Speculative Decoupled Software Pipelining

Neil Vachharajani Ram Rangan[†] Easwaran Raman
Matthew J. Bridges Guilherme Ottoni David I. August

Department of Computer Science
Princeton University
Princeton, NJ 08540
{nvachhar,ram,eraman,mbridges,ottoni,august}@princeton.edu

Abstract

In recent years, microprocessor manufacturers have shifted their focus from single-core to multi-core processors. To avoid burdening programmers with the responsibility of parallelizing their applications, some researchers have advocated automatic thread extraction. A recently proposed technique, Decoupled Software Pipelining (DSWP), has demonstrated promise by partitioning loops into long-running, fine-grained threads organized into a pipeline. Using a pipeline organization and execution decoupled by inter-core communication queues, DSWP offers increased execution efficiency that is largely independent of inter-core communication latency.

This paper proposes adding speculation to DSWP and evaluates an automatic approach for its implementation. By speculating past infrequent dependences, the benefit of DSWP is increased by making it applicable to more loops, facilitating better balanced threads, and enabling parallelized loops to be run on more cores. Unlike prior speculative threading proposals, speculative DSWP focuses on breaking dependence recurrences. By speculatively breaking these recurrences, instructions that were formerly restricted to a single thread to ensure decoupling are now free to span multiple threads. Using an initial automatic compiler implementation and a validated processor model, this paper demonstrates significant gains using speculation for 4-core chip multiprocessor models running a variety of codes.

1 Introduction

For years, steadily increasing clock speeds and uniprocessor microarchitectural improvements reliably enhanced performance for a wide range of applications. Although this approach has recently faltered, the exponential growth in transistor count remains strong, leading microprocessor manufacturers to add value by producing chips that incorporate multiple processors. Multi-core processors can improve system throughput and speed up multithreaded applications, but single-threaded applications receive no benefits.

While the task of producing multithreaded code could be left to the programmer, there are several disadvantages to this approach. First, writing multithreaded codes is inherently more difficult than writing single-threaded codes. Multithreaded programming requires programmers to reason about concurrent accesses to shared data and to insert sufficient synchronization to ensure proper behavior while still permitting enough parallelism to improve performance. Active research in automatic tools to identify deadlock, livelock, and race conditions [3, 4, 5, 14, 21] in multithreaded programs is a testament to the difficulty of this task. Second, there are many legacy applications that are single-threaded. Even if the source code for these applications is available, it would take enormous programming effort to translate these programs into well-performing parallel versions.

A promising alternative approach for producing multithreaded codes is to let the compiler automatically convert single-threaded applications into multithreaded ones. This approach is attractive as it takes the burden of writing multithreaded code off the programmer. Additionally, it allows the compiler to automatically adjust the amount and type of parallelism extracted depending on the underlying architecture, just as instruction-level parallelism (ILP) optimizations relieved the programmer of the burden of targeting complex single-threaded architectures.

Unfortunately, compilers have been unable to extract thread-level parallelism (TLP) despite the pressing need. While success of this type has not been achieved yet, progress has been made. Techniques dedicated to parallelizing scientific and numerical applications, such as DOALL and DOACROSS, are used routinely in such domains with good results [10, 25]. Such techniques perform well on counted loops manipulating very regular and analyzable structures consisting mostly of predictable array accesses. Since these techniques were originally proposed for scien-

[†]Currently with the Performance and Tools Group, IBM Austin Research Laboratory, rrangan@us.ibm.com.

tific applications, they generally do not handle loops with arbitrary control flow or unpredictable data access patterns that are the norm for general-purpose applications.

Because dependences tend to be the limiting factor in extracting parallelism, speculative techniques, loosely classified as thread-level speculation (TLS), have dominated the literature [1, 2, 8, 9, 11, 12, 15, 22, 23, 26, 28]. Speculating dependences that prohibit DOALL or DOACROSS parallelization increases the amount of parallelism that can be extracted. Unfortunately, speculating enough dependences to create DOALL parallelization often leads to excessive misspeculation. Additionally, as will be discussed in Section 2, core-to-core communication latency combined with the communication pattern exhibited by DOACROSS parallelization often negates the parallelism benefits offered by both speculative and non-speculative DOACROSS parallelization.

Decoupled Software Pipelining (DSWP) [16, 20] approaches the problem differently. Rather than partitioning a loop by placing distinct iterations in different threads, DSWP partitions the loop body into a pipeline of threads, ensuring that critical path dependences are kept thread-local. The parallelization is tolerant of both variable latency within each thread and long communication latencies between threads. Since the existing DSWP algorithm is non-speculative, it must respect all dependences in the loop. Unfortunately, this means many loops cannot be parallelized with DSWP.

In this paper, we present *Speculative Decoupled Software Pipelining* (SpecDSWP) and an initial *automatic* compiler implementation of it. SpecDSWP leverages the latency-tolerant pipeline of threads characteristic of DSWP and combines it with the power of speculation to break dependence recurrences that inhibit DSWP parallelization. Like DSWP, SpecDSWP exploits the fine-grained *pipeline parallelism* hidden in many applications to extract long-running, concurrently executing threads, and can do so on more loops than DSWP. The speculative, decoupled threads produced by SpecDSWP increase execution efficiency and may also mitigate design complexity by reducing the need for low-latency inter-core communication. Additionally, since effectively extracting fine-grained pipeline parallelism often requires in-depth knowledge of many microarchitectural details, the compiler's automatic application of SpecDSWP frees the programmer from the difficult and even counter-productive involvement at this level.

Using an initial automatic compiler implementation and a validated processor model, this paper demonstrates that SpecDSWP provides significant performance gains for a multi-core processor running a variety of codes. Ottoni et al. demonstrated DSWP's initial promise by applying it to 9 key application loops [16]. Here, we extend their work by extracting significant speedup, 40% on average, from four additional key application loops using *only* control and silent store speculation.

In summary, the contributions of this paper are:

- A new technique, Speculative Decoupled Software Pipelining, that can extract parallel threads from previously unparallelizable loops.
- A compiler implementation of SpecDSWP.
- An evaluation of several general-purpose applications on a cycle-accurate hardware simulator.

The rest of the paper is organized as follows. Section 2 examines existing speculative and non-speculative parallelization techniques to put this work in context. Section 3 provides an overview of the original non-speculative DSWP technique, which is followed by a discussion of the speculative DSWP technique in Section 4. Sections 5 and 6 detail how dependences are chosen for speculation and how misspeculation is handled. Section 7 provides experimental results, and Section 8 concludes the paper.

2 Motivation

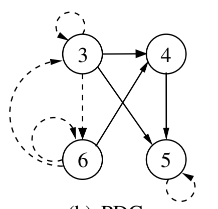
Three primary non-speculative loop parallelization techniques exist: DOALL [10], DOACROSS [10], and DSWP [16, 20]. Of these techniques, only DOALL parallelization yields speedup proportional to the number of cores available to run the code. Unfortunately, in general-purpose codes, DOALL is often inapplicable. For example, consider the code shown in Figure 1. In the figure, program dependence graph (PDG) edges that participate in *dependence recurrences* are shown as dashed lines. Since the statements on lines 3, 5, and 6 are each part of a dependence recurrence, consecutive loop iterations cannot be independently executed in separate threads. This makes DOALL inapplicable.

DOACROSS and DSWP, however, are both able to parallelize the loop. Figure 2(a) and 2(b) show the parallel execution schedules for DOACROSS and DSWP respectively. These figures use the same notation as Figure 1, except the nodes are numbered with both static instruction numbers and loop iteration numbers. After an initial pipeline fill time, both DOACROSS and DSWP complete one iteration every other cycle. This provides a speedup of 2 over single-threaded execution. DOACROSS and DSWP differ, however, in how this parallelism is achieved. DOACROSS alternately schedules entire loop iterations on processor cores. DSWP, on the other hand, partitions the loop code, and each core is responsible for a particular piece of the loop across all the iterations.

The organization used by DOACROSS forces it to communicate dependences that participate in recurrences (dashed lines in the figure) from core to core. This puts communication latency on the critical path. DSWP's organization allows it to keep these dependences thread-local (in fact the algorithm requires it) thus avoiding communica-

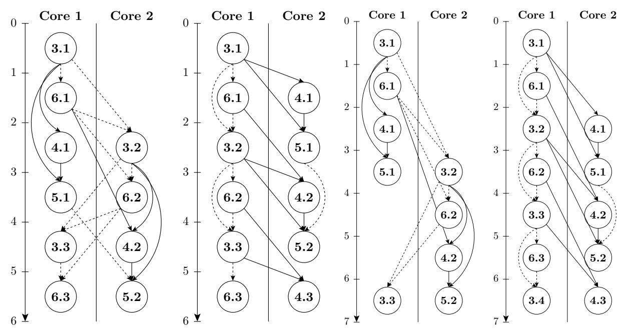
```
1  cost=0;
2  node=list->head;
3  while(node) {
4    ncost=doit(node);
5    cost += ncost;
6    node=node->next;
7  }
```

(a) Loop



(b) PDG

Figure 1. Parallelizable loop.



(a) DOACROSS (b) DSWP (c) DOACROSS (d) DSWP
Comm. latency = 1 cycle Comm. latency = 2 cycles

Figure 2. DOACROSS and DSWP schedules.

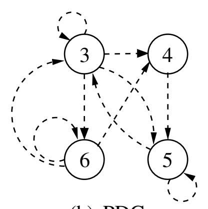
tion latency on the critical path. Figure 2(c) and 2(d) show the execution schedules if communication latency were increased by one cycle. Notice that DSWP still completes one iteration every two cycles. Only its pipe-fill time increased. DOACROSS, however, now only completes one iteration every three cycles. While not shown in the figure, by using decoupling queues, DSWP is also tolerant to variable latency within each thread. Often, this increases overall memory-level parallelism (MLP) by overlapping misses in different threads. These features make DSWP a promising approach to automatic thread extraction.

Unfortunately, both DOACROSS and DSWP are not able to parallelize all loops. For example, consider the loop shown in Figure 3, which is identical to the loop in Figure 1 except this loop can exit early if the computed cost exceeds a threshold. Since all the loop statements participate in a single dependence recurrence (they form a single strongly-connected component in the dependence graph), DSWP is unable to parallelize the loop. For similar reasons, a DOACROSS parallelization would obtain no speedup; no iteration can begin before its predecessor has finished.

In response to this, there have been many proposals for thread-level speculation (TLS) techniques which speculatively break various loop dependences [1, 2, 8, 9, 11, 12, 15, 22, 23, 26, 28]. Once these dependences are broken, DOACROSS and sometimes even DOALL parallelization is possible. In the example, if TLS speculatively breaks the loop exit control dependences (the dependences originating from statement 3), then the execution schedule shown in Figure 4(a) is possible. This parallelization offers a

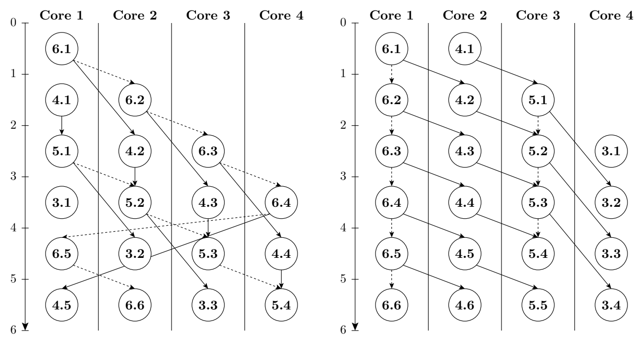
```
1  cost=0;
2  node=list->head;
3  while(cost<T && node) {
4    ncost=doit(node);
5    cost += ncost;
6    node=node->next;
7  }
```

(a) Loop



(b) PDG

Figure 3. Speculatively parallelizable loop.



(a) TLS (b) SpecDSWP

Figure 4. TLS and SpecDSWP schedules.

speedup of 4 over single threaded execution. Notice, however, that just as for non-speculative DOACROSS parallelization, TLS must communicate dependences that participate in recurrences. Just as before, this places communication latency on the critical path. For many codes, this fact coupled with the non-unit communication latency typical of multi-core architectures negates most, if not all, of the expected parallelism.

The Speculative DSWP[1] technique described in this paper can be considered the DSWP analogue of TLS. Just as adding speculation to DOALL and DOACROSS expanded their applicability, adding speculation to DSWP allows it to parallelize more loops. Figure 4(b) shows the execution schedule achieved by applying SpecDSWP to the loop in Figure 3. Similar to the TLS parallelization, SpecDSWP offers a speedup of 4 over single-threaded execution. However, SpecDSWP retains the pipelined organization of DSWP. Notice, in the execution schedule, that all dependence recurrences are kept thread-local. This assures, in the absence of misspeculation, SpecDSWP's performance is independent of communication latency and tolerant of intra-thread memory stalls. The remainder of this paper will review DSWP and will then describe Speculative DSWP in detail.

[1]Speculative Decoupled Software Pipelining and Speculative Pipelining [18] have similar names, but the two techniques are different. The latter is a prepass applied to code before applying TLS. This prepass could also be applied before DSWP or Speculative DSWP to improve performance.

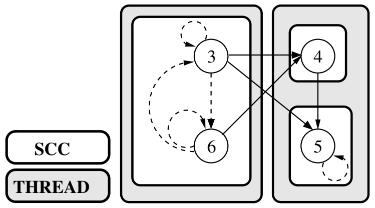


Figure 5. DSWP transformation example.

3 Decoupled Software Pipelining

This section describes the DSWP transformation, from which the Speculative DSWP algorithm is built. DSWP is a *non-speculative* pipelined multithreading transformation that parallelizes a loop by partitioning the loop body into stages of a pipeline. Like conventional software pipelining (SWP), each stage of the decoupled software pipeline operates on a different iteration of the loop, with earlier stages operating on later iterations. DSWP differs from conventional SWP in three principal ways. First, DSWP relies on TLP, rather than ILP, to run the stages of the pipeline in parallel. Each pipeline stage executes within a thread and communicates to neighboring stages via communication queues. Second, since each pipeline stage is run in a separate thread and has an independent flow of control, DSWP can parallelize loops with complex control flow. Third, since inter-thread communication is buffered by a queue, the pipeline stages are *decoupled* and insulated from stalls in other stages. Variability in the execution time of one stage does not affect surrounding stages provided sufficient data has been buffered in queues for later stages and sufficient space is available in queues fed by earlier stages [20].

The DSWP algorithm partitions operations into pipeline stages and has three main steps (see [16] for a detailed discussion). Figure 5 illustrates each of these steps on the loop from Figure 1. First, the program dependence graph (PDG) is constructed for the loop being parallelized. The PDG contains all register, memory, and control dependences present in the loop.[2] Second, all dependence recurrences are found in the PDG by identifying its strongly-connected components (SCCs). To ensure that there are no cyclic cross-thread dependences after partitioning, the SCCs will be the minimum scheduling units. Lastly, each SCC is allocated to a thread while ensuring that no cyclic dependences are formed between the threads. SCCs are partitioned among the desired number of threads according to a heuristic that tries to balance the execution time of each thread [16].

[2]Register anti- and output-dependences are ignored since each thread will have an independent set of registers.

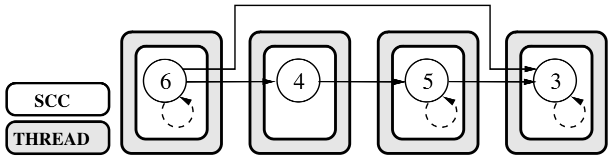


Figure 6. Speculative DSWP example.

4 Speculative DSWP

Despite the success of DSWP, the non-speculative transformation does not leverage the fact that many dependences are easily predictable or manifest themselves infrequently. If these dependences were *speculatively* ignored, large dependence recurrences (SCCs) may be split into smaller ones with more balanced performance characteristics. As the example from Section 2 showed, these smaller recurrences provide DSWP with more scheduling freedom, leading to greater applicability, scalability, and performance.

4.1 Execution Model

Before describing the compiler transformation used by speculative DSWP, this section outlines the basic execution paradigm and hardware support necessary. SpecDSWP transforms a loop into a pipeline of threads with each thread's loop body consisting of a portion of the original loop body. SpecDSWP speculates certain dependences to ensure no dependence flows between a later thread and an earlier thread. In the absence of misspeculation, SpecDSWP achieves decoupled, pipelined multithreaded execution like DSWP.

To manage misspeculation recovery, threads created by SpecDSWP conceptually checkpoint architectural state each time they initiate a loop iteration. When misspeculation is detected, each thread is resteered to its recovery code, and jointly the threads are responsible for restoring state to the values checkpointed at the beginning of the misspeculated iteration and re-executing the iteration non-speculatively. Our current SpecDSWP implementation uses software to detect misspeculation and recover register state. It, however, relies on hardware *versioned memory* (see Section 6.1) to rollback the effects of speculative stores.

4.2 Compiler Transformation Overview

To generate parallel code, a Speculative DSWP compiler should follow the steps below.

1. Build the PDG for the loop to be parallelized.
2. Select the dependence edges to speculate.
3. Remove the selected edges from the PDG.
4. Apply the DSWP transformation [16] using the PDG with speculated edges removed.
5. Insert code necessary to detect misspeculation.
6. Insert code to recover from misspeculation.

Recall the loop from Figure 3(a). Applying step 1 yields the PDG shown in Figure 3(b). In step 2, the compiler

```
1  node=list->head;
2  produce(node);
3  while(TRUE) {
4    node=node->next;
5    produce(node);
6  }
7
```

(a) Thread 1

```
1  node=consume();
2  while(TRUE) {
3     ncost=
4       doit(node);
5     produce(ncost);
6     node=consume();
7  }
```

(b) Thread 2

```
1  cost=0;
2  produce(cost);
3  while(TRUE) {
4     ncost=consume();
5     cost += ncost;
6     produce(cost);
7  }
```

(c) Thread 3

```
1  node=consume();
2  cost=consume();
3  while(cost<T && node) {
4     cost=consume();
5     node=consume();
6  }
7  FLAG_MISSPECULATION();
```

(d) Thread 4

Figure 7. The code from Figure 3(a) after SpecDSWP is applied.

would select the loop exit control dependences to be speculated. Step 3 would produce the PDG in Figure 6. After steps 4 and 5, the code in Figure 7 would be produced.

The next two sections will provide more details on the SpecDSWP transformation. Steps 2 and 5 are described in Section 5, and step 6 is described in Section 6.

5 Selecting Edges to Speculate

Since the scheduling freedom enjoyed by SpecDSWP, and consequently its potential for speedup, is determined by the number and performance (i.e., schedule height) of a loop's dependence recurrences, SpecDSWP should try to speculate dependences which break recurrences. However, since speculating one dependence alone may not break a recurrence, SpecDSWP ideally would simultaneously consider speculating all sets of dependences. Since exponentially many dependence sets exist, SpecDSWP uses a heuristic solution. First, SpecDSWP provisionally speculates all dependences which are highly predictable. Next, the partitioning heuristic allocates instructions to threads using a PDG with the provisionally speculated edges removed. Once the partitioning is complete, SpecDSWP identifies the set of provisionally speculated edges that are cross-thread *and* that originate from a thread later in the pipeline to an earlier thread. These dependences are the only edges that need to be speculated to ensure acyclic communication between the threads. Consequently, only these edges are speculated. All other provisionally speculated dependences are not speculated and are added back to the PDG before invoking the non-speculative DSWP transformation (step 4 from Section 4.2).

A SpecDSWP implementation can speculate any dependence that has an appropriate misspeculation detection mechanism and, for value speculation, that also has an appropriate value predictor. Each misspeculation detection mechanism and value predictor can be implemented either in software, hardware, or a hybrid of the two. Our current implementation relies solely on software for value prediction and misspeculation detection. The remainder of this section will detail the speculation carried out by our SpecDSWP compiler.

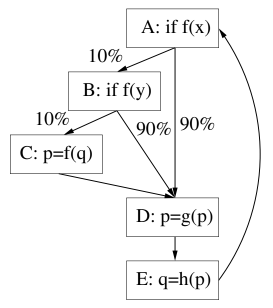
5.1 Biased Branches

As the example presented earlier (Figures 3, 6, and 7) showed, speculating biased branches can break dependence recurrences. Recall from the example that the loop terminating branch (statement 3) was biased provided that the loop ran for many iterations. Consequently, the compiler can speculatively break the control dependences between the branch and other instructions. Figures 3(b) and 6 show the dependence graph before and after speculation, respectively. This speculation is realized by inserting an unconditional branch in each thread that was dependent on the branch. The `while(TRUE)` statements in Figures 7(a), 7(b), and 7(c) are the unconditional branches. Misspeculation detection is achieved by making the taken (fall through) path of the speculated branch be code that flags misspeculation assuming the branch was predicted not taken (taken). In the example, the not-predicted path is the loop exit, so misspeculation is flagged there (line 7 of Figure 7(d)).

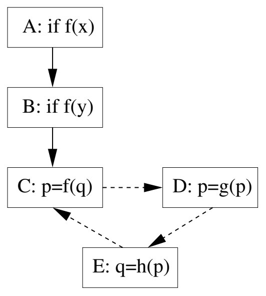
Our SpecDSWP compiler assumes that all branches that exit the loop being SpecDSWPed are biased and speculates the loop will not terminate. For other branches, if their bias exceeds a threshold, the branch is speculated. Finally, inner loop exit branches are handled specially. If all inner loop exits are speculatively removed, each time the inner loop is invoked it will eventually flag misspeculation. Since our misspeculation recovery model forces rollback to the beginning of a loop iteration for the loop being SpecDSWPed, this is tantamount to speculating the inner loop will not be invoked. Consequently, if the compiler does not speculate that an inner loop will never be invoked, it preserves some exit from the inner loop.

5.2 Infrequent Basic Block Speculation

While speculating biased branches is successful in the previous example, consider the code shown in Figure 8. Assume, in this example, that our branch bias threshold is 95%. This implies that neither operation A nor B will be speculated. Further, notice that, despite not being the target of a sufficiently biased branch, operation C only has a 1% chance of execution. Since its execution is very unlikely, SpecDSWP will speculatively break the dependences between operation C and D. Speculating these dependences



(a) CFG



(b) PDG

Figure 8. Infrequent basic block speculation.

breaks the dependence recurrence between operations C, D, and E. Note, in this example, even if branches A and B were sufficiently biased, breaking the dependences from A to B or from B to C would not have broken the dependence recurrence between C, D, and E.

In general, SpecDSWP will break all dependences originating from operations in infrequently executed basic blocks. Just as for biased branches, SpecDSWP uses a threshold parameter to determine if a basic block is infrequently executed. To detect misspeculation, operations in the speculated basic block are replaced with code that flags misspeculation.

5.3 Silent Stores

In addition to control speculation, our SpecDSWP compiler speculates memory flow dependences originating at frequently silent stores [13]. To enable this speculation, the compiler first profiles the application to identify silent stores. Then, the compiler transforms a silent store into a hammock that first loads the value at the address given by the store, compares this value to the value about to be stored, and only if the two differ perform the store. After this transformation, the biased branch speculation mechanism described above is applied. The compiler will predict that the store will never occur, and if it does occur, misspeculation will be flagged.

5.4 False Memory Dependences

Finally, our compiler disregards all loop-carried memory anti- and output-dependences. Since our recovery mechanism relies on versioned memory (see Section 6.1) to recover from misspeculation, each loop iteration can explicitly load from and store to a particular version of the memory. Consequently, loop-carried false memory dependences no longer need to be respected. Since the compiler is removing edges from the PDG, this is similar to speculation, but since these dependences truly do not need to be respected, there is no need to detect misspeculation or initiate recovery.

6 Misspeculation Recovery

Recall from Section 4.1 that SpecDSWP handles misspeculation at the iteration level. When thread j in a pipeline of T threads detects misspeculation, several actions must be taken to recover. In this discussion, assume that, when misspeculation is detected, thread j is executing iteration n_j. These are the actions that need to be performed:

1. The first step in recovery is waiting for all threads to complete iteration $n_j - 1$.
2. Second, speculative state must be discarded and non-speculative state must be restored. This includes reverting the effects of speculative stores to memory, speculative writes to registers, as well as speculative produces to cross-thread communication queues. Since values produced to communication queues in a particular iteration are consumed in the same iteration, it is safe to flush the communication queues (i.e., after all threads have reached iteration n_j, there are no non-speculative values in the queues).
3. Third, the misspeculated iteration must be re-executed. Since the speculative code is deterministic, re-executing it will again result in misspeculation. Consequently, a non-speculative version of the iteration must be executed.
4. Finally, speculative execution can recommence from iteration $n_j + 1$.

To orchestrate this recovery process, SpecDSWP relies on an additional *commit thread*, which receives state checkpoints and status messages from each of the worker threads. Pseudo-code for a worker thread and the commit thread are shown in Figure 9.

To support rolling back of speculative memory updates, the worker threads and the commit thread rely on versioned memory. Consequently, during normal speculative execution, the first action taken in each loop iteration is to advance to the next sequential memory version. Details regarding how versioned memory works are described in Section 6.1.

After checkpointing memory state, each worker thread collects the set of live registers that need to be checkpointed and sends them to the commit thread. The commit thread receives these registers and locally buffers them. Section 6.2 details which registers need to be checkpointed.

Next, each worker thread executes the portion of the original loop iteration allocated to it. Execution of the loop iteration generates a status: the loop iteration completed normally, misspeculation was detected, or the loop iteration exited the loop. The worker thread sends this status to the commit thread, and then, based on the status, either continues speculative execution, waits to be redirected to recovery code, or exits the loop normally.

The commit thread collects the status messages sent by each thread and takes appropriate action. If all worker threads successfully completed an iteration, then the cur-

```
while (true) {
  move_to_next_memory_version();
  produce_register_checkpoint(commit_thread);
  status = execute_loop_iteration();
  produce(commit_thread, status);
  if (status == EXIT)
    break;
  else if (status == MISSPEC)
    wait_for_resteer();
  else if (status == OK)
    continue;

recovery:
  produce_resteer_ack(commit_thread);
  flush_queues();
  regs = consume_register_checkpoint(commit_thread);
  restore_registers(regs);

  if (MULTI_THREAD_RECOVERY)
    execute_synchronized_loop_iteration();
}
```

(a) Worker Thread

```
do {
  move_to_next_memory_version();
  regs = consume_register_checkpoints(threads);

  status = poll_worker_statuses(threads);

  if (status == MISSPEC) {
    resteer_threads(threads);
    consume_resteer_acks(threads);
    rollback_memory();

    if (SINGLE_THREAD_RECOVERY)
      regs = execute_full_loop_iteration(regs);

    produce_register_checkpoints(threads, regs);
  } else if (status == OK || status == EXIT)
      commit_memory();
} while (status != EXIT);
```

(b) Commit Thread

Figure 9. Pseudo-Code for (a) Worker and (b) Commit threads

rent memory version is committed, and the register checkpoint is discarded.[3] If any thread detected misspeculation, the commit thread initiates recovery.

By collecting a status message from all worker threads each iteration, it is guaranteed that no worker thread is in an earlier iteration than the commit thread. Consequently, when recovery is initiated, step 1 is already complete. Recovery thus begins by asynchronously resteering all worker threads to thread-local recovery code. Once each thread acknowledges that it has been resteered, step 2 begins. The resteer acknowledgment prevents worker threads from speculatively modifying memory or producing values once memory state has been recovered or queues flushed. To recover state, the commit thread informs the versioned memory to discard all memory writes to the current or later versions. Additionally, the worker threads flush all queues used by the thread.

Lastly, the misspeculated iteration is re-executed. Two possibilities exist for this re-execution: the worker threads can run a non-speculative multithreaded version of the loop body (this version will have cyclic cross-thread dependences), or one thread (usually the commit thread) can run the original single threaded loop body. In the first case, the register-checkpoints collected at the beginning of the iteration are sent back to each thread, and the iteration is re-executed. In the second case, the commit thread will use the register checkpoint to execute the single-threaded loop body. The register values *after* the iteration has been re-executed will then be distributed back to the worker threads. The tradeoffs between single-threaded and multi-threaded recovery are discussed in Section 6.3.

[3]Since the register checkpoint is saved in virtual registers in the commit thread, no explicit action is required to discard the register checkpoint.

Note that the commit thread incurs overhead that scales with the number of worker threads. While this code is very light weight, there is a point at which using additional worker threads will result in worse performance, since one iteration in the commit thread takes longer than any worker thread. However, various solutions to this problem exist. First the commit thread code is parallelizable. Additional threads can be used to reduce the latency of committing a loop iteration. Second, the problem can be mitigated by unrolling the original loop. This effectively increases the amount of time the commit thread has to complete its bookkeeping. This can potentially increase the misspeculation penalty, but provided misspeculation is infrequent, the tradeoff should favor additional worker threads.

6.1 Versioned Memory

Speculative DSWP leverages versioned memory to allow for speculative writes to memory. Like transactional memories or memory subsystems used in TLS architectures [6, 7, 19], SpecDSWP's versioned memory buffers speculative stores. One version in the versioned memory roughly corresponds to a transaction in a transactional memory. If a memory version is committed, then the stores buffered in that version are allowed to update non-speculative architectural state. Otherwise, if the version is rolled back, the buffered stores are discarded.

SpecDSWP's versioned memory differs from transactional memories in three principal ways. First, the versioned memory does not provide any mechanism for detecting conflicting memory accesses across two versions. Any cross-version dependences that need to be enforced must explicitly be synchronized in the software. Second, many threads (and cores) can simultaneously read from and update the *same* memory version. Third, reads to addresses not explicitly written in a particular version are obtained from the latest version where an explicit write has taken place *even* if that version has *not* yet been committed.

The architectural semantics of the versioned memory are as follows. Note that this is a description of the semantics, *not* an implementation, of versioned memory. Each thread maintains a current version number. All threads in a process share a committed version number. Memory is addressed with (version number, address) pairs. Memory cells (i.e., a particular address in a given version) that have never been written are initialized with a sentinel value, ϵ. Stores from a particular thread update the specified address in the current version of the thread. Loads from a thread read from the specified address in the current version of the thread. If ϵ is read, the load will read from previous versions until a non-ϵ value is found. If, in the committed version number, ϵ is found, then the result of the load is undefined. Each thread can increment its current version number, update the process's committed version number, and discard all versions past the committed version number. The rollback operation resets all addresses in all versions past the committed version to ϵ and asynchronously updates all threads' current version number to the committed version number.

Notice, in addition to buffering speculative stores, the versioned memory semantics allow anti- (WAR) and output- (WAW) memory dependences to be safely ignored provided the source of the dependence executes in a different memory version than the destination of the dependence.

Implementation details of versioned memory are beyond the scope of this paper. However, observe that the relation to transactional memories suggests that using caches with augmented tags and coherence protocols is a viable implementation strategy. Garzarán et al. explore a vast design space [6] and many of the designs they discuss could be modified to implement versioned memory.

6.2 Saving Register State

Since recovery occurs at iteration boundaries, each thread need only checkpoint those registers which are live into its loop header, since these are the only registers necessary to run the loop and code following the loop. (Callee-saved registers must be handled specially, but due to space limitations the details have been omitted.) In addition to high-level compiler temporaries (virtual registers), it is necessary to checkpoint machine-specific registers which may be live into the loop header such as the stack pointer, global pointer, or register window position.

The set of registers to be checkpointed can be optimized by recognizing that registers that are live into the loop entry, but that are not modified in the loop (loop invariants in the speculative code), need not be checkpointed each iteration, since their values are constant for the loop execution. Instead, these registers can be checkpointed once per loop invocation. While these loop invariant registers can be checkpointed once per invocation, upon misspeculation, they *must* be recovered since, due to misspeculation, unexpected code may have executed and modified their values.

Core	Functional Units - 6-issue, 6 ALU, 4 memory, 2 FP, 3 branch L1I Cache - 1 cycle, 16 KB, 4-way, 64B lines L1D Cache - 1 cycle, 16 KB, 4-way, 64B lines, write-through L2 Cache - 5,7,9 cycles, 256KB, 8-way, 128B lines, writeback Maximum Outstanding Loads - 16
Shared L3 Cache	$>$ 12 cycles, 1.5 MB, 12-way, 128B lines, writeback
Main Memory	141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration
SyncArray	256 32-entry queues, 1-cycle inter-core latency

Table 1. Machine details

6.3 Iteration Re-execution

After restoring state, the commit thread must ensure that the misspeculated iteration executes nonspeculatively. We explored two options for how to perform this iteration: multi-threaded and single-threaded. With multi-threaded recovery, each thread runs the same portion of the original loop that it does during speculative execution except the speculated dependences are synchronized with cross-thread communication. This communication creates cyclic dependences between the threads, thus putting communication latency on the critical path for the recovery iteration. However, the sequential ordering of operations from the single-threaded loop guarantees that the cyclic communication pattern does not result in deadlock.

On the other hand, single-threaded recovery executes the original single-threaded loop body in the commit thread. For the recovery iteration, any potential benefit of thread parallelism is lost. However, the cost of communication on the critical path often outweighs the benefits of parallelism, potentially making this faster than multithreaded recovery. Single-threaded recovery also allows worker threads to overlap some of their recovery with the execution of the non-speculative iteration. As will be discussed in Section 7.4, our experiments have shown that the time taken to flush queues and restore certain machine specific registers can sometimes dominate the recovery time. Unfortunately, single-threaded recovery suffers from cold microarchitectural structures. In particular, the core running the commit thread does not have any application data loaded into its caches. These tradeoffs make it unclear which recovery style is superior.

7 Evaluation

This section evaluates our initial implementation of Speculative DSWP targeting a quad-core chip multiprocessor. Speculative DSWP was implemented in the VELOCITY backend optimizing compiler [24]. VELOCITY contains an aggressive classical optimizer, which includes global versions of partial redundancy elimination, dead and

Benchmark	Function	% of Runtime	# of Loop Ops.	Average Iterations per Invocation	SCCs Non-Spec.	SCCs Spec.	# of Comm. Ops.	# of Saved Regs	Max # of Versions	Speculation Types	% of Iters. with Misspec.
164.gzip	`deflate`	53%	586	24444826.0	9	336	677	3	-	CF	12.4%
256.bzip2	`fullGtU`	38%	99	123.4	2	51	17	25	32	CF	0.8%
181.mcf	`primal_net_simplex`	75%	254	17428.8	3	78	112	10	22	CF, SS	0.2%
052.alvinn	`main`	98%	181	30.0	57	63	82	9	2	None	0.0%
mpeg2enc	`dist1`	56%	136	6.3	1	101	4	10	20	CF	15.9%

CF = Control Flow, SS = Silent Store

Table 2. Benchmark Details

unreachable code elimination, copy and constant propagation, reverse copy propagation, algebraic simplification, constant folding, strength reduction, and redundant load and store elimination. VELOCITY also performs an aggressive inlining, superblock formation, superblock-based scheduling, and register allocation. Speculative DSWP was applied to benchmarks from the SPEC-CPU2000, SPEC-CPU92, and Mediabench suites. We report results for all benchmark loops in which the non-speculative DSWP transformation found no suitable partition, but SpecDSWP found a suitable partition, and the loop accounted for more than a third of the program runtime.

To evaluate the performance of SpecDSWP, we used a validated cycle-accurate Itanium 2 processor performance model (IPC accurate to within 6% of real hardware for benchmarks measured [17]) to build a multi-core model comprising four Itanium 2 cores connected by a *synchronization array* [20]. The models were built using the Liberty Simulation Environment [27]. Table 1 provides details about the simulation model.

The synchronization array (SA) in the model works as a set of low-latency queues. The Itanium ISA was extended with `produce` and `consume` instructions for inter-thread communication. As long as the SA queue is not empty, a `consume` and its dependent instructions can execute in back-to-back cycles. The ISA was additionally augmented with a `resteer` instruction that interrupts the execution of the specified core and restarts its execution at the given address. Finally, our model includes a versioned memory which supports 32 simultaneous outstanding versions. Speculative state is buffered both in the private L1 and L2 caches and in the shared L3 cache. In our experiments, speculative state never exceeded the size of the L3 cache.

The detail of the validated Itanium 2 model prevented whole program simulation. Instead, detailed simulation was restricted to the loops in question in each benchmark. The remaining sections were functionally simulated keeping the caches and branch predictors warm.

7.1 Experimental Results

Table 2 presents compilation and profile statistics for the chosen loops. These loops account for between 38% and 98% of the total benchmark execution time, and the loops contain between 99 and 586 static operations (excluding operations in called procedures). Table 2 also presents the number of SCCs in the PDG for each loop both before and after speculating dependences. Notice that for all the benchmarks (except `052.alvinn`), speculation dramatically increases the number of SCCs in the PDG. This increase in the number of SCCs directly translates into scheduling freedom for the partitioning heuristic.

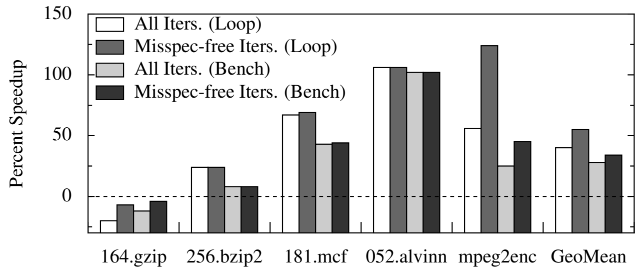


Figure 10. Speedup vs. single threaded code.

All loops were compiled to generate 3 worker threads and a commit thread, for a total of 4 threads. Figure 10 shows the speedup over single-threaded execution for these parallelizations. For each loop, the loop speedup and the benchmark speedup (assuming only the given loop is parallelized) are shown. Speedup is further broken down into speedup obtained ignoring iterations that misspeculated (i.e., assuming these iterations were not executed) and speedup for all iterations. All benchmarks, except `164.gzip`, exhibit considerable loop speedup ranging from 1.24x to 2.24x. The average speedup over all parallelized loops was 1.40x. The rest of this section examines the results of several of benchmarks in more detail to highlight interesting features of SpecDSWP.

7.2 A Closer Look: Communication Cost

The slowdown seen in `164.gzip` can be attributed to a poor partition, large amount of communication, and the high rate of misspeculation. While SpecDSWP (and DSWP in general) prevents communication latency from appearing on the critical path, the effects of communication instructions within a core *are* significant. In `164.gzip`, the schedule height of one iteration of the slowest thread increased (compared to the schedule height of a single-threaded loop iteration) due to resource contention between original loop instructions and communication instructions.

This increase was not offset by the reduction in schedule height due to parallelization. Improvements in the partitioning heuristic and additional optimizations may be able to improve the performance of SpecDSWP on this benchmark.

7.3 A Closer Look: Versioned Memory

While all parallelized loops rely on versioned memory to rollback speculative stores, the loops from `181.mcf` and `052.alvinn` benefit from the effective privatization provided by versioned memory. In `181.mcf`, execution of the function `refresh_potential` could be overlapped with other code within the `primal_net_simplex` loop by speculating that `refresh_potential` would not change a node's potential (using silent store speculation). The versioned memory allows subsequent iterations of the loop to modify the tree data structure used by `refresh_potential` without interfering with `refresh_potential`'s computation. Due to the complexity of the left-child, right-sibling tree data structure, it is unlikely that compiler privatization could achieve the effects of the versioned memory.

In `052.alvinn`, versioned memory privatized several arrays. The parallelized loop contains several inner loops. Each inner loop scans an input array and produces an output array that feeds the subsequent inner loop. While the loop is not DOALL (dependences exist between invocations of the inner loops), each inner loop can be put in its own thread. Each stage in the SpecDSWP pipeline, therefore, is one of the inner loops. However, privatization is needed to allow later invocations of an early inner loop to execute before earlier invocations of a later inner loop have finished. Note that `052.alvinn` did not require any true speculation. False dependences broken by the versioned memory were sufficient to achieve parallelism. Consequently, in addition to its role in SpecDSWP, versioned memory also has potential in improving parallelism for non-speculative DSWP.

7.4 A Closer Look: Misspeculation

As Figure 10 shows, of the loops that were parallelized, only `mpeg2enc` observed a significant performance loss due to misspeculation. This performance loss is directly attributable to the delay in detecting misspeculation, the significant misspeculation rate, and the cost of recovery. Only loop exits were speculated to achieve the parallelization in `mpeg2enc`. However, as Table 2 shows, `mpeg2enc` has only a few iterations per invocation. This translates to significant misspeculation due to loop exit speculation.

Despite the significant misspeculation, the parallelization benefit outweighs the recovery cost. However, optimization may be able to mitigate the deleterious effects of misspeculation. As SpecDSWP is currently implemented, upon misspeculation, state is rolled back to the state at the beginning of the iteration, and the iteration is re-executed non-speculatively. However, in the case of a loop exit misspeculation, observe that no values have been incorrectly computed, only additional work has been done. Consequently, provided no live-out values have been overwritten, it is only necessary to squash the speculative work; the iteration does not have to be rolled back and re-executed. With single-threaded recovery, the savings can be significant. For example, in `mpeg2enc`, on average a single-threaded loop iteration takes 120 cycles. The parallelized loop iteration, conversely, takes about 53 cycles. After one misspeculation, due to a cold branch predictor, the recovery iteration took 179 cycles to execute. Reclaiming these 179 cycles would significantly close the performance gap between the iterations without misspeculation and all the iterations. We are currently working to extend our SpecDSWP implementation to recognize these situations and to avoid unnecessarily rolling back state and re-executing the last loop iteration.

Given the deleterious effects of cold architectural state on recovery code and the significant parallelism present in this loop, it is natural to ask if multithreaded recovery would fare better. Our initial experiments with multithreaded recovery in `mpeg2enc` showed that all the parallelization speedup is lost due to recovery overhead. In particular, on Itanium, it was necessary to recover the state of the register stack engine (RSE) after misspeculation. Manipulating certain RSE control registers (e.g., CFM), requires a function call. The overhead of this recovery code was significant in comparison to the execution time of one loop iteration, significantly slowing recovery. By moving to single-threaded recovery, the compiler was able to overlap this recovery in the worker threads with the execution of the recovery iteration in the commit thread. In our experience this balance yielded better performing recovery code.

8 Summary

This paper introduced Speculative DSWP, a speculative extension to the pipelined multithreading technique DSWP. Contrary to existing thread-level speculation techniques, the parallelism extracted is neither DOALL nor DOACROSS. Instead, by combining speculation and pipeline parallelism, SpecDSWP is able to offer significant speedup on a variety of applications even in the presence of long inter-core communication latency. Our proof-of-concept implementation extracted an average of 40% loop speedup on the applications explored with support for only control and silent store speculation. As the implementation is refined and support for more types of speculation, such as memory alias or value speculation, are added, we expect to be able to target more loops and offer still more performance.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. Additionally, we thank the anonymous reviewers for their insightful comments. The authors acknowledge the support of the GSRC Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program. This work has been supported by Intel Corporation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of our sponsors.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[2] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[3] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[4] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.

[5] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, New York, NY, USA, 1988. ACM Press.

[6] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture Code Optimization*, 2(3):247–279, 2005.

[7] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[8] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[9] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 59–70, 2004.

[10] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[11] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.*, 22(3):326–379, 2004.

[12] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[13] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 22–31, New York, NY, USA, 2000. ACM Press.

[14] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience*, 14(11):911–932, 2002.

[15] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, pages 365–372, New York, NY, USA, 1999. ACM Press.

[16] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[17] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[18] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[19] R. Rajwar and J. R. Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125, Nov-Dec 2003.

[20] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[22] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[23] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[24] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[25] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 176–185, New York, NY, USA, 1986. ACM Press.

[26] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[27] 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*, pages 271–282, November 2002.

[28] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual International Symposium on Microarchitecture*, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.