Compiler Optimization of Scalar Value Communication Between Speculative Threads

Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{*zhaia,colohan,steffan,tcm*}*@cs.cmu.edu*

ABSTRACT

While there have been many recent proposals for hardware that supports *Thread-Level Speculation* (TLS), there has been relatively little work on compiler optimizations to fully exploit this potential for parallelizing programs optimistically. In this paper, we focus on one important limitation of program performance under TLS, which is stalls due to forwarding scalar values between threads that would otherwise cause frequent data dependences. 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. In addition, we contrast our compiler techniques with related hardware-only approaches. With our most aggressive compiler and hardware techniques, we improve performance under TLS by 6.2–28.5% for 6 of 14 applications, and by at least 2.7% for half of the other applications.

1. INTRODUCTION

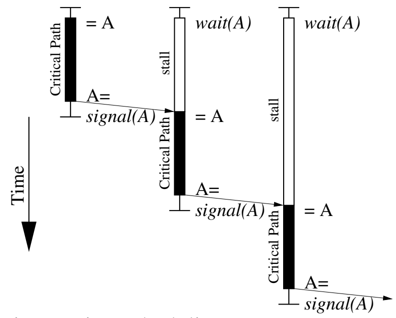
Multithreading within a chip is becoming increasingly commonplace: examples include the IBM Power4 [17], Sun MAJC [33], Alpha 21464 [9], HP PA-8800, and Sibyte BCM-1250 [4]. While using this multithreaded hardware to improve the *throughput* of a workload is straightforward, using it to improve the performance of a *single application* requires parallelization. The ideal solution would be to convert sequential programs into parallel programs automatically, but unfortunately this is difficult (if not impossible) for many *general-purpose* programs due to their use of pointers, complex data and control structures, and run-time inputs.

Thread-Level Speculation (**TLS**) [1, 7, 13, 14, 15, 19, 20, 25, 30, 34] is a potential solution to this problem since it allows the compiler to create parallel threads without having to prove that they are independent. The underlying hardware ensures that inter-thread dependences through memory are satisfied, and re-executes any thread for which they are not.

The key to extracting parallelism from these programs and hence improving performance is in the efficiency of speculative execution. While recent research has investigated hardware optimization for TLS [7, 19, 21, 31, 23], there has been relatively little work on compiler optimization in this area. One potential opportunity

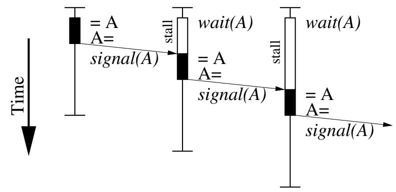
ASPLOS X 10/02 San Jose, CA, USA


```
do {
   work1();
   wait(A);
   if (condition(A)) {
      A = A + *p;
   } else {
      B = 2;
      work2();
   }
   A = A + 1;
   signal(A);
   work3();
} while (1);
```



(a) Before instruction scheduling.

```
do {
   work1();
   wait(A);
   if (condition(A)) {
      A = A + *p;
      A = A + 1;
      signal(A);
   } else {
      A = A + 1;
      signal(A);
      B = 2;
      work2();
   }
   work3();
} while (1);
```



(b) After instruction scheduling.

Figure 1: Impact of scheduling on the critical forwarding path.

for optimization focuses on data dependences between speculative threads that occur frequently: if the compiler is able to identify the source and the destination of a frequent inter-thread data dependence, then it is beneficial to insert synchronization and forward that value explicitly to avoid failed speculation. Figure 1(a) shows an example loop that the compiler has speculatively parallelized by partitioning the loop into speculative threads (aka *epochs*). Since the variable `A` is read and written in every iteration, the compiler decides to synchronize and forward `A` by inserting a `wait` operation before the first *use* of `A`, and a `signal` operation after the last *definition* of `A`—we describe, implement, and evaluate this algorithm in Section 3. The synchronization results in the partially-parallel execution shown in Figure 1(a), where each epoch stalls until the value of `A` is produced by the previous epoch. The flow of the value of `A` between epochs serializes the parallel execution, and so we refer to it as a *critical forwarding path*. In the next section, we show that the overall performance of speculation is limited by the size of this critical forwarding path.

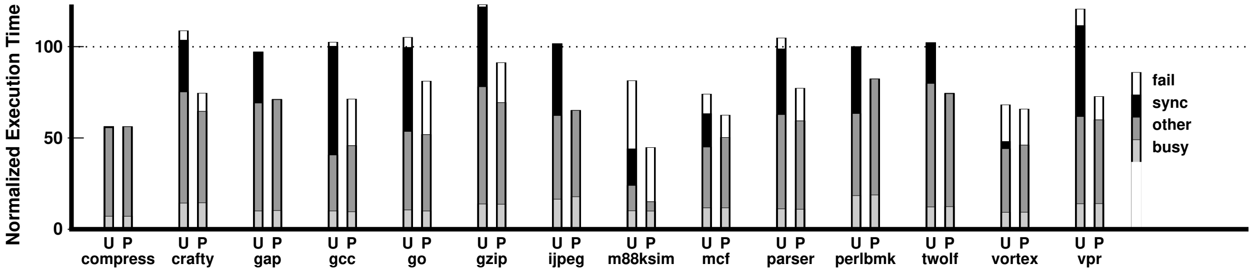


Figure 2: Potential impact of reducing the critical forwarding path. For each benchmark we show execution time on four processors for the speculatively parallelized regions of code normalized to that of the original sequential versions. *U* is the unscheduled speculative version and *P* shows the impact of perfect prediction of forwarded values.

1.1 The Importance of 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. Figure 2 shows the potential impact of reducing the critical forwarding path on a four-processor chip multiprocessor—we will explain the details of this experiment later in Section 2. The *U* bars show the unscheduled TLS version of each application run speculatively in parallel. Each bar is normalized to the execution time of the original sequential version, such that bars less than 100 are speeding up. The best we can possibly do with scheduling is to eliminate the critical forwarding path altogether. We measure this ideal behavior with a model that can perfectly predict all forwarded values such that there is no synchronization (*P*). We see that for most applications, removing the synchronization bottleneck results in great performance improvements.

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 conditional control flow. Figure 1(b) 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 will be performed before reaching each `signal` (by deferring the computation of `B=2` and `work2()`). As shown in the figure, this reduces the stall time for each epoch, thereby improving overall execution time. We describe an algorithm for reducing the critical path in Section 4.1.

1.2 More Aggressive Instruction Scheduling

All of the transformations that we have described so far will preserve the control and data dependences within each epoch: the transformed code will perform the same operations as the original, but possibly reordered within each control structure and between ambiguous data dependences. However, it is potentially beneficial to move code past control and data dependences [5, 10, 12, 24] 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 1(b), we could schedule "`A=A+1;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 intra-epoch control speculation and data dependence speculation in Section 4.2.

1.3 Related Work

Parallelization of a loop where the compiler synchronizes a loop-carried data dependence is known as a *DOACROSS* [8, 26] parallelization and has been exploited in previous work [6, 22, 38]. All schemes for TLS support include some form of DOACROSS synchronization, although few use the compiler to optimize this aspect of speculative execution.

The most relevant related work is the Wisconsin Multiscalar [11, 27, 35] compiler, which performs synchronization and scheduling for register values [35]. (The Multiscalar effort also evaluated hardware support for automatically detecting and synchronizing data dependences [23].) 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 (aka *epochs*) are much larger on average than Multiscalar tasks and contain complex control flow. This inspired the dataflow-based scheduler presented in this paper, 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 is extremely difficult, we schedule producer instructions as early as possible. Another difference is that our more general approach to scheduling handles loop induction 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 4.2). We modified our scheduler to mimic the Multiscalar scheduler, and we contrast the performance impact of both approaches later in Section 5.3.

Other schemes for TLS hardware support provide the means to synchronize and forward values between speculative threads but do not use the compiler to optimize loop-induction variables or synchronize frequent dependences [1, 14, 15, 20], while others provide such support but do not schedule instructions to reduce the critical forwarding path [7, 34]. Research on TLS hardware support has shown the importance of the critical forwarding path and how the prediction of forwarded values may be used to increase parallelism [21, 31]; it also showed that hardware is ineffective at improving performance by scheduling the critical forwarding path [31]. Other hardware techniques for improving the efficiency of speculation include prediction of loads to memory, dynamic synchronization, and squashing of *silent stores* (which overwrite memory with the same value that is already there) [1, 7, 20, 31].

Concurrent with our work, Zilles and Sohi [39] recently 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 paper could potentially be applied to the distilled code in the master thread.

Our algorithm for reducing the critical forwarding path builds upon previous dataflow approaches to code motion, namely *partial redundancy elimination* [18], path-sensitive dataflow analysis [16], and *hot paths* [2]. Previous work on speculative code motion to exploit a frequently executed path includes trace scheduling [10] and superblock scheduling [5]. There has also been work on aggressive load/store reordering where the runtime check and recovery are performed entirely in software [24] or through a hybrid hardware/software approach [12].

1.4 Contributions

In the context of thread-level speculation, this work makes the following contributions. First, we show that the critical forwarding path is a significant performance bottleneck for many applications. Second, we present novel dataflow 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. Finally, we compare and contrast our compiler scheduling techniques with hardware techniques for optimizing the critical forwarding path [31]; comparison with hardware techniques for automatically synchronizing dependences [23] is beyond the scope of this paper.

2. INFRASTRUCTURE FOR TLS

In this section we describe our compiler infrastructure and target hardware support for TLS, as well as our simulation infrastructure and experimental framework.

2.1 Compiler Infrastructure

Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [32]. In addition to scheduling the critical forwarding path, our compiler also performs the tasks described below when automatically transforming a program to exploit TLS.

Deciding Where to Speculate: For this paper, we focus solely on loops (at any nesting depth) as candidates for parallel execution (i.e. speculative *regions*), where loop iterations are the units of parallelism. Based on profile information, we first discarded loops which did not iterate, had a program coverage of less than 0.1%, executed fewer than 30 instructions per loop invocation, or had more than 16384 instructions per iteration, since these were unlikely to produce significant interesting parallelism. We then parallelized each of the remaining loops and profiled them using a simple in-order single instruction per cycle simulator. During this profiling we ignored the effects of all synchronization, thereby measuring an optimistic upper bound on performance—this assumes that the techniques in this paper will remove any negative impact due to

Table 1: 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)

Memory Parameters	
Cache Line Size	32B
Instruction Cache	32KB, 4-way set-assoc
Data Cache	32KB, 2-way set-assoc, 2 banks
Unified Secondary Cache	2MB, 4-way set-assoc, 4 banks
Miss Handlers	16 for data, 2 for insts
Crossbar Interconnect	8B 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

synchronization. The loops that performed best under these ideal conditions were then selected for use in this study. We found that some of the loops chosen based on our simple simulator performed poorly when run on a realistic simulator, since the simple simulator takes neither communication latency nor the cost of thread creation into account. We also show results for a subset of the loops that perform well—the details of the selection process are discussed below in Section 2.2.

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 a special portion of the stack called the *forwarding frame* which supports the communication of values between epochs, and inserts `wait` and `signal` primitives according to the algorithms described in the remainder of this paper. The `wait` and `signal` primitives combine synchronization with communication, acting as loads from and stores to the forwarding frame.

Generating Object Code: Our compiler outputs `C` source code which 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.

2.2 Region Selection

Since some of the regions selected using the process above performed poorly on our realistic simulator, we created a second set of regions that elides poor performers so that the performance of those regions with potential is not overwhelmed. Throughout this paper we therefore evaluate performance on two sets of speculative regions for each application: (i) a set selected by the criteria given above ("*all regions*"), and (ii) a subset where we have pruned the regions for which speculative parallelization is ineffective despite the most aggressive compiler and hardware optimizations ("*good regions*"). Our results show that most of the pruned regions were not dominated by synchronization overhead, and hence they could not benefit from the techniques described in this paper.

The GCC benchmark stands out as an exception: we pruned a few loops (with a combined program coverage of 4.4%) which were dominated by synchronization yet were not improved by our techniques. Some of the corresponding critical forwarding paths

Table 2: Benchmark statistics.

Suite	Application Name	All Regions: Portion of Dynamic Execution Parallelized	All Regions: Number of Unique Parallelized Regions	All Regions: Average Epoch Size (dynamic insts)	All Regions: Average Number of Epochs Per Dynamic Region Instance	Good Regions: Portion of Dynamic Execution Parallelized	Good Regions: Number of Unique Parallelized Regions
SpecINT2000	186.CRAFTY	28.7%	24	184.3	7.9	16.9%	8
	254.GAP	10.9%	1	94.0	4.8	10.9%	1
	176.GCC	43.2%	104	627.3	160.3	24.1%	57
	164.GZIP	7.9%	2	40.5	27229.8	2.5%	1
	181.MCF	49.9%	11	56.7	306.8	39.5%	3
	197.PARSER	58.1%	36	579.2	23644.5	10.8%	17
	253.PERLBMK	29.9%	10	363.9	405.0	16.3%	2
	300.TWOLF	26.1%	12	212.3	4.3	11.1%	3
	255.VORTEX	14.0%	6	4775.6	4.1	4.3%	4
	175.VPR	78.5%	5	236.9	5.8	65.3%	2
SpecINT95	129.COMPRESS	38.5%	1	125.0	863.0	38.5%	1
	099.GO	86.2%	73	1370.6	31.8	17.4%	24
	132.IJPEG	95.1%	21	245.4	95.8	60.7%	13
	124.M88KSIM	59.5%	6	790.1	60.3	54.6%	5

contained inner loops with a significant amount of computation, while others contained lengthy function calls—hence our scheduling techniques were unable to reorder the instructions effectively.

2.3 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 [29, 30]. 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 [1, 7, 13, 14, 15, 19, 20, 25, 30, 34].

2.4 Experimental Framework

We evaluate our compilation techniques using a detailed machine model which simulates 4-way issue, out-of-order, superscalar processors similar to the MIPS R10000 [36], 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 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 1.

After skipping over the initialization phases, we simulate up to the first billion instructions of the applications described in Table 2. (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.) We simulate all of the SPECint95 and SPECint2000 benchmarks [28] 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; 147.VORTEX, which is identical to 255.VORTEX; and 256.BZIP2, 130.LI, and 134.PERL, the three of which have no loops that both comprise an interesting portion of execution and also are speculatively parallelizable by our current techniques.

3. INSERTING SYNCHRONIZATION

This section presents a general algorithm for inserting synchronization to communicate values between epochs. In this paper, we focus on the set of local (i.e. defined in the scope of the enclosing procedure) *communicating scalars*, which we define as the subset of local scalar variables that meet one of the following two criteria. First, any scalar in the intersection of the set of scalars with a downwards-exposed definition and an upwards-exposed use (i.e., the scalar is *live* between epochs). Second, any scalar that is live when the loop exits. In contrast with local scalars, global scalar and pointer references may be modified by calls to other procedures; hence synchronizing them is beyond the scope of this paper.

For each synchronized scalar, the compiler inserts instructions that perform the synchronization and communicate the value. This is performed by the `wait` and `signal` instructions, each of which is associated with the scalar in question through an architected register. The `wait` instruction stalls execution until the value is produced by the previous epoch, which communicates that value through the `signal` instruction. For the first epoch of a speculatively-parallelized loop, the `wait` instruction does not stall (since there is no producer). In remainder of this section, we describe where to place the synchronization instructions to ensure correct execution.

3.1 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 path;
2. a `signal` must occur after the last definition of the scalar on any path.

In reality, if a signal was omitted on a given path then the waiting epoch could continue executing once all previous epochs were complete (rather than stalling indefinitely)—however, we ignore this to allow the placement algorithms to be symmetric. Hence, we have the additional constraint:

3. a `signal` must occur for each synchronized scalar on every possible path.

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.

3.2 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` downwards through the code. When a branch is encountered, the `wait` can be duplicated and pushed down on either side 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 dataflow 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 dataflow 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 technical report [37].)

We define our dataflow analysis over the set of communicating scalars V on the control flow graph $G = (N, E, s, e)$ 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 node* and *end node* of G (note that the start node 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 make our analysis difficult, we break any such edges into two edges using *synthetic nodes* [18].

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 node n ($NMD(n)$) to be the set of scalars that are not defined on any execution path from n to e. This function can be computed using dataflow analysis on the following equation:

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

For the example in Figure 3, the shaded boxes in Figure 3(c) indicate where $NMD(n)$ is true for the variable `a`. Note that there are two definitions of `a`, hence $\mathtt{a} \in NMD(n)$ for all nodes n dominated by these two nodes.

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 `signal`s. We avoid redundant `signal`s through the function $signal(n)$ which determines the placement of `signal` instructions:

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

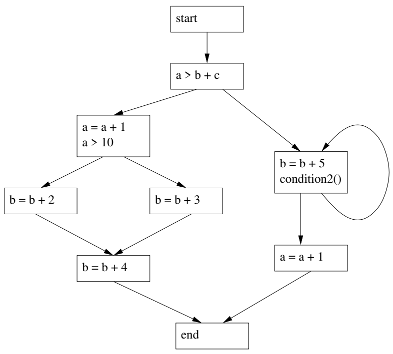
Figure 3(c) shows the synchronization points for variables `a` and `b` for the original code in Figure 3(a).

3.3 Performance Evaluation

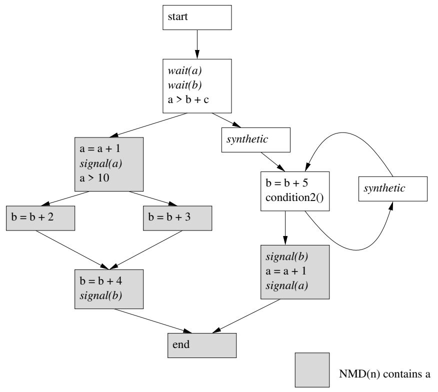
The *U* bars in Figure 2 show the performance of the applications when synchronization is placed using the techniques described in this section, using the good set of regions. Each bar is normalized to the execution time of the original sequential version, such that bars less than 100 are speeding up. The *P* bar shows the impact of perfectly predicting all forwarded values—this shows an upper bound on the gains we can realize by optimizing synchronization.

```
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.

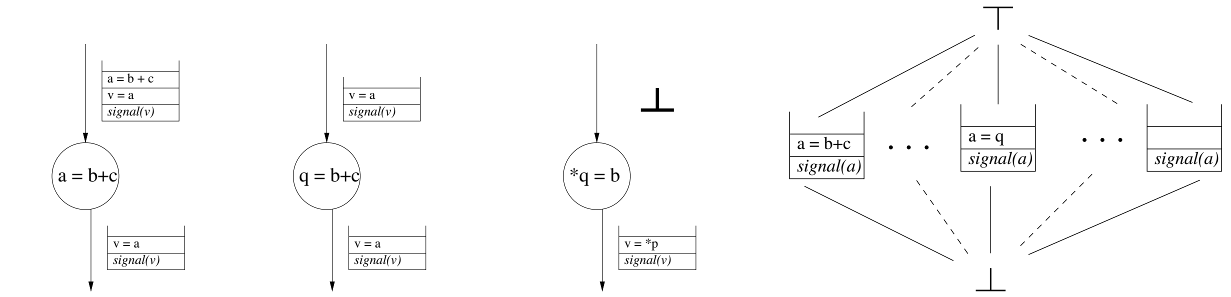


(c) Inserting waits and signals.

Figure 3: Example of how waits and signals are inserted.

Each bar in Figure 2 is broken down into four segments explaining what happened during all potential *graduation slots*. The number of graduation slots is the product of: (i) the issue width (4 in this case), (ii) the number of cycles, and (iii) the number of processors (4 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 a forwarded location; and the *other* segment is all other slots where instructions cannot graduate.

As we see in Figure 2, synchronization is a significant bottleneck for most applications; hence placing the `signal` instructions as early as possible and the `wait` instructions as late as possible is not good enough—the critical forwarding path is still too large.



(a) Dependence exists. (b) No dependence. (c) Unresolved dependence. (d) Lattice.

Figure 4: Illustration of the $transfer$ function (parts (a), (b), and (c)) used for computing the value of $stack$ in equation (3), and the lattice (part (d)) over which the $stack$ dataflow analysis is defined.

There is much potential for improvement—as indicated by the *P* bars—by reducing the critical forwarding path. It is interesting to note that in half of the applications, eliminating the synchronization bottleneck *increases* the amount of *failed speculation*, since the resulting increase in parallel overlap can lead to new occurrences of data dependence violations.

In many cases, the bulk of the computation in the critical forwarding path is not directly related to the forwarded scalar in question. If we can relocate this unrelated computation, the resulting smaller critical forwarding path will lead to improved performance. In the remainder of this paper, we investigate the use of code motion to reduce the critical forwarding path.

4. INSTRUCTION SCHEDULING

The compiler can improve the performance of speculatively parallelized code by using scheduling techniques to move the `signal` operations (and the code that these operations depend upon) upwards through the control flow graph to reduce the length of the critical forwarding path and expose more parallelism. For example, closer examination of Figure 3 reveals that the forwarded value for variable `a` depends only on the result of a single addition. While the forwarding path between the `wait` and the `signal` shown in Figure 3 contains many instructions, only the following three instructions are really necessary to wait for, compute, and forward the new value of `a`:

```
wait(a);
a = a + 1;
signal(a);
```

In the next section, we describe a scheduling algorithm that achieves this minimum critical forwarding path for this example.

4.1 Conservative Scheduling Algorithm

Similar to our algorithm for inserting synchronization, we also define our conservative scheduling algorithm over the set of communicating scalars V on the control flow graph $G = (N, E, s, e)$, with critical edges broken by synthetic nodes as described in the previous section. We initialize the algorithm by placing all `signals` at the exit node e. (It is equivalent to start all `signals` in the position indicated by our placement algorithm, but placing them at the end node simplifies the proof of correctness significantly.) Note that in our implementation of this algorithm, we have chosen only to move `signal` instructions (and the instructions they depend upon) upwards in the control flow graph; although the converse of this algorithm can be applied to moving `wait` instructions (and the instructions that depend upon them) downwards in the control flow graph, our experiments with moving waits downward showed little 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 technical report [37].)

As we schedule the instructions, we must identify at each node the computation that the eventual `signal` depends upon. Since we cannot represent these computations as binary values, bit-vector 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 computations necessary to produce the value of a communicating scalar v if it is to be sent from the corresponding node.

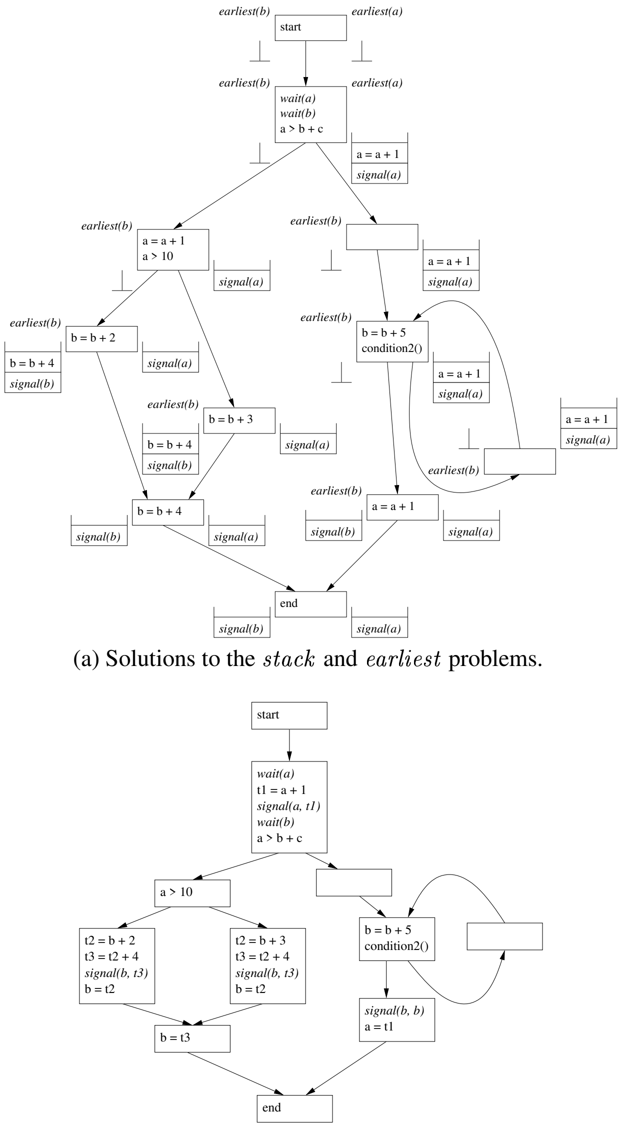
The domain of the $stack$ dataflow problem is the set of all possible configurations of the computation stack. This domain, along with the meet operator (described later), defines a semi-lattice. All nodes are initialized to $\top$. If a given node is found to be a safe place for the `signal` instruction then $stack$ returns a non-empty stack of computations, otherwise $stack$ returns $\perp$. The following dataflow equation computes $stack(n, v)$ at the exit of each node:

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

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 4(a).
- 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 4(b).
- 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 4(c).
- 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 over the set of all possible configurations of the computation stack. The lattice shown in Figure 4(d) defines the following operations for



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

(b) After code transformation.

Figure 5: Example of our conservative scheduling algorithm applied to the code in Figure 3.

meet: 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 semi-lattice of height three, thus our dataflow problem is well-defined.

We also define the dataflow 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:

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

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

Code Transformation: For each node that is both $safe$ and $earliest$ for a variable 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 5(a) illustrates solutions for $stack$ and $earliest$ for the example shown earlier in Figure 3. $Earliest$ is true for variable `a` only at the top node. The stack for the variable `a` at the top node contains only the two instructions required to compute `a`—this matches the optimal result we derived manually at the beginning of this section. Figure 5(b) shows the transformed program. 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 code segment size is increased by less than 1.3% for all benchmarks.

4.2 Aggressive Scheduling Algorithms

In the scheduling algorithm that we just described, the backward motion of `signal` operations (and the instructions on which they depend) can be obstructed for the following two reasons:

Control Dependences: If incompatible computation stacks from multiple execution paths meet at a single node during the backwards dataflow analysis, then code motion stops. This implies that our conservative scheduling algorithm cannot move instructions out of the `then` or `else` parts of an `if-then-else` statement unless those same instructions are executed along both conditional paths.

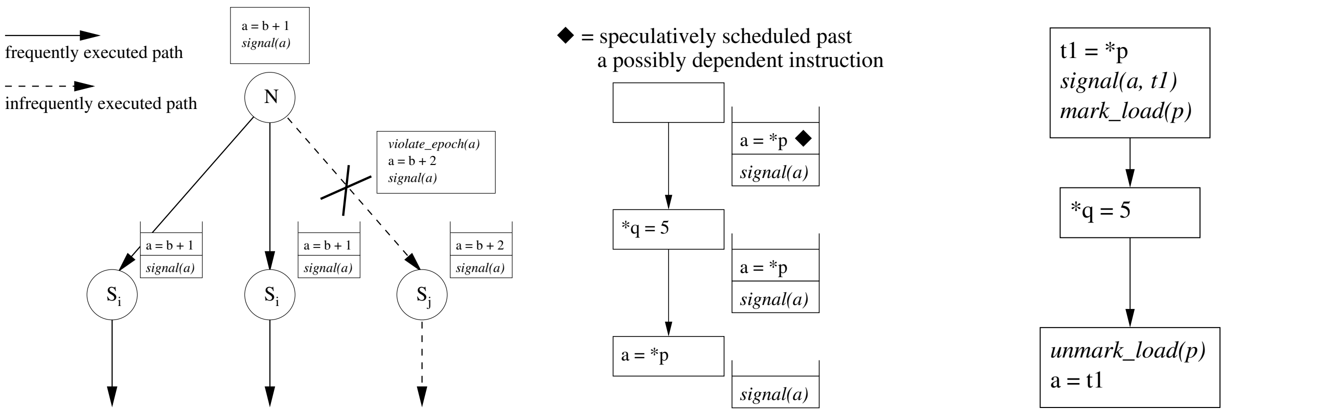
Data Dependences: A computation stack cannot be moved past a store instruction whose target address may alias locations referenced in the computation stack. This scenario often arises with writes through pointers or other forms of indirection.

Hence instructions are only scheduled at program points where the intra-epoch control and data dependences mentioned above have been resolved. Since both of these cases occur frequently in our programs, we would like to make scheduling more aggressive. In this section we will discuss both the compiler techniques and the hardware support necessary to allow for instruction scheduling beyond intra-epoch control and data dependences.

4.2.1 Scheduling Past Control Dependences

Dataflow 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 run-time. 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. Ball and Larus [3] proposed efficient methods to record execution paths that are taken by the program at run time, which allows us to identify the frequently executed paths.

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



(a) Modified meet operator for control speculation.

(b) Modified transfer function for data speculation.

(c) Code generation for data speculation.

Figure 6: Modified dataflow analysis for speculative instruction scheduling.

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. When the epoch reaches a `wait` instruction, it will stall until the correct value is received. 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 non-speculatively scheduled copy of the code should then be executed to ensure that the exception was real.

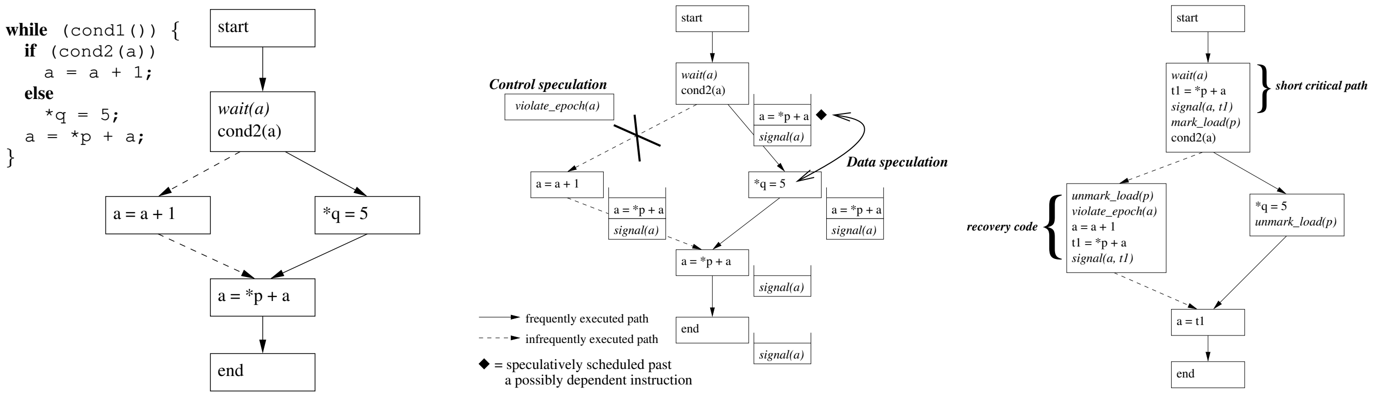
We have modified the scheduling algorithm from Section 4.1 to speculate on control dependences. We make the algorithm more aggressive by modifying the meet operator $\prod$ used in the $stack$ dataflow analysis in equation (3), and add new nodes on infrequent edges which contain the recovery code. First, all possible execution paths through an epoch are enumerated, and a profiling run reports the number of times the epoch is executed as well as the number of times each execution path is taken. An execution path is considered *frequently executed* if the probability of taking this execution path when the epoch is executed is greater than a certain threshold.

The meet operator $\prod$ for $stack$ is modified as shown in Figure 6(a). When evaluating the meet operator $\prod$ at node n for the scalar v, we first operate on the set of successors, where the edge (n, s_i) is on a frequently executed path. Then for each node s_j, where (n, s_j) is not on any frequently executed path, we verify whether $transfer(s_j, v, stack(s_j, v))$ is compatible with the partially evaluated $stack(n, v)$. 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)): it 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 6(a) 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 re-executed with the correct value.

4.2.2 *Scheduling Past Data Dependences*

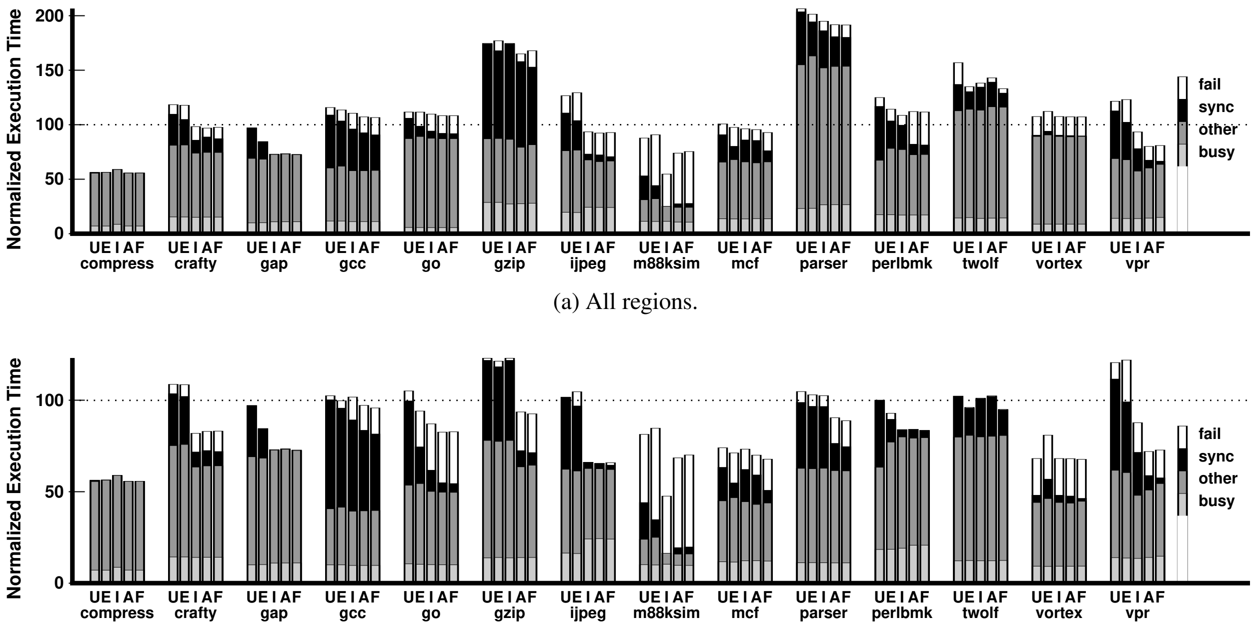
We now consider how our conservative scheduling algorithm can be extended to allow code motion beyond potential data dependences. Using the output from an automatic data dependence profiling tool, our compiler can reason about the likelihood of data dependence problems at run-time if the code associated 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 run-time, 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—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 [12] 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 [37].)

To implement scheduling across potential data dependences, we modify the *transfer* function described earlier in Section 4.1 (and used in equation (3)) as shown in Figure 6(b). 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 $\prod$, any *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.



(a) Original code. (b) Control flow graph. (c) Transformed code.

Figure 7: Illustration of how speculation on control and data dependences can be complementary.



(a) All regions.

(b) Good regions.

Figure 8: 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, *E* builds on *U* with hardware optimization of forwarded values, *I* schedules only loop induction variables, *A* schedules all forwarded variables, and *F* builds on *A* with hardware optimization of forwarded values.

4.2.3 Complementary Effects

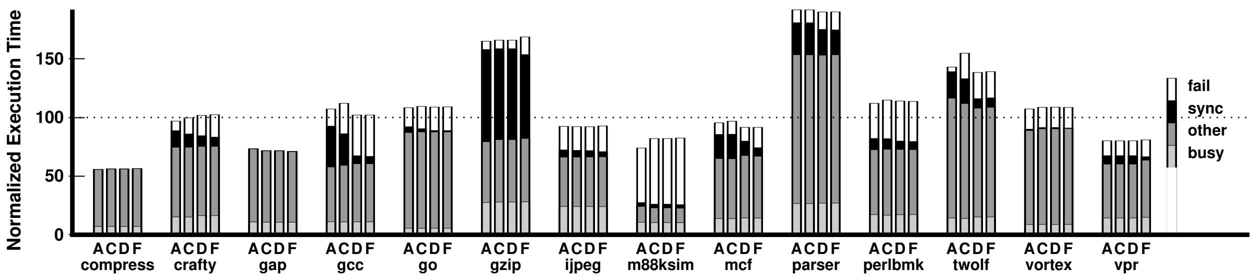
Control and data dependence speculation can be complementary. Figure 7 shows an example where the combination of a control hazard and a data hazard prevent the code from being scheduled early, and where speculation on either type of hazard alone will not yield any benefit. By speculating on both control and data dependences in tandem, the computation of variable `a` can be moved upwards next to the `wait` operation, thereby resulting in a much shorter critical forwarding path for the common case.

5. EXPERIMENTAL RESULTS

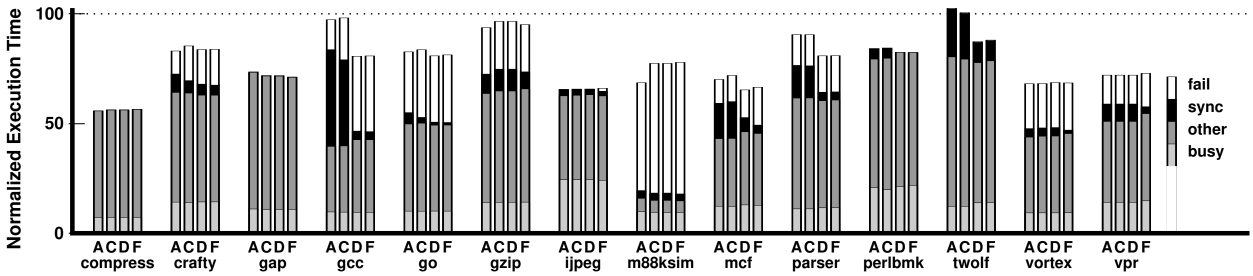
We now present our experimental results to quantify the performance impact of the scheduling algorithms described in the previous section. We include a comparison with hardware-based techniques that are also designed to reduce synchronization stalls under TLS [31], as well as a comparison between our conservative algorithm and the Multiscalar scheduling algorithm [35].

5.1 Impact of Conservative Scheduling

Figure 8 shows the impact of our conservative scheduling algorithm on parallelized region performance. (Recall the definitions of *"all regions"* and *"good regions"* presented earlier in Section 2.2.) 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).



(a) All regions.



(b) Good regions.

Figure 9: Impact of aggressive instruction scheduling and hardware optimization on region execution time. *A* is conservatively scheduled, *C* has aggressive instruction scheduling past control dependences, *D* has aggressive instruction scheduling past both control and data dependences, and *F* builds on D with hardware optimization of forwarded values.

We first evaluate the performance of a hardware technique (*E*) that we described in an earlier publication [31] to schedule the critical forwarding path and predict forwarded values for the sake of eliminating synchronization stalls. We observe significant improvements for GAP, PERLBMK, and TWOLF, as well as for GO when poor regions are pruned (i.e. *good regions* in Figure 8(b)).

When our scheduling algorithm is applied to loop induction variables alone (*I*), the synchronization stall times decrease significantly (more than 50% for 9 of the 14 applications), and many applications now enjoy significant speedups. On average, *all regions* are improved by 11.8% and the *good regions* by 13.6%. Only COMPRESS performs slightly worse under this optimization. We note that reduction of synchronization stall time does not always translate directly into improved performance. For example, synchronization time is greatly reduced for GO when loop induction variables are scheduled (*I*), but the resulting increased parallel overlap exposes more data dependences across threads, and failed speculation becomes a new bottleneck that offsets much of the potential performance gain.

When all forwarded variables are scheduled (*A*), we see in Figure 8 that many applications enjoy additional improvement, while the performance of three applications is degraded. The worst degradation occurs in M88KSIM for the same reason mentioned above: the reduced critical forwarding path exposes inter-thread memory dependences that previously were synchronized indirectly, thereby resulting in a large increase in failed speculation for this particular case. This example illustrates the complex interactions that can occur among different potential bottlenecks under TLS, and suggests that there is still room for improvement in future research by attacking these other bottlenecks.

We now consider the effectiveness of the compiler versus the hardware at optimizing the critical forwarding path. Our first observation from Figure 8 is that our conservative scheduling algorithms (*I* and *A*) outperform the hardware-only technique (*E*) in nearly every case. To evaluate whether the compiler and the hardware are complementary, we supplemented the compiler's efforts with hardware support (*F*) for optimizing forwarded values (as was done for (*E*)). These hardware mechanisms offer a slight additional benefit for only a few cases, suggesting that hardware mechanisms for optimizing the critical forwarding path are largely unnecessary given proper compiler support.

In summary, we observe that code motion, even if it is conservative, is an effective way to reduce the critical forwarding path, and that the compiler appears to be better suited to this optimization than hardware. While most applications in Figure 8 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.

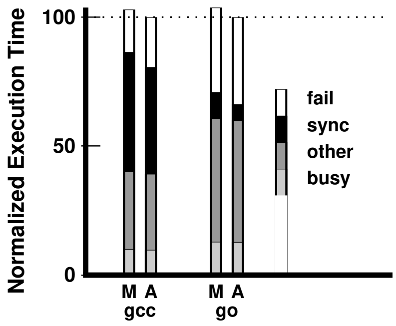
5.2 Impact of Aggressive Scheduling

Our control and data dependence speculation algorithms exploit path frequency information and data dependence information gathered from a profile of each application. For control speculation, we consider any path through an epoch that was executed at least 5% of the time to be "*frequently executed*" (as discussed earlier in Section 4.2.1). For data dependence speculation, we speculatively move code back across stores or function calls unless there is more than a 15% chance of this resulting in a data dependence violation. Although experimentation with these threshold values showed that the best values vary between applications, we chose to use these fixed values throughout this paper.

Figure 9 shows the impact of aggressive instruction scheduling and hardware optimization on region execution time. The first bar (*A*) for each application shows the performance of conservative

```
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) Simplified loop from GCC (`reorg.c`:2680).



(b) Performance comparison.

Figure 10: Comparison with the Multiscalar scheduling algorithm. *M* approximates the Multiscalar scheduling technique, and *A* is our conservative scheduling algorithm. Execution time of *good regions* in part (b) is normalized to *A*.

scheduling (as seen earlier in Figure 8); the *sync* portion of these bars shows the potential gain from better scheduling, with GCC, MCF, PARSER and TWOLF being the most significant. The second bar (*C*) shows the performance of speculatively scheduling past control dependences alone. We observe that performance improves or degrades slightly for several applications, except for M88KSIM and TWOLF (*all regions*) for which the degradation is more severe: these contain cases where the code has not moved a significant distance, and where the benefits of the code motion are not sufficient to overcome the costs of misspeculation. Our scheduling algorithm could make more informed tradeoffs if it took this code motion distance into account, rather than always moving code whenever possible (as it does now).

For the *good regions* cases in Figure 8(b), speculatively scheduling past both control and data dependences (*D*) decreases synchronization time by an average of 19.8%. Speculative scheduling results in a 0.3% average performance degradation for the *all regions* cases, and a 5.8% average performance improvement for the *good regions* of applications for which more than 6% of execution time is spent on synchronization—most notably, GCC, PARSER, and TWOLF, which speed up by 17.1%, 19.1%, and 14.8% respectively. These observations suggest that the overheads of speculative scheduling are too large to apply this technique liberally, but that it can be quite effective for certain regions of code (where synchronization stalls remained a problem after conservative scheduling) if applied selectively.

Finally, we again supplement the compiler's efforts with hardware support (*F*) that schedules the critical forwarding path and predicts forwarded values to eliminate synchronization [31]. We observe that improvements from this additional hardware support are negligible, and that such hardware support is not necessary with sufficiently aggressive compiler optimization.

5.3 Comparison of Conservative Scheduling with the Multiscalar Algorithm

Since the Multiscalar scheduler [35] is essentially a dataflow algorithm that only traverses the control flow diagram 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 dataflow analysis will converge during the first iteration.

Figure 10(a) shows a simplified version of a loop in GCC (at line `reorg.c`:2680) that highlights the advantage of the more general dataflow approach of our conservative scheduling algorithm over the Multiscalar algorithm [35]. While the original version of this loop has multiple variables that are forwarded, we focus on the variable `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 dataflow solution where it can.

Figure 10(b) shows a performance comparison of our conservative scheduling technique with that of the Multiscalar algorithm for the *good regions* cases of the two applications where there was a significant difference in performance: i.e. GCC and GO. Compared with the Multiscalar algorithm, our conservative scheduling approach reduces synchronization time by 10% for GCC and by 39% for GO, which in turn reduces the respective region execution times by 3.0% and 3.7% relative to the Multiscalar approach. Again, this result is not surprising since the Multiscalar algorithm was designed for smaller, simpler regions.

5.4 Impact on Overall Program Performance

The goal of TLS and our techniques for improving its efficiency is to exploit chip multiprocessors and other multithreaded machines to improve the performance of *programs*. Up to this point, we have evaluated the performance of our optimizations on the *regions* of each program that have been speculatively parallelized. Figure 11 shows the impact on program execution time of TLS with compiler and hardware optimization when only the *good regions* are speculatively parallelized. The first bar (*U*) shows the unscheduled version, the second (*A*) shows conservative scheduling. We observe that IJPEG and VPR enjoy a tremendous benefit from conservative scheduling while CRAFTY, GO, and M88KSIM show more modest improvements. The third bar (*D*) shows aggressive scheduling past control and data dependences, which shows a significant improvement for GCC but degrades performance for several other applications, indicating that we must be more selective when applying this technique. The fourth bar (*H*) shows the additional impact of hardware optimization of both forwarded values and memory values [1, 7, 20, 31], which improves GCC, COMPRESS, IJPEG, and M88KSIM significantly. For the remaining applications, our hardware techniques cannot significantly improve upon the the performance of the compiler. (Note that our hardware technique for automatically synchronizing data dependences [31] is not the most aggressive approach possible [23].) With our most aggressive compiler and hard-

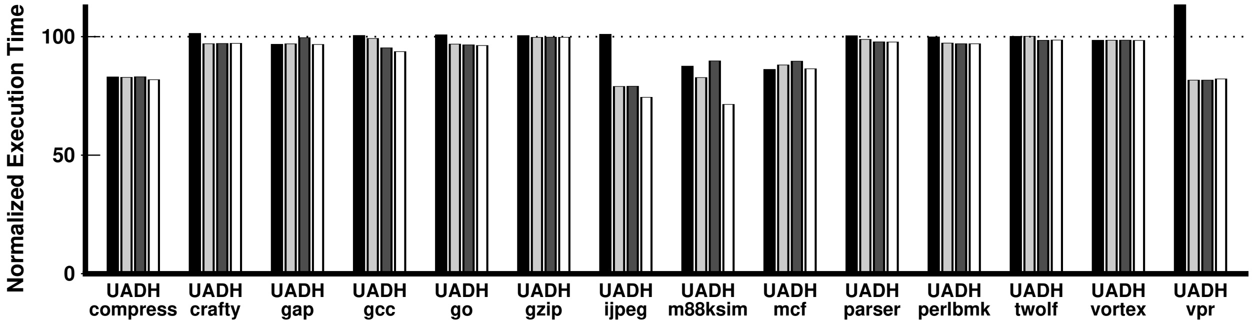


Figure 11: Impact on program execution time of aggressive instruction scheduling and hardware optimization when only the good regions are speculatively parallelized. *U* is unscheduled, *A* is conservatively scheduled, *D* has aggressive instruction scheduling past both control and data dependences, *H* builds on *D* with hardware optimization of memory and forwarded values.

ware techniques, we have improved program performance of 6 of 14 applications by 6.2–28.5%, and we improved half of the others by 2.7–3.6%.

6. CONCLUSIONS

The critical forwarding path is an important bottleneck to overcome when trying to extract parallelism from many important programs using TLS. In this paper, we have proposed and evaluated a range of scheduling algorithms that the compiler can use to reduce the impact of the critical forwarding path. Loop induction variables were the largest performance bottleneck, and our conservative scheduling technique effectively eliminates their impact on performance. By applying conservative scheduling to all synchronized variables, we saw additional performance gains. These results demonstrate 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.

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 upon speculative code motion that require some additional hardware support to preserve correctness. We found that scheduling speculatively past control dependences alone offered little performance benefit. However, scheduling speculatively past both control and data dependences resulted in substantial performance gains for a number of applications. In particular, GCC—which is the most challenging application among the set that we considered—was the largest benefactor from this speculative scheduling. GCC also highlighted the performance advantages of our robust dataflow-based approach to scheduling compared with the previous state-of-the-art technique.

The bottom line from this study is that the critical forwarding path bottleneck for TLS is best addressed by the compiler rather than through elaborate hardware mechanisms. If hardware resources are to be devoted to this problem, they are best spent on implementing the instructions necessary to support speculative scheduling of `signal` operations (and the instructions they depend upon) past both control and data dependences.

7. ACKNOWLEDGMENTS

This research is supported by grants from IBM, Intel, and NASA. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship.

8. REFERENCES

[1] AKKARY, H., AND DRISCOLL, M. A Dynamic Multithreading Processor. In *MICRO-31* (December 1998).

[2] AMMONS, G., AND LARUS, J. R. Improving data-flow analysis with path profiling. In *Proc. ACM SIGPLAN 98 Conference on Programming Language Design and Implementation* (1998).

[3] BALL, T., AND LARUS, J. R. Efficient path profiling. In *Proceedings of Micro-29* (1996).

[4] BROADCOM CORPORATION. The Sibyte SB-1250 Processor. http://www.sibyte.com/mercurian.

[5] CHANG, P. P., WARTER, N. J., MAHLKE, S. A., CHEN, W. Y., AND HWU, W. W. *Three Superblock Scheduling Models for Superscalar and Superpipelined Processors*. Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, 1991.

[6] CHEN, D. K., AND YEW, P. C. Statement re-ordering for DOACROSS loops. In *International Conference on Parallel Processing* (Aug. 1994), pp. 24–28.

[7] CINTRA, M., MARTÍNEZ, J. F., AND TORRELLAS, J. Learning Cross-Thread Violations in Speculative Parallelization for Scalar Multiprocessors. In *Proceedings of the 8th HPCA* (February 2002).

[8] CYTRON, R. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing* (1986).

[9] EMER, J. Ev8: The post-ultimate alpha.(keynote address). In *International Conference on Parallel Architectures and Compilation Techniques* (2001).

[10] FISHER, J. A. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers 13* (June 1981).

[11] FRANKLIN, M. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin – Madison, 1993.

[12] GALLAGHER, D. M., CHEN, W. Y., MAHLKE, S. A., GYLLENHAAL, J. C., AND HWU, W. W. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In *Proceedings of the 6th ASPLOS* (October 1994), pp. 183–195.

[13] GOPAL, S., VIJAYKUMAR, T., SMITH, J., AND SOHI, G. Speculative Versioning Cache. In *Proceedings of the 4th HPCA* (February 1998).

[14] GUPTA, M., AND NIM, R. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98*

(November 1998).

[15] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of ASPLOS-VIII* (October 1998).

[16] HOLLEY, L. H., AND K. ROSEN, B. Qualified data flow problems. *IEEE Transactions on Software Engineering 7*, 1 (Jan. 1981).

[17] KAHLE, J. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum '99* (October 1999).

[18] KNOOP, J., AND RUTHING, O. Lazy code motion. In *Proc. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation* (92).

[19] KRISHNAN, V., AND TORRELLAS, J. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *Proceedings of PACT '99* (October 1999).

[20] MARCUELLO, P., AND GONZLEZ, A. Clustered Speculative Multithreaded Processors. In *Proc. of the ACM Int. Conf. on Supercomputing* (June 1999).

[21] MARCUELLO, P., TUBELLA, J., AND GONZSSLEZ, A. Value prediction for speculative multithreaded architectures. In *Proceedings of Micro-32* (Haifa, Israel, Nov. 1999).

[22] MIDKIFF, S. P., AND PADUA, D. A. Compiler algorithms for synchronization. *IEEE Transactions on Computers C-36*, 12 (1987), 1485–1495.

[23] MOSHOVOS, A. I., BREACH, S. E., VIJAYKUMAR, T., AND SOHI, G. S. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th ISCA* (June 1997).

[24] NICOLAU, A. Run-time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers 38* (May 1989), 663–678.

[25] OPLINGER, J., HEINE, D., AND LAM, M. S. In Search of Speculative Thread-Level Parallelism. In *Proceedings of PACT '99* (October 1999).

[26] PADUA, D., KUCK, D., AND LAWRIE, D. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computing* (September 1980).

[27] SOHI, G. S., BREACH, S., AND VIJAYKUMAR, T. Multiscalar processors. In *Proceedings of the 22nd ISCA* (June 1995).

[28] STANDARD PERFORMANCE EVALUATION CORPORATION. The SPEC Benchmark Suite. http://www.specbench.org.

[29] STEFFAN, J. G., COLOHAN, C. B., AND MOWRY, T. C. Architectural Support for Thread-Level Data Speculation. Tech. Rep. CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[30] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th ISCA* (June 2000).

[31] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. Improving Value Communication For Thread-Level Speculation. In *Proceedings of the 8th HPCA* (February 2002).

[32] TJIANG, S., WOLF, M., LAM, M., PIEPER, K., AND HENNESSY, J. *Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Germany, 1992, pp. 137–151.

[33] TREMBLAY, M. MAJC: Microprocessor Architecture for Java Computing. *HotChips '99* (August 1999).

[34] TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D., AND YEW, P.-C. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures 48*, 9 (September 1999).

[35] VIJAYKUMAR, T. *Compiling for the Multiscalar Architecture*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Jan. 1998.

[36] YEAGER, K. The MIPS R10000 superscalar microprocessor. *IEEE Micro* (April 1996).

[37] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler Optimizations to Accelerate Scalar Value Communication Between Speculative Threads. Tech. Rep. CMU-CS-02-162, School of Computer Science, Carnegie Mellon University, August 2002.

[38] ZHU, C.-Q., AND YEW, P.-C. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering 13*, 6 (June 1987), 726–739.

[39] ZILLES, C. B., AND SOHI, G. S. Master/Slave Speculative Parallelization with Distilled Programs. Tech. Rep. TR-1438, Computer Sciences Department, University of Wisconsin-Madison, April 2002.