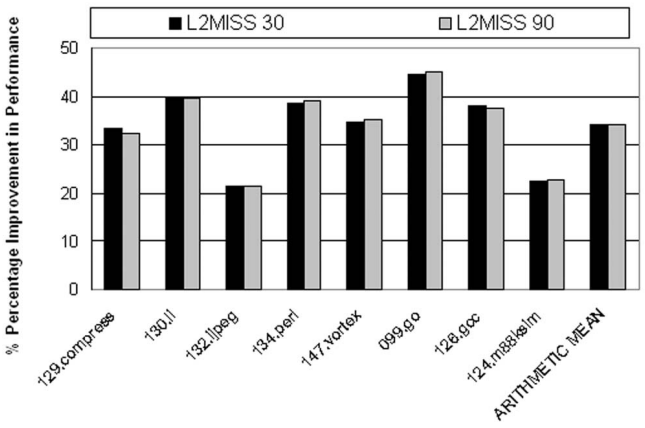

Fig. 17. Performance results for L2 miss latency variance.

126.gcc, the performance improvement actually increases as the branch latency increases. On the other hand, the percentage improvement in *134.perl* drops as the branch penalty increases because the speculative thread may not progress.

3.6 Effects of L2 Cache Miss Penalties

Similarly, we examine the effects of varying the latency of the second-level (L2) cache to main memory. We use a base latency of 30 cycles and increase it to 90 cycles to watch the effects of longer L2 miss latencies on performance in the dual-thread *Weld* architecture. For both runs, we assume a BP of five cycles. The results are given in Fig. 17. The average improvement in performance increases by 0.4 percent across all benchmarks in spite of the increased L2 miss latency. This is attributable to the fact that the speculative thread takes advantage of the increase in empty issue slots resulting from the longer stalls in the main thread, compensating for the loss in performance. 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 effect is observed because the speculative thread may not progress.

3.7 Effects of Register File Copy Cycle Time

So far, we assume 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 if the register file is organized as clusters. 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. Fig. 18 shows the performance results with 1, 10, 20, 30, 40, 50, and 60 cycle register file copy time. The percentage improvement in performance with 1-cycle register transfer time is known to be 34 percent. From 1 cycle to 10 and 20 cycles, the percentage improvement drops to 28 percent and 22 percent, respectively. There is a linear relationship between the percentage drop in performance and the increase in the register file copy cycle time. At every 10-cycle increase in the copy time, the percentage improvement drops by 6 percent. Until the copy time is 60 cycles, improvement in performance is possible. However, at 60 cycles and after, performance degradation is observed. To support this argument, the average distance between a *bork* and thread merge is computed in terms of average cycle time and tabulated in Table 3. This distance is calculated for each benchmark by adding each interval between a taken *bork* and thread merge and then taking the arithmetic mean. For instance, the average distance between a *bork* and thread merge time for *129.compress* is 29 cycles. *129.compress* experiences performance degradation when the register file copy time becomes 30 or more cycles. A similar effect can be observed for *124.m88ksim*, in which the average distance is 39 cycles and performance degradation begins after 40 or more register file copy cycles.

In summary, the register file copy time has a deep impact on performance when designing a VLIW processor using the general *Weld* architecture, which has multiple threads and multiple register files. On the other hand, a careful design of the dual-register files, as pointed out in Section 2.1, makes the register file transfer fast and, therefore, not critical on performance for the dual-thread VLIW processor.

4 Related Work

There are two primary relevant pieces of work on the topic of multithreading for VLIW architectures. In *processor coupling*, multiple threads are scheduled statically and interleaved into execution clusters, consisting of a set of function units and a common register file, at runtime [6], [7]. Threads, which are generated by the compiler through explicit *fork* and *forall* operations, communicate through registers and memory and are nonspeculative, unlike the

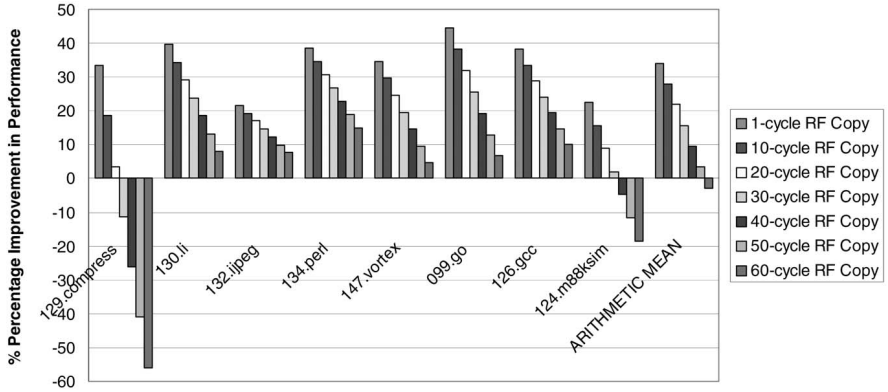

Fig. 18. Performance results with variance of register file copy cycles.

TABLE 3
Average Number of Cycles between a *bork* and a Thread Merge

Benchmark	# of Cycles
129.compress	29
130.li	88
132.ijpeg	131
134.perl	108
147.vortex	92
099.go	74
126.gcc	101
124.m88ksim	39
AVERAGE	**83**

dual-thread *Weld* model. XIMD [8] has multiple functional units and a large global register file similar to VLIW/EPIC architecture, and each functional unit has a dedicated instruction sequencer to fetch instructions. A program is partitioned into several threads by the compiler or by a specialized partitioning tool. The XIMD compiler takes each thread and schedules it separately. Threads are then merged statically to increase static code density or to optimize for execution time. No speculative threads are allowed in XIMD. Recently, VLIW processors using simultaneous multithreading have been proposed in [29], [30] to improve throughput by running different applications simultaneously.

There have also been several multithreading techniques proposed for dynamically-scheduled architectures. In the MultiScalar paradigm, there are multiple superscalar cores, called *processing units*, consisting of their own private register file, I-cache, and functional units [2], [3]. 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 graph of the program. During the execution of a program, register values can flow from one task to another. SPSM (Single-Program Speculative Multithreading) speculatively spawns multiple threads within a single program and simultaneously executes those threads on a superscalar core [1]. Thread spawning is performed by inserting *fork* and *suspend* instructions into the static code. There is a main thread that can spawn multiple speculative threads. When the main thread merges with a speculative thread, the speculative thread dies and the main thread continues. This requires merging the register state of the dying thread with the main thread. This is somewhat different from the dual-thread *Weld* model in which the speculative thread simply becomes the main thread, eliminating the need for register file copy and other burdens. TME (Threaded Multiple Path Execution) executes multiple alternate paths on an SMT superscalar processor [4], [5]. It uses free hardware contexts to assign paths of conditional branches. Slipstream processors execute two instances of a program on a chip-multiprocessor or a simultaneous multithreaded processor (SMT) to improve performance and provide fault tolerance [13]. Similarly, Balasubramonian et al. describe a double-thread superscalar architecture model [14]. When the primary thread stalls, the future thread starts executing speculatively. The future thread does not modify the processor state, but prefetches load data and helps resolve branches in the primary thread. Steffan et al. present a thread-level speculation technique for a CMP system [15]. Essentially, the compiler creates threads speculatively and the hardware detects memory and register use violations. The invalidation-based cache coherence protocol is extended to detect violations caused by speculative memory operations. Data forwarding between threads is performed using memory, instead of special forwarding hardware. Data-driven Multithreading preexecutes some long latency instructions that might cause branch mispredictions or data cache load misses as separate threads [16]. Those threads run simultaneously with the main thread. When the main thread merges with those instructions, it does not reexecute them if they were correctly speculated. Luk proposed a software-based preexecution technique that tolerates long-latency memory operations in SMT processors [17]. The software spawns a preexecution thread that executes and brings the data into the cache. This reduces data cache misses within the main thread. However, the preexecution thread does not merge with the main thread. Its results are discarded when the thread stops. The preexecution thread is created and killed by special instructions in the compiler.

5 Conclusion

This paper presents a novel microarchitectural and compiler technique for latency tolerance in VLIW architectures. The proposed dual-thread VLIW processor model based on the *Weld* architecture paradigm tolerates latencies by creating a speculative thread from a single application and running it with the nonspeculative main thread simultaneously. The compiler decides where and when to create the speculative thread that can be both control and memory speculative at the time of thread creation.

The dual-thread *Weld* model with memory disambiguation hardware for speculative loads causes a large number of thread squashes. Thus, we have also investigated the viability of eliminating the memory disambiguation hardware for speculative loads in order to reduce the hardware cost further and potentially to improve performance by eliminating thread squashes caused by speculative loads. Our compiler schedules threads in such a way that no load mispeculation occurs at runtime. Hence, the speculative load hardware and thread squashes can be completely eliminated. Consequently, this allows a cost-effective design of a dual-thread *Weld* processor. The performance results have shown that the low-cost dual-thread *Weld* processor architecture model without the speculative load hardware could have performance improvement of up to 34 percent with respect to a single-threaded VLIW processor.

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," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques,* June 1995.

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

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

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

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

[6] S.W. Keckler and W.J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism," *Proc. 19th Ann. Int'l Symp. Computer Architecture,* May 1992.
[7] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee, "The M-Machine Multicomputer," *Proc. 28th Ann. Int'l Symp. Microarchitecture,* Dec. 1995.
[8] A. Wolfe and J.P. Shen, "A Variable Instruction Stream Extension to the VLIW Architecture," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Apr. 1991.
[9] W.A. Havanki, "Treegion Scheduling for VLIW Processors," master's thesis, Dept. of Electrical and Computer Eng., North Carolina State Univ., Raleigh, North Carolina, July 1997.
[10] W.A. Havanki, S. Banerjia, and T.M. Conte, "Treegion Scheduling for Wide-Issue Processors," *Proc. Fourth Int'l Symp. High Performance Computer Architecture,* Feb. 1998.
[11] B.R. Rau, "Dynamically Scheduled VLIW Processors," *Proc. 26th Ann. Int'l Symp. Microarchitecture,* Dec. 1993.
[12] M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Computers,* May 1996.
[13] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Nov. 2000.
[14] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi, "Dynamically Allocating Processor Resources between Nearby and Distant ILP," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* June 2001.
[15] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* June 2000.
[16] A. Roth and G.S. Sohi, "Speculative Data-Driven Multithreading," *Proc. Sixth Conf. High-Performance Computer Architecture,* Jan. 2000.
[17] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* June 2001.
[18] E. Özer, T.M. Conte, and S. Sharma, "Weld: A Multithreading Technique towards Latency-Tolerant VLIW Processors," *Proc. Eighth Int'l Conf. High Performance Computing,* Dec. 2001.
[19] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for High-Performance Out-of-Order Execution Machines," *IEEE Trans. Computers,* vol. 36, no. 12, Dec. 1987.
[20] M. Franklin and G.S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Computers,* May 1996.
[21] M. Tremblay, "A Microprocessor Architecture for the New Millennium," *Hot Chips 11,* Aug. 1999.
[22] Transmeta, CrusoeTM, http://www.transmeta.com, 2005.
[23] Intel, Intel Itanium Processor at 800MHZ and 733MHZ Data Sheet, May 2001.
[24] T. Sukemura, "FR500 VLIW-Architecture High-Performance Embedded Microprocessor," *FUJITSU Scientific and Technical J.,* vol 36, no. 1, June 2000.
[25] StarCore, SC140 DSP Core Reference Manual, 2001.
[26] Texas Instruments, TMS320C62XX CPU and Instruction Set Reference Guide, July 1997.
[27] Philips, TM 1000 Preliminary Data Book, 1997.
[28] P. Faraboschi, G. Brown, J.A. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," *Proc. 27th Int'l Symp. Computer Architecture (ISCA-2000),* 2000.
[29] S. Kaxiras, A.D. Berenbaum, and G. Narlikar, "Simultaneous Multithreaded DSPs: Scaling from High Performance to Low Power," *Bell Laboratories Technical Memorandum 10009639-001024-06TM,* 2000.
[30] H.P. Rao, S.K. Nandy, and M.N. V. S. Kiran, "Simultaneous MultiStreaming for Complexity Effective VLIW Architectures," *Proc. Advances in Computer System Architecture (ACSAC 2003),* Sept. 2003.
[31] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Water, and W.-m.W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," *Proc. 18th Ann. Int'l Symp. Computer Architecture,* May 1991.
[32] S. Aditya, V. Kathail, and B.R. Rau, "Elcor's Machine Description System: Version 3.0," HP Technical Report HPL-98-128, Oct. 1998.

Emre Özer received the bachelor and master of science degrees from the Department of Computer Science and Engineering at Yildiz Technical University, Istanbul, Turkey, in 1993 and 1996. He received the PhD degree in the Department of Electrical and Computer Engineering at North Carolina State University, Raleigh, North Carolina, in 2001. He accomplished two different summer industrial internships during his PhD. In the summer of 1999, he worked in the MAJC Architecture Group at Sun Microsystems, Sunnyvale, CA. He was also a summer student researcher in HP Labs., Cambridge, MA, in the summer of 2000. After completing the PhD degree, he joined Motorola Inc. as a computer architect at Star*Core Design Center in Atlanta, GA. In December 2002, he joined the Department of Computer Science in Trinity College Dublin, Ireland, as a research fellow and worked there for two years. Currently, he is a senior engineer in the R&D Department at ARM Ltd., Cambridge, UK. He has been a reviewer of several journals and conferences on computer architectures, embedded systems, and compilers. His research interests are microarchitecture design, optimizing compilers, multithreading, multiprocessing, embedded processor design, reconfigurable architectures, and hardware compilation.

Thomas M. Conte (S'84-M'92-SM'99-F'05) received the bachelor of electrical engineering degree from the University of Delaware in 1986. He went on to receive the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in 1988 and 1992, respectively. He is currently a professor of electrical and computer engineering and Director of the Center for Embedded Systems Research at North Carolina State University. While on leave from North Carolina State University from 2000-2001, he served as the Chief Microarchitect and Manager of Back End Compiler Development for DSP vendor BOPS, Inc. He is editor-in-chief of the *Journal of Instruction-Level Parallelism* and an associate editor of both the *ACM Transactions on Architecture and Compiler Optimization* and the *ACM Transactions on Embedded Computer Systems*. Conte is also the chair of the IEEE CS Technical Committee on Microprogramming and Microarchitecture (TC-ARCH). He currently directs several PhD students on the TINKER project in topics spanning advanced microarchitecture, compiler optimization, and performance analysis. His research is or has been supported by Compaq, DARPA, HP, IBM, Intel, Sun Microsystems, TI, and the US National Science Foundation. He is the recipient of a US National Science Foundation CAREER award and the IBM T.J. Watson Partnership Award for Faculty Development. He is a fellow of the IEEE.