Decoupled Software Pipelining with the Synchronization Array

Ram Rangan Neil Vachharajani Manish Vachharajani David I. August
Department of Computer Science
Princeton University
{ram, nvachhar, manishv, august}@cs.princeton.edu

Abstract

Despite the success of instruction-level parallelism (ILP) optimizations in increasing the performance of microprocessors, certain codes remain elusive. In particular, codes containing recursive data structure (RDS) traversal loops have been largely immune to ILP optimizations, due to the fundamental serialization and variable latency of the loop-carried dependence through a pointer-chasing load. *To address these and other situations, we introduce decoupled software pipelining (DSWP), a technique that statically splits a single-threaded sequential loop into multiple non-speculative threads, each of which performs useful computation essential for overall program correctness. The resulting threads execute on thread-parallel architectures such as simultaneous multithreaded (SMT) cores or chip multiprocessors (CMP), expose additional instruction level parallelism, and tolerate latency better than the original single-threaded RDS loop. To reduce overhead, these threads communicate using a* synchronization array, *a dedicated hardware structure for pipelined inter-thread communication. DSWP used in conjunction with the synchronization array achieves an 11% to 76% speedup in the optimized functions on both statically and dynamically scheduled processors.*

1. Introduction

Variable latency instructions are problematic for achieving high levels of parallelism in modern processors. Static scheduling is very effective at hiding fixed latencies by placing independent instructions between a long latency instruction and its consumer. Unfortunately, the compiler does not know how many instructions must be placed between a variable latency instruction and its consumer. Assuming best, average, or worst case latency for scheduling can detrimentally affect run-time performance of the code. Out-of-order (OOO) execution mitigates this problem to an extent. Rather than stalling when a consumer, whose dependences are not satisfied, is encountered, the OOO processor will execute instructions from after the stalled consumer. Thus, the compiler can now safely assume average latency since actual latencies longer than the average will not necessarily lead to execution stalls.

Unfortunately, the predominant type of variable latency instruction, memory loads, have worst case latencies (i.e., cache-miss latencies) so large that it is often difficult to find sufficiently many independent instructions after a stalled consumer. As microprocessors become wider and the disparity between processor speeds and memory access latencies grows [8], this problem is exacerbated since it is unlikely that dynamic scheduling windows will grow as fast as memory access latencies.

Despite these limitations, static and dynamic instruction scheduling combined with larger and faster caches have been largely successful in improving instruction-level parallelism (ILP) for many programs. However, the problems described above become particularly pronounced in certain programs that have unpredictable memory access patterns and few independent instructions near long latency loads. Loops that traverse recursive data structures (e.g., linked lists, trees, graphs, etc.) exhibit exactly these characteristics. Data items are not typically accessed multiple times and subsequent data accesses are referenced through pointers in the current structure, resulting in poor spatial and temporal locality. Since most instructions in a given iteration of the loop are dependent on the pointer value that is loaded in the previous iteration, these codes prove to be difficult to execute efficiently.

To address these concerns, software, hardware, and hybrid prefetching schemes have been proposed [10, 14, 15]. These schemes increase the likelihood of cache hits when traversing recursive data structures, thus reducing the latency of data accesses when the data is actually needed. Software techniques insert prefetching load instructions that attempt to predict future accesses using information statically available to the compiler [10]. Hardware techniques dynamically analyze memory access traces and send prefetch requests to the memory subsystem to bring data into the processor caches [14]. Despite these efforts, the performance of recursive data structure loop code is still far from ideal. Since prefetching techniques are speculative, they may not always end up fetching the right addresses at the right time. Achieving good coverage (i.e., the ratio of the number of useful addresses prefetched to the total number of addresses accessed) without performing an excessive number of loads is extremely difficult. The prefetching of useless data (i.e., poor prefetch accuracy) not only does not help performance but may in fact hinder it by causing useful data to be evicted from the cache and by occupying valuable memory access bandwidth. Consequently, the accuracy of prefetchers is extremely important. Achieving high coverage *and* accuracy is very difficult in these speculative frameworks.

In this paper, we present a technique called *decoupled software pipelining* (DSWP) designed to tolerate latency by statically splitting sequential programs into multiple non-speculative threads. We motivate DSWP by demonstrating the technique on recursive data structure traversal codes, which are notoriously difficult to optimize in other ways. Threaded execution of decoupled software pipelined codes ensures that many independent instructions are available for execution in the event of a traversal stall without requiring extremely large instruction windows. Increased parallelism is achieved by leveraging split-window issue in the multithreaded core and by using the *synchronization array*, a low-latency communication structure between the cores.

The remainder of this paper is organized as follows. Section 2 examines the pattern of recursive data structure traversals and illustrates why current static and dynamic scheduling techniques are not effective in identifying parallelism. Section 3 describes how DSWP parallelizes single-threaded pointer-chasing loops in the context of an SMT or CMP core using the synchronization array. That section further describes how the synchronization array can be used to achieve compiler-controlled low-latency thread-to-thread communication. Section 4 presents simulation results from running the resulting programs on both in-order and out-of-order machines. Section 5 discusses related work. Section 6 summarizes the contributions of this paper.

2. RDS Loops and Latency Tolerance

As was mentioned earlier, recursive data structure loops suffer from poor cache locality, requiring aggressive strate-

```
while(ptr = ptr->next) {
  ptr->val = ptr->val + 1;
}
```

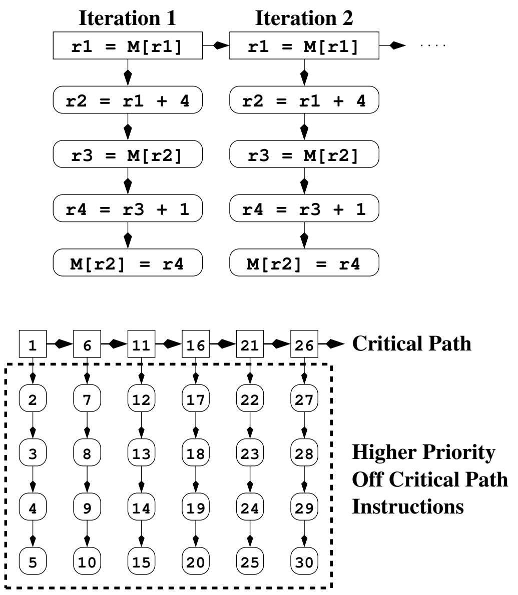

Figure 1. RDS loop traversal illustrating mis-prioritized instruction execution

gies to tolerate latency. Figure 1 illustrates a typical RDS loop. Each iteration of the loop processes a node in the data structure and contains code to fetch the next node to be processed. The diagram below the program code illustrates the data dependences that exist among the instructions in the program. As is evident from the figure, the critical path in this loop's execution is the chain of load instructions resulting from the loop-carried dependence of `r1`. Ideally, all the loop's computation could be overlapped with the computation along this critical path.

Unfortunately, modern single-threaded processors, both in-order and out-of-order, implement a sequential fetch and dispatch policy. Thus, rather than attempting to execute along the loop's critical path, the processor fetches sequential instructions, delaying the execution of the next critical path instruction. In the figure, this in-order fetch policy results in the twenty-four instructions inside the dashed box to be fetched and dispatched to allow the fetch and dispatch of six instructions on the critical path. This misappropriation of resources to instructions off the critical path is not restricted to fetch. Dynamic schedulers typically give priority to the oldest instruction in the dynamic scheduling window. Thus, even during execution in an out-of-order processor, priority is given to relatively unimportant instructions rather than those on the critical path.

This misprioritization of instructions can result in lost opportunities for overlapping latency. Consider, for example, the choice between executing instruction 6 or instruction 2 following the completion of instruction 1. If instruction 2 is selected in favor of instruction 6, and instruction 6, when executed, would miss in the cache, then executing instruction 2 first will not only delay this miss by one cycle but will also consume an instruction (i.e., itself) whose execution could have been overlapped with the miss. Even if instruction 6 were to hit in the cache, executing down the 2-3-4-5 path of instructions would delay the execution of instruction 6, which ultimately will delay a cache miss on one of the critical path loads. Each instruction executed preferentially over a ready critical-path instruction is an instruction that delays the occurrence of a future critical-path cache miss *and* one that removes an opportunity for latency tolerance by decreasing the pool of independent instructions.

Note that the delays incurred by executing off-critical-path instructions need not be short. For example, as the length of a loop increases, the number of cycles required to fetch a subsequent copy of the loop-carried load may become significant. Alternatively, cache misses off the critical path could cause chains of dependent instructions to collect in the machine's issue window thus forcing the system to stall until the load returns from the cache. For example, if, in Figure 1, instructions 3 and 8 miss in the cache, a 9 instruction reorder buffer (ROB) would be necessary to start the next critical path load. A smaller ROB would result in the subsequent critical path load being delayed until the cache misses resolve.

Avoiding such delays is possible in aggressive out-of-order processors but would require the addition of significant resources. In the tiny example loop shown in Figure 1, promptly bringing a critical-path load into the instruction window would cost five instructions of issue/dispatch width and five entries in the ROB. While possibly not impractical for loops of size five, as loop lengths increase, the cost of tracking *all* instructions between consecutive critical-path instructions will become excessive.

The same effect can be achieved without requiring excessive resources in the pipeline. In a multi-threaded architecture, it is possible to partition the loop into multiple threads such that one thread is constantly fetching and executing instructions from the critical path, thus maximizing the potential for latency tolerance. The next section proposes a technique to create these threads from a sequential loop and describes the hardware necessary to support it.

3. RDS Parallelization

As illustrated in the previous section, RDS loops typically consist of two relatively independent chains of instructions. The first, which makes up the critical path of execution, is the traversal code. The second is the chain of computations performed on each node returned by the traversal. While the data dependence of these sequences is often unidirectional (i.e., the computation chain is dependent on the traversal chain, but not vice-versa), variable latency instructions coupled with processor resource limitations create artificial dependencies between the two chains. While the resulting stalls are more pronounced in in-order processors, they can cause prolonged stalls even on aggressive out-of-order machines.

```
while(ptr = ptr->next) {
  ptr->val = ptr->val + 1;
}
```

(a) Recursive Data Structure Loop

```
while(ptr = ptr->next) {
  produce(ptr);
}
```

(b) Traversal Loop

```
while(ptr = consume()) {
  ptr->val = ptr->val + 1;
}
```

(c) Computation Loop

Figure 2. Splitting RDS Loops

To overcome this, it is necessary for the processor architecture to allow for a more decoupled execution of the two RDS loop pieces. Ideally, variable latency instructions from one piece should not impact code from the other piece unless the instructions from the two pieces are in fact dependent. To achieve this decoupling, we propose decoupled software pipelining (DSWP) which statically splits the original RDS loop into two distinct threads and executes the threads on a thread-parallel processor such as an SMT [20] core or chip-multiprocessor (CMP) [7] system. This will allow the traversal code to run unhindered by computation code. Since the traversal code defines the longest data dependence chain in the loop, executing instructions from this piece as quickly as possible is critical to obtaining maximum performance in RDS traversal loops. The rest of this section presents DSWP and the hardware necessary for its efficient execution.

3.1. Parallelism in RDS Loops

Consider the loop shown in Figure 1, reproduced in Figure 2a. The traversal slice consists of the critical path code, `ptr=ptr->next` and the computation slice consists of `ptr->val=ptr->val+1`. A natural way to thread this loop into a traversal and computation thread is shown in Figure 2. In the figure, the `produce` function enqueues the pointer onto a queue and the `consume` function dequeues the pointer. If the queue is full, the `produce` function will block waiting for a slot in the queue. The `consume` function will block waiting for data, if the queue is empty. In this way, the traversal and computation threads behave as a traditional decoupled producer-consumer pair.

The above parallelization lets the traversal thread make forward progress even in the event of stalls in the computation thread, and thus the traversal thread will have an opportunity to buffer data for the computation thread's consumption. The buffered data also allows the computation thread to be relatively independent of the stalls in the traversal thread since its dependence on the traversal thread is only through the `consume` function, and its dependences will be satisfied by one of the buffered values. Thus, this parallelization effectively decouples the behavior of the two code slices and allows useful code to be overlapped with long variable-latency instructions without resorting to speculation or extremely large instruction windows.

In addition to tolerating latency in the computation slice, the proposed threading also allows the traversal thread to execute far ahead of the corresponding single-threaded program due to the reduced size of the traversal loop compared to the original loop. On machines with finite issue width, this reduced size translates into more rapid initiation of the RDS traversing loads provided that the previous dynamic instance of the loads have completed. Thus, the reduced size loop allows the program to take better advantage of traversal cache hits to initiate traversal cache misses early. From an ILP standpoint, this allows for an overlap between traversal and computation instructions from distant iterations.

We call this threading technique decoupled software pipelining to highlight the parallel execution of the threads and the decoupled flow of data from the traversal to the work "stages" through a structure to be described next.

3.2. The Synchronization Array

While decoupled software pipelining as presented above appears promising, such partitioning of loops will usually lead to performance degradation on current hardware. Early experiments with decoupled software pipelining on a Pentium 4 Xeon processor with HyperThreading (Intel's SMT implementation) showed that the overhead in communication and synchronization negates any benefits from the threading, and in fact usually causes performance degradation for several reasons. First, synchronization must occur via the operating system (OS) or via spin-locks. In the OS case, the trap and return time is so large that it negates any performance advantage. Worse still, the communication between the threads must occur via shared memory, increasing the demand for a limited number of memory ports and polluting the cache. The demand on the memory subsystem is even greater when spin-locks are used. Also, communication between threads must occur in the commit stage to avoid expensive hardware for speculative memory writes. This increases the latency between data production and availability, creating extra unnecessary stalls in the computation thread when the queue is empty. Thus, for DSWP to be successful, it is imperative that the implementation of the queue and the queue access functions be efficient.

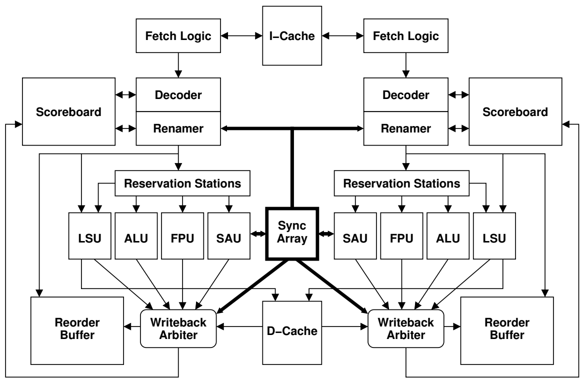

Figure 3. Implementation: Hardware model

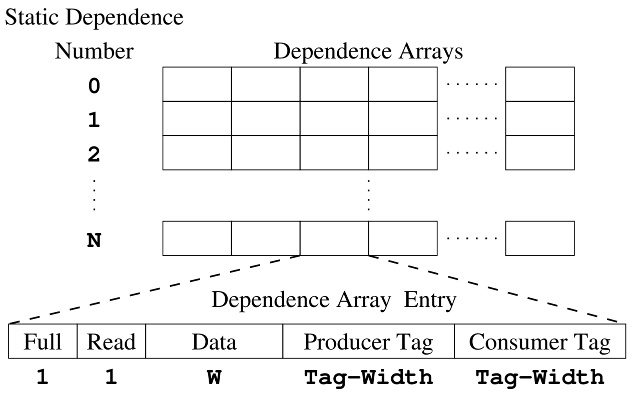

Figure 4. Synchronization Array structure

Since much of the problem related to the implementation of the `produce` and `consume` functions is due to the use of shared memory, we propose the *synchronization array*, a non-memory-based structure for communication between various hardware threads of execution on CMP or SMT machines. The ISA abstraction of the synchronization array is a set of blocking queues accessed via `produce` and `consume` instructions. The `produce` instruction takes an immediate dependence number and a register as operands. The value in the register is enqueued in the virtual queue

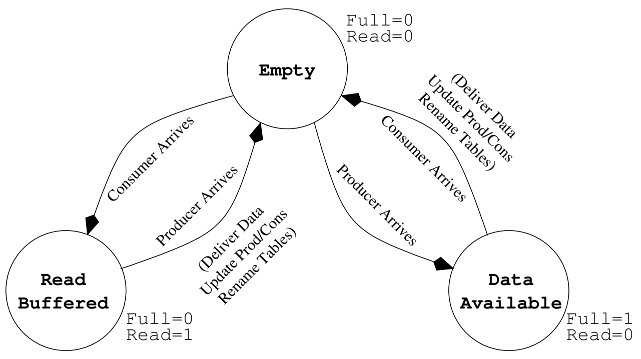

Figure 5. Synchronization Array State Machine

identified by the dependence number. The `consume` instruction dequeues data in a similar fashion. While the ISA model of the synchronization array (SA) is a queue, the hardware itself supports out-of-order and speculative execution of these instructions. Figure 3 illustrates an out-of-order CMP system with the SA coupling the two cores. The SA and its associated datapaths are highlighted in dark. Figure 4 shows the structure of the SA in more detail.

Synchronization between `produce` and `consume` instructions involves two main considerations - which locations to access and when to access these locations. The SA rename logic, which is distributed across the two cores, renames dependent `produce` and `consume` instructions to the correct (same) dependence array entries. This logic also reclaims dependence array slots after a successful synchronization or on a misspeculation. This is very similar to the rename logic in modern out-of-order processors. However, free-list maintenance is simplified in the SA rename logic, since there is only one unique `produce` and `consume` instruction per allocated slot. In the remainder of this section, the terms renaming and rename logic refer to the SA rename logic.

The compiler ensures that dynamically there is a one-to-one correspondence between `produce` and `consume` instructions across the two threads. This guarantee combined with the fact that the `produce` and `consume` instructions flow through the rename stage in program order in their respective cores, ensures that both instructions of every `produce`-`consume` dependence-pair are renamed to the same unique dependence array slot.

The state machine in each dependence array entry, shown in Figure 5, ensures that the `consume` instruction reads the data value only *after* the corresponding `produce` instruction is done producing. When the `produce` instruction executes, it writes the data into the data field, writes the producer tag field with its own tag and updates other architectural state as shown in Figure 5. When a `consume` is ready to execute, the SA checks to see if data is available for read (i.e., checks to see if the `Full` bit is set). If it is, it immediately returns the data and frees the entry. If not, the SA remembers the tag of the `consume` instruction in the consumer tag field. When the corresponding `produce` instruction arrives, the SA delivers the data to the writeback stage of the processor along with the consumer tag. This simple protocol ensures correct synchronization between the two instructions.

The SA notifies each core with the appropriate tag (`produce` or `consume`) when an array slot reverts to the `Empty` state. This allows the renamer to reclaim dependence array slots for future allocations. In this way, the synchronization array and rename logic preserve the illusion of a queue. While the decentralized hardware avoids lookups into the SA, in the absence of global broadcasts of issue and completion activity from all other cores, the rename logic of a core can only deliver accurate information about slots allocated to and accessed by dependence numbers for which this core is the exclusive producer or consumer. As a result, for a given dependence number, there must be a unique core that produces data and a unique core that consumes it. The designated producer-consumer pair of cores for a particular dependence number is allowed to change only after all issued `produce` and `consume` instructions corresponding to that dependence number have retired. In order to support multiple dependencies between the two threads, the SA is organized as an array of dependence arrays as shown in Figure 4. This provides the compiler with a lot of scheduling flexibility with respect to `produce` and `consume` instructions in each thread. For example, by allocating a unique dependence number to every true inter-thread dependence, `produce` and `consume` instructions in a control block can be moved to any location within that block. The compiler can also constrain SA instruction movement by assigning them the same dependence number.

Note that, while we have presented the SA for two cores, the technique is easily extended to support multiple cores. Each core can use any static dependence number to communicate with any other core, provided that the above restrictions are met. Furthermore, in an SMT/CMP system, the fetch and rename hardware is completely partitioned, so the above scheme fits in naturally.

The synchronization array has implications on the operating system. The SA namespace is used for communication between the threads of a process and must therefore become part of the process-context. Accordingly, the contents of the SA (including the bookkeeping information) must be saved and restored on context switches.

3.3. Decoupled Software Pipelining

Thus far, it was assumed that the hardware would be provided with properly parallelized code. This parallelization is a tedious process and should be performed in the compiler. This section presents a decoupled software pipelining algorithm suitable for inclusion in a compiler.

Although RDS loop parallelization would be typically performed at the assembly code level, we illustrate this process here in `C` for clarity. Consider the sample code of Figure 2a. As was mentioned previously, this loop structure is typical of recursive data structure access loops. Line 1 fetches the next data item for the computation to work on, and line 2 performs the computation in the loop. In order to parallelize the loop, we must first identify which pieces of the code are responsible for the traversal of the recursive data structure. Since a data structure is recursive if elements in the data structure point to other instances of the data structure, either directly or indirectly, RDS loops can be identified by searching for this pattern in the code. Specifically, we can search for load instructions that are data dependent on previous instances of the same instruction. These induction pointer loads (IPL) form the kernel of the traversal slice [13]. IPLs can be identified using augmented techniques for identifying induction variables [6, 13]. In the example in Figure 2a, the assignment within the *while* condition expression on line 1 is the IPL.

Once the IPL is identified, the backward slice of the IPL forms the base of the traversal thread. In Figure 2a, the backward slice of the IPL consists of the loop (i.e., the backward branch), the IPL, and the initialization of `ptr`. To complete the thread, a `produce` instruction is inserted to communicate the value loaded by the IPL and, if initialization code exists, an instruction to communicate this initial value. Figure 2b illustrates the traversal thread code that would arise from applying this technique to the loop shown in Figure 2a.

The computation thread is essentially the inverse of the traversal thread. Both loops share identical structure. Those instructions that are not part of the backward slice of the IPL but within the loop being split are inserted into the computation thread. In place of the IPL, a `consume` instruction is inserted. Just as in the traversal thread, it is necessary to include `consume` instructions to account for loop initialization. Figure 2c shows the code computation thread corresponding to the original loop shown in Figure 2a. For each data dependent pair of instructions split across the two threads, a dependence identifier is assigned to the corresponding producer-consumer pair.

In order for this algorithm to be effective, the compiler must be able to identify the traversal slice, including dependent memory operations. Existing memory dependence analysis techniques identify if there are stores in the loop that will affect later traversals (i.e., loops that modify the RDS). DSWP must handle these loops appropriately or exclude them from optimization. In cases where memory analysis is too conservative, not enough instructions will be placed in the computation thread and the benefits of the technique will be negated. Further, the compiler must balance the work of the original loop between the traversal and computation thread to realize optimal performance.

While memory dependence analysis is important for decoupled software pipelining, existing analysis used to compute load-store dependences is sufficient. DSWP does not attempt to identify completely independent code to split into threads. The analysis required for a general parallelization technique is much more sophisticated. Indeed, that problem has been heavily studied and has proved extremely difficult to solve. Instead, DSWP builds two threads that operate in a pipelined fashion, where the traversal thread feeds information to the computation thread. Though we focus only on RDS loops in this paper, DSWP can be extended to other variable latency producer-consumer scenarios as well, such as splitting loops to tolerate loop-carried floating-point operations. The concepts remain largely unchanged, except that loop slicing begins at different, appropriately chosen points.

4. Evaluating RDS Loop Parallelization

To evaluate DSWP applied to RDS codes, we simulate several benchmarks using an IA-64 processor model. The model was built using the Liberty Simulation Environment [22], using the processor configuration parameters shown in Table 1. In this section, single-threaded (ST), multithreaded (MT), in-order (INO), and out-of-order (OOO) processors are examined. The MT processors were implemented with two copies of the ST core connected to shared caches and to the synchronization array as depicted in Figure 3. To handle out-of-order processing in the presence of predication, a technique similar to that in [24] was used.

The benchmarks that were selected for this study were chosen because they contain recursive data structures. Since DSWP was manually applied, optimizations were performed on specific functions from a variety of benchmark suites (SPEC2000, Olden, and the Wisconsin Pointer-Intensive Benchmark Suite) and were selected to explore various environments and possibilities for threading. Table 2 lists the specific benchmarks used, the functions to which DSWP was applied, and the basic structure of the loops that were optimized. All single-threaded codes were aggressively optimized using the Intel C/C++ compiler for Itanium version 6.0 (Electron). The multi-threaded codes with DSWP were also compiled using Electron version 6.0. In cases were pathological situations arose in Electron's optimizer, we manually fixed the gener-

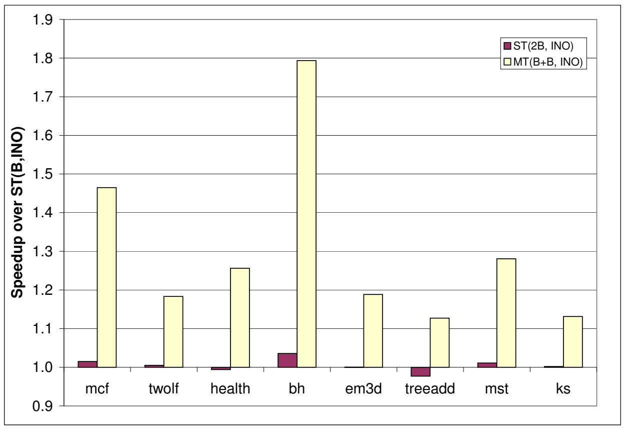

Figure 6. Function speedups of in-order issue processors vs. ST(B,INO)

```
 1  while( node != root ) {
 2    while( node ) {
 3      if( node->orientation == UP )
 4        node->potential = node->basic_arc->cost +
 5                          node->pred->potential;
 6      else {
 7        node->potential = node->pred->potential -
 8                          node->basic_arc->cost;
 9        checksum++;
10      }
11      tmp = node;
12      node = node->child;
13    }
14
15    node = tmp;
16    while( node->pred ) {
17      tmp = node->sibling;
18      if( tmp ) {
19        node = tmp;
20        break;
21      } else
22        node = node->pred;
23    }
24  }
```

Figure 7. Traversal loop in `181.mcf`

ated codes to ensure that the results were unbiased. On balance, this led to more improvements in the single-threaded codes than in the multi-threaded codes.

4.1. Threading on In-Order Processors

The preliminary study was conducted on a baseline in-order processor (ST(B,INO)) and compared to an in-order processor with twice the issue width (ST(2B,INO)) and a chip multiprocessor with two cores (MT(B+B,INO)), each identical to the baseline core with shared caches. To ensure fairness, the single-threaded code was occasionally hand-modified to overlap pointer chasing loads with the loop body. From Table 1, observe that the total number of functional units available in MT(B+B,INO) is equal to that in ST(2B,INO), yet is split between the two cores. Further, note that the number of cache ports present in MT(B+B,INO) and ST(2B,INO) are equal, but the ports are split between the two cores in MT(B+B,INO). Figure 6 shows the function speedups normalized to ST(B,INO).

As the graph indicates, very little additional ILP over the baseline machine is extracted by increasing the width of the in-order processor. Most speedups are negligible with a maximum speedup of 3% obtained in `bh`. However, despite the static distribution of resources between the two cores, the MT processor facilitates speedups of 12% to 80%.

These results are obtained by achieving an out-of-order scheduling effect on an in-order processor. For example, consider the traversal loop for `181.mcf` shown in Figure 7. The calculation to identify the next node to visit in the tree often involves traversing not just a child pointer (line 12) in the innermost computation loop, but also sibling and predecessor pointers (lines 17 and 22 respectively) in a loop following the computation loop. Each iteration of the innermost computation loop performs two pairs of dependent loads (lines 4-5 and 7-8). Cache misses in these instructions followed by subsequent uses force the in-order processor to delay the execution of the sibling pointer traversal in the loop following the computation loop. Subsequent misses in the sibling traversal are taken *after* the load-use stalls, rather than being overlapped with the stalls. Static scheduling techniques fail to allow the sibling traversal loads to be overlapped with the computation loads because the loads exist in two independent loops.

In the multithreaded implementation, the child *and* predecessor/sibling traversal loads are moved into the traversal thread. Stalls incurred in the computation loop do *not* affect the traversal thread, allowing the misses incurred in both loops to be overlapped. This is similar to what would occur in an out-of-order processor. Independent instructions that occur logically after the stalled instruction can be executed while stalled instructions wait for their source operands.

4.2. Threading on Out-of-Order Processors

Since DSWP achieves out-of-order effects on in-order processors, it is only logical to compare this technique with out-of-order execution. Figure 8 compares the performance of a single-threaded out-of-order core to both the in-order multithreaded configuration and an out-of-order multithreaded configuration.

From the figure we observe that the parallelized programs running on in-order cores perform on par with or outperform their single-threaded counterparts running on out-of-order cores. In many cases, adding out-of-order execution to the multithreaded core yields further speedup, thus

Model name	Characteristics	
ST(B, INO)	Number of Threads	1
	Instruction cache	32 KByte, 4-way set associative, 64-byte lines
	Data cache	32 KByte, 4-way set associative, 64-byte lines, 2-cycle load-use penalty
	Unified L2 cache	512 KByte, 4-way set associative, 128-byte lines, 8-cycle latency
	Cache ports	2
	Main memory	75 cycle latency
	Pipeline	In-order issue, out-of-order completion, 6 stages
	Functional Units	6 integer, 3 floating-point (4-cycle latency), 2 load-store
	Issue Width	6
	Branch Predictor	36Kbit PAp predictor
ST(2B, INO)	Functional Units	12 integer, 6 floating-point, 4 load-store
	Cache ports	4
	Issue Width	12
	Other characteristics	same as ST(B, INO)
ST(2B, OOO, 64ROB)	Pipeline	Out-of-order issue and completion, 6 stages
	ISW Size	64-instruction instruction window
	ROB Size	64-instruction reorder buffer
	Other characteristics	same as ST(2B, INO)
ST(2B, OOO, 128ROB)	ISW Size	128-instruction instruction window
	ROB Size	128-instruction reorder buffer
	Other characteristics	same as ST(2B, 64ROB, OOO)
MT(B+B, INO)	Number of Threads	2
	Pipeline	2x(In-order issue, out-of-order completion, 6 stages)
	Functional Units	(6 integer, 3 floating-point, 2 load-store) per thread
	Cache ports	4 (shared between threads)
	Issue Width	6 per thread
	Branch Predictor	36Kbit PAp predictor per thread
	Sync. Array Rows	8
	Effective Queue Size	32
	Array Access Time	1 cycle
	Other characteristics	same as ST(B, INO)
MT(B+B, OOO, 32+32ROB)	Pipeline	Out-of-order issue and completion, 6 stages
	ISW Size	32-instruction instruction window/thread
	ROB Size	32-instruction reorder buffer/thread
	Other characteristics	same as MT(B+B, INO)
MT(B+B, OOO, 64+64ROB)	ISW Size	64-instruction instruction window/thread
	ROB Size	64-instruction reorder buffer/thread
	Other characteristics	same as MT(B+B, 32ROB, OOO)

Table 1. Simulation models

Benchmark Name	Benchmark Suite	Function Optimized	Loop Description
181.mcf	SPECInt2000	`refresh_potential`	Depth first tree traversal. Tree is maintained with parent pointers, first child pointers, and sibling pointers.
300.twolf	SPECInt2000	`new_dbox_a`	Nested linked list traversals
bh	Olden	`walksub`	Recursive depth-first tree traversal
em3d	Olden	`traverse_nodes`	Linked list traversal
health	Olden	`check_patients_waiting`	Linked list traversal
mst	Olden	`BlueRule`	Separate chaining hash lookup embedded in linked list traversal
treeadd	Olden	`TreeAdd`	Recursive depth-first tree traversal
ks	Pointer-Intensive Benchmarks	`FindMaxGpAndSwap` and `CAiBj`	Four nested loop linked list traversals

Table 2. Benchmarks used during experimentation

indicating that the two techniques complement each other.

Analysis of the execution traces indicates that this speedup is achieved because the multithreaded program is able to more rapidly execute successive pointer-chasing loads than out-of-order execution, but does not affect local stalls in either thread of execution. For example, in `181.mcf`, when the traversal slice transitions from the child traversal to the sibling traversal, a cache miss on the child traversal load (which will yield `NULL` indicating the end of the loop) will cause a stall on a subsequent predicate define instruction that consumes the load's value. Conversely, on the out-of-order processor, the dynamic scheduler will issue instructions around the stalled predicate define, speculate across the branch instruction that consumes the predicate, and begin executing pointer-chasing code from the sibling traversal. Similar effects were observed in the other benchmarks.

To gain further insight into how the threaded program outperforms the single-threaded variant, we measured and analyzed the amount of time between successive iterations of a pointer-chasing load in the single-threaded program and the corresponding instructions in the traversal (pro-

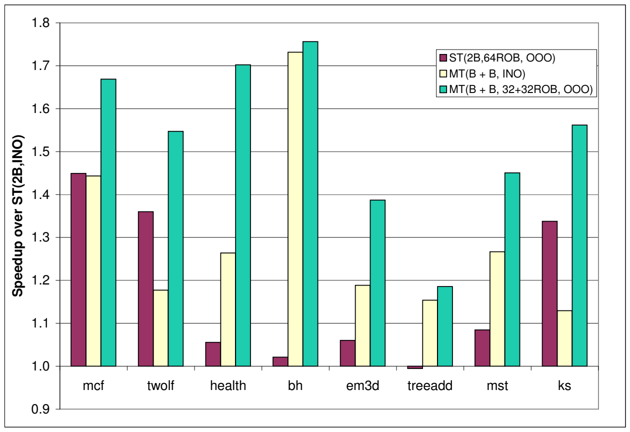

Figure 8. Speedups of out-of-order issue processors vs. ST(2B,INO)

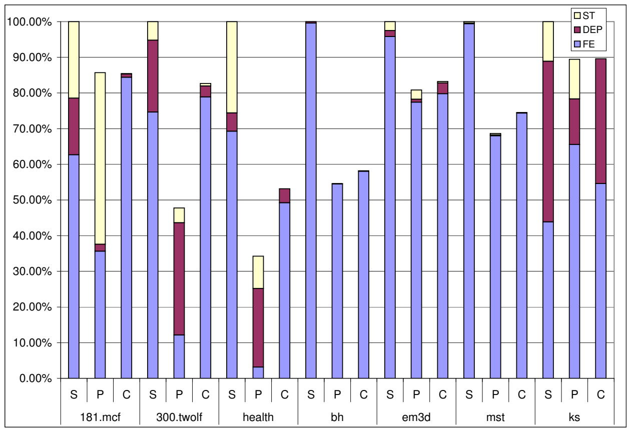

Figure 9. Distribution of iteration time. **S=single-threaded, P=producer, C=consumer.**

ducer) slice and computation (consumer) slice of the multithreaded program. Figure 9 shows the average time per iteration for each loop normalized to the average time per iteration of the single-threaded loop. The time is broken down into three categories: FE, DEP, and ST. FE (front-end stall or ROB filled up) is the time between the moment when the previous iteration's load was sent to the cache and the next iteration's load enters the instruction window. DEP (dependence stall) is the amount of time the load instruction waits for its dependences to be satisfied. Finally, ST (structural stalls) is the amount of time the load spends waiting to be issued after all dependences are satisfied. Note that the predominant reason for ST cycles is the presence of too many outstanding misses in the cache. It is for this reason that structural stalls are mostly absent from the consumer slice. It obtains data from the synchronization array rather than going to the data cache.

Notice from the figure that, for most benchmarks, the single-threaded loop iteration time was dominated by front-end cycles and dependence stalls. In comparison, the producer thread in the multithreaded version spent far less time in the front end of the machine despite the fact that its width and instruction window size are *half* that of the single-threaded core. This occurs for reasons described in Section 2. Since the traversal loop is smaller than the original loop and contains fewer instructions that can potentially "jam" the reorder buffer, the pointer-chasing load in the traversal thread (producer) is able to enter the instruction window much faster than in the single-threaded version. This data illustrates how prioritized execution of the critical path load instructions allows for more rapid completion of the loop's critical path by avoiding stalls due to limited resource availability.

In addition, this graph illustrates how initiating the

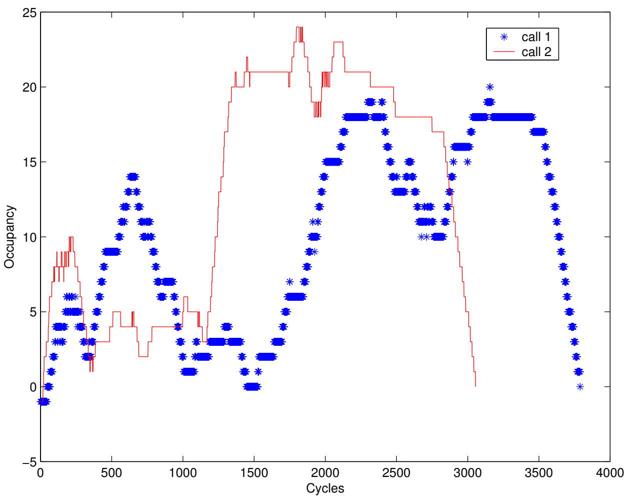

Figure 10. Synchronization Array occupancy in 300.twolf

pointer-chasing loads early affects overall performance. In most cases, the work thread (consumer) spends few cycles stalled on dependences. The data for `300.twolf` demonstrate this effect clearly. Due to the size of the producer loop, it is able to rapidly execute with the clear majority of its time stalled on the pointer-chasing load. The consumer thread, while spending a comparable amount of time in the front-end as the single-threaded version, spends significantly less time stalled on dependences, and thus is able to achieve faster iteration times.

To see this effect more clearly, Figure 10 illustrates the occupancy of the dependence array entry for the innermost threaded loop for two invocations of the `new_dbox_a` function in `300.twolf`. A ramp-up indicates a producer thread

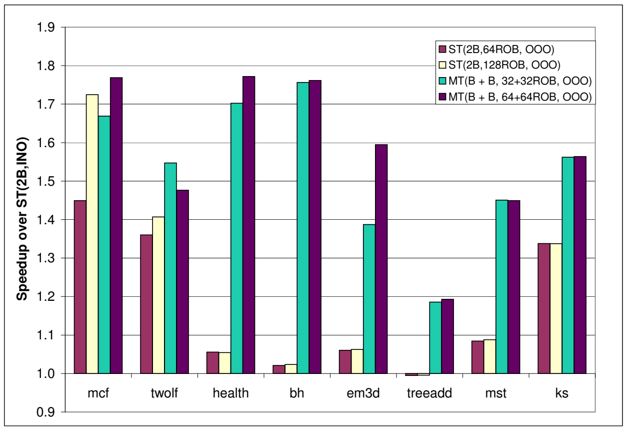

Figure 11. Comparisons with larger instruction windows

run-ahead while a ramp-down indicates a consumer thread catch-up, either due to a stalled producer or because the producer thread has completed all its loop iterations. Negative occupancy in the graph indicates that consumer instructions have issued prior to their corresponding produce instructions and are waiting for data. Data buffered in the synchronization array decouples the behavior of the traversal thread and the computation thread, allowing misses in each thread to be overlapped with work in the other thread. Note from the figure that there are periods of time when the occupancy of the synchronization array dips by approximately 10 entries. This drop indicates that the consumer was able to execute 10 more iterations than the producer. To avoid stalling in this situation, it is necessary that at least 10 instances of the pointer-chasing load have completed ahead of the computation code. Since in a single-threaded processor only one copy of the pointer-chasing load appears in the instruction window per iteration of the loop, 10 entire copies of the loop would have to appear in the instruction window to avoid a stall. This loop in `300.twolf` contains 28 instructions, which means that avoiding this stall requires an instruction window of at least 280 instructions.

Figure 11 demonstrates that the benefit of the DSWP technique and synchronization array is due to the decoupling of execution, and *not* simply the effective increase in instruction window size in the multiple core machine. Doubling the instruction window size does not dramatically improve the performance for most benchmarks, while DSWP does. The slightly lower speedup for 300.twolf on MT (B+B,64+64ROB,OOO) compared to MT (B+B,32+32ROB,OOO) is a result of an increased cross-thread misspeculation rate in the former (due to a larger instruction window and a small inner loop in the producer).

5. Related Work

Simultaneous Multi-threaded (SMT) processors [20] were designed to achieve higher overall throughput by executing instructions from multiple threads (possibly from different programs) in parallel and making better use of available functional units. Chip multiprocessors (CMP) [7] were proposed as a simple way of implementing multiple execution cores on a single chip.

Since the advent of these multithreaded machines, a significant amount of research [4, 5, 9, 23, 25] has gone into finding ways to exploit the additional hardware contexts available on these machines to improve single-program performance, especially for general-purpose integer benchmarks which are difficult to parallelize. The majority of these techniques focus on using the additional hardware context to prefetch [4, 5, 9, 23]. While the specific mechanisms vary, most of these techniques use the additional threads to execute loads that are likely to miss in the primary thread. One concern that the reader may have is that the technique presented in this paper achieves its benefit primarily from this effect since the traversal thread does issue load instructions that may spatially overlap with loads that the computation thread will perform later.

To detect whether this was in fact the dominant factor in the performance gains reported in Section 4, we re-threaded the original program to have a traversal thread similar to the ones used in the previous MT experiments and to have a computation thread nearly identical to the original program. The traversal threads were modified to remove the produce instructions, and the original code was modified to add triggers to initiate prefetching. Figure 12 shows the results from this experiment. The results indicate that this implementation of prefetching yields very little to no performance advantage in most cases, thus indicating that the speedup observed by the proposed technique is not primarily due to prefetching. As a further comparison, Figure 13 compares the performance of the proposed technique to perfect prefetching that was implemented by replacing the cache hierarchy with a single-cycle main memory. The figure shows that in many cases perfect prefetching combined with the proposed threading technique yields performance greater than just perfect prefetching alone, suggesting possible synergy between the two. Note that in all the experiments, in the code with DSWP applied, the effective architectural latency between a loop-carried load in the producer and its ultimate consumer (an instruction that consumes the `consume` instruction's output) is 3 cycles (2 cycles for load plus 1 cycle for the `consume` instruction). However, in the single-threaded code the architectural load-use latency is only 2 cycles. Despite this, DSWP performs better thanks to the scheduling benefits that stem from efficient overlap of traversal and computation instructions. In addi-

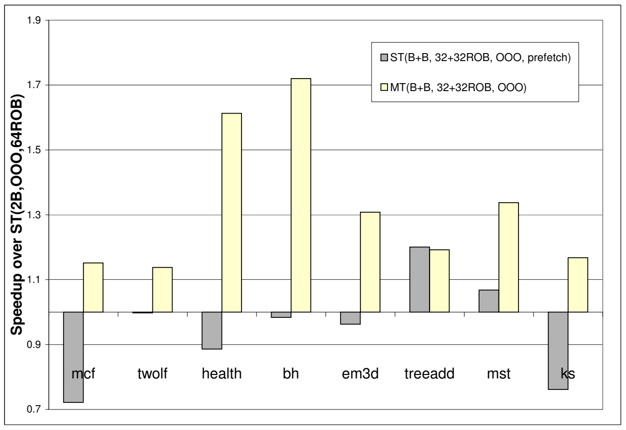

Figure 12. Comparison to subordinate thread based prefetching

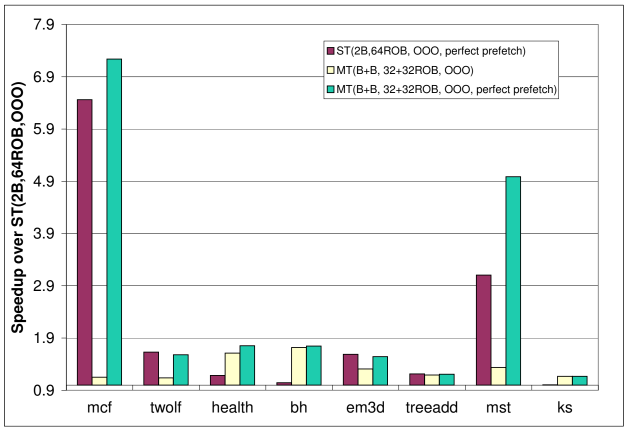

Figure 13. Comparison to perfect prefetching

tion to the prefetching techniques described above, other software, hardware, and hybrid prefetching techniques have been proposed [10, 12, 14, 15]. While DSWP is orthogonal to prefetching, prefetching can complement DSWP for RDS by reducing the effective memory access latency on the producer side, enabling the producer to run ahead faster.

In addition to prefetching, other uses of dual-core architectures have been presented. Barnes et al. [1] propose a dual-core architecture with decoupling queues between cores. They address the problem of long-latency load-value-use stalls blocking in-order instruction issue by propagating phantom values for such instructions and re-executing such instructions and their dependence chain selectively on the second core. While this technique effectively hides the effects of variable latency instructions, it does not handle long chains of such instructions common in RDS loops.

Techniques to address long latency instructions in out-of-order processors have also been proposed. Brekelbaum et al. [2], in order to reduce the complexity of dynamic schedulers, move long-latency instructions to a larger and slower instruction window and maintain less latency-tolerant instructions in a small fast window. This reduction in complexity allows for the creation of larger instruction windows but comes at the cost of performance.

Thread-level speculation (TLS) techniques [3, 18] speculatively extract and execute threads from sequential programs on parallel hardware and commit threads in program order. Other techniques have been proposed to improve fine-grain value communication between such threads [19].

Moshovos et al. propose a technique [11] that identifies producer-consumer patterns among communicating stores and loads at run-time and assigns unique dependence numbers to each store-load pair. They do this to avoid stalls due to memory aliasing. While the communication is similar, the technique proposed in this paper identifies the communication channels statically and uses them to tolerate variability in instruction latency.

Tullsen et al. proposed synchronization functional units to implement fine-grained blocking synchronization among threads executing on a SMT machine [21]. The special functional units acquire and release memory-based locks to bring about the synchronization

Finally, the synchronization array is similar to hardware queues proposed by Smith in decoupled access/execute microarchitectures [17]. Roth et al. [16] highlight the usefulness and relevance of such decoupling in modern-day architectures and modify the architecture described in Smith's work so that it works as a speculative microarchitectural technique. While the functionality of the synchronization array proposed in our paper is very similar to hardware queues, it is important to note that the synchronization array implementation provides for out-of-order accesses and software-controlled directionality.

6. Conclusion

Loops that traverse and manipulate recursive data structures (RDSs) commonly contain instructions with long and variable latency on the critical computation path. Techniques to hide latency, such as out-of-order execution, are often unable to find enough useful work to hide the effect of these long-latency instructions. Current compiler-directed techniques, such as static scheduling, are not well-suited to deal with variable latencies. Prefetching techniques may also prove ineffective if they are unable to fetch the proper addresses well in advance and at the appropriate rate.

This paper presents decoupled software pipelining (DSWP) to avoid stalls resulting from long variable-latency instructions in RDS loops. DSWP threads these loops for parallel execution on SMT or CMP processors. However, for this threading technique to be effective, a very low overhead communication and synchronization mechanism between the threads is required. To accomplish this, the paper presents the *synchronization array* which appears to the ISA as a set of queues capable of supporting both out-of-order execution and speculative issue.

Careful analysis reveals that DSWP exposes parallelism and hides latency in a manner unlike other techniques. Even in the presence of perfect prefetching, this technique improves performance by exploiting far ILP. The large amount of useful work exposed routinely masks the adverse effects of multiple sequential load misses to main memory. Overall, the technique gives an 11%-76% performance improvement for optimized functions.

Acknowledgments

We thank John Sias, David Penry, and the entire Liberty Research Group for their support and feedback during this work. We would also like to thank the anonymous reviewers for their insightful comments. This work has been graciously supported by Intel Corporation.

References

[1] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. W. Hwu. Beating in-order stalls with 'flea-flicker' two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture*, December 2003.

[2] E. Brekelbaum, J. R. II, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 27–36, November 2002.

[3] M. Cintra, J. F. Martinez, and J. Torellas. Architectural support for scalable speculative parallelization on shared-memory multiprocessors. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 13–24, June 2000.

[4] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 306–317, December 2001.

[5] 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. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[6] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[7] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, September 1997.

[8] J. L. Hennesey and N. P. Jouppi. Computer technology and architecture: An evolving interaction. pages 18–29, September 1991.

[9] C.-K. Luk. Tolerating memory latnecy through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[10] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 222–233. ACM Press, 1996.

[11] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 235–245, 1997.

[12] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Arechitecture*, February 2003.

[13] E. M. Nystrom, R. D. Ju, and W. W. Hwu. Characterization of repeating data access patterns in integer benchmarks. In *Proceedings of the 28th International Symposium on Computer Architecture*, September 2001.

[14] A. Roth, A. Moshovos, and G. S. Sohi. Dependence-based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[15] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[16] A. Roth, C. B. Zilles, and G. S. Sohi. Microarchitectural miss/execute decoupling. In *Proceedings of MEDEA Workship*, October 2000.

[17] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 112–119, April 1982.

[18] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, June 2000.

[19] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 65–80, February 2002.

[20] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[21] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 54–58, January 1999.

[22] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.

[23] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen. Memory latency-tolerance approaches for Itanium processors: Out-of-order execution vs speculative precomputation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 187–196, February 2002.

[24] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 15–25, January 2001.

[25] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.