# The Impact of Incorrectly Speculated Memory Operations in a Multithreaded Architecture

Resit Sendag, *Member*, *IEEE*, Ying Chen, *Student Member*, *IEEE*, and David J. Lilja, *Senior Member*, *IEEE*

**Abstract**—The speculated execution of threads in a multithreaded architecture, plus the branch prediction used in each thread execution unit, allows many instructions to be executed speculatively, that is, before it is known whether they actually will be needed by the program. In this study, we examine how the load instructions executed on what turn out to be incorrectly executed program paths impact the memory system performance. We find that *incorrect speculation* (wrong execution) on the instruction and thread-level provides an indirect prefetching effect for the later correct execution paths and threads. By continuing to execute the mispredicted load instructions *even after* the instruction or thread-level control speculation is known to be incorrect, the cache misses observed on the correctly executed paths can be reduced by 16 to 73 percent, with an average reduction of 45 percent. However, we also find that these extra loads can increase the amount of memory traffic and can pollute the cache. We introduce the small, fully associative *Wrong Execution Cache* (WEC) to eliminate the potential pollution that can be caused by the execution of the mispredicted load instructions. Our simulation results show that the WEC can improve the performance of a concurrent multithreaded architecture up to 18.5 percent on the benchmark programs tested, with an average improvement of 9.7 percent, due to the reductions in the number of cache misses.

**Index Terms**—Speculation, multithreaded architecture, mispredicted loads, wrong execution, prefetching, wrong execution cache.

## 1 Introduction

Several studies have proposed methods to exploit more instruction and thread-level parallelism and to hide the latency of the main memory accesses, including speculative execution [6], [17], [24], [25], [26] and data prefetching [11], [13], [14], [18], [27], [28], [29]. In this study, we have used the SuperThreaded Architecture (STA) [2], which is a concurrent multithreaded architecture [1], as our base architecture model. The STA can exploit loop-level and instruction-level parallelism from a broad range of applications. It consists of a number of thread processing elements (superscalar cores) interconnected with some tightly integrated communication network [2]. Each superscalar processor core can use branch prediction to speculatively execute instructions beyond basic block-ending conditional branches. If a branch prediction ultimately turns out to be incorrect, the processor state must be restored to the state prior to the predicted branch and execution must be restarted down the correct path. Simultaneously, a concurrent multithreaded architecture can aggressively fork speculative successor threads to further increase the amount of parallelism that can be exploited in an application program. If a speculated control dependence turns out to be incorrect, the nonspeculative head thread must kill all of its speculative successor threads.

With both instruction and thread-level control speculations, a multithreaded architecture may issue many memory references which turn out to be unnecessary since they are issued from what subsequently is determined to be a mispredicted branch path or a mispredicted thread. However, these incorrectly issued memory references may produce an indirect prefetching effect by bringing data or instruction lines into the cache that are needed later by correctly executed threads and branch paths. On the other hand, these additional memory operations will increase the memory traffic and may pollute the cache with unneeded blocks [3], [17].

Existing superscalar processors with deep pipelines and wide issue units do allow memory references to be issued speculatively down wrongly predicted branch paths. In this study, however, we go one step further and examine the effects of continuing to execute the loads issued from both mispredicted branch paths and mispredicted threads even after the speculative operation is known to be incorrect. These instructions are marked as being from the mispredicted branch path or a mispredicted thread when they are issued so they can be squashed to prevent them from altering the target register after they access the memory system.

The execution of these extra loads can make a significant performance improvement when there exists a large disparity between the processor cycle time and the memory speed. However, executing these loads can reduce the performance in systems with small data caches and low associativities due to cache pollution. This cache pollution occurs when the loads from mispredicted branch paths and mispredicted threads move blocks into the data cache that are never needed by the correct execution paths. It is also

- *R. Sendag is with the Department of Electrical and Computer Engineering, University of Rhode Island, 4 East Alumni Ave., Kingston, RI 02881. E-mail: sendag@ele.uri.edu.*
- *Y. Chen and D.J. Lilja are with the Department of Electrical and Computer Engineering, University of Minnesota, 200 Union St. S.E., Minneapolis, MN 55455. E-mail: wildfire@ece.umn.edu, lilja@umn.edu.*

*Manuscript received 1 July 2003; revised 2 Apr. 2004; accepted 4 Aug. 2004; published online 23 Feb. 2005.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0102-0703.*

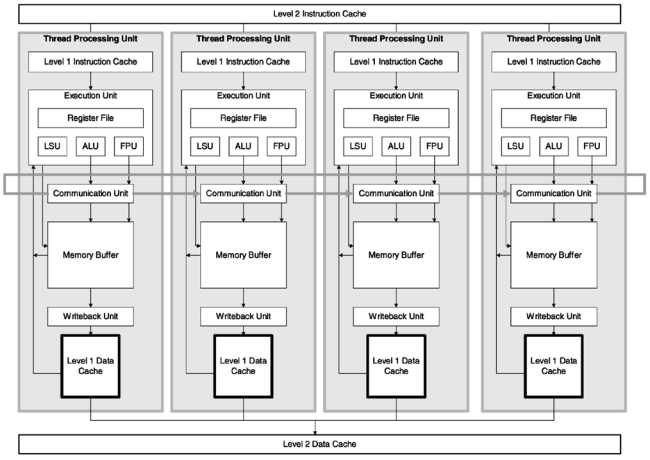



Fig. 1. A superthreaded architecture processor with four thread units.

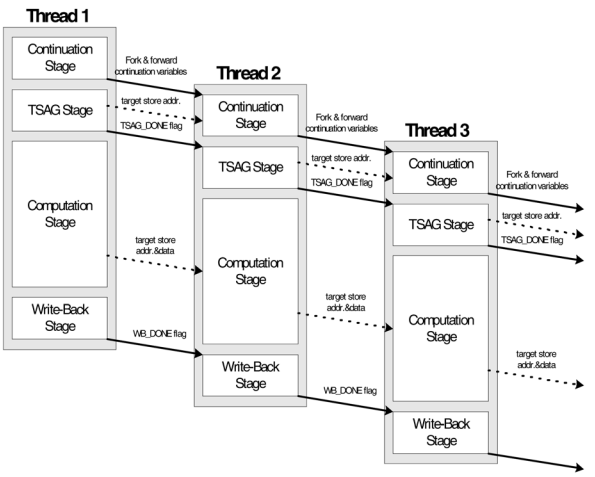



Fig. 2 The thread pipelining execution model.

possible for the cache blocks fetched by these loads to evict blocks that are still required by the correct paths.

We propose the *Wrong Execution Cache* (WEC) to eliminate the potential cache pollution caused by executing the wrong-path and wrong-thread loads. This small, fully associative cache stores the values fetched by the wrong execution loads plus the blocks evicted from the data cache. This work shows that the execution of wrong-path or wrong-thread loads with a WEC can produce a significant performance improvement.

In the remainder of the paper, Section 2 presents an overview of the STA [2], which is a concurrent multithreaded architecture used as the base architecture for this study. Section 3 describes wrong execution loads and Section 4 gives the implementation of the WEC in the base processor. Our experimental methodology is presented in Section 5 with the corresponding results given in Section 6. Section 7 discusses some related work and Section 8 concludes.

## 2 The Superthreaded Architecture

The superthreaded architecture (STA) integrates compilation techniques and runtime hardware support to exploit both thread-level and instruction-level parallelism in programs [21], [22]. It uses a thread pipelining execution model to enhance the overlap between threads.

### 2.1 Base Architecture Model

The STA consists of multiple thread processing units (TUs), with each TU connected to its successor by a unidirectional communication ring, as shown in Fig. 1. Each TU has its own private level-one (L1) instruction cache, L1 data cache, program counter, register file, and execution units. The TUs share the unified second-level (L2) instruction and data cache. A private memory buffer is used in each TU to cache speculative stores for runtime data dependence checking.

When multiple threads are executing on an STA processor, the oldest thread in the sequential order is called the *head thread* and all other threads derived from it are called *successor threads*. The program execution starts from its entry thread while all other TUs are idle. When a parallel code region is encountered, this thread activates its downstream thread by forking. It also transfers all the data the downstream thread needs for one iteration of the loop. This forking continues until there are no idle TUs. When all TUs are busy, the youngest thread delays forking another thread until the head thread retires and its corresponding TU becomes idle. A thread can be forked either speculatively or nonspeculatively. A speculatively forked thread will be aborted by its predecessor thread if the speculative control dependence subsequently turns out to be false.

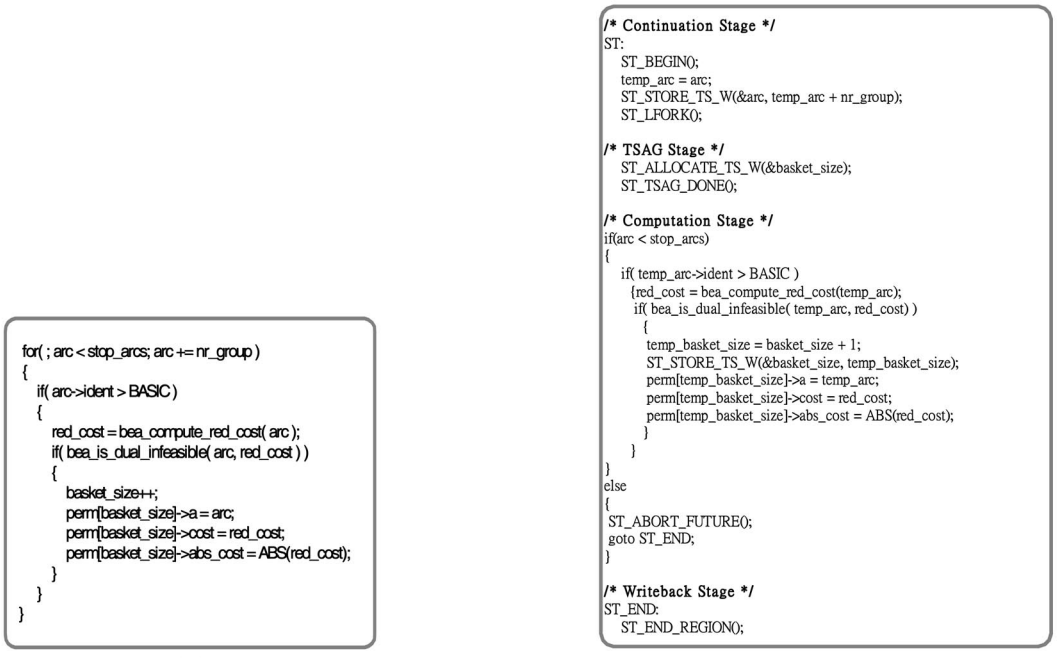
### 2.2 Thread Pipelining Execution Model

The execution model for the STA architecture is thread pipelining, which allows threads with data and control dependences to be executed in parallel. Instead of speculating on data dependences, the thread execution model facilitates runtime data dependence checking for load instructions. This approach avoids the squashing of threads caused by data dependence violations. It also reduces the hardware complexity of the logic needed to detect memory dependence violations compared to some other CMA execution models [4], [5]. As shown in Fig. 2, the execution of a thread is partitioned into the *continuation* stage, the *target-store address-generation (TSAG)* stage, the *computation* stage, and the *write-back* stage.

The main function of the continuation stage is to compute recurrence variables such as loop index variables needed to fork a new thread on the next thread unit. This stage ends with a *fork* instruction, which initiates a new speculative or nonspeculative thread on the next TU. The *abort* instruction is used to kill the successor threads when it is determined that a speculative execution was incorrect. Note that the continuation stages of two adjacent threads can never overlap.

The TSAG stage computes the addresses of store instructions on which later concurrent threads *may* have data dependences. These special store instructions are called *target stores* and are identified using conventional data dependence analysis. The computed addresses are stored in the memory buffer of each TU and are forwarded to the memory buffers of all succeeding concurrent threads units.

The computation stage performs the actual computation of the loop iteration. Loads on addresses that may have cross-iteration dependences, as previously identified by the target stores, are checked against the addresses in a TU's



```
for( ; arc < stop_arcs; arc += nr_group )
{
   if( arc->ident > BASIC )
   {
      red_cost = bea_compute_red_cost( arc );
      if( bea_is_dual_infeasible( arc, red_cost ) )
      {
         basket_size++;
         perm[basket_size]->a = arc;
         perm[basket_size]->cost = red_cost;
         perm[basket_size]->abs_cost = ABS(red_cost);
      }
   }
}
```

(a)

```
/* Continuation Stage */
ST:
   ST_BEGIN();
   temp_arc = arc;
   ST_STORE_TS_W(&arc, temp_arc + nr_group);
   ST_LFORK();

/* TSAG Stage */
   ST_ALLOCATE_TS_W(&basket_size);
   ST_TSAG_DONE();

/* Computation Stage */
if(arc < stop_arcs)
{
   if( temp_arc->ident > BASIC )
    {red_cost = bea_compute_red_cost(temp_arc);
     if( bea_is_dual_infeasible( temp_arc, red_cost) )
       {
        temp_basket_size = basket_size + 1;
        ST_STORE_TS_W(&basket_size, temp_basket_size);
        perm[temp_basket_size]->a = temp_arc;
        perm[temp_basket_size]->cost = red_cost;
        perm[temp_basket_size]->abs_cost = ABS(red_cost);
       }
    }
}
else
{
 ST_ABORT_FUTURE();
 goto ST_END;
}

/* Writeback Stage */
ST_END:
   ST_END_REGION();
```

(b)

Fig. 3. (a) An example code segment from 181.mcf and (b) its transformed superthreaded code.

memory buffer. If a dependence is detected, but the data has not yet arrived from the upstream thread, the out-of-order superscalar core will execute instructions that are independent of the load operation that is waiting for the upstream data value. This stalled load will be executed after the dependent value is passed from the upstream thread.

In the write-back stage, all the store data (including target stores) in the memory buffer will be committed and written to the memory hierarchy. The write-back stages are performed in the original program order to preserve nonspeculative memory state and to enforce output and antidependences between threads.

### 2.3 An Example Superthreaded Program

The example code segment shown in Fig. 3a is one of the most time-consuming loops in the SPEC2000 benchmark *181.mcf*. This is a for-loop with exit conditions in the loop head. There is potential read-after-write data dependence across loop iterations caused by variable *basket_size*.

The transformed STA code is shown in Fig. 3b. In the continuation stage, loop index *arc* is increased and forwarded to the next thread-processing unit with a ST_STORE_TS_W instruction. The original value is stored in *temp_arc* for later use. The continuation stage ends with an ST_LFORK instruction to fork a new thread speculatively without checking loop exit condition. The address of the variable *basket_size* is forwarded to the next thread in the TSAG stage by an ST_ALLOCATE_TS_W instruction. In the computation stage, the loop exit condition is checked. If it is true, it will abort the successor threads by instruction, an ST_ABORT_FUTURE instruction, which causes the control flow to jump out of the loop. Otherwise, the thread unit will continue computation in the loop body. The update of the variable *basket_size* is performed with an ST_STORE_TS_W instruction, which will forward the result to the next thread. If the exit condition is false, the thread unit will execute an ST_END_REGION instruction. This instruction will force a write-back of the contents of the memory buffer back to the shared main memory.

## 3 Wrong Execution

There are two types of *wrong execution* that can occur in a concurrent multithreaded architecture such as the STA processor. The first type occurs when instructions continue to be issued down the path of what turns out to be an incorrectly predicted conditional branch instruction within a single thread. We refer to this type of execution as *wrong-path execution*. The second type of wrong execution occurs when instructions are executed from a thread that was speculatively forked, but is subsequently aborted. We refer to this type of incorrect execution as *wrong-thread execution*. Our interest in this study is to examine the effects on the memory hierarchy of load instructions that are issued from both of these types of wrong executions.

### 3.1 Wrong-Path Execution

Before a branch is resolved, some load instructions on wrongly predicted branches may not be ready to be issued because they are waiting either for the effective address to be calculated or for an available memory port. In wrong path execution, however, they are allowed to access the memory system as soon as they are ready even though they are known to be from the wrong path. These instructions are marked as being from the wrong execution path when they are issued so they can be squashed in the pipeline at the write-back stage. A wrong-path load that is dependent upon another instruction that gets flushed after the branch is resolved also is flushed in the same cycle. In this manner, the processor is allowed to continue accessing memory with

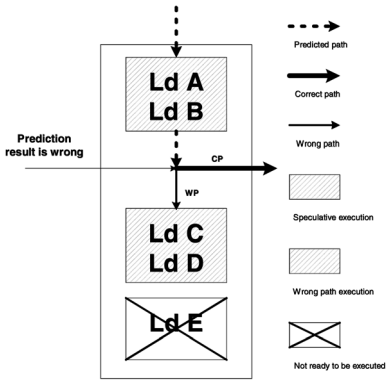



Fig. 4. The difference between speculative and wrong-path execution.

loads that are known to be from the wrong branch path. No store instructions are allowed to alter the memory system, however, since they are known to be invalid.

An example showing the difference between traditional speculative execution and our definition of wrong-path execution is given in Fig. 4. In this example, there are five loads (A, B, C, D, and E) fetched down the predicted execution path. In a typical pipelined processor, loads A and B become ready and are issued to the memory system speculatively before the branch is resolved. After the branch result is known to be wrong, however, the other three loads, C, D, and E, are squashed before being able to access the memory system.

In a system with wrong-path execution, however, ready loads are allowed to continue execution (loads C and D in Fig. 4) in addition to the speculatively executed loads (A and B). These *wrong-path loads* are marked as being from the wrong path and are squashed later in the pipeline to prevent them from altering the destination register. However, they are allowed to access the memory to move the value read into the levels of the memory hierarchy closer to the processor. Since load E is not ready to execute by the time the branch is resolved, it is squashed as soon as the branch result is known.

### 3.2 Wrong-Thread Execution

When executing a loop in the normal execution mode of the superthreaded execution model described in Section 2, the head thread executes an *abort* instruction to kill all of its successor threads when it determines that the iteration it is executing satisfies the loop exit condition. To support wrong-thread execution in this study, however, the successor threads are marked as wrong threads instead of killing them when the head thread executes an *abort*. These specially marked threads are not allowed to fork new threads, yet they are allowed to continue execution. As a result, after this parallel region completes its normal execution, the wrong threads continue execution in parallel with the following sequential code. Later, when the wrong threads attempt to execute their own *abort* instructions, they kill themselves before entering the write-back stage.

If the sequential region between two parallel regions is not long enough for the wrong threads to determine that they are to be aborted before the beginning of the next parallel region, the *begin* instruction that initiates the next parallel region will abort all of the still-executing wrong threads from the previous parallel region. This modification of the *begin* instruction allows the head thread to fork without stalling. Since each thread's store data are put in a speculative memory buffer local to each TU and wrong threads do not execute their write-back stages, no stores from the wrong threads can alter the shared memory.

Fig. 5 shows this wrong-thread execution model with four TUs. Although wrong-path and wrong-thread execution have similarities, the main difference between them is that, once a branch is resolved, the ready loads that are not

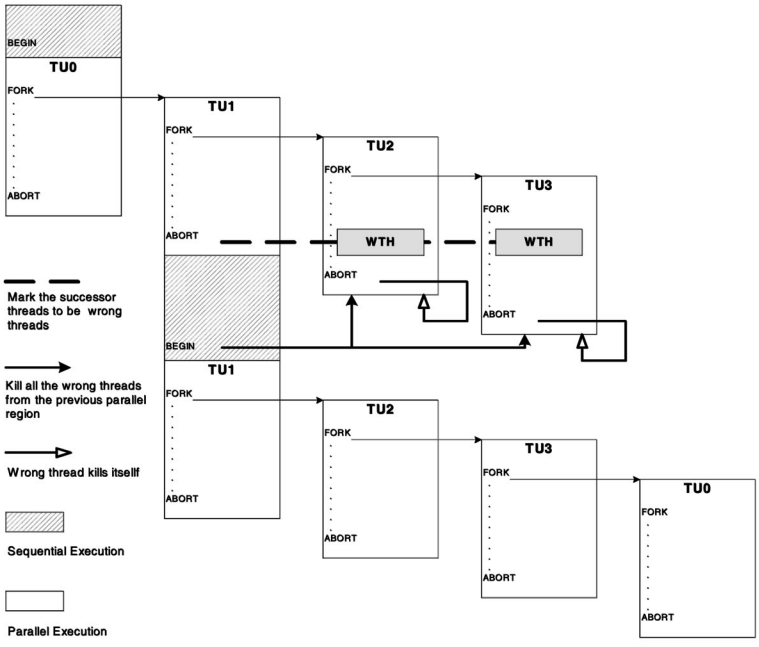



Fig. 5. The wrong-thread execution model with four TUs.

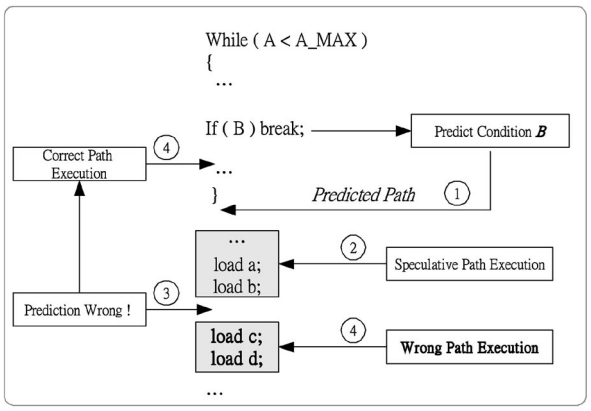



Fig. 6. *Wrong-Path Execution*. The prediction on condition *B* (1) breaks the while-loop. Loads *c* and *d* are issued (4) to the cache memory, however, even after the prediction result is known to be wrong (3). These loads might bring data blocks closer to the processor to assist the subsequent execution of these same loads down the correct path when the while-loop exits normally. Note that *c* and *d* are not written within the loop.

yet ready to execute on a wrong path are squashed, while wrong-thread loads are allowed to continue their execution.

### 3.3 Using Wrong Execution to Prefetch Data in the STA

In this section, we present two fabricated code examples to show how wrong execution can help improve the performance of a superscalar and a multithreaded architecture processor. While these examples do not show all of the benefits of incorrect speculation, they do demonstrate the sources of much of the potential of wrong execution.

#### 3.3.1 Prefetch by Wrong-Path Execution within a TU

In the example in Fig. 6, a misprediction on condition B causes the program to break the *while* loop and begin fetching from the end of the loop. Loads *a*, *b*, *c*, and *d* then are fetched speculatively. Loads *a* and *b* are issued speculatively before the branch result is known. On the other hand, loads *c* and *d* are ready to be issued to the memory system by the time branch is resolved. Since the prediction is determined to be wrong, the processor must squash these two loads before they are issued. However, they are marked as being issued in a wrong execution mode. If loads *c* and *d* miss in the cache, their results are moved into the data cache. Meanwhile, the execution continues with the instruction that follows the branch condition. After the *while*-loop exits normally, loads *c* and *d* will be requested down the correct execution path. The values of these loads have already been brought into the cache by the previous wrong execution. This is only one example to show how *wrong-path execution* can help in hiding the memory latency.

#### 3.3.2 Prefetch by Wrong-Thread Execution in the STA

It might be counter intuitive that executing the wrong threads can produce any benefit since the parallelization is based on loop iterations and wrong threads execute the wrong iterations. However, in the case of nested loops, the wrong iterations of the inner loop may actually need to be executed in later iterations of the outer loop. A similar situation may occur with nonnested loops since the wrong iterations from one loop could move data into the caches for use by the correct iterations of subsequently executed loops.

Fig. 7a shows a code segment in which the inner loop is parallelized, while Fig. 7b shows the threads that are spawned. We can see that, when $i = 4$, $y[4]$ is loaded down the wrong thread. In the next iteration, when $i = 5$, $y[4]$ is accessed down the correctly forked thread. The wrong-thread execution prior to the correct thread's execution, in this case, will help in hiding the latency when reading $y[4]$.

## 4 Wrong Execution Cache

The indirect prefetching effect provided by the execution of loads down the wrong paths and the wrong threads may be able to reduce the number of subsequent correct-path

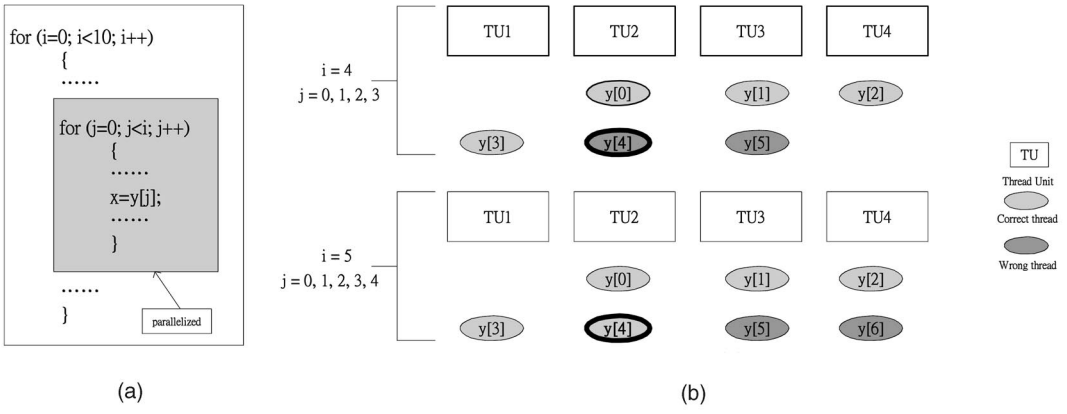



Fig. 7. *Wrong-Thread Execution*: (a) An example code segment in which the inner loop is parallelized. (b) Threads spawned and loads issued down correctly and incorrectly forked threads. When $i = 4$, $j = 0, 1, 2, 3$, and $y[0]$, $y[1]$, $y[2]$, and $y[3]$ are loaded down the correctly forked threads executing in TU2, TU3, TU4, and TU1, respectively. On the other hand, $y[4]$ and $y[5]$ are issued to the cache system down the wrongly forked threads executing in TU2 and TU3, respectively. When $i = 5$, $j = 0, 1, 2, 3, 4$, and $y[0]$, $y[1]$, $y[2]$, $y[3]$, and $y[4]$ are loaded down the correctly forked threads. Since the result of $y[4]$ has already been brought into TU2's L1 data cache by the wrongly forked thread's execution in the previous iteration of the loop (for $i = 4$), the request for $y[4]$ by TU2's processor (when $i = 5$) can be serviced faster than when the incorrectly forked threads are not allowed to continue executing.

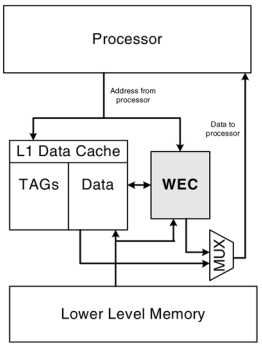



Fig. 8. The placement of the Wrong Execution Cache in the memory hierarchy.

misses. However, these additional wrongly executed loads also may reduce the performance since the cache pollution caused by these loads might offset the benefits of their indirect prefetching effect. This cache pollution can occur when the wrong execution loads move blocks into the data cache that are never needed by the correct execution path. It is also possible for the cache blocks fetched by the wrong execution loads to evict blocks that are still required by the correct path. This effect is likely to be more pronounced for low-associativity caches. In order to eliminate this cache pollution, we introduce the *Wrong Execution Cache* (WEC).

### 4.1 Basic Operation of the WEC

The WEC is used to store cache blocks fetched by wrong execution loads separately from those fetched by loads known to be issued from the correct path, which are stored in the regular L1 data cache. The WEC is accessed in parallel with the L1 data cache. Only those loads that are known to be issued from the wrong execution path, that is, after the control speculation result is known, are handled by the WEC. The data blocks fetched by loads issued before the control speculation is cleared are put into the L1 data cache. After the speculation is resolved, however, a wrong execution load that causes a miss in both the L1 data cache and the WEC will cause an access to be made to the next level memory. The required block is moved into the WEC to eliminate any cache pollution that might be caused by the wrong execution load. If a load causes a miss in the L1 data cache, but a hit in the WEC, the block is simultaneously transferred to both the processor and the L1 data cache.

A load from the correct path that hits on a block previously fetched by a wrong execution load also initiates a next-line prefetch. The block fetched by this next-line prefetch is placed into the WEC. When a correct-execution load causes a miss, the data block is moved into the L1 data cache instead of the WEC, as would be done in a standard cache configuration. The WEC also acts as a victim cache [11] by caching the blocks evicted from the L1 cache by cache misses from the correct execution path. In summary, the WEC is a combination of a prefetch buffer for wrong execution loads and a victim cache for evictions from the L1 data cache. The placement of the WEC in the memory hierarchy is shown in Fig. 8 and its operation is summarized in Fig. 9.

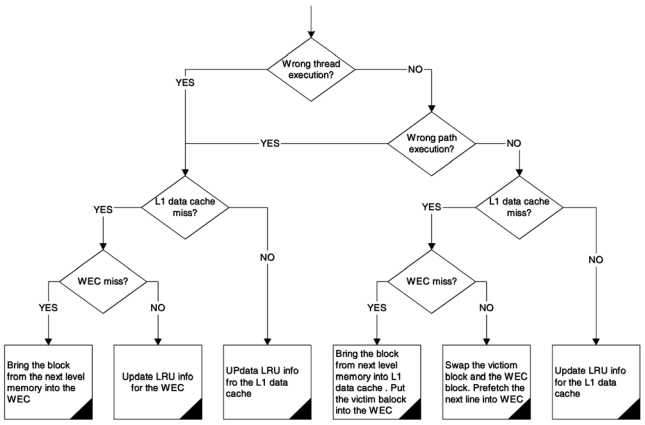



Fig. 9. Flowchart of a WEC access.

### 4.2 Discussion: Future Considerations for WEC

The speculative loads before the branch resolution are placed into L1 data cache as discussed in the previous sections. Only loads that are known to be issued from wrong paths or wrong threads are placed into WEC. Our algorithm can easily be extended to the case where WEC is capable of storing the results of speculative loads before they are known to be from the wrong paths or wrong threads. This can be done by adding a mechanism such as the Load Miss Queue (LMQ) in the Alpha21264 and the IBM Power 4. The Alpha 21264 and the IBM Power4, both incorporate a load miss queue (LMQ) to keep track of L1 cache misses. Instructions are flushed from the pipeline when a branch misprediction occurs. However, entries in the LMQ are simply marked instead of being flushed. When the data returns for an entry that is marked in the LMQ, it is put into the L1 cache, but it is not written to the register file. In the WEC design presented in this paper, we go one step further than existing processors by intentionally continuing to execute as many ready loads as possible down the mispredicted branch path and mispredicted thread even after the speculative operation is known to be wrong. A mechanism such as the LMQ can be used to determine whether a value should be put into the WEC after the miss is serviced rather than making this decision at issue time. This delay in determining what should be placed in the WEC should further increase its potential performance. In this study, however, our intent was to show the potential of our definition of wrong execution. The specific improvements to help its performance were left to be considered in future.

### 4.3 Incorporating the WEC into the STA

Each TU in the STA used in this study has its own private L1 data cache. In addition, a private WEC is placed in parallel with each of the L1 caches. To enforce coherence among the caches during the execution of a parallel section of code, all possible data dependences in a thread's execution path are conservatively identified. These potentially shared data items are stored in each TU's private speculative memory buffer until the write-back stage is executed. Updates to shared data items made by a thread during the execution of a parallel

TABLE 1
Program Transformations Used in Manually Transforming the Benchmark Programs Tested to the Thread-Pipelining Execution Model

| Transformations | 164.gzip | 175.vpr | 197.parser | 181.mcf | 183.equake | 177.mesa |
|---|---|---|---|---|---|---|
| Loop Coalescing | | | | | | ✓ |
| Loop Unrolling | ✓ | | | | ✓ | ✓ |
| Statement Reordering to Increase Overlap | | ✓ | ✓ | ✓ | ✓ | |

section of code are passed to downstream threads via a unidirectional communication ring.

During sequential execution, a simple update protocol is used to enforce coherence. When a cache block is updated by the single thread executing the sequential code, all the other idle threads that cache a copy of the same block in their L1 caches or WECs are updated simultaneously using a shared bus. This coherence enforcement during sequential execution creates additional traffic on the shared bus. This traffic is directed only to what would otherwise be idle caches, however, and does not introduce any additional delays.

## 5 Experimental Methodology

This study uses the SIMCA (SImulator for Multithreaded Computer Architecture) simulator [7] to model the performance effects of incorporating the WEC into the STA. This simulator is based on the cycle-accurate SimpleScalar simulator, *sim-outorder*, version 3.0 [9]. SIMCA is execution driven and performs both functional and timing simulation.

### 5.1 TU parameters

Each TU uses a 4-way associative branch target buffer with 1,024-entries and a fully associative speculative memory buffer with 128 entries. The distributed L1 instruction caches are each 32KB and 2-way associative. The default unified L2 cache is 512KB, 4-way associative, with a block size of 128 bytes, although we do experiment with other L2 cache sizes where noted. The round-trip memory latency is 200 cycles. The L1 cache parameters are varied as described in Section 6.

The time required to initiate a new thread (the fork delay) in the STA includes the time required to copy all of the needed global registers to a newly spawned thread's local register file and the time required to forward the program counter and the necessary *target-store* addresses from the memory buffer. We use a fork delay of four cycles [2] in this study plus two cycles per value to transfer data between threads after a thread has been forked.

### 5.2 Benchmark Programs

Four SPEC2000 integer benchmarks (*vpr*, *gzip*, *mcf*, *parser*) and two SPEC2000 floating-point benchmarks (*equake*, *mesa*) are evaluated in this study. All of these programs are written in C. The compiler techniques shown in Table 1 were used to manually parallelize these programs for execution on the STA following the steps that would be applied by an actual compiler [20]. The loops chosen for parallelization were identified with runtime profiling as the most time-consuming loops in each program. Table 2 shows the fraction of each program that we were able to parallelize.

The GCC compiler, along with modified versions of the GAS assembler and the GAD loader, were used to compile the parallelized programs. The resulting parallelized binary code was then executed on the simulator. Fig. 10 shows the complete compilation process. Each benchmark was optimized using optimization level O3 and run to completion. To keep simulation times reasonable, the MinneSPEC [16] reduced input sets were used for several of the benchmark programs, as shown in Table 2.

### 5.3 Processor Configurations

The following STA configurations are simulated to determine the performance impact of executing wrong-path and wrong-thread loads and the performance enhancements attributable to the WEC.

*orig*: This is the baseline supertheaded architecture described in the previous sections.

*vc*: This configuration adds a small fully-associative victim cache in parallel with the L1 data cache to the *orig* configuration.

*wp*: This configuration adds more aggressive speculation to a TU's execution as described in Section 3.1.

*wth*: This configuration is described in detail in Section 3.2. To summarize, speculatively forked threads in this configuration are allowed to continue execution *even after* they are known to have been mispredicted. Wrong threads are

TABLE 2
The Dynamic Instruction Counts of the Benchmark Programs Used in this Study and the Fraction of These Instructions that Were Executed in Parallel

| Benchmark | Suite/Type | Input Set | Whole Benchmark Instruction Count (M) | Targeted loops Instruction Count (M) | Fraction of Dynamic Instructions Parallelized |
|---|---|---|---|---|---|
| **175.vpr** | SPEC2000/INT | SPEC test | 1126.5 | 97.2 | 8.6% |
| **164.gzip** | SPEC2000/INT | MinneSPEC large | 1550.7 | 243.6 | 15.7% |
| **181.mcf** | SPEC2000/INT | MinneSPEC large | 601.6 | 217.3 | 36.1% |
| **197.parser** | SPEC2000/INT | MinneSPEC medium | 514.0 | 88.6 | 17.2% |
| **183.equake** | SPEC2000/FP | MinneSPEC large | 716.3 | 152.6 | 21.3% |
| **177.mesa** | SPEC2000/FP | SPEC test | 1832.1 | 319.0 | 17.3% |

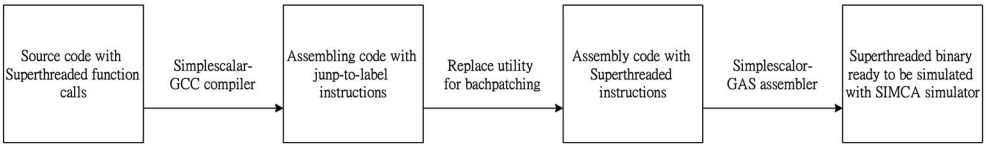



Fig. 10. The compilation process for generating parallelized superthreaded code for the simulations.

squashed before they reach the write-back stage of the thread-execution pipeline to prevent wrongly executed loads from altering the target register. Since each thread's store data is put into the speculative memory buffer during a thread's execution and wrong threads cannot execute their write-back stages, no wrong-thread store alters the memory system. The speculative load execution within a correct TU (superscalar core) remains the same in this configuration as in the *orig* configuration.

*wth-wp*: This is a combination of the *wp* and *wth* configurations.

*wth-wp-vc*: This configuration is the *wth-wp* configuration with the addition of a victim cache. It is used to compare against the performance improvement made possible by caching the wrong-path and wrong-thread loads in the WEC.

*wth-wp-wec*: This is the *wth-wp* configuration with the addition of a small, fully associative WEC in parallel with each TU's L1 data cache. The details of the WEC are given in Section 4.1.

*nlp*: This configuration implements next-line tagged prefetching [12] with a fully associative prefetch buffer, but without any other form of speculative execution. A prefetch is initiated on a miss and on the first hit to a previously prefetched block. The results of these prefetches are put into the prefetch buffer. Tagged prefetching has previously been shown to be more effective than prefetching only on a miss [13]. The model is very similar to the stream buffer first proposed by Farkas et al. [24], but without enhancements such as allocation filters and hardware support for dynamic strides. There are enhancements that could be made to both the prefetch buffer used here and the WEC, but a more detailed comparison of these enhancements is beyond the scope of this paper. We have used this configuration to compare against the ability of the WEC to successfully prefetch blocks that will be used by subsequently executed loads issued from a correct execution path.

## 6 Evaluation of Simulation Results

We first examine the baseline performance of the STA, followed by an evaluation of the performance of the WEC when using different numbers of TUs. The effects of wrong execution, both with and without a WEC, on the performance of the STA are subsequently examined. We also study the sensitivity of the WEC to several important memory system parameters and analyze the reduction in the number of L1 data cache misses and the increase in the memory traffic due to the WEC.

The overall execution time is used to determine the percentage change in performance of the different configurations tested relative to the execution time of the baseline configuration. Average speedups are calculated using the execution time weighted average of all of the benchmarks [10]. This weighting gives equal importance to each benchmark program independent of its total execution time.

### 6.1 Baseline Performance of the STA

The system parameters used to test the baseline performance, and to determine the amount of parallelism actually exploited in the benchmark programs [8] are shown in Table 3. The size of the distributed 4-way associative L1 data cache in each TU is scaled from 2K to 32K as the number of TUs is varied to keep the total amount of L1 cache in the system constant at 32K.

The baseline for these initial comparisons is a single-thread, single-issue processor which does not exploit any parallelism. The single-thread-unit, 16-issue processor corresponds to a very wide issue superscalar processor that is capable of exploiting only instruction-level parallelism. In the 16TU superthreaded processor, each thread can issue only a single instruction per cycle. Thus, this configuration exploits only thread-level parallelism. The other configurations exploit a combination of both instruction and thread-level parallelism. Note that the total amount of parallelism available in all of these configurations is constant at 16 instructions per cycle.

Fig. 11 shows the amount of instruction and thread-level parallelism in the parallelized portions of the benchmarks to thereby compare the performance of the superthreaded processor with a conventional superscalar processor. The single TU configuration at the left of each set of bars is capable of issuing 16 instructions per cycle within the single thread. As you move to the right within a group, there are more TUs, but each can issue a proportionally smaller number of instructions per TU so that the total available parallelism is fixed at 16 instructions per cycle.

In these baseline simulations, *164.gzip* shows high thread-level parallelism with a speedup of 14x for the 16TU X 1-issue configuration. A 1TU X 16-issue configuration gives a speedup less than $4x$ when executing this program. The *175.vpr* benchmark appears to have more instruction-level parallelism than thread-level parallelism since the speedup of the parallelized portion of this

TABLE 3
Simulation Parameters Used for Each TU

| # of TUs<br>Issue rate | 1<br>1 | 1<br>16 | 2<br>8 | 4<br>4 | 8<br>2 | 16<br>1 |
|---|---|---|---|---|---|---|
| Reorder buffer size | 8 | 128 | 64 | 32 | 16 | 8 |
| INT ALU | 1 | 16 | 8 | 4 | 2 | 1 |
| INT MULT | 1 | 8 | 4 | 2 | 1 | 1 |
| FP ALU | 1 | 16 | 8 | 4 | 2 | 1 |
| FP MULT | 1 | 8 | 4 | 2 | 1 | 1 |
| L1 data cache size (K) | 2 | 32 | 16 | 8 | 4 | 2 |

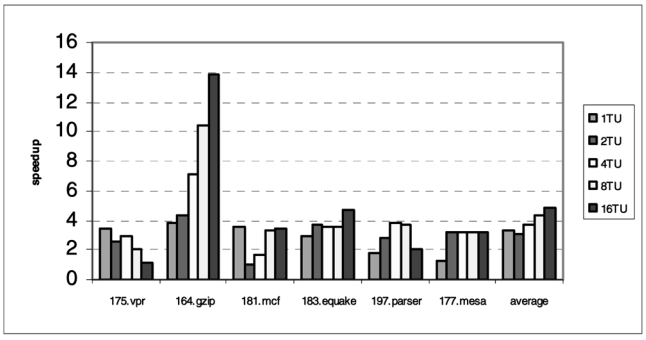



Fig. 11. Performance of the superthreaded processor for the parallelized portions of the benchmarks with the hardware configurations shown in Table 3. The baseline configuration is a single-threaded, single-issue processor.

program decreases as the number of TUs increases. For most of the benchmark programs, the performance tends to improve as the number of TUs increases. This behavior indicates that there is more thread-level parallelism in the parallelized portions of the benchmark programs than simple instruction-level parallelism.

In the cases where the pure superscalar model achieves the best performance, it is likely that the clock cycle time of the very wide issue superscalar processor would be longer than the combined models or the pure superthreaded model. This increase in cycle time would occur since the 16-issue superscalar model would require a very large instruction reorder buffer for dynamic instruction scheduling. On average, we see that the thread-level parallelization tends to outperform the pure instruction-level parallelization.

### 6.2 Performance of the Superthreaded Processor with the WEC

Based on the results in the previous section and considering what is expected for future processor development, we use eight TUs, where each TU is an 8-issue out-of-order processor, in the remainder of the study. In some of the experiments, however, we vary the number of TUs to study the impact of varying the available thread-level parallelism on the performance of the WEC.

Each of the TUs has a load/store queue size of 64 entries. The reorder buffer also is 64 entries. The processor has eight integer ALU units, four integer multiply/divide units, eight floating-point (FP) adders, and four FP multiply/divide units. The default L1 data cache in each TU is 8KB, direct-mapped, with a block size of 64 bytes. The default WEC has eight entries and is fully associative with the same block size as the L1 data cache.

Since our focus is on improving the performance of on-chip direct-mapped data caches in a speculative multithreaded architecture, most of the following comparisons for the WEC are made against a victim cache. We also examine the prefetching effect of wrong execution with the WEC by comparing it with next-line tagged prefetching.

Fig. 12 shows the performance of the *wth-wp-wec* configuration as the number of TUs is varied. Each TU is an 8-issue out-of-order processor in Figs. 12 and 13. So, as the number of TUs increases, the total issue width increases. The results are for the entire benchmark program, not just the parallelized loops. The baseline is the *orig* configuration with a single TU. We see that the speedup of the *wth-wp-wec* configuration can be as much as 39.2 percent (*183.equake*). For most of the benchmarks, we see that even a two-thread-unit *wth-wp-wec* performs better than the *orig* configuration with 16 TUs.

The single-thread *wth-wp-wec* configuration shows that adding the WEC to the STA can improve the performance significantly, up to 10.4 percent for *183.equake*, for instance. When more than one TU is used, we see even greater improvements when using the WEC due to the larger number of wrong loads issued by executing the wrong threads. For example, in Fig. 13, we see that the performance of *181.mcf* improves from 6.2 percent compared to the baseline configuration when executing with a single TU to a 20.2 percent increase over the baseline configuration when using 16 TUs. On average, the performance of the *wth-wp-wec* configuration increases with the number of threads because the total size of the WEC and the L1 cache increases in the parallel configurations, although the ratio of the WEC size to the L1 data cache size remains constant. Once the total cache and WEC sizes match the benchmark's memory footprint, the performance improvement levels off.

The *175.vpr* program slows down on the *orig* configuration compared to the single-thread performance because there is not enough overlap among threads when using

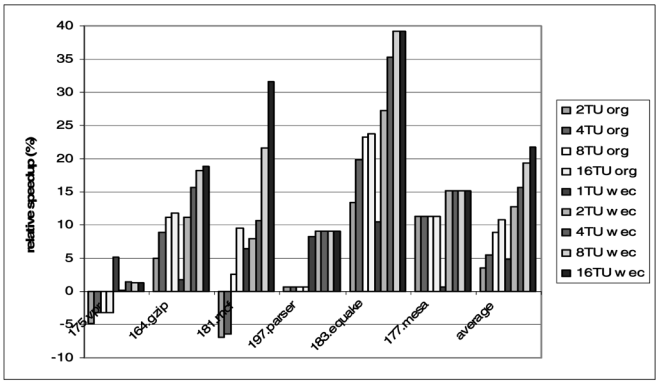



Fig. 12. Performance of the entire benchmark programs as the number of TUs is varied. The baseline processor is a superthreaded processor with a single TU. Each TU is an 8-issue out-of-order processor.

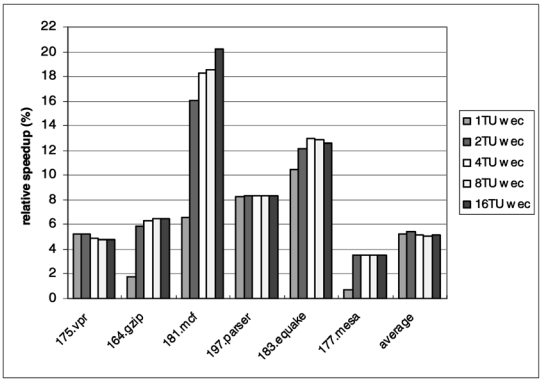



Fig. 13. Performance of the *wth-wp-wec* configuration on top of the parallel execution. The baseline processors are one to 16-TU superthreaded processors, with the number of TUs corresponding to the number of threads used in the *wth-wp-wec* configuration. Each TU has an 8-issue core.

more than one TU. As a result, the superthreading overhead overwhelms the benefits of executing the program in parallel. The *181.mcf* program also shows some slowdown for two and four TUs because of contention for thread units.

Figs. 12 and 13 showed the performance improvement obtained by executing wrong-path and wrong-load instructions with a WEC in each TU as the total number of available TUs was varied. Fig. 14, in contrast, compares the relative speedup obtained by all of the different processor configurations described in Section 4.3 compared to the baseline processor, *orig*. All of these configurations use eight TUs.

We see from this figure that the combination of wrong execution plus the WEC (*wth-wp-wec*) gives the greatest speedup of all the configurations tested. The use of only wrong-path or wrong-thread execution alone or in combination (*wp*, *wth*, or *wth-wp*) provides very little performance improvement. When they are used together (*wth-wp*), for instance, the best speedup is only 2.2 percent (for *183.equake*), while there is some slowdown for *177.mesa*. It appears that the cache pollution caused by executing the wrong loads in these configurations offsets the benefit of their prefetching effect. When the WEC is added, however, the cache pollution is eliminated, which produces speedups of up to 18.5 percent (*181.mcf*), with an average speedup of 9.7 percent. Compared to a victim cache of the same size, the configurations with the WEC show substantially better performance. While the WEC (*wth-wp-wec*) and the victim cache (*wth-wp-vc*) both reduce conflict misses, the WEC further eliminates the pollution caused by executing loads from the wrong paths and the wrong threads. The WEC also performs better than conventional next-line tagged prefetching (*nlp*) with the same size prefetch buffer. Note that the extra hardware cost of both configurations would be approximately the same. On average, conventional next-line prefetching (*nlp*) produces a speedup of 5.5 percent, while the WEC (*wth-wp-wec*) produces a speedup of 9.7 percent.

### 6.3 The Overhead Due to the Wrong Execution

The overhead of wrong execution is shown in Figs. 15 and 16. Fig. 15 shows the percentage of extra threads executed that are wrong threads. On average, there is a 3.2 percent increase in the number of threads executed due to wrong-thread execution compared to when not allowing wrong-thread execution. The execution of these wrong threads is overlapped with the execution of the following sequential code. As a result, these wrong threads do not compete for hardware resources with the single correct execution thread since the wrong threads are executing on what would otherwise be idle thread units.

Fig. 16 shows that the performance increase from executing wrong threads comes at the cost of an increase in the traffic between the processor and the L1 cache. This increase in cache traffic is a side effect of issuing more load instructions from both the wrong execution path and the mis-speculated threads. This traffic increase can be as high as 30 percent in *175.vpr*, with an average increase of 14 percent. This relatively small average increase in cache traffic would appear to be more than offset by the increase in performance provided by using the WEC, though. Fig. 16 also shows that the WEC can significantly reduce the number of misses in the L1 data cache. This reduction is as high as 73 percent for *177.mesa*, although the miss count reduction for *181.mcf* is not as significant as the others.

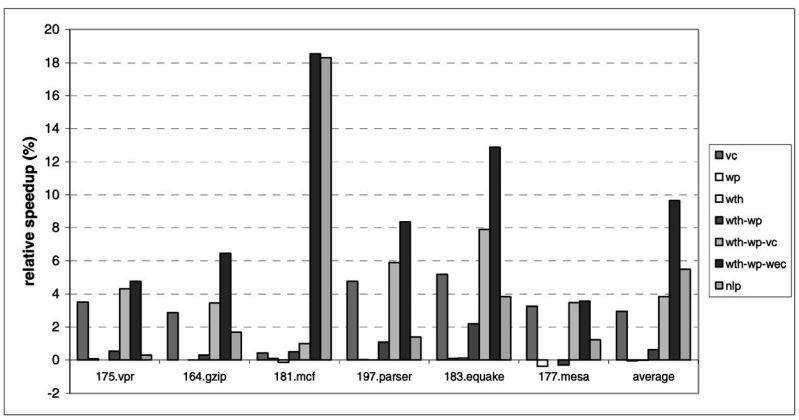



Fig. 14. Relative speedups obtained by the different processor configurations' parallel execution with eight TUs. The baseline is the original superthreaded parallel execution with eight TUs.

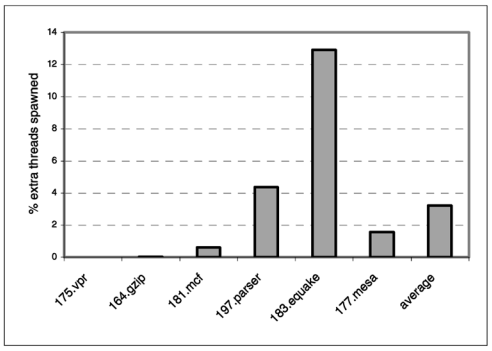



Fig. 15. Percentage of extra threads executed due to wrong-thread execution. The baseline is the total number of executed threads when not allowing wrong-thread execution.

### 6.4 Quantifying Useful Data in the WEC

In Figs. 17 and 18, we show the amount of wrong-path and wrong-thread data in WEC and how much of it is used. Fig. 17 shows the WEC data saved from wrong paths and wrong threads. We can see that the amount of data that are moved into WEC by wrong-thread execution is negligible compared to the amount that are moved by wrong-path execution. The only exceptions are *183.equake* and *181.mcf* with 13 percent and 4 percent, respectively. However, the fraction of dynamic instructions in the benchmarks that were executed in parallel is also low for most of the benchmarks (see Table 2 in Section 5.1).

From Fig. 18, we see that approximately 32-60 percent of the blocks that are moved into the WEC are used before being evicted by subsequent accesses. However, not all of these accesses are useful to the correct execution path. The lighter portion of each bar in this figure shows the fraction of all of the blocks accessed in the WEC that are actually used by loads issued from the correct execution path. For *181.mcf*, for example, 52 percent of the blocks in the WEC are accessed before being evicted. Furthermore, of these blocks, only 25 percent are actually useful to the execution of the correct branch path. This result suggests that there is space for improvement since the pollution in WEC can also be reduced. However, we did not consider a filtering mechanism to improve the effectiveness of WEC in this study.

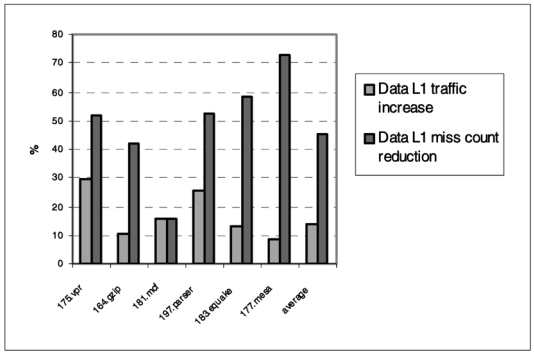



Fig. 16. Increases in the L1 cache traffic and the reduction in L1 misses when using wrong-thread execution.

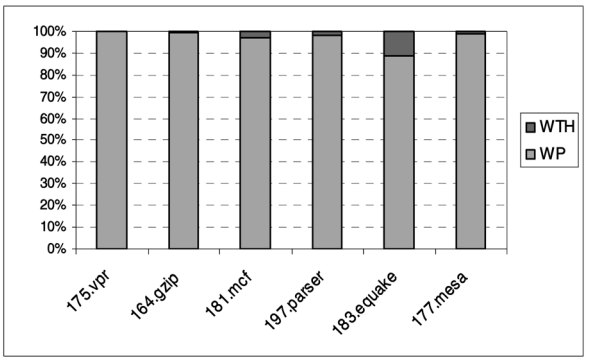



Fig. 17. The percentage of the wrong-execution blocks in WEC saved from Wrong Thread (WTH) or Wrong Path (WP).

### 6.5 Sensitivity Analysis

In this section, we study the effects of varying the L1 data cache associativity, the L1 data cache size, and the shared L2 cache size on the performance of the WEC. Each simulation in this section uses eight TUs.

#### 6.5.1 Impact of the L1 Data Cache Associativity

Increasing the L1 cache associativity typically tends to reduce the number of L1 misses for both correct execution [14] and wrong execution [3]. The reduction in misses in the wrong execution paths reduces the number of indirect prefetches issued during wrong execution, which then reduces the performance improvement from the WEC, as shown in Fig. 19. The baseline configuration is this figure is the *orig* processor with a direct-mapped and 4-way associative L1 data corresponding to the direct-mapped and 4-way WEC results. When the associativity of the L1 cache is increased, the speedup obtained by the victim cache (*vc*) disappears. However, the configuration with the wrong execution cache, *wth-wp-wec*, still provides significant speedup. This configuration also substantially outperforms the *wth-wp-vc* configuration, which issues loads from the wrong execution paths, but uses a standard victim cache instead of the WEC.

#### 6.5.2 The Effect of the L1 Data Cache Size

Fig. 20 shows the speedups for the *orig* and *wth-wp-wec* configurations when the L1 data cache size is varied. The

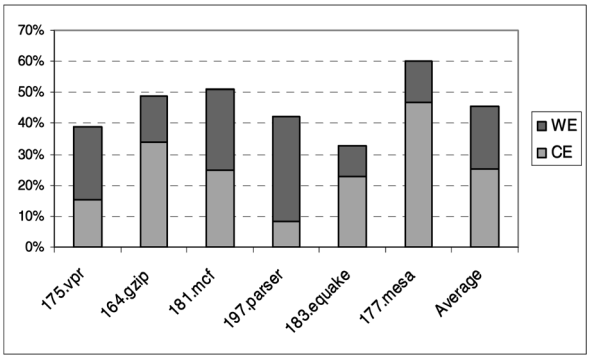



Fig. 18. Percentage of the blocks that are moved into the WEC that are used before being evicted. The proportion of hits used by the Wrong Execution (WE) and the Correct Execution (CE) are also given.

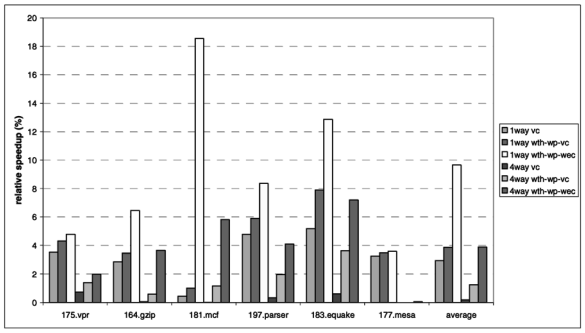



Fig. 19. Performance sensitivity of an eight-TU superthreaded processor with 8-issue superscalar cores and a WEC as the associativity of the L1 data cache is varied (direct-mapped, 4-way).

baseline for the comparison is the *orig* configuration with 4K L1 data cache. The speedups by *wth-wp-wec* are layered on top of the corresponding non-*wth-wp-wec* ones (*orig*). We see that the relative speedup produced by the WEC (*wth-wp-wec*) decreases as the L1 data cache size is increased. Note, however, that the WEC size is kept constant throughout this group of simulations so that the relative size of the WEC compared to the L1 data cache is reduced as the L1 size is increased. With a larger L1 cache, the wrong execution loads produce fewer misses compared to the configurations with smaller caches. The smaller number of misses reduces the number of potential prefetches produced by the wrong execution loads, which thereby reduces the performance impact of the WEC.

We can see from Fig. 20 that, for all of the test programs, a small 8-entry WEC with an 8K L1 data cache exceeds the performance of the baseline processor (*orig*) when the cache size is doubled, but without the WEC. Furthermore, on average, the WEC with a 4K L1 data cache performs better than the baseline processor with a 32K L1 data cache. These results suggest that incorporating a WEC into the processor is an excellent use of chip area compared to simply increasing the L1 data cache size.

### 6.5.3 The Effect of the WEC Size

Fig. 21 shows that, in general, the configuration that is allowed to issue loads from both the wrong paths and the wrong threads with a 4-entry victim cache (*wth-wp-vc*) outperforms the *orig* configuration with a 16-entry victim cache. Furthermore, replacing the victim cache with a 4-entry WEC causes the *wth-wp-wec* configuration to outperform the configuration with a 16-entry victim cache, *wth-wp-vc*. This trend is particularly significant for *164.gzip*, *181.mcf*, and *183.equake*.

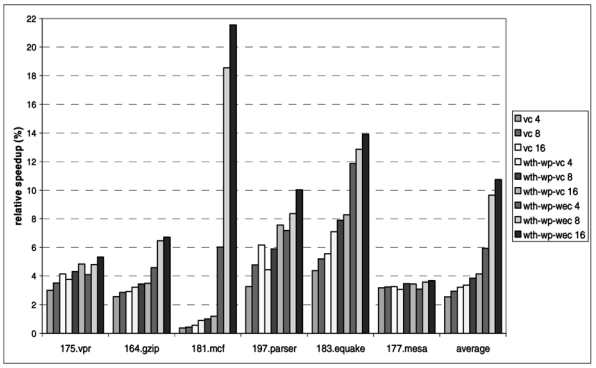



Fig. 21. Performance sensitivity of an eight-TU superthreaded processor with 8-issue superscalar cores and a WEC to changes in the size of the WEC (4, 8, 16 entries) compared to a *vc*.

Fig. 22 compares the WEC approach to a tagged prefetching configuration that uses a prefetch buffer that is the same size as the WEC. It can be seen that the *wth-wp-wec* configuration with an 8-entry WEC performs substantially better than traditional next-line prefetching (*nlp*) with a 32-entry prefetch buffer. This result indicates that the WEC is actually a more efficient prefetching mechanism than a traditional next-line prefetching mechanism.

## 7 Related Work

Pierce and Mudge [17] suggested that the additional loads issued from mispredicted branch paths could provide some performance benefits and proposed a wrong-path instruction prefetching scheme [18] in which instructions from both possible branch paths were prefetched. Currently, wasted resource, wrong-path execution has recently received attention from researchers. Akkary et al. [33] used wrong-path execution to improve branch prediction in superscalar processors with very deep pipelines. Their method, recycling waste, used wrong-path branch informa-

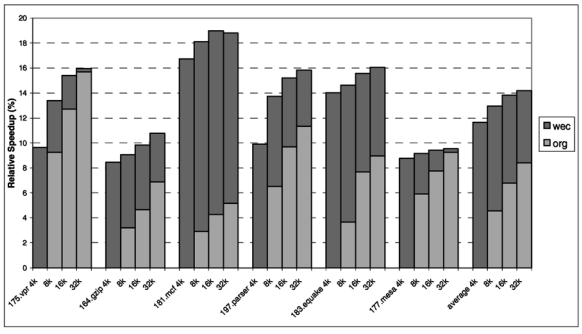



Fig. 20. Performance sensitivity of an eight-TU superthreaded processor with 8-issue superscalar cores and a WEC as the L1 data cache size is varied (4K, 8K, 16K, 32K). The baseline is the original superthreaded processor with 4K L1 data cache.

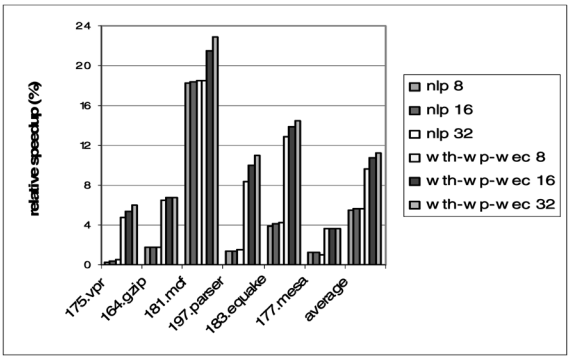



Fig. 22. Performance sensitivity of an eight-TU superthreaded processor with 8-issue superscalar cores and a WEC to changes in the size of the WEC (4, 8, 16 entries) compared to *nlp*.

tion along with the correct path branch history and extended the branch prediction to allow the recycling of wrong-path branch outcomes at the fetch stage. While recycling waste was proposed for superscalar processors, the authors claimed that their method was applicable to other types of speculative execution, as well. In [33], the authors also mentioned extending their work in the future to allow wrong-path execution to continue after the branch recovery, which we have discussed previously in [3] and [23] for superscalar processors and concurrent multithreaded architectures, respectively.

Instruction fetch supply disruptions caused by branch mispredictions limit the performance. There has been a great effort to improve the branch prediction accuracies. However, studies that looked into simultaneous multipath execution are more related to our work in this paper. Ahuja et al. [31] discussed, in the context of single thread execution, the limits of multipath execution, the fetch-bandwidth needed for multipath execution, and various dynamic confidence-prediction schemes that gauged the likelihood of branch mispredictions. This work indicated that multipath execution could exploit extra hardware in multithreading and other forms of parallelism, with other multithreading mechanisms themselves cannot.

In [30], Tyson et al. proposed the limited dual path execution. Although executing down both paths (correct/wrong paths, taken/not-taken paths) of a conditional branch enables the branch penalty to be minimized, it is infeasible because instruction fetch rates far exceed the capability of the pipeline to retire a single branch before others must be processed. This approach also consumes a substantial amount of processor resources. The work in [30] used a two-level branch predictor's history table as a dynamic confidence predictor to restrict the dual path execution and thus make it feasible.

All of the above-mentioned studies were for single-thread execution and their wrong-path execution definition is different than our definition in this paper. The previous studies called a prior speculatively executed path a wrong path after the branch result is known and the execution does not continue after the path is known to be wrong. In our definition, however, the wrong path starts after the speculative path is known to be wrong and the execution continues down this path in parallel with the correct path, but only for loads that are ready at this time.

There has been no previous work that has examined the impact of executing loads from a mispredicted thread in a concurrent multithreaded architecture. A few studies have examined different forms of speculative execution, such as loop-continuation in the dynamic multithreading (DMT) processor [32] and prefetching in the Simultaneous MultiThreading (SMT) architecture [6], [19].

Akkary and Driscoll [32] presented a dynamic multithreading processor with lookahead execution beyond mispredicted branches. In this study, the hardware automatically broke up a program into loops and procedure threads that executed as different threads in the SMT pipeline on the superscalar processor. The control logic kept a list of the thread order in the program, along with the start PC of each thread. A thread would stop fetching instructions when its PC reached the start of the next thread in the order list. If an earlier thread never reached the start PC of the next thread in the order list, the next thread was considered to be mispredicted and was squashed.

Collins et al. [6] studied the use of idle thread contexts to perform prefetching based on a simulation of the Itanium processor that had been extended to perform simultaneous multithreading. Their approach speculatively precomputed future memory accesses using a combination of software, existing Itanium processor features, and additional hardware support. Similarly, using idle threads on an Alpha 21464-like SMT processor to preexecute speculative addresses and, thereby, prefetch future values to accelerate the main thread has also been proposed [19].

These previous studies differ from our work in this paper in several important ways. First, this current study extends these previous evaluations of superscalar and SMT architectures to a concurrent multithreading architecture. Second, our mechanism requires only a small amount of extra hardware, which is transparent to the processor—no extra software support is needed. Third, while we also use threads that would be idle if there was no wrong-thread execution, our goal is not to help the main thread's current execution, but rather to accelerate the future execution of the currently idle threads.

Our initial results on the effect of executing load instructions down the mispredicted branch paths [3] and mispredicted threads [23] are promising and should encourage and motivate further study of the topic.

## 8 Conclusions

In this study, we have examined the effect of executing load instructions issued from a mispredicted branch path (wrong-path) or from a misspeculated thread (wrong-thread) on the performance of a speculative multithreaded architecture. We find that we can reduce the cache misses for subsequent correctly predicted paths and threads by continuing to execute the mispredicted load instructions even after the instruction or thread-level control speculation is known to be incorrect.

Executing these additional loads causes some cache pollution by fetching never needed blocks and by evicting useful blocks needed for the later correct execution paths and threads. This effect is likely to be more pronounced for low associativity caches. In order to eliminate the potential pollution caused by the mispredicted load instructions, we introduced the small, fully associative *Wrong Execution Cache* (WEC). Our simulation results show that the WEC can improve the performance of a concurrent multithreaded architecture up to 18.5 percent on the benchmark programs tested, with an average improvement of 9.7 percent. This performance improvement comes from reducing the number of cache misses by, typically, 42-73 percent. While this study has examined the effects of several parameters on the performance of the WEC, there are still many important factors left to be considered, such as the effects of memory latency, the block size, and the relationship of the branch prediction accuracy to the performance of the WEC.

The WEC proposed in this work is one possible structure for exploiting the potential benefits of executing mispre-

dicted load instructions. Although this current study is based on a multithreaded architecture that exploits loop level parallelism, the ideas presented in this paper can be easily used in all types of multithreaded architectures executing general workloads.

## ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation grants EIA-9971666 and CCR-9900605, IBM Corporation, Compaq's Alpha development group, and the Minnesota Supercomputing Institute. Resit Sendag was supported in part by the University of Rhode Island new faculty startup fund. Preliminary versions of this work were presented at the ACM Euro-Par 2002 Conference [3] and the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003) [23]. The authors would like to thank the anonymous reviewers for their thoughtful comments on earlier versions of this paper. Their suggestions helped us to improve the presentation of this paper significantly.

## REFERENCES

[1] T. Ungerer, B. Robic, and J. Silc, "Multithreaded Processors," *The Computer J.,* vol. 45, no. 3, pp. 320-348, 2002.

[2] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew, "The Superthreaded Processor Architecture," *IEEE Trans. Computers,* vol. 48, no. 9, pp. 881-902, Sept. 1999.

[3] R. Sendag, D.J. Lilja, and S.R. Kunkel, "Exploiting the Prefetching Effect Provided by Executing Mispredicted Load Instructions," *Proc. Eighth Int'l Euro-Par Conf. Parallel Processing,* pp. 468-480, 2002.

[4] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 414-425, 1995.

[5] J.G. Steffan and T.C. Mowry, "The Potential for Thread-Level Data Speculation in Tightly-Coupled Multiprocessors," Technical Report CSRI-TR-350, Computer Science Research Inst., Univ. of Toronto, Feb. 1997.

[6] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* pp. 14-25, 2001.

[7] J. Huang, "The SImulator for Multithreaded Computer Architecture," Technical Report ARCTiC 00-05, Laboratory for Advanced Research in Computing Technology and Compilers, Univ. of Minnesota, June 2000.

[8] J.-Y. Tsai, Z. Jiang, E. Ness, and P.-C. Yew, "Performance Study of a Concurrent Multithreaded Processor," *Proc. Fourth Int'l Symp. High-Performance Computer Architecture,* pp. 24-35, 1998.

[9] D.C. Burger, T.M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Technical Report CS-TR-96-1308, Univ. of Wisconsin-Madison, July 1996.

[10] D.J. Lilja, *Measuring Computer Performance.* Cambridge Univ. Press, 2000.

[11] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 364-373, 1990.

[12] J.E. Smith and W.-C. Hsu, "Prefetching in Supercomputer Instruction Caches," *Proc. Supercomputing Conf. '92,* pp. 588-597, 1992.

[13] S.P. VanderWiel and D.J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys,* vol. 32, no. 2, pp. 174-199, June 2000.

[14] A.J. Smith, "Cache Memories," *ACM Computing Surveys,* vol. 14, no. 3, pp. 473-530, Sept. 1982.

[15] D. Lee, J.-L. Baer, B. Calder, and D. Grunwald, "Instruction Cache Fetch Policies for Speculative Execution," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 357-367, 1995.

[16] A.J. Kleinosowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters,* vol. 1, May 2002.

[17] J. Pierce and T. Mudge, "The Effect of Speculative Execution on Cache Performance," *Proc. Eighth Int'l Symp. Parallel Processing,* pp. 172-179, 1994.

[18] J. Pierce and T. Mudge, "Wrong-Path Instruction Prefetching," *Proc. 29th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 165-175, 1996.

[19] C.-K. Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* pp. 40-51, 2001.

[20] B. Zheng, J. Tsai, B. Zang, T. Chen, B. Huang, J. Li, Y. Ding, J. Liang, Y. Zhen, P. Yew, and C. Zhu, "Designing the Agassiz Compiler for Concurrent Multithreaded Architectures," *Proc. Workshop Languages and Compilers for Parallel Computing,* pp. 380-398, 1999.

[21] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, "Program Optimization for Concurrent Multithreaded Architecture," *Proc. Workshop Languages and Compilers for Parallel Computing,* pp. 146-162, Aug. 1997.

[22] J.-Y. Tsai and P.-C. Yew, "The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation," *Proc. 1996 Conf. Parallel Architectures and Compilation Techniques,* pp. 35-46, 1996.

[23] Y. Chen, R. Sendag, and D.J. Lilja, "Using Incorrect Speculation to Prefetch Data in a Concurrent Multithreaded Processor," *Proc. IEEE Int'l Parallel and Distributed Processing Symp.,* p. 76, 2003.

[24] K.I. Farkas, N.P. Jouppi, and P. Chow, "How Useful Are Non-Blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?" Technical Report WRL RR 94/8, Digital Western Research Laboratory, Palo Alto, Calif., Dec. 1994.

[25] G. Reinman, T. Austin, and B. Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 234-245, 1999.

[26] S. Wallace, D. Tullsen, and B. Calder, "Instruction Recycling on a Multiple-Path Processor," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture,* pp. 44-53, 1999.

[27] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors," *Proc. First IEEE Symp. High-Performance Computer Architecture,* pp. 68-77, 1995.

[28] T.F. Chen and J.L. Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Trans. Computers,* vol. 44, no. 5, pp. 609-623, May 1995.

[29] G. Reinman, B. Calder, and T. Austin, "Fetch Directed Instruction Prefetching," *Proc. 32nd Int'l Symp. Microarchitecture,* pp. 16-27, 1999.

[30] G. Tyson, K. Lick, and M. Farrens, "Limited Dual Path Execution," Technical Report CSE-TR-346-97, Dept. of Electrical Eng. and Computer Science, Univ. of Michigan, Ann Arbor, 1997.

[31] P.S. Ahuja, K. Skadron, M. Martonosi, and D.W. Clark, "Multipath Execution: Opportunities and Limits," *Proc. 12th Int'l Conf. Supercomputing,* pp. 101-108, 1998.

[32] H. Akkary and M.A. Driscoll, "A Dynamic Multithreading Processor," *Proc. 31st Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 226-236, 1998.

[33] H. Akkary, S.T. Srinivasan, and K. Lai, "Recycling Waste: Exploiting Wrong-Path Execution to Improve Branch Prediction," *Proc. 17th Ann. Int'l Conf. Supercomputing,* pp. 12-21, 2003.

**Resit Sendag** received the PhD degree in electrical engineering from the University of Minnesota in Minneapolis, the MS degree in electrical engineering from Cukurova University in Adana, Turkey, and the BS degree in electronics engineering from Hacettepe University in Ankara, Turkey. He is currently an assistant professor of electrical and computer engineering at the University of Rhode Island in Kingston. His research interests include high-performance computer architecture, memory systems performance issues, and parallel computing. He is a member of the IEEE.

**Ying Chen** received the MS degree in electrical engineering from the University of Minnesota and the BS degree in electrical engineering from Tsinghua University. Currently, she is a PhD candidate in electrical engineering at the University of Minnesota. Her main research interests include hardware verification methodology, multiprocessors, multithreaded architectures, memory systems, and reconfigurable computing. She is a student member of the IEEE.

**David J. Lilja** received the PhD and MS degrees, both in electrical engineering, from the University of Illinois at Urbana-Champaign and the BS degree in computer engineering from Iowa State University. He is currently a professor of electrical and computer engineering and a fellow of the Minnesota Supercomputing Institute at the University of Minnesota in Minneapolis. He also serves as a member of the graduate faculties in computer science and scientific computation. He has been a visiting senior engineer in the Hardware Performance Analysis Group at IBM in Rochester, Minnesota, and a visiting professor at the University of Western Australia in Perth, supported by a Fulbright award. Previously, he worked as a research assistant at the Center for Supercomputing Research and Development at the University of Illinois and as a development engineer at Tandem Computers Inc. (now, a division of Hewlett-Packard) in Cupertino, California. He has chaired and served on the program committees of numerous conferences, was a distinguished visitor of the IEEE Computer Society, is a member of the ACM and a senior member of the IEEE, and is a registered professional engineer in electrical engineering in Minnesota and California. His primary research interests are in high-performance computer architecture, parallel computing, hardware-software interactions, nano-computing, and performance analysis.