Compiler and Hardware Support for Reducing the Synchronization of Speculative Threads

ANTONIA ZHAI

University of Minnesota

J. GREGORY STEFFAN

University of Toronto

CHRISTOPHER B. COLOHAN

Google

and

TODD C. MOWRY

Carnegie Mellon University

Thread-level speculation (TLS) allows us to automatically parallelize general-purpose programs by supporting parallel execution of threads that might not actually be independent. In this article, we focus on one important limitation of program performance under TLS, which stalls as a result of synchronizing and forwarding scalar values between speculative threads that would otherwise cause frequent data dependences and, hence, failed speculation. Using SPECint benchmarks that have been automatically transformed by our compiler to exploit TLS, we present, evaluate in detail, and compare both compiler and hardware techniques for improving the communication of scalar values. We find that through our dataflow algorithms for three increasingly aggressive instruction scheduling techniques, the compiler can drastically reduce the *critical forwarding path* introduced by the synchronization and forwarding of scalar values. We also show that hardware techniques for reducing synchronization can be complementary to compiler scheduling, but that the additional performance benefits are minimal and are generally not worth the cost.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors)—*Multiple-instruction-stream, multiple-data-stream processors (MIMD)*; D.3.4 [Programming Languages]: Processors—*Compilers*; C.1.4 [Processor Architectures]: Parallel Architectures

Authors' address: Antonia Zhai, Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455. zhai@cs.umn.edu. J. Gregory Steffan, University of Toronto, Toronto, Canada. Christopher B. Colohan, Google, Ann Arbor, Michigan 48104. Todd C. Mowry, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org. © 2008 ACM 1544-3566/2008/05-ART3 \$5.00 DOI 10.1145/1369396.1369399 <http://doi.acm.org/10.1145/1369396.1369399>

ACM Transactions on Architecture and Code Optimization, Vol. 5, No. 1, Article 3, Publication date: May 2008.

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Thread-level speculation, chip-multiprocessing, automatic parallelization, instruction scheduling

ACM Reference Format:

Zhai, A., Steffan, J. G., Colohan, C. B., and Mowry, T. C. 2008. Compiler and hardware support for reducing the synchronization of speculative threads. *ACM Trans. Architect. Code Optim.* 5, 1, Article 3 (May 2008), 33 pages. DOI = 10.1145/1369396.1369399 <http://doi.acm.org/10.1145/1369396.1369399>

1. INTRODUCTION

Chip-multiprocessors have become nearly commonplace [IBM Corporation 2007; Intel Corporation 2005; AMD Corporation 2005; Sun Corporation 2005]. Using this multithreaded hardware to improve the throughput of a workload is relatively straightforward, but to improve the performance of a single application is still an open problem and generally requires some form of parallelization. How can we parallelize all of the applications that we care about? Writing parallel software can be a daunting task; we would much rather have the compiler parallelize our code for us. Traditionally, compilers have parallelized by proving that potential threads are independent [Blume et al. 1996; Hiranandani et al. 1993; Wilson et al. 1994]—but this is extremely difficult, if not impossible, for many general-purpose programs because of their complex data structures and control flow and use pointers and runtime inputs. One promising alternative for overcoming this problem is *thread-level speculation* (TLS) [Knight 1986; Akkary and Driscoll 1998; Franklin and Sohi 1992; Cintra and Torrellas 2002; Dubey et al. 1995; Sohi et al. 1995; Gupta and Nim 1998; Hammond et al. 1998; Krishnan and Torrellas 1999; Marcuello and Gonzales 1999; Oplinger et al. 1999; Steffan et al. 2000; Tsai et al. 1999], which allows the compiler to create parallel threads without having to prove that they are independent. The underlying hardware ensures that interthread dependences through memory are satisfied and reexecutes any thread for which they are not.

1.1 The TLS Execution Model

In TLS, the compiler partitions a program into speculatively parallel threads without having to decide at compile time whether they are independent. At runtime, the underlying hardware determines whether interthread data dependences are preserved and reexecutes any thread for which they are not. This execution model allows the parallelization of programs that were previously nonparallelizable, as demonstrated by the following example.

The most straightforward way to parallelize a loop is to execute multiple iterations of that loop in parallel. With TLS, the loop in Figure 1a can be parallelized by the compiler without deciding whether the pointer p points to the same memory location as pointer q from any previous iteration. Figure 1b demonstrates the speculative parallel execution of the loop on a four-processor shared-memory multiprocessor that supports TLS, where each thread corresponds to a single iteration of the loop. Speculation will succeed as long as no

```
do {
    ...
    load *p;
    ...
    store *q;
    ...
} while (condition)
```

(a) A loop with ambiguous loop-carried data dependencies.

(b) Speculatively parallel threads.

Fig. 1. Loops with potential inter-thread data dependences can be parallelized under TLS.

load (through pointer p) executes out-of-order with respect to a store to the same address (through pointer q) by a logically earlier thread. In the example, the load in *thread 4* accesses the location 0x88 out-of-order with respect to a store in *thread 1*. Hence, speculation fails for *thread 4*, which is then squashed and reexecuted to ensure correctness. The key advantage of TLS is that correctness is preserved while extracting whatever parallelism exists between ambiguous dependences between threads.

1.2 The Importance of Value Communication for TLS

In the context of TLS, value communication refers to the satisfaction of any true (read-after-write) dependence between *epochs* (sequential chunks of work performed speculatively in parallel). From the compiler's perspective, there are two ways to communicate the value of a given variable. First, the compiler may speculate that the variable is not modified (Figure 2a). However, if at runtime the variable actually *is* modified then the underlying hardware ensures that the misspeculated epoch is reexecuted with the proper value. This method only works well when the variable is modified infrequently, since the cost of misspeculation is high. Second, if the variable is frequently modified, then the compiler may instead synchronize and forward¹ the value between epochs (Figure 2b). However, such synchronization creates a serialization called a *critical forwarding path* (Figure 2c). Since a parallelized region of code will contain many variables, the compiler will employ a combination of speculation and synchronization as appropriate. In the case of synchronization, the compiler can also schedule instructions to reduce the *critical forwarding path* (Figure 2c), increasing parallel overlap and improving performance.

To further improve upon static compile-time choices between speculating or synchronizing for specific memory accesses, we can exploit dynamic runtime behavior to make value communication more efficient. For example, we might employ *value prediction* [Akkary and Driscoll 1998; Gabbay and Mendelson 1996; Lipasti and Shen 1996; Marcuello et al. 1999a; Rotenberg

¹This is also known as *doacross* [Cytron 1986; Padua et al. 1980] parallelization.

Fig. 2. A memory value may be communicated between two epochs (E1 and E2) through (a) speculation, (b) synchronization, or (d) prediction. In the case of synchronization, we can further accelerate value communication by reducing the critical forwarding path (c).

et al. 1997; Sazeides and Smith 1997; Wang and Franklin 1997], as illustrated in Figure 2d.

To get a sense of the potential upside of enhancing value communication under TLS, let us briefly consider the ideal case. From a performance perspective, the ideal case would correspond to a value predictor that could perfectly predict the value of any interthread dependence. In such a case, speculation would never fail and synchronization would never stall. While this perfect-prediction scenario is unrealistic, it does allow us to bound the potential impact of improving value communication in TLS.

Figure 3 shows the impact of perfect prediction on a set of speculatively parallelized loops for several SPECint benchmarks, running on a four-processor CMP that implements hardware support for TLS [Steffan et al. 2000]. Each bar is normalized to the execution time of the original sequential version, such that bars less than 100 are speeding up. Each bar is broken down into four segments explaining what happened during all potential *graduation slots*. The number of graduation slots is the product of: (1) the issue width (four in this case), (2) the number of cycles, and (3) the number of processors (four in this case). The *fail* segment represents all slots wasted on failed thread-level speculation and the remaining three segments represent slots spent on successful speculation. The *busy* segment is the number of slots where instructions graduate; the *sync* portion represents slots spent waiting for synchronization for

Fig. 3. Potential for improved value communication. For each experiment, we show normalized execution time of loops that can potentially speed up when speculatively parallelized (smaller is better). U is the unscheduled TLS version run speculatively in parallel and F shows perfect prediction of forwarded values.

a forwarded location; the *other* segment is all other slots where instructions cannot graduate. More details of our experimental framework are given later in Section 4.

In Figure 3, the U experiment shows the performance of the speculatively parallelized regions of code after basic compiler insertion of synchronization and forwarding for frequent dependences, as described later in Section 2. Disappointingly, this basic synchronization approach, on average, does not result in any speedup. The F experiment shows the impact of perfect prediction of forwarded values. In effect, this means that there will be no time spent waiting for synchronization of forwarded values. Most applications improve tremendously, by an average of 33.7% across all applications. However, LI suffers from a significant increase in failed speculation: perfect forwarded value prediction has increased parallel overlap, but exposed new cross-epoch dependences, causing speculation to fail more often. Overall, this experiment indicates that reducing the critical forwarding path between parallel threads has significant potential for improving performance.

1.3 Improving Value Communication for TLS

Given the importance of efficient value communication for TLS, what solutions can we implement to approach the ideal results of Figure 3? Figure 2 shows the spectrum of possibilities: i.e., speculation, synchronization, reducing the critical forwarding path, and prediction. Hardware support for efficient *speculation* has already been addressed in a number of papers on TLS [Akkary and Driscoll 1998; Cintra et al. 2000; Gopal et al. 1998; Gupta and Nim 1998; Hammond et al. 1998; Krishnan and Torrellas 1999; Marcuello and Gonzales 1999; Oplinger et al. 1999; Steffan et al. 2000; Tsai et al. 1999]. For any interthread data dependence that occurs frequently, it is most efficient to synchronize the producer and consumer and explicitly forward the value between them (Figure 2b) to avoid failed speculation; in Section 2, we describe a basic algorithm for inserting such synchronization for scalar variables. While techniques for reducing failed speculation can have a significant impact on performance [Steffan et al. 2002; Colohan et al. 2006; Cintra and Torrellas 2002], in

this article we focus on techniques for reducing synchronization, since the results in Figure 3 indicate that this will lead to the greater overall performance benefit. In particular, we explore the following two approaches.

1.3.1 Reducing the Critical Forwarding Path. Once synchronization is introduced to explicitly forward values across epochs, it creates a dependence chain across the threads that may ultimately limit the parallel speedup. We can potentially improve performance in such cases by using scheduling techniques to reduce the critical path between the first use and last definition of the dependent value, as illustrated in Figure 2c. We present and evaluate dataflow algorithms for three increasingly aggressive instruction scheduling techniques that reduce the *critical forwarding path* introduced by the synchronization associated with this data forwarding. We also evaluate a hardware method for prioritizing instructions to reduce the critical forwarding path.

1.3.2 Value Prediction. We can exploit value prediction by having the consumer of a potential dependence use a predicted value instead, as illustrated in Figure 2d. After the epoch completes, it will compare the predicted value with the actual value; if the values differ, then the normal speculation recovery mechanism will be invoked to squash and restart the epoch with the correct value. In this article, we explore using value prediction as a replacement for *synchronization*. We refer to this technique as “*forwarded value prediction*,” where successful prediction avoids the need to stall waiting for synchronization.

1.4 Contributions

This article makes the following contributions:

1. We evaluate a comprehensive set of techniques for reducing the impact of synchronization of scalar values within a system that supports thread-level speculation and demonstrate that many can result in significant performance gains. While we evaluate these techniques within the context of our own implementation of TLS, we expect to see similar trends within other TLS environments, since the results are largely dependent on application behavior rather than the details of how speculation support is implemented.
2. We present three increasingly aggressive data-flow scheduling algorithms for reducing the critical forwarding path and show that scheduling loop-induction variables and other scalars results in significant performance gains for most applications.
3. We evaluate hardware techniques both before and after the compiler has eliminated obvious data dependences and scheduled critical forwarding paths, thereby removing the “easy” bottlenecks to achieving good performance.
4. We compare and contrast compiler and hardware techniques and find that while certain techniques can be complementary, compiler scheduling is generally more effective at improving the performance of value communication for TLS than hardware techniques, such as value prediction.

2. BASIC COMPILER INSERTION OF SYNCHRONIZATION

In the previous section, we described how for every store-load pair the compiler has the important decision of whether to speculate or synchronize. For store-load pairs that are frequently dependent, the best alternative is to synchronize and forward the value between speculative threads. In this section, we present a general algorithm for inserting this synchronization to communicate scalar values between speculative threads. We target the set of scalar values that are defined in the enclosing scope of the parallelized loop, do not have their addresses taken, and satisfy one of the following criteria: 1) belong to the set of scalars with downward-exposed definitions *and* upward-exposed uses (i.e., the value is *live* between threads); or 2) is defined in the loop and is *live* when the loop exits. These scalars are referred to as *communicating scalars*. In contrast to these scalars, global variables and values referenced with pointers may have aliased accesses and be modified by instructions from outside of the loop body. Developing compiler optimizations to improve value communication for these other types of values is beyond the scope of this article [Zhai et al. 2004].

2.1 Forwarding Primitives

We employ several primitives (implemented as new TLS instructions), which define the location and timing of synchronization and value forwarding. In our approach, the compiler allocates forwarded variables on a special portion of the stack called the *forwarding frame* [Steffan et al. 2005] that supports the communication of values between speculative threads. The forwarding frame is defined by a base address within the stack frame and an offset from that base address; this way, any regular load or store to an address within the pre-defined forwarding frame address range can be treated appropriately. The address range of the forwarding frame is defined through the software interface at the beginning of every speculative region. Accesses to the forwarding frame are exempt from the data-dependence tracking mechanisms of the underlying TLS hardware support.

The compiler uses two value communication primitives, *wait* and *signal*, to forward values. The *wait* instruction stalls execution until the value is produced by the previous thread, which communicates that value through the *signal* instruction. For the first thread of a speculatively parallelized loop, the *wait* instruction does not stall (since there is no producer). These primitives implement fine-grained synchronization, since we synchronize on each individual value (rather than stalling the entire thread before the first use of any forwarded value and sending after the last definition of any forwarded value). This granularity also allows the processor to issue instructions out-of-order with respect to a blocked wait for value instruction.

In the remainder of this section, we describe how the compiler inserts instructions that perform the synchronization and communicate values.

2.2 Constraints on Placement

The proper placement of *wait* and *signal* instructions can be described by a series of constraints which we describe here for a single synchronized scalar;

the same constraints can be applied to each synchronized scalar individually. First, we want the last write to a scalar in an epoch to execute before the next epoch reads that scalar, regardless of the path of execution taken by either epoch. Hence, we have the first two constraints:

1. a wait must occur before any use of the scalar on any execution path;
2. a signal must occur after the last definition of the scalar on any execution path.

If a signal is omitted on a certain execution path, the waiting epoch can potentially stall indefinitely, although, in reality, the waiting epoch only stalls until all previous epochs are completed. To avoid such unnecessary stalls, we require:

3. a signal must occur on every possible path through an epoch.

Given these first three constraints, a correct program can be created by trivially placing all wait instructions at the top of each epoch, and all signal instructions at the bottom of each epoch. However, such a transformation would completely serialize execution. To remedy this situation, we apply two additional constraints for the sake of improving performance:

4. each wait should be placed as late as possible;
5. each signal should be placed as early as possible.

2.3 Placement Algorithm

Intuitively, a placement algorithm for wait instructions would involve putting a wait for a scalar at the top of the epoch and then pushing the wait downward through the code. When a branch is encountered, the wait can be duplicated and pushed down on both sides of the branch. The motion stops when a use of the scalar is encountered. For placement of signal instructions, the converse of this algorithm is used. Deciding which basic block should contain a wait or signal can be implemented as a data-flow analysis (described in detail below): within a basic block, the wait is placed directly above the first use of the scalar and the signal is placed directly below the last definition of the scalar.

We now present a data-flow algorithm for placing wait and signal instructions in accordance with the above constraints. While we only show the algorithm for placing signal instructions, note that the converse of this algorithm is used to place wait instructions. (A proof of the correctness of this algorithm can be found in our previous work [Zhai 2005].)

We define our data-flow analysis over the set of communicating scalars V on the control-flow graph (CFG) $G = (N, E, s, e)$ of the epoch, where N is the set of nodes which represent basic blocks, E is the set of edges, and s and e are the unique *start* and *end node* of G (note that the start and end node contain no code). Since *critical edges* (i.e., any edge connecting a node with more than one successor to a node with more than one predecessor) would block certain code motion and make our analysis difficult, we break any such edges into two edges using *synthetic nodes* [Knoop et al. 1992].


```

while (condition()) {
    if (a > b + c) {
        if (++a > 10)
            b = b + 2;
        else
            b = b + 3;
        b = b + 4;
    } else {
        do {
            b = b + 5;
        } while (condition2());
        ++a;
    }
}
    
```

(a) Original code.

(b) Control-flow graph.

(c) Inserting waits and signals.

Fig. 4. Example of how waits and signals are inserted.

At each node $n \in N$ we define a predicate $LocalDef(n)$ to be the set of communicating scalars that are defined at n . Since the `signal` instruction that forwards the value of $v \in V$ must occur after the last definition to v on all possible execution paths, we define *no-more-definitions* at the exit of each node n ($NMD(n)$) to be the set of scalars that are not defined on any execution path from the exit of n to e . This function can be computed using the data-flow analysis defined in the following equations:

$$NMD(n) = \begin{cases} V & \text{if } n = e \\ \bigcap_{s \in succ(n)} (NMD(s) - LocalDef(s)) & \text{otherwise} \end{cases} \quad (1)$$

For the example in Figure 4, the shaded boxes in Figure 4c indicate where $a \in NMD(n)$.

While it would be correct to insert `signal` instructions at all nodes n for which $NMD(n) \neq \{\}$, this may cause a single execution path from n to e to have many signals. We avoid redundant signals through the function $signal(n)$, defined at the exit of node n , which determines the placement of `signal` instructions:

$$signal(n) = \begin{cases} \{\} & \text{if } n = s \\ NMD(n) - \bigcap_{p \in pred(n)} NMD(p) & \text{otherwise} \end{cases} \quad (2)$$

Fig. 5. Reducing the critical forwarding path through instruction scheduling.

We only compute *signal* for forwarded scalars at the exit of each node, so that within a basic block, the signals can be pushed as far back as possible and placed directly below the last definition of these scalars. Figure 4c shows the synchronization points for scalars *a* and *b* for the original code in Figure 4a.

3. COMPILER TECHNIQUES FOR REDUCING THE CRITICAL FORWARDING PATH

Although synchronization is better than speculation for a data dependence that occurs frequently, the resulting serialization can still limit performance. In fact, the performance of many applications that exploit TLS is limited by the critical forwarding path. What can the compiler do to shrink the critical forwarding path? The key idea is to reduce the number of instructions between each wait/signal pair. However, this becomes more difficult in the presence of intrathread control flow and data dependences. Figure 5a shows an example loop that the compiler has speculatively parallelized by partitioning the loop into epochs. The compiler decides to synchronize and forward the scalar *A*, which is read and written in every iteration of the loop, by inserting a wait operation before the first *use* of *A*, and a signal operation after the last *definition* of *A*. Figure 5a shows that the flow of the value of *A* between epochs serializes the parallel execution, and so we refer to it as a *critical forwarding path*. Figure 5b

shows the example loop after the compiler has scheduled the code to reduce the critical forwarding path. The scheduling algorithm has duplicated the computation of $A=A+1$ as well as the signal and moved them into the conditional structure. If the condition on A is rarely true, then less work is performed before reaching each signal (by deferring the computation of $B=2$ and $\text{work2}()$). As shown in the figure, this reduces the stall time for each epoch, thereby improving overall execution time. We describe an algorithm for scheduling instruction to reduce the critical path in Section 3.2. The transformations described above preserve the control and data dependences within each epoch and thus is referred to as the conservative instruction scheduling algorithm.

It is potentially beneficial to move code past control and data dependences [Chang et al. 1991; Fisher 1981; Gallagher et al. 1994; Nicolau 1989] to further reduce the critical forwarding path. For example, if a certain path is executed more frequently than alternative paths, then it is advantageous to speculatively schedule the critical forwarding path to exploit this fact. To illustrate, if the *else* clause is more frequently executed than the *then* clause in Figure 5b, we could schedule “ $A=A+1$; $\text{signal}(A)$,” from the *else* clause above the *if* structure to further shrink the critical forwarding path in the common case. Thus, our new schedule involves control speculation and requires the ability to recover whenever our speculation is incorrect. Similarly, we can schedule code from the critical forwarding path past ambiguous data dependences, given the additional hardware support to detect when such speculation has failed. We describe and evaluate schemes for scheduling the critical forwarding path using intraepoch control speculation and data-dependence speculation in Section 3.3.

3.1 Related Work

The most relevant related work is the Wisconsin Multiscalar [Sohi et al. 1995; Vijaykumar 1998] compiler, which performs synchronization and scheduling for register values [Vijaykumar 1998] (the Multiscalar effort also evaluated hardware support for automatically detecting and synchronizing data dependences [Moshovos et al. 1997]). The Multiscalar scheduler was designed with Multiscalar tasks in mind and these usually consist of a few basic blocks that do not contain procedure calls or loops. In contrast, our speculative threads are much larger, on average, than Multiscalar tasks and contain complex control flow. This inspired the data flow-based scheduler presented in this article, which can move instructions past inner loops and procedure calls. The Multiscalar compiler does not schedule code beyond the point within a task where it is no longer critical, as determined by a simplified machine model; in contrast, because we believe that accurate determination of this point at compile-time in an out-of-order machine is extremely difficult, we schedule producer instructions as early as possible. Another difference is that our more general approach to scheduling handles loop index variables automatically (by scheduling them at the top of the loop), rather than having to treat them as a special case. A final difference is that we evaluate the benefits of speculatively scheduling code past control and data dependences (as discussed later in Section 3.3). We

modified our scheduler to mimic the Multiscalar scheduler and we contrast the performance impact of both approaches later in Section 5.2.

Concurrent with our work, [Zilles and Sohi 2002; Zilles 2002] proposed decomposing a program into speculative threads by having a master thread execute a *distilled* version of the program that orchestrates and predicts values for slave threads. In this scheme, values are precomputed by the master thread and distributed to the slave threads (as opposed to being updated and forwarded between consecutive speculative threads). A potential advantage of this master/slave approach is that it effectively removes interprocessor communication from the critical forwarding path. We note that the scheduling techniques that we present later in this article could potentially be applied to the distilled code in the master thread.

Other schemes for TLS hardware support provide the means to synchronize and forward values between speculative threads, but provides relatively little compiler support for optimizing interthread data dependences [Akkary and Driscoll 1998; Gupta and Nim 1998; Hammond et al. 1998; Marcuello et al. 1999b; Steffan et al. 2002; Moshovos et al. 1997], while others provide such support but do not schedule instructions to reduce the critical forwarding path [Cintra and Torrellas 2002]. Other papers have proposed manual optimizations that are able to perform sophisticated optimizations to deal with complex interthread data dependences [Prabhu and Olukotun 2005, 2003; Colohan et al. 2005], however, it is not clear how such optimizations can be integrated into a compiler.

Our algorithm for reducing the critical forwarding path builds upon previous data-flow approaches to code motion, namely *partial redundancy elimination* [Knoop et al. 1992]. Previous work on speculative code motion to exploit frequently executed paths includes trace scheduling [Fisher 1981], superblock scheduling [Chang et al. 1991], and hot-path analysis [Ammons and Larus 1998]. There has also been work on aggressive load/store reordering where the runtime check and recovery are performed entirely in software [Nicolau 1989] or through a hybrid hardware/software approach [Gallagher et al. 1994].

3.2 Conservative Scheduling Algorithm

Similar to the synchronization placement algorithm described in Section 2, we define our conservative instruction scheduling algorithm as a set of data-flow analyses over the set of communicating scalars V on the control-flow graph $G = (N, E, s, e)$. We initialize the algorithm by placing all signals at the exit node e . Note that in our implementation of this algorithm, we have chosen only to move signal instructions (and the instructions they depend upon) upward in the CFG; although the converse of this algorithm can be applied to moving wait instructions (and the instructions that depend upon them) downward in the CFG, our experiments showed little additional performance benefit since downward code motion is often blocked by data-dependent control dependences. (A proof of correctness for this algorithm can be found in our previous work [Zhai 2005].)

As we schedule the instructions, we must identify at each node the computation that the eventual signal depends upon. Since we cannot represent

Fig. 6. Illustration of the semilattice (a) over which the *stack* data-flow analysis is defined and the *transfer* function (b, c, and d) used for computing the value of *stack* in Equation (3).

these computations as binary values, bit vector-based data-flow analysis is inadequate. Hence, at each node n , we keep a *stack*—denoted as $stack(n, v)$ —of computation for each communicating scalar. This stack records the computation necessary to produce the value of a communicating scalar v if it is to be sent from the node n .

The domain of the *stack* data-flow problem is the set of all possible configurations of the computation stack. This domain, along with the meet operator (described later), defines a semilattice (shown in Figure 6a). All nodes are initialized to $\top$. If a given node is found to be a safe place for the *signal* instruction, then *stack* returns a nonempty stack of computation, otherwise *stack* returns $\perp$. The following data-flow equation computes $stack(n, v)$ at the exit of each node:

$$stack(n, v) = \begin{cases} \boxed{\text{signal } v} & \text{if } n = e \\ \prod_{m \in succ(n)} transfer(m, v, stack(m, v)) & \text{otherwise} \end{cases} \quad (3)$$

where the *transfer* function is defined as follows:

- If $stack(m, v) = \top$, then $transfer = \top$.
- If the computation chain for v in the stack $stack(m, v)$ depends on a value w produced by node m , then the computation that produces w is added to the computation stack, as illustrated in Figure 6b.
- If the computation chain in the stack $stack(m, v)$ does not depend on a value produced by the computation at node m , then $transfer = stack(m, v)$, as illustrated in Figure 6c.
- If we cannot resolve the dependence between the computation chain for v and the computation in node m , we should stop the code motion; hence, $transfer = \perp$, as illustrated in Figure 6d.
- If a *wait* is issued for any exposed scalar in the computation chain, the code motion should stop; hence, $transfer = \perp$.

The meet operator $\prod$ for the *stack* problem is defined as follows: if any input is $\perp$ then the output is $\perp$; if any input stack differs from any other input stack,

then the output is $\perp$; otherwise, the meet operator returns the input stack, or $\top$ if all inputs are $\top$. The meet operator combined with the domain of the *stack* function defines a semilattice of height three, thus our data-flow problem is well-defined.

We also define the data-flow problem *earliest* to find the earliest synchronization point for each communicating scalar. *Earliest* is a bit-vector problem defined over the set of communicating scalars V on the control-flow graph G . The *earliest*(n, v) function is true at node n for v if no node prior to n is a safe place to schedule the signal on some execution path starting at s :

$$earliest(n, v) = \begin{cases} true & \text{if } n = s \\ \bigvee_{m \in pred(n)} (\neg safe(m, v) \wedge earliest(m, v)) & \text{otherwise} \end{cases} \quad (4)$$

where $safe(m, v) = (stack(m, v) \neq \perp)$, and all nodes are initialized to false.

For each node that is both *safe* and *earliest* for a scalar v , we insert the contents of v 's stack either at the beginning of the node, or immediately after the computation that stopped code motion (a wait instruction or ambiguous pointer reference) if it exists. We replace references to v with temporary variables, and update the unscheduled computation to use these temporaries.

Figure 7a illustrates solutions for *stack* and *earliest* for the example shown earlier in Figure 4. *Earliest* is true for scalar a only at the top node. The stack for the scalar a at the top node contains only the one instruction required to compute a and the `signal(a)` instruction required to forward the value. Figure 7b shows the transformed program. This transformation can potentially introduce redundant copy instructions into the program, such as the `b=t2` instruction in Figure 7b. We expect scalar optimization passes to remove these instructions. Note that this transformation can either expand code size (by duplicating computations at join points), or reduce code size (by performing a form of common subexpression elimination at branch points). We observe in our experiments that the maximum code expansion because of instruction scheduling is around 1% for all benchmarks.

3.3 Aggressive Scheduling Algorithms

In the conservative scheduling algorithm from the previous section, the backward motion of `signal` operations (and the instructions on which they depend) are often obstructed by control dependences and ambiguous data dependences. To make scheduling more aggressive, we will discuss both the compiler techniques and the hardware support necessary to allow for instruction scheduling beyond intraepoch control and data dependences.

3.3.1 Scheduling Past Control Dependences. Data-flow analysis conservatively assumes that all execution paths are possible and finds the minimal solution that satisfies all possible execution paths. In practice, however, only a small number of execution paths are frequently executed at runtime. By taking this into account, we can schedule instructions aggressively for the common cases at the cost of possibly incurring an expensive recovery operation on the less frequently executed paths.

(a) Solutions to the *stack* and *earliest* problems.

(b) Code transformation.

Fig. 7. Instruction scheduling for the code in Figure 4.

Fig. 8. Modified data-flow analysis for speculative instruction scheduling.

When we optimize for the common case, we will schedule code as early as possible and `signal` the values as soon as they are available. If a less frequent path is taken, then this `signal` will have forwarded the wrong value to the next epoch—we need a mechanism to recover from this. For recovery, we first notify the next epoch that it received an invalid value and then we forward the correct value to the next epoch. The notification of the next epoch is done using the `violate_epoch` instruction, which passes the identity of the communicated scalar—this instruction first discards the previously forwarded value, and then checks to see if the wrong value has already been consumed. If the incorrect value was consumed, then the epoch is violated and restarts; otherwise it is allowed to proceed. If instructions are speculatively scheduled past branches (e.g., NULL pointer checks), then exceptions may occur in the scheduled code. When an exception occurs it should cause a violation, and a nonspeculatively scheduled copy of the code should then be executed to ensure that the exception was real.

We have modified the conservative scheduling algorithm from Section 3.2 to speculate on control dependences. We make the algorithm more aggressive by modifying the meet operator $\sqcap$ used in the *stack* data-flow analysis in Equation (3). A profiling run is conducted to report the number of epochs executed, a.k.a. epoch count, for each loop, as well as the number of times each branch is taken/not taken. The speculative instruction scheduling is only attempted for nodes that are frequently executed.

The meet operator $\sqcap$ for *stack* is modified as shown in Figure 8a. When evaluating the meet operator $\sqcap$ at node n for the scalar v , we first operate on the set of successors, where the edge (n, s_i) is a frequently taken branch. Then, for each node s_j , where (n, s_j) is not a frequently taken branch, we verify whether $\text{transfer}(s_j, v, \text{stack}(s_j, v))$ is compatible with the partially evaluated $\text{stack}(n, v)$. If this verification fails, then we add a new node on the edge (n, s_j) which contains a single `violate_epoch` instruction. We also make a minor change to the definition of *earliest* (shown earlier in Equation (4)): *earliest* is always true for these new `violate_epoch` nodes, thereby making the scheduling algorithm

automatically insert the signal stack at the appropriate point on the execution paths starting at the edge (n, s_j) . Figure 8a illustrates how the two compatible computations on the frequently executed nodes are scheduled above node N , while the infrequently executed node on the right causes the next thread to be violated and reexecuted with the correct value.

3.3.2 Scheduling Past Data Dependences. We now consider how our conservative scheduling algorithm can be extended to allow code motion beyond potential data-dependences. Using the output from an automatic data-dependence profiling tool, our compiler can reason about the likelihood of data-dependence violations at runtime if the code associated with generating a particular signal operation is speculatively moved back ahead of a given potentially conflicting store instruction. If a data-dependence does occur at runtime, we must first *detect* this situation, and then *recover* from our misspeculation. We *detect* data-dependences by defining two new instructions: `mark_load` instructs the hardware to remember (i.e., “mark”) the specified memory location. If any subsequent store modifies a marked location then the speculation fails. Once we have reached a point where the potential data-dependence has been resolved then the `unmark_load` clears the mark on the memory location. If speculation fails or when an exception occurs, we *recover* by violating the current epoch and its successors. When the epoch restarts, it runs a different version of the code without speculative scheduling past data-dependences. It is worth noting that this architectural support for speculative loads is quite similar to the LD.A and CHK.A instructions [Gallagher et al. 1994] available in the Intel IA-64 architecture. One important difference, however, is that when the speculative code motion fails, in our case, the underlying TLS recovery mechanism rewinds execution to the start of the epoch; in contrast, under IA-64 the results of an LD.A instruction must be explicitly validated by a CHK.A instruction. (Further details on the implementation of `mark_load` and `unmark_load` can be found in our technical report [Zhai et al. 2002].)

To implement scheduling across potential data-dependences, we modify the *transfer* function described earlier in Section 3.2 (and used in Equation 3), as shown in Figure 8b. When scheduling a stack of instructions across a potentially dependent store, we mark all potentially conflicting loads in the stack as being *possibly conflicting*. When two stacks are merged at node n through the meet operator $\sqcap$, *possibly conflicting* marks are merged using logical *or*. At the time of code generation, we add a `mark_load` instruction after each *possibly conflicting* load. For all load instructions that are marked as *possibly conflicting*, an `unmark_load` is inserted at the original location of the load instruction.

3.3.3 Complementary Effects. Control and data-dependence speculation can be complementary. Figure 9 shows an example where the combination of a control and a data hazard prevent the code from being scheduled early and where speculation on either type of hazard alone will not yield any benefit. By speculating on both control and data-dependences in tandem, the computation

Fig. 9. Speculation on control and data dependences can be complementary.

of variable a can be moved upward next to the `wait` operation, thereby resulting in a much shorter critical forwarding path for the common case.

4. INFRASTRUCTURE

We now describe our basic compiler infrastructure and target hardware support for TLS, as well as our simulation infrastructure and experimental framework.

4.1 Compiler Infrastructure

Our compilation infrastructure is based on the Stanford SUIF 1.3 compiler system [Wilson et al. 1994]. In addition to scheduling the critical forwarding path, our compiler also performs the tasks described below when automatically transforming a program to exploit TLS.

Each iteration of the selected loop corresponds to a single epoch and is assigned to a separate processor during parallel execution. The selected loops can potentially contain nested loops and/or function calls.

4.1.1 Deciding Where to Speculate. For this article, we focus solely on loops (at any nesting depth) as candidates for parallel execution—although we expect that many of our techniques for improving value communication would be

applicable to speculative threads constructed from code structures other than loops. Each iteration of the loop corresponds to a single epoch and is assigned to a separate processor during parallel execution. The loops can potentially contain nested loops and/or function calls. With the help of automatically gathered profile information, the compiler selects loops to maximize coverage while meeting heuristics for epoch size and loop trip counts: each loop must comprise at least 0.1% of overall execution time and have an average of at least 1.5 iterations per loop invocation, as well as an average of at least 15 instructions per iteration. Once the key loops are selected, the compiler automatically applies loop unrolling to small loops to help amortize the overheads of speculative parallelization.

4.1.2 Inserting TLS-Specific Instructions. Once speculative regions are chosen, the compiler inserts new TLS-specific instructions that interact with hardware to create and manage epochs. The compiler allocates forwarded variables on the *forwarding frame* (described in Section 2.1), which supports the communication of values between epochs and inserts wait and signal primitives according to the algorithms described in the remainder of this article. The wait and signal primitives combine synchronization with communication, acting as loads from and stores to the forwarding frame.

4.1.3 Generating Object Code. Our compiler outputs C source code encodes our new TLS instructions as in-line MIPS assembly code using gcc’s “asm” statements. This source code is then compiled with gcc 2.95.2 using the “-O3” flag to produce optimized, fully functional MIPS binaries with TLS instructions.

4.2 Underlying Hardware Support

TLS hardware support must implement two important features: buffering speculative modifications from regular memory and detecting and recovering from failed speculation, which we implement using the first-level data caches and an extended version of invalidation-based cache coherence [Steffan et al. 2005]. While we evaluate our compiler support on this specific implementation of TLS, we expect that our conclusions would be similar for other TLS hardware implementations [Akkary and Driscoll 1998; Cintra and Torrellas 2002; Gopal et al. 1998; Gupta and Nim 1998; Hammond et al. 1998; Krishnan and Torrellas 1999; Marcuello and Gonzales 1999; Oplinger et al. 1999; Tsai et al. 1999].

4.3 Experimental Framework

We evaluate our compilation techniques using a detailed machine model which simulates four four-way issue, out-of-order, superscalar processors, each similar to the MIPS R10000 [Yeager 1996], but modernized to have a 128-entry reorder buffer. Each processor has its own physically private data and instruction caches, connected to a unified second-level cache that is shared by all processors by a crossbar switch. Register renaming, the reorder buffer, branch prediction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled and are parameterized, as shown in Table I.

Table I. Simulation Parameters

Pipeline parameters	
Issue width	4
Functional units	2 Int, 2 FP, 1 Mem, 1 Branch
Reorder buffer size	128
Integer multiply	12 cycles
Integer divide	76 cycles
All other integer	1 cycle
FP divide	15 cycles
FP square root	20 cycles
All other FP	2 cycles
Branch prediction	GShare (16KB, 8 history bits)
TLS parameters	
Synchronization latency	10 cycles
Thread creation overhead	10 cycles
Memory parameters	
Cache line size	32 B
Instruction cache	32 KB, 4-way set-associative
Data cache	32 KB, 2-way set-associative, two banks
Unified secondary cache	2 MB, 4-way set-associative, four banks
Miss handlers	16 for data, two for insts
Crossbar interconnect	8 B per cycle per bank
Minimum miss latency to secondary cache	10 cycles
Minimum miss latency to local memory	75 cycles
Main memory bandwidth	1 access per 20 cycles

Table II summarizes the benchmark applications that we evaluate. We have studied all of the SPECint95 and SPECint2000 benchmarks except for the following: 252.EON, which is written in C++ and, therefore, not handled by SUIF; 126.GCC, which is similar to 176.GCC; and 147.VORTEX, which is identical to 255.VORTEX. We have broken 256.BZIP2 into compress and decompress phases and similarly broken 175.VPR into place and route phases. In this article, we do not report results for 129.COMPRESS, 164.GZIP, 300.TWOLF, nor 254.GAP since, for these applications, we found no loops that both comprise an interesting portion of execution and also are speculatively parallelizable by our baseline hardware and compiler support. We have also elided results for the route phase of 175.VPR because of compiler errors. In order for the simulations to terminate within a reasonable amount of time, we simulate up to the first billion instructions using the *ref* inputs after skipping over the initialization phases. Since the sequential and TLS versions of each application are compiled differently, the compiler instruments them to ensure that they terminate at the same point in their executions relative to the source code. As an estimated measure of the coverage of simulating the first billion instructions of the benchmark applications, the last column of Table II shows the number of unique loops that are simulated versus the number of unique loops that are executed in a full run of the benchmark. The results show that our simulations capture 66% of executed loops, on average, and more than 60% of executed loops for 9/13 benchmarks.

Table II. Benchmark Statistics

Suite	Application name	Portion of sequential dynamic execution parallelized (%)	Number of unique parallelized loops	Average epoch size (dynamic insts)	Average number of epochs Per parallelized loop invocation	Number of static loops simulated vs. executed
SpecINT2000	BZIP2_COMP	29	14	272.9	40475.7	88/138
	BZIP2_DECOMP	13	3	186.4	29451.9	44/138
	CRAFTY	9	7	1022.8	7.4	191/311
	GCC	11	58	720.9	79.0	1508/1629
	MCF	97	4	125.5	1977.0	16/35
	PARSER	50	24	341.6	363.9	365/422
	PERLBMK	8	5	967.3	25.8	417/532
	VORTEX	4	5	6543.7	12.3	53/127
SpecINT95	VPR_PLACE	76	8	340.8	153.9	41/113
	GO	17	26	1247.4	33.7	545/660
	LJPEG	84	12	455.8	61.3	49/83
	LI	8	2	5767.6	16.9	54/71
	M88KSIM	55	5	424.1	34.1	220/222

Fig. 10. Impact of scheduling the critical forwarding path. For each benchmark, we show execution time for the speculatively parallelized regions of code normalized to that of the original sequential versions. *U* is unscheduled, *I* only schedules instructions for induction variables, and *B* conservatively schedules all forwarded scalars (referred to as the “baseline” experiment).

5. IMPACT OF COMPILER SCHEDULING TECHNIQUES

We now present our experimental results to quantify the performance impact of our compiler scheduling algorithms. We also include a comparison between our conservative algorithm and the Multiscalar scheduling algorithm [Vijaykumar 1998].

5.1 Impact of Conservative Scheduling

Figure 10 shows the impact of our conservative scheduling algorithm on parallelized region performance. Note that, in most cases, the unscheduled version (*U*) slows down relative to the original sequential version (i.e., the height of the bar is greater than 100). When all forwarded scalars are scheduled (*B*), the performance of every application is significantly improved. For all benchmarks, with the exception of *LI* and *MCF*, time spent waiting for synchronization (*sync*) is significantly reduced and, in some cases, even eliminated. It is interesting

```

for (insn = target; insn; insn = next) {
    rtx this_jump_insn = insn;
    next = NEXT_INSN(insn);
    switch (GET_CODE(insn)) {
    case ...: ...
    case INSN:
        if (GET_CODE(PATTERN(insn)) == USE) {
            ...; continue;
        } else if (GET_CODE(PATTERN(insn)) == CLOBBER) {
            continue;
        } else if (GET_CODE(PATTERN(insn)) == SEQUENCE) {
            for (i = 0; i < XVECLEN(PATTERN(insn), 0); i++) {
                ...
            }
        }
    }
}

```

(a) A loop from GCC (reorg.c:2680).

(b) Performance comparison.

Fig. 11. Comparison with the Multiscalar scheduling algorithm. M approximates the Multiscalar scheduling technique and B is our conservative scheduling algorithm. Execution time in (b) is normalized to that of B .

to note that, in some benchmarks, such as BZIP2_COMP, the reduced critical forwarding path exposes interthread memory dependences that previously were synchronized indirectly, thereby resulting in an increase in failed speculation (*fail*). This implies that reducing the critical forwarding path will likely work even better when combined with techniques for reducing failed speculation. When our scheduling algorithm is applied to loop induction variables alone (I), the synchronization stall time also decreases significantly. However, for five benchmarks, BZIP_COMP, GCC, GO, JPEG, and VPR_PLACE, scheduling instructions for all forwarded scalars gives at least 5% additional performance improvement. Note that, MCF suffers a slight performance degradation. This is caused by the extra instructions created by the instruction scheduling process.

In summary, we observe that code motion, even if it is conservative, is an effective way to reduce the critical forwarding path. While most applications in Figure 10 have enjoyed substantial reductions in synchronization stall times (*sync*), there are still a handful of cases where this bottleneck remains significant. We now investigate whether our more aggressive scheduling algorithms (based on control and data-dependence speculation) can reduce these stall times further.

5.2 Comparison of Conservative Scheduling with the Multiscalar Algorithm

Since the Multiscalar scheduler [Vijaykumar 1998] is essentially a data-flow algorithm that only traverses the CFG once, we can estimate its operation by constraining our conservative scheduling algorithm: we modify the meet operator such that it returns $\perp$ whenever $\top$ meets with any value that is not $\top$ —this way, the modified data-flow analysis will converge during the first iteration.

Figure 11a shows a simplified version of a loop in GCC (at line reorg.c:2680) that highlights the advantage of the more general data-flow approach of our conservative scheduling algorithm over the Multiscalar algorithm [Vijaykumar 1998]. While the original version of this loop has multiple scalars that are

Fig. 12. Impact of aggressive instruction scheduling on region execution time. *B* is conservatively scheduled, *C* has aggressive instruction scheduling past control dependences, *D* has aggressive instruction scheduling past data-dependences, and *E* has aggressive instruction scheduling past both control and data-dependences.

forwarded, we focus on the scalar insn. The Multiscalar scheduler cannot move the update and forward of insn above the inner loop in the case statement, while our approach iterates to a data-flow solution where it can.

Figure 11b shows a performance comparison of our conservative scheduling technique with that of the Multiscalar algorithm (only benchmarks with significant performance differences are shown). Compared with the Multiscalar algorithm, our conservative scheduling approach significantly reduces synchronization time for BZIP_COMP, GCC, GO, PARSER, and PERLBMK, which, in turn, reduces the respective region execution times relative to the Multiscalar approach. Again, this result is not surprising, since the Multiscalar algorithm was designed for smaller, simpler speculative regions.

5.3 Impact of Aggressive Scheduling

Our control and data-dependence speculation algorithms exploit branch taken frequency and data-dependence information gathered from profile runs of each application. For control speculation, we only speculatively schedule instruction across branches with an execution count that is greater than 5% of the epoch count, since speculating across infrequently executed nodes has little performance impact. An edge exiting one of these nodes is considered frequently taken if the number of times this edge is taken is greater than 5% of the epoch count *and* is greater than 5% of the node's execution count. Note that the intuition behind this heuristic is different from that of branch prediction: *both* paths existing a node are considered frequently taken if the node contains a branch that is taken 10% of the times. For data-dependence speculation, we speculatively move computations back across stores and function calls unless there is a more than 5% chance of this resulting in a data-dependence violation (we assume function calls do not conflict with any computation except for another function call). Although experimentation with these threshold values showed that the best values may vary for some applications, we chose to use these fixed values throughout this article.

Figure 12 shows the impact of aggressive instruction scheduling. The first bar (*B*) for each benchmark shows the performance of the conservative scheduling (as seen earlier in Figure 10). The *sync* portion of these bars shows the potential

Fig. 13. Impact of profiling accuracy on speculatively instruction scheduling. *E* shows the performance of speculative instruction scheduling with realistic profile and *e* shows the performance of speculative instruction scheduling with perfect profile. Execution time is normalized to that of *E*.

gain from better scheduling. Compared with conservative instruction scheduling, VORTEX achieves 10.7% parallel loop speedup when speculating on control dependences (“C” bars), and PERLBK and CRAFTY achieve 3.6 and 2.2% parallel loop speedup when speculating on data-dependences (“D” bars). We should also point out that control-dependence and data-dependence speculation are complementary, and by combining the two techniques, we are always able to get the benefits of both optimizations (“E” bars). Furthermore, for some benchmarks, such as GCC and MCF, speculating on neither control nor data-dependence offers any performance improvement, however, speculating on both (“E” bars) is able to provide additional performance gain.

5.3.1 Sensitivity of Speculative Instruction Scheduling to Profiling Accuracy. All results in Figure 12 assumed *realistic* profiling information: i.e., profiling information is collected with the train input set and performance evaluation is done using the ref input set. Can we further improve program performance with more accurate profiling information? The sensitivity of our optimization techniques to the accuracy of profiling information is evaluated with a new set of experiments that perform optimizations using accurate profiling information collected with the ref input set. When perfect profiling information is available, only five benchmarks scheduled instructions differently: although the performance for three of them is improved by 1%, it is degraded by 0.2 and 4.6% for GCC and MCF, respectively. The results of this experiment are shown in Figure 13. Thus, we conclude that speculative instruction scheduling is insensitive to the accuracy of dependence profiling information.

5.4 Summary

We have demonstrated that the compiler is effective at inserting synchronization, but that to obtain decent performance it is mandatory for the compiler to also schedule the resulting critical forwarding paths. We presented a data-flow algorithm for performing this scheduling and showed that it is effective. We also presented and evaluated two more aggressive forms of scheduling and found that speculative instruction schedule can improve the performance further for some applications.

6. HARDWARE TECHNIQUES FOR REDUCING SYNCHRONIZATION

In this section, we describe and evaluate hardware techniques for further improving the communication of scalar values between speculative threads. We first focus on the benefits of predicting values: we describe the issues related to value prediction in the midst of speculation, and describe how to predict forwarded values. We then investigate a technique for using hardware to reduce the critical forwarding path.

6.1 Predicting Forwarded Value

For a frequently occurring cross-epoch dependence that the compiler has decided to synchronize, an attractive alternative is to instead predict the value, eliminating any synchronization stall time. In this section, we investigate techniques for predicting such values for TLS. We begin with a comparison of related work and follow with a description of the issues involved in predicting values for TLS (as compared with uniprocessor value prediction). We then quantify the impact of prediction of forwarded values.

6.1.1 *Related Work.* Value prediction in the context of a uniprocessor is fairly well understood [Gabbay and Mendelson 1996; Lipasti and Shen 1996; Sazeides and Smith 1997; Wang and Franklin 1997], while value prediction for thread-speculative architectures is relatively new. Marcuello et al. [1999a] evaluated the potential for value prediction when speculating at a thread-level on the innermost loops from SPECint95 benchmarks and concluded that predicting synchronized register values provided the greatest benefit. In this section, we demonstrate that predicting forwarded values is indeed beneficial when the synchronized accesses are not scheduled—however, in Section 6.1.3, we show how good compiler scheduling can mostly eliminate the need for hardware prediction of forwarded values. Cintra et al. [Cintra and Torrellas 2002] investigated the impact of value prediction after the compiler has optimized loop-induction variables in floating-point applications, while several other works evaluate the impact of value prediction without such compiler optimization. Oplinger et al. [1999] evaluate the potential benefits to TLS of memory, register, and procedure return value prediction, and Akkary et al. [Akkary and Driscoll 1998] and Rotenberg et al. [1997] also describe designs that include value prediction.

6.1.2 *Predictor Design and Operation.* Predicting values for TLS has similar issues to predicting values in the midst of branch speculation, but at a larger scale. With branch speculation, we do not want to update the predictor for loads on the mispredicted path. Also, when a value is mispredicted, we need only squash a relatively small number of instructions, so the cost of misprediction is not large. Similarly, in TLS, we only want to update the predictor for values predicted in successful epochs, but this will require either a larger amount of buffering or the ability to backup and restore the state of the value predictor. Furthermore, the cost of a misprediction is high for TLS: the entire epoch must be reexecuted if a value is mispredicted, because a prediction cannot be verified until the end of the epoch when all modifications by previous

epochs have been made visible.² Finally, for TLS we require that each epoch has a logically-separate value predictor. For SMT or other shared-pipeline speculation scheme, this does not mean that each requires a physically separate value predictor, but that the prediction entries must be kept separate by incorporating the epoch context identifier into the indexing function. This is necessary since multiple epochs may need to simultaneously predict different versions of the same location.

We model an aggressive hybrid predictor that combines a $1K \times 3$ -entry context predictor with a 1K-entry stride predictor, using 2-bit, up/down, saturating confidence counters to select between the two predictors [Sazeides and Smith 1997]. We use such an aggressive predictor to demonstrate the maximum potential for this form of prediction; finding the smallest and simplest predictor that produces good results is beyond the scope of this work.

Rather than predicting every value that was synchronized by the compiler, instead, we only predict when the prediction confidence is at the maximum value. Any misprediction is not detected until the actual scalar value is forwarded, at which point the predicted value is verified; hence, an epoch must first verify any outstanding predictions before committing and we must squash and reexecute any epoch that has used a mispredicted value.

6.1.3 Impact of Prediction of Forwarded Values. Recall that forwarded values are the communicating scalars identified and synchronized by the compiler, as described in Section 3. In Figure 14a, the *U* experiment shows the performance when the compiler *has not* scheduled the critical forwarding paths (as evidenced by the large synchronization stall, *sync*, and poor performance). In the *P* experiment, we show the performance impact of predicting forwarded values with perfect confidence—i.e., a prediction is only made when the predicted value is correct, giving an upper bound on the performance of aggressive forwarded value prediction. We observe that forwarded values are indeed predictable since synchronization stall (*sync*) and, hence, execution time is reduced considerably for most applications, although four of them still do not speed up. This result is intuitive, since, for unscheduled code, the synchronization of loop index variables will dominate and such variables are very predictable. Note that for several benchmarks we observe an increase in failed speculation: dependences which previously were indirectly synchronized by value forwarding are now exposed because of the increased overlap resulting from successfully predicted forwarded values—we observed a similar trend for compiler scheduling in Section 5.

In the *F* experiment, we model a more realistic confidence mechanism (described above), and see that we achieve close to the performance of perfect confidence (*P*) for all applications except CRAFTY, GCC, and VPR_PLACE. We attempted to remedy this problem by tracking which forwarded loads have caused the pipeline to stall waiting for synchronization and only predicting those values (when confident): however, this approach only maintained the performance of

²Some schemes support selective-squashing of instructions that have used a mispredicted value [Akkary and Driscoll 1998], but this requires a large amount of buffering.

Fig. 14. Performance of prediction of forwarded values. *U* is the unscheduled version, *P* builds on *U* by predicting forwarded values with perfect confidence, and *F* predicts forwarded values with realistic confidence.

the *F* experiment and did not solve the problem for CRAFTY, GCC, and VPR_PLACE. In the case of LI, the idealistic *P* experiment performs worse than the realistic *F* experiment because of an increase in the time spent on failed speculation. This is because when synchronization serializes two threads of execution and it can prevent some data-dependence violations by delaying the consumer instruction. This phenomenon introduces nonlinear effect in the performance of TLS—reducing synchronization leads to performance decrease. This phenomenon is repeatedly observed in our experiments.

In Figure 14b we investigate the performance of forwarded value prediction where the compiler *has* conservatively scheduled the critical forwarding paths: the *B* experiment, shown previous in Figure 10, is scheduled by the compiler, but has no value prediction, the *P* experiment predicts forwarded values with perfect confidence, and the *F* experiment predicts with realistic confidence. Looking at the perfect confidence experiment (*P*), we see that forwarded value prediction is much less effective when the compiler has scheduled the critical forwarding paths. For MCF, the further reduction in synchronization is significant (25.2%), while the reduction is more modest for VORTEX (7.6%). The remaining applications either do not benefit or perform slightly worse; overall this technique improves performance by only 1.5% after the compiler has scheduled the critical forwarding paths.

In summary, hardware support for predicting forwarded values can be beneficial when there is minimal or no compiler support for TLS; however, if the compiler is instead able to schedule the critical forwarding path and, hence,

Fig. 15. Prioritization of the critical forwarding path. The hardware marks the instructions on the input chain of a signal instruction and the pipeline's issue logic gives them high priority.

Fig. 16. Performance impact of prioritizing the critical forwarding path: *U* is unscheduled by the compiler, *Us* builds on *U* by prioritizing the critical forwarding path, *B* is the baseline experiment (scheduled by the compiler), and *Bs* builds on *B* by prioritizing the critical forwarding path.

avoid synchronizing easy-to-predict loop induction variables, then hardware support for forwarded value prediction is, for the most part, unnecessary.

6.2 Prioritizing the Critical Forwarding Path

In Section 6.1.3, we observed that even after aggressive prediction of *forwarded* values, synchronization is still an impediment to good speedup for some benchmarks. For hardware, when it is not possible to eliminate synchronization through prediction, an alternative is to instead prioritize *critical instructions* to help reduce the size and, hence, performance impact of the critical forwarding path. Our compiler already performs this optimization to the best of its ability, but there may be more that can be done dynamically by hardware at runtime.

We begin by modeling an aggressive hardware prioritization scheme, as shown in Figure 15. For any store associated with a signal instruction (as transformed by the previous compiler passes), we estimate a backwards, slice from the store value by marking all instructions with register outputs on the input chain of the signal instruction, and by also tracking the critical forwarding path through memory. Ideally, we would also mark any instructions on the input chain of an unpredictable conditional branch as being on the critical forwarding path, but this is beyond the scope of this work. The pipeline issue logic then gives priority to marked instructions so that the associated signal may be issued as early as possible. This algorithm could be implemented using techniques described by Fields et al. [2001], but, for now, we focus on the potential impact.

The impact of prioritizing the critical forwarding path is shown in Figure 16. Note that we model a 128-entry combined issue-window/reorder-buffer (see

Table III. Prioritization of the Critical Forwarding Path^a

Application	Issued insts that are high priority and issued early (%)	Improvement in avg. start-to-signal time (cycles)		
		Unprioritized	Prioritized	Speedup
BZIP2_COMP	15.2	215.6	416.6	1.07
BZIP2_DECOMP	1.2	1511.3	351.0	0.98
CRAFTY	6.9	70.7	117.4	1.02
GCC	12.2	113.0	218.8	1.08
GO	5.0	95.7	273.0	0.94
IJPEG	6.1	37.2	449.0	0.99
LI	7.3	3206.1	3193.3	1.00
M88KSIM	0.0	20.5	607.6	1.04
MCF	1.2	1707.8	2251.0	1.07
PARSER	2.9	178.5	395.8	1.00
PERLBMK	1.5	1021.2	565.3	1.01
VORTEX	7.2	317.0	287.3	1.10
VPR_PLACE	15.9	138.0	134.1	1.03

^aWe show some statistics, namely, the fraction of issued instructions that are given high priority by our algorithm and issue early, and also the improvement in the average number of cycles from the start of the epoch until each signal.

Table I), so the issue logic has significant opportunity to reorder prioritized instructions. We first compare the impact of prioritization when the compiler has *not* scheduled the critical forwarding path: the *U* experiment is unscheduled, and the *Us* experiment builds on *U* by prioritizing the critical forwarding paths in hardware. We observe that prioritization has a modest performance impact, although it is not enough to result in speedup for benchmarks that were initially slowing down. Comparing the applications for which the critical forwarding paths have been scheduled by the compiler (*B*, as shown previously in Figure 10), the prioritized execution (*Bs*) shows that additional impact of prioritization is negligible.

To clarify the impact of prioritization, Table III shows the fraction of issued instructions that are given high priority by our algorithm and also issue early, which averages 10.0% across all benchmarks. Table III also shows the change in the average number of cycles from the start of an epoch to the issue of each signal, for which the results are mixed: eight benchmarks are improved by at least 5%, while four benchmarks are either unchanged or slowed down slightly. Since hardware support for prioritizing the critical forwarding path can only schedule instructions within the reorder buffer, the speculative threads that we evaluate are too large and, hence, the hardware is unable to move the instructions far enough. In contrast, compiler support is able to analyze the entire program and move instruction across a comparatively long distance. Overall, our conclusion is that this technique does not appear to be worth the expense.

7. COMBINING COMPILER AND HARDWARE TECHNIQUES

In the previous sections, we investigated both compiler and hardware techniques for improving value communication between speculative threads—in this section, we compare the two sets of techniques in greater detail and also

Fig. 17. Performance of all techniques when combined. *U* has no compiler scheduling, *E* implements our most aggressive compiler scheduling (speculative scheduling past both control and data-dependences), the *H* experiment builds on *E* with prediction of forwarded values in hardware, *F* models perfect prediction of forwarded values.

evaluate the impact of combining them. Figure 17 shows the performance impact of combining the various techniques, where the *U* experiment has no compiler scheduling, the *E* experiment implements our most aggressive compiler scheduling (speculative scheduling past both control and data-dependences), the *H* experiment builds on *E* with prediction of forwarded values in hardware, and the *F* experiment models perfect prediction of forwarded values.

Looking at speculatively parallel region performance in Figure 17a, we observe that, on average, aggressive compiler scheduling (*E*) provides a significant performance benefit (31.2%), and even outperforms the perfect prediction experiment (*F*) for some benchmarks, which suffers from dependences that were indirectly synchronized by value forwarding and are now exposed because of the increased overlap resulting from perfectly predicted forwarded values; this effect is most pronounced for *LI*. While hardware prediction of forwarded values (*H*) does reduce the remaining synchronization stall time significantly for one benchmarks (*MCF*), on average, performance is marginally worse as a result of this same effect.

Looking at program performance in Figure 17b, we see that compiler scheduling (*E*) has a significant impact on many applications, while others (*LI*, *GCC*, *CRAFTY*, *PERLBMK*, and *VORTEX*) are limited by low coverage of speculative parallelization (see Table II). For *PARSER*, although 50% of dynamic execution were parallelized, the parallelized regions only show moderate performance gain, thus, the program speedup is also moderate. When considering program performance, hardware prediction of forwarded values (*H*) further improves performance slightly because of a significant impact on *MCF*—an 22.8%

improvement over compiler scheduling alone (E). Both cases are very close to the perfect prediction of forwarded values experiment (F), indicating that our techniques for reducing synchronization are effective.

8. CONCLUSIONS

We have shown that reducing synchronization for TLS can yield large performance improvements and proposed several techniques for doing so. Our analysis provides several important lessons. First, we found that the critical forwarding path is an important bottleneck to overcome when trying to extract parallelism from many important programs using TLS. In this article, we have proposed and evaluated a range of scheduling algorithms that the compiler can use to reduce the impact of the critical forwarding path. By applying conservative scheduling to all synchronized variables, we observed that the compiler can be effective in reducing the performance impact of the critical forwarding path without requiring any additional hardware support beyond what is normally needed for TLS. Second, to further reduce the critical forwarding path for the handful of applications where synchronization stalls were still a concern, we proposed and evaluated scheduling techniques based on speculative code motion that require some additional hardware support to preserve correctness. We found that scheduling speculatively past control and data-dependences offered a modest additional performance benefit. Third, we found that hardware prioritization to reduce the critical forwarding path does not have a significant performance impact, even though a good number of instructions can be reordered. Finally, we found that predicting forwarded values in hardware can be effective when the compiler has not scheduled the critical forwarding paths, but that otherwise this hardware support is not well motivated. Our overall conclusion is that reducing the impact of synchronization on TLS is a challenge that is best addressed by the compiler.

ACKNOWLEDGMENTS

This research has been supported by grants from the National Science Foundation, NASA, IBM and Intel.

REFERENCES

- AKKARY, H. AND DRISCOLL, M. 1998. A Dynamic Multithreading Processor. In *MICRO-31*.
- AMD CORPORATION. 2005. Leading the industry: Multi-core technology & dual-core processors from AMD. <http://multicore.amd.com/en/Technology/>.
- AMMONS, G. AND LARUS, J. R. 1998. Improving data-flow analysis with path profiling. In *ACM SIGPLAN 98 Conference on Programming Language Design and Implementation (PLDI98)*.
- BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAK, Y., POTTINGER, B., RAUCHWERGER, L., AND TU, P. 1996. Parallel programming with polaris. *IEEE Computer* 29, 12, 78–82.
- CHANG, P. P., WARTER, N. J., MAHLKE, S. A., CHEN, W. Y., AND HWU, W. W. 1991. Three superblock scheduling models for superscalar and superpipelined processors. Tech. Rept. CRHC-91-29, Center for Reliable and High-Performance Computing, University of Illinois.
- CINTRA, M. AND TORRELLAS, J. 2002. Learning cross-thread violations in speculative parallelization for multiprocessors. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)*.

- CINTRA, M., MARTÍNEZ, J. F., AND TORRELLAS, J. 2000. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of ISCA 27*.
- COLOHAN, C. B., AILAMAKI, A., STEFFAN, J. G., AND MOWRY, T. C. 2005. Optimistic intra-transaction parallelism on chip multiprocessors. In *31st International Conference on Very Large Data Bases (VLDB 2005)*.
- COLOHAN, C. B., AILAMAKI, A., STEFFAN, J. G., AND MOWRY, T. C. 2006. Hardware support for large speculative threads. In *33rd Annual International Symposium on Computer Architecture (ISCA '06)*.
- CYTRON, R. 1986. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*.
- DUBEY, P., O'BRIEN, K., O'BRIEN, K., AND BARTON, C. 1995. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1995)*.
- FIELDS, B. A., RUBIN, S., AND BODIK, R. 2001. Focusing processor policies via critical-path prediction. In *28th Annual International Symposium on Computer Architecture (ISCA '01)*.
- FISHER, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.* 13.
- FRANKLIN, M. AND SOHI, G. S. 1992. The expandable split window paradigm for exploiting fine-grain parallelism. In *19th Annual International Symposium on Computer Architecture (ISCA '92)*. 58–67.
- GABBAY, F. AND MENDELSON, A. 1996. Speculative execution based on value prediction. Tech. Rept. EE Department TR #1080, Technion–Israel Institute of Technology.
- GALLAGHER, D. M., CHEN, W. Y., MAHLKE, S. A., GYLLENHAAL, J. C., AND HWU, W. W. 1994. Dynamic memory disambiguation using the memory conflict buffer. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*. 183–195.
- GOPAL, S., VIJAYKUMAR, T., SMITH, J., AND SOHI, G. 1998. Speculative versioning cache. In *4th International Symposium on High-Performance Computer Architecture (HPCA-4)*.
- GUPTA, M. AND NIM, R. 1998. Techniques for speculative run-time parallelization of loops. In *Supercomputing '98*.
- HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of ASPLOS-VIII*.
- HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. 1993. Preliminary experiences with the Fortran D compiler. In *Supercomputing '93*.
- IBM CORPORATION. 2007. IBM unleashes world's fastest chip in powerful new computer. <http://www-03.ibm.com/press/us/en/pressrelease/21580.wss>.
- INTEL CORPORATION. 2005. Intel's Dual-Core Processor for Desktop PCs. http://www.intel.com/personal/desktopcomputer/dual_core/index.htm.
- KNIGHT, T. 1986. An architecture for mostly functional languages. In *Proceedings of the ACM Lisp and Functional Programming Conference*. 500–519.
- KNOOP, J., RUTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *ACM SIGPLAN 92 Conference on Programming Language Design and Implementation (PLDI'92)*.
- KRISHNAN, V. AND TORRELLAS, J. 1999. The need for fast communication in hardware-based speculative chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.
- LIPASTI, M. H. AND SHEN, J. P. 1996. Exceeding the data-flow limit via value prediction. In *International Symposium on Microarchitecture*.
- MARCUELLO, P. AND GONZALES, A. 1999. Clustered speculative multithreaded processors. In *Proceedings of the ACM International Conference on Supercomputing*.
- MARCUELLO, P., TUBELLA, J., AND GONZALEZ, A. 1999a. Value prediction for speculative multithreaded architectures. In *International Symposium on Microarchitecture*.
- MARCUELLO, P., TUBELLA, J., AND GONZALEZ, A. 1999b. Value prediction for speculative multithreaded architectures. In *Micro 32*, Haifa, Israel.
- MOSHOVOS, A. I., BREACH, S. E., VIJAYKUMAR, T., AND SOHI, G. S. 1997. Dynamic speculation and synchronization of data-dependences. In *24th Annual International Symposium on Computer Architecture (ISCA '97)*.

- NICOLAU, A. 1989. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Trans. Comput.* 38, 663–678.
- OPLINGER, J., HEINE, D., AND LAM, M. S. 1999. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*.
- PADUA, D. A., KUCK, D. J., AND LAWRIE, D. H. 1980. High-speed multiprocessors and compilation techniques. *IEEE Trans. Computers* 29, 9, 763–776.
- PRABHU, M. AND OLUKOTUN, K. 2003. Using thread-level speculation to simplify manual parallelization. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming*.
- PRABHU, M. AND OLUKOTUN, K. 2005. Exposing speculative thread parallelism in spec2000. In *ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming*.
- ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. 1997. Trace processors. In *Proceedings of Micro 30*.
- SAZEIDES, Y. AND SMITH, J. E. 1997. The predictability of data values. *Proceedings of Micro 13*, 248–258.
- SOHI, G. S., BREACH, S., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture (ISCA '95)*. 414–425.
- STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *27th Annual International Symposium on Computer Architecture (ISCA '00)*.
- STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2002. Improving value communication for thread-level speculation. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)*.
- STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2005. The stampede approach to thread-level speculation. *ACM Transactions on Computer System (TOCS)*, 23, 3, 253–300.
- SUN CORPORATION. 2005. Throughput computing—Niagara. <http://www.sun.com/processors/throughput/>.
- TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D., AND YEW, P.-C. 1999. The superthreaded processor architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures* 48, 9 (Sept.).
- VIJAYKUMAR, T. 1998. Compiling for the multiscalar architecture. Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison.
- WANG, K. AND FRANKLIN, M. 1997. Highly accurate data value prediction using hybrid predictors. In *30th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-30)*. Research Triangle Park, NC.
- WILSON, R. P., FRENCH, R. S., WILSON, C. S., AMARASINGHE, S. P., ANDERSON, J.-A. M., TJANG, S. W. K., LIAO, S.-W., TSENG, C.-W., HALL, M. W., LAM, M. S., AND HENNESSY, J. L. 1994. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices* 29, 12, 31–37.
- YEAGER, K. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro* 16, 2 (Apr.), 28 – 40.
- ZHAI, A. 2005. Compiler optimization of value communication for thread-level speculation. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.
- ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2002. Compiler optimizations to accelerate scalar value communication between speculative threads. Tech. Rept. CMU-CS-02-162, School of Computer Science, Carnegie Mellon University. August.
- ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2004. Compiler optimization of memory-resident value communication between speculative threads. In *The 2004 International Symposium on Code Generation and Optimization*, Palo Alto, CA.
- ZILLES, C. 2002. Master/slave speculative parallelization and approximate code. Ph.D. thesis, University of Wisconsin—Madison.
- ZILLES, C. AND SOHI, G. 2002. Master/slave speculative parallelization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-35)*.

Received March 24, 2006; revised November 14, 2006 and June 4, 2007; accepted November 30, 2007