Compiler Optimization of Memory-Resident 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

†*Department of Electrical & Computer Engineering*
University of Toronto
Toronto, Ontario M5S 3G4

ABSTRACT

Efficient inter-thread value communication is essential for improving performance in *Thread-Level Speculation* (TLS). Although several mechanisms for improving value communication using hardware support have been proposed, there is relatively little work on exploiting the potential of compiler optimization. Building on recent research on compiler optimization of scalar value communication between speculative threads, we propose compiler techniques for the optimization of memory-resident values.

In TLS, data dependences through memory-resident values are tracked by the underlying hardware and preserved by re-executing any speculative thread that violates a dependence; however, re-execution incurs a large performance penalty and should be used only to resolve data dependences that are infrequent. In contrast, value communication for frequently-occurring data dependences must be very efficient.

In this paper, we propose using the compiler to first identify frequently-occurring memory-resident data dependences, then insert synchronization for communicating values to preserve these dependences. We find that by synchronizing frequently-occurring data dependences we can significantly improve the efficiency of parallel execution. A comparison between compiler-inserted and hardware-inserted memory synchronization reveals that the two techniques are complementary, with each technique benefitting different benchmarks.

1.. INTRODUCTION

Hardware support for *Thread-Level Speculation* (TLS) proposed in previous work [1, 8, 10, 12, 13, 15, 19, 24, 28] empowers the compiler to parallelize general-purpose programs despite their use of pointers, runtime inputs, complex data structures and control flow. Under TLS, the compiler partitions the program into parallel speculative threads (*a.k.a. epochs*) without having to prove that they are independent, allowing instructions to be fetched and executed long before their data and control dependences are resolved. The underlying hardware checks whether inter-epoch dependences are satisfied and re-executes any epoch for which they are not. Thus, we are able to parallelize programs that were previously non-parallelizable.

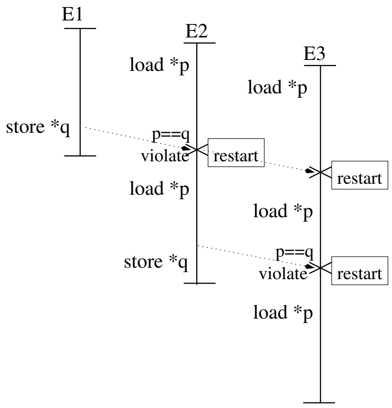
Since speculation failure incurs a high cost it should only be invoked occasionally. We must seek alternative methods to deal with frequently occurring data dependences. One way to avoid speculation failures caused by data dependence violations is to synchronize frequently-occurring data dependences. Figure 1 shows a loop example that the compiler has speculatively parallelized by turning each loop iteration into an epoch. In each epoch a value is loaded through the pointer `p` and another value is stored through the pointer `q`. When `p` in a later epoch points to the same memory location as `q` in an earlier epoch, there is a *read-after-write* dependence. Figure 1(b) and 1(c) show two methods to communicate a value between the two epochs to satisfy this dependence. The first method is *speculation*: the consumer epoch executes assuming there is no data dependence with previous threads and is re-executed if the hardware detects a dependence violation. The second method is *synchronization*: the consumer epoch stalls and waits for the producer epoch to produce and forward the correct value. Synchronization serializes parallel execution and only allows partial overlap between parallel epochs, but is more efficient than speculation when data dependences occur frequently since restarts are avoided.

The existence of aliases between memory accesses makes it more difficult to synchronize accesses to memory-resident values than accesses to scalar values. Previous work on compiler optimization for inter-epoch value communication [32] focuses on communicating register-resident *scalar* values. It shows that: (i) compiler-inserted synchronization and forwarding can communicate scalar values efficiently between epochs; and (ii) instruction scheduling techniques are essential for reducing the *critical forwarding path* created by such synchronization. However, these techniques cannot be directly applied to communicate memory-resident values since the compiler is unable to identify the producer and the consumer of a data dependence statically. Figure 3(a) shows three epochs running speculatively in parallel. *Load *p* can potentially depend on any of the five stores in the figure, although each access to the memory uses a different pointer. The compiler must prove that *load *p* depends on *store *q* in all possible executions before synchronizing the two instructions and directly forwarding a value between them. Such a proof is difficult and sometimes impossible to construct. If the compiler decides to synchronize *store *q* and *load *p* without such a proof, we must confirm at runtime that (i) p and q refer to the same memory location, and that (ii) stores through pointers y and z do not modify this location.

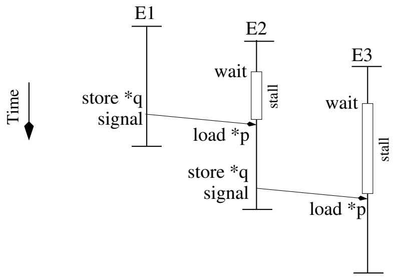
Previously, a number of studies [8, 18, 25] propose using hardware implementations to dynamically insert synchronization for frequently occurring and unpredictable data dependences in TLS. Moshovos *et. al.* [18] demonstrated how to identify frequently occurring data dependences with a centralized structure. However, a centralized structure can limit performance [11] and is difficult to scale. On the other hand, the distributed version of this scheme is complex since it involves replicating the tables which predict/synchronize load-store pairs and keeping them coherent via broadcast. In a distributed environment, it is relatively easy for the hardware to dynamically identify loads that frequently cause speculation to fail using hardware lookup tables, but more involved for the hardware to identify the corresponding stores. For the hardware to dynamically identify an inter-epoch dependence pair it has to (i) compare the addresses accessed by loads and stores in dif-

```
do {
   work1();
    ... = *p;
   work2();
   *q = ... ;
   work3();
} while (1);
```

(a) Code with communication through memory-resident values.



(b) Communicating through speculation: if p and q alias between epochs then the later epochs are violated and restarted with the correct value.



(c) Communicating through synchronization: the later epochs always stall until the value is available.

Figure 1: Performance trade-off of using speculation versus synchronization under TLS.

ferent epochs, and (ii) dynamically determine whether a store is the last store that modifies an address in an epoch. To avoid this complexity, recent proposals for hardware-inserted synchronization [8, 25] choose to delay the load instructions until previous epochs have committed. However, such simplification tends to over-synchronize parallel execution, trading time spent on failed speculation for time spent on synchronization. In contrast, compilers have the advantage of knowing the entire program, thus can determine which stores are more likely to produce the desired value and, therefore, only stall the consumer until the value is produced (rather than waiting for the entire producer epoch to complete). Compilers can also schedule instructions to produce the forwarded value early to reduce synchronization time. Furthermore, compiler-inserted synchronization avoids hardware complexity by eliminating lookup tables used by hardware proposals to identify frequently occurring loads.

1.1 Our Approach: Compiler-Inserted Synchronization for Memory-Resident Values

In this paper, we propose to use the compiler to insert explicit synchronization to communicate values more efficiently for inter-epoch data dependences that occur frequently. In our approach, we first identify frequently occurring data dependences using profiling information, then insert *signal* and *wait* instruction pairs, the same synchronization primitive as sued for synchronizing communicating scalars [32], to create point-to-point synchronization and to forward the values involved in the dependences. We also describe the hardware support required to verify that the synchronized load and store are indeed dependent at runtime and to guarantee recovery from incorrect execution if they are not. Details of this hardware support are in Section 2.2.

The compiler decides where to insert synchronization based on the output of a software-only instrumentation-based tool. In our experiment this tool records all accesses to the memory and matches all dependent load and store instructions. Pointer analysis [17, 29], especially probabilistic, inter-procedural and context-sensitive pointer analysis [3, 5, 14] could help us obtain this information with less detailed profiling information. Data dependence profiling and compiler insertion of synchronization are described in more detail in Section 2.3.

1.2 Performance Impact of Failed Speculation

To estimate the performance potential of compiler-inserted synchronization for memory-resident values, we study TLS execution with optimal memory-resident value communication. Figure 2 shows the potential impact of reducing failed speculations in the parallelized regions of a program on a four-processor chip multiprocessor that supports TLS (detailed in Section 3). Each bar in Figure 2 is broken down into four segments explaining what happens 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 scalar values (scalar values are communicated using explicit synchronization); and the *other* segment is all other slots where instructions cannot graduate. The **U** bars represent the execution time of the benchmark when run in parallel using TLS. Each bar is normalized to the execution time of the original sequential version, and hence bars less than 100 are speeding up. The best we can possibly do to reduce speculation failure is to prevent any data dependence speculation from failing. We measure this ideal behavior by running the same benchmarks with a hypothetical model that perfectly forwards the values needed by all load instructions such that no failed speculation nor synchronization stall ever occur due to accesses to the memory (**O** bars). We find that for most benchmarks, eliminating failed speculation results in a substantial performance gain.

1.3 Related Work

Previous work on synchronizing loop-carried data dependences for DOACROSS loops [2, 4, 9, 20, 30] only focuses on array-based numeric codes. Our technique applies to arbitrary control flow and memory access patterns in general-purpose programs, and is able to (i) forward data for dependences that may or may not occur, and (ii) ensure correct execution if subsequent stores invalidate the data that have already been forwarded. Prior to our work, Sura *et. al.* [26] used the compiler to insert memory fence instructions to map the consistency model at the programming language level to the consistency model offered by the hardware. Correct execution must be ensured through this mapping, hence, their compiler analyses are conservative. In our case, correctness is ensured by the

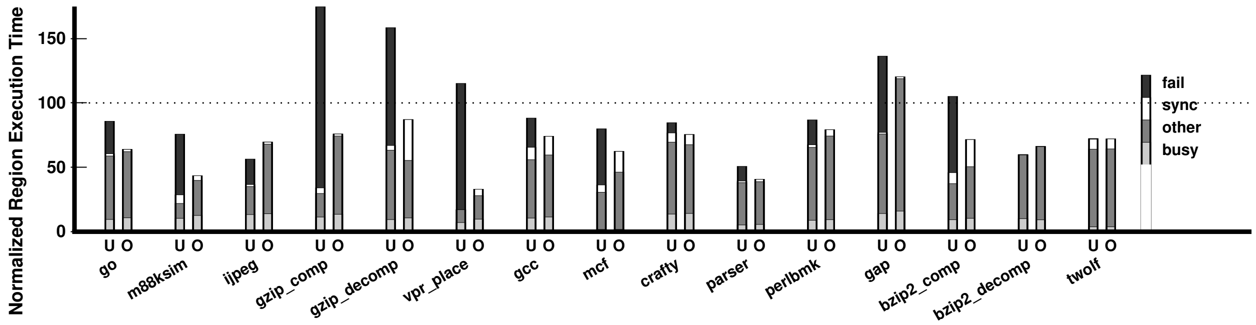


Figure 2: Potential impact of reducing failed speculation. For each benchmark we show execution time on four processors (for the speculatively parallelized regions of code) normalized to that of the original sequential version. U is unoptimized, without synchronization of memory resident values; O shows the impact of perfect (oracle) prediction of loads of memory resident values.

underlying hardware and we synchronize frequently occurring data dependences strictly to improve performance, hence, the compiler analyses used in this work to insert synchronization can be more aggressive.

To avoid excessive failed speculation when using TLS, two types of hardware mechanisms: *value prediction* [8, 16, 18, 19, 21, 25] and *synchronization* [8, 18, 25] have been proposed. Value prediction allows the consumer of a potential data dependence to use a predicted value, avoiding a dependence violation if the prediction is correct. Hardware support for automatic synchronization identifies store-load dependences that frequently cause violations and attempts to synchronize them dynamically. The various implementations of these two hardware mechanisms are discussed below.

Dynamic synchronization of memory accesses can benefit both uniprocessors and multiprocessors. In superscalars, loads are usually issued as early as possible, but no earlier than prior stores that write to the same memory address to avoid memory-order violations. Chrysos and Emer [7] present a design that uses a prediction table for synchronizing dependent store-load pairs in an out-of-order issue uniprocessor. Moshovos *et. al.* [18] demonstrate how to implement a hardware-based synchronization mechanism in the context of a Multiscalar processor (a thread-speculative chip-multiprocessor) using centralized lookup tables to match dependent load/store pairs from different processing units.

A major drawback of previous proposals is the need for centralized lookup tables which can limit performance and are difficult to scale. Two groups [8, 25] propose alternative implementations to manage synchronization information in a distributed manner. Cintra and Torrellas [8] propose building a distributed hardware lookup table to keep track of frequently occurring violations. They divide data dependences into three categories and handle them accordingly. For violations caused by false dependences, they optimistically allow the consumer to proceed and use the per-word access bits in its cache hierarchy to check for correctness before committing. In the case of a true dependence where the value is predictable, the consumer uses a predicted value and later verifies the value before committing. In the case of a true dependence with an unpredictable value, violations are avoided by stalling the consumer until the producer has committed. Their evaluation shows that these optimizations can substantially improve value communication for floating point benchmarks.

In prior work we evaluated the use of value prediction to communicate predictable values and the use dynamically inserted synchronization to communicate unpredictable values [25]. We found that value prediction and dynamic synchronization can incur a significant cost and should only be applied to those dependences that limit performance. Loads that frequently cause violations are delayed until the producer epoch commits rather than until the desired value is produced, due to the difficulty in identifying dependent store-load *pairs*. Thus, this dynamically inserted synchronization tends to serialize parallel execution more than necessary.

In another prior work we explored the compiler's ability to improve scalar value communication, and showed that compilers can communicate scalar values efficiently between epochs [32]. By targeting scalar values, we have been able to use traditional data-flow analysis to find all reads/writes to the same data item and identify the producer and the consumer of a data dependence. We conclude that the key to efficiently communicating scalar values between epochs is to reduce the *critical forwarding path* created by synchronization, a task effectively accomplished by the compiler through instruction scheduling. Although this paper focuses more on reducing the cost of violations instead of reducing the impact of the synchronization we insert to avoid violations, we attempt to evaluate the significance of reducing the cost synchronization for communicating memory-resident values in Section 4 through idealized experiments.

1.4 Contributions

In the context of thread-level speculation, this paper makes the following three contributions. First, this is the first attempt to explore a compiler-based approach to improving the communication of memory-resident values. We demonstrate the automatic insertion of synchronization and forwarding primitives, and also how to ensure correct execution when forwarding potentially aliased values. Second, we show that compiler-inserted synchronization can reduce the amount of failed speculation caused by frequently-occurring dependences, and hence improve performance significantly for some applications. Finally, we compare and contrast our approach for compiler-inserted synchronization of memory resident values with a recently proposed hardware technique [25] and demonstrate that the hardware and compiler can work in tandem.

2.. SYNCHRONIZING MEMORY-RESIDENT DEPENDENCES

Previous research [32] has shown that compiler-inserted synchronization can effectively communicate scalar values between epochs and improve program performance by boosting the efficiency of parallel execution. In this paper, we extend this work to communicate memory-resident values. Synchronizing frequently-occurring memory-resident values is, however, more complicated due to the existence of potential aliasing (i.e., a pointer through which the memory location in question is unexpectedly modified).

In this section, we first describe how the compiler identifies and synchronizes register-resident scalar values, then point out the differences between communicating register-resident values and memory-resident values. We also describe how the compiler can explicitly synchronize accesses to memory-resident values and avoid failed speculation.

2.1 Synchronizing Register-Resident Values

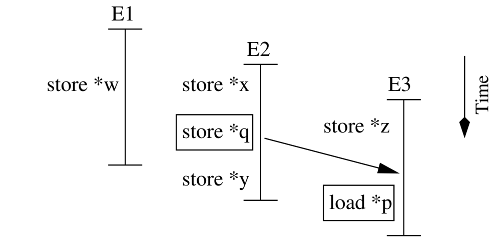
We can identify scalars that require synchronization using traditional data flow analysis. Scalar synchronization [32] is applied to the set of local *communicating scalars* (i.e. those defined in the scope of the enclosing procedure), which we define as any scalar which is *live* between epochs and does not have its address taken. Since each communicating scalar is allocated to a register (assuming it is not spilled), we also refer to the values they hold as *register-resident* values. For each communicating scalar, the compiler inserts `wait` and `signal` instructions to synchronize and forward the value. The `wait` instruction stalls until a value is produced and forwarded by the producer epoch through a `signal` instruction.

The following characteristics of register-resident values make them easier to synchronize than memory-resident values: (i) there is no aliasing in accessing scalar values, all accesses (reads or writes) must explicitly refer to the single register name; and (ii) static instructions that access communicating scalars only occur in the loop body being optimized, not in the procedures called from the loop body. Thus, it is relatively straightforward to identify all accesses to these values and to use data-flow techniques to determine the last definitions and the first exposed uses within an epoch.

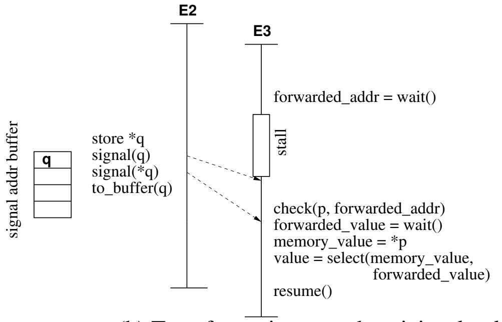
2.2 Synchronizing Memory-Resident Values

Unfortunately, the mechanism for forwarding register-resident scalar values with `signal` and `wait` instructions cannot be directly applied to forwarding memory-resident values for two reasons. First, we are unable to decide whether two memory accesses refer to the same data item using traditional data-flow analysis when the same location can be accessed using different names through pointers. Second, the existence of potential aliasing in accessing memory-resident values make it difficult and sometimes even impossible to determine the last definition and the first exposed use of a data item within an epoch. Thus, as opposed to only synchronizing frequent dependences at definite program points, we now synchronize *probable* data dependences for memory-resident values.

Now we take a close look at how aliasing in memory accesses makes our problem more difficult. An inter-epoch dependence occurs between a store and a load if: (i) the store occurs in a logically earlier epoch, (ii) both the store and the load access the same memory address, and (iii) no other store, between the store and load in question, modifies this address. Figure 3(a) shows three epochs executing speculatively in parallel. Assume that *load *p* could depend on any of the five store instructions while, however, it depends on *store *q* most frequently, thus, we want to synchronize and forward a value between this pair. Traditional pointer analysis [3, 5, 29] may help us reduce the set of pointers that *p* aliases to, but could not provide the set of *frequently* dependent instructions that we need. For instance, *must*-alias pointer analysis could not identify *likely* dependences, such as *load *p* and *store *q*, hence would not synchronize them. On the other hand, *may*-alias pointer analysis would indicate that *load *p* may depend on any of the five store instructions, hence they should all be synchronized. Since neither provides us with the desired information in this situation, we need profiling-based tools that identify *likely* dependences [6, 5]. We also need mechanisms that allows us to synchronize these likely data dependences, and to ensure correct execution for whatever dependences actually occur at runtime.



(a) Original program: *load *p* often depends on *store *q*.



(b) Transformation: synchronizing *load *p* and *store *q*.

Synchronization operation	Description
store *q;	The original store operation.
signal(q);	Forward the address **q** to the next epoch.
signal(*q);	Forward the value stored as well.
to_buffer(q);	Save the address **q** in the signal address buffer.
forwarded_addr = wait();	Wait for the address to arrive from the previous epoch.
check(p, forwarded_addr);	If *p* equals *forwarded_addr* set the *use_forwarded_value* flag. Loads issued while this flag is set will not cause violations.
forwarded_value = wait();	Wait for the value to arrive from the previous epoch.
memory_value = *p;	Load a value from the memory system using the load operation. If the address *p* has been previously modified by the current epoch, this instruction clears the *use_forwarded_value* flag. If the *use_forwarded_value* flag is set when this load is issued, this instruction only accesses the speculative cache and will *not* cause a violation.
value = select (memory_value, forwarded_value);	If *use_forwarded_value* flag is set, select *forwarded_value*, otherwise, select *memory_value*. The selected value is placed in *value*.
resume();	Reset the *use_forwarded_value* flag.

(c) Description of operations inserted for synchronization.

Figure 3: Program transformation to synchronize frequently occurring memory-resident dependences between epochs.

In the rest of this section we describe the hardware mechanism for synchronizing memory-resident values while preserving correct execution, using Figure 3(b) as our guide.

The producer of the forwarded value still has to store the value to memory, since it may still be read from memory by other parts of the program. The producer also has to communicate the forwarded value and its address, through the *signal* instructions. In addition, the producer has to be able to detect if the wrong value was forwarded—this is done by storing the address in the *signal_address_buffer*, a small per-cpu buffer which is used to make sure that no later store in the epoch writes to the same memory location.

The consumer of the forwarded value first has to wait for the value and its address to arrive, through the *wait* instructions. The consumer then checks to see if the addresses match (to make sure a useful value was received), and if so sets a cpu-local flag called *use_forwarded_value*. The consumer then issues a load to the speculative cache. If the *use_forwarded_value* flag is set when this load is issued, this instruction only accesses the speculative cache and will *not* cause a violation. This load also checks to see if the value has been overwritten locally and clears the *use_forwarded_value* flag if it is. The value of the *use_forwarded_value* flag then determines whether the forwarded value or the value loaded from memory is used in subsequent computation, and when we are finished the *use_forwarded_value* flag is reset.

We now describe how correctness is ensured by describing all possible data dependences that may occur. When a true data dependence occurs between *store *q* and *load *p*, the forwarding mechanism forwards the correct address and value, then the forwarded value is used. If *load *p* depends on *store *w* or *store *x*, the forwarded address q cannot point to the same location as p. Thus, the *use_forwarded_value* flag is not set, the *select* instruction will choose *memory_value*, and the underlying hardware that supports TLS will ensure correct execution. If p, q and y all point to the same memory location, the forwarding instruction will forward the correct address, but a wrong value. The producer epoch will notice that it is storing to an address that is already in the *signal_address_buffer*, and send a signal which restarts the consumer epoch. If *load *p* depends on *store *z*, *use_forwarded_value* flag is reset by the load instruction and we will use the value loaded from the memory. This is correct, since this memory access is not exposed and the local cache holds the correct value.

It is possible that on some paths through an epoch the value is never produced. In this case, the producer epoch should still signal the consumer epoch by sending a `NULL` value in the address field, so that the consumer does not wait indefinitely. If p points to a valid address then it will not match this `NULL` pointer, and the load in the consumer epoch will be read from memory. If p happens to be a `NULL` pointer as well then the program will dereference this `NULL` pointer just like the original untransformed program did, and cause an exception (depending on the policy of the host operating system).

The size of the *signal_address_buffer* is equal to the number of forwarded values. In practice, the number of values requiring forwarding is small. Our experiments show that we never need a buffer larger than 10-entries.

2.3 Compiler Support

In our approach to TLS support, the compiler is able to both detect and synchronize frequently-occurring data dependences. In this section we demonstrate how the compiler inserts synchronization using the example shown in Figure 4. In this example we parallelize a loop that calls the procedures *free_element()* and *use_element()* to add and remove members of a linked list called *free_list*. In every iteration of the loop, the global variable *free_list* is read and modified, potentially causing frequent data dependences and failed speculation unless prevented by proper synchronization. Note that this example is complicated by the fact that the variable *free_list* can be accessed using other names (i.e., *aliases*). Our compiler performs the following steps to synchronize the accesses to this variable:

Profiling dependences: The compiler identifies frequently-occurring, memory-resident, data dependences by profiling all inter-epoch data dependences for each parallelized loop (this profile information is context-sensitive but flow-insensitive). To acquire the profile information, we first associate a unique identifier with each static load and store instruction, and each procedure call point. During execution each load and store instruction can be named by the combination of the instruction identifier and the current *call stack* (the call stack for an instruction, rooted at the parallelized loop, is the list of procedures calls invoked when that instruction is executed). During profiling, each load is matched with any store on which it depends, and the frequency of each dependence is recorded. In Figure 4(a), *ld_1*, *ld_3*, *st_2* and *st_4* all access the same memory location denoted by *free_list*, and their dependence relation is illustrated in Figure 5. Note that a two memory references with the same identification number but different call stacks are treated separately (i.e., represented by two different vertices in the graph).

Identifying frequently occurring dependences: Unlike scalar values, the same memory-resident value can be accessed with multiple names (through pointer aliasing)—hence we *group* together loads and stores that access the same memory location. It is important to understand that a *group* is different from an alias set. An alias set of pointers is defined conservatively to be a set of pointers that *may* point to the same memory locations. In contrast, (i) pointers in a group will *definitely* access the same memory locations frequently, and (ii) pointers that access the same location might not be grouped if the corresponding data dependences are infrequent.

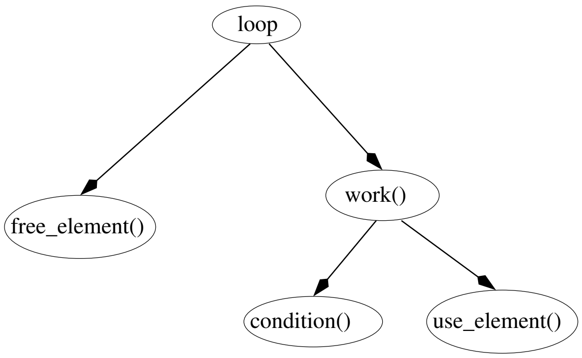
The compiler chooses groups of pointers by using the dependence profiling information described above to construct a dependence graph, where each load or store instruction with a different call stack is represented by a vertex, and each frequently-occurring dependence is represented by an edge. In the resulting graph, each connected component represents a *group*, and all loads and stores belonging to the same group are then synchronized by the compiler as a single entity. Note that we ignore infrequent dependences for performance reasons: if we were to additionally include infrequent data dependences in the graph then our groups would be much larger (as shown in Figure 5), leading to over-synchronization and poor performance.

Cloning: For best performance, we want synchronization code to be executed only when necessary to avoid data dependence violations. For example, when a load with a particular call stack is chosen for synchronization, ideally the corresponding synchronization code would only be executed when the load has been reached on a path matching that call stack—the synchronization code should not be executed when the load is reached through some other call path.

Our compiler uses the following steps to implement this code specialization, which basically clones the appropriate procedures on the call stack of a synchronized memory reference. First we build a call tree with the parallelized loop as the root and each call instruction as a decedent of this loop, as shown in Figure 4(a). Second, we identify the location in the tree of all frequently-occurring data dependences: for any node containing frequently-occurring dependences, that node and its parents are all cloned, and the original call instructions are modified to refer to these cloned procedures. In our example, the synchronized load and store occurs on the call stack *call_3*, hence the procedure *free_element* is cloned as shown in Figure 4(b). Code expansion due to such cloning is negligible (less than 1% on average), since only a small number of procedures are cloned in each application.

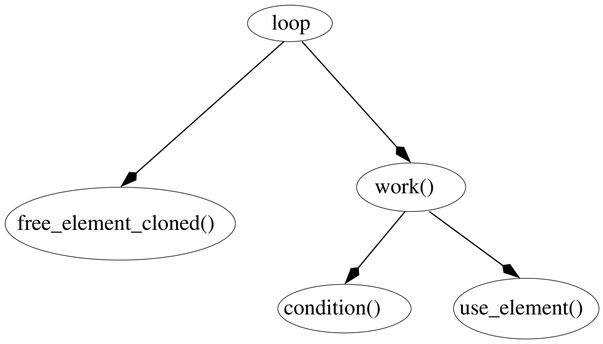
Inserting synchronization: *Wait* instructions are inserted before each load instruction to be synchronized, as shown in Figure 3(b). However, *Signal* instructions cannot be inserted after every store instruction, since multiple store instructions belonging to the same *group* could occur on a single execution path through an epoch. A signal instruction must occur at least once for each group on every execution path through the epoch, and should occur *after* the last store instruction from that group has been issued. We perform

```
void  free_element(element) {
   element->next = free_list;                 st_1, ld_1
   free_list = element;                       st_2, ld_2
}
int  use_element() {
   element = free_list;                       st_3, ld_3
   free_list = element->next;                 st_4, ld_4
   return element;
}
void  work() {
   if(condition())                            call_1
      use_element(some_element);              call_2
}
main() {
   do {
      free_element(some_element);             call_3
      work();                                 call_4
   } while (test);
}
```



(a) The original program and the corresponding call tree. Function calls, loads and stores are instrumented with labels to identify them.

```
void free_element_cloned(element) {
   f_addr = wait();
   check(f_addr, &free_list);
   f_value = wait();
   m_value = free_list;
   actual_value = select(f_value, m_value);
   resume();
   element->next = actual_value;
   free_list = element;
   signal(&free_list);
   signal(free_list);
}
...free_element(), use_element() and work()
functions omitted for brevity...
main() {
   do_parallel {
      free_element_clone(some_element);
      work();
   } while (test);
}
```



(b) The cloned call tree and the program with synchronization inserted.

Figure 4: Compiler-directed procedural cloning and synchronization insertion.

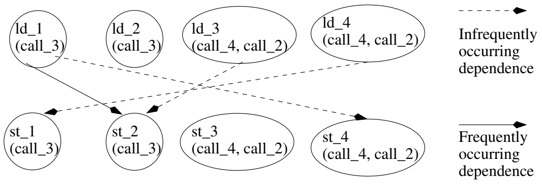


Figure 5: An example dependence graph. Each vertex represents a load or store, identified by the combination of a unique number and call stack. Each edge shows a true data dependence between memory references. Ignoring infrequent data dependences, a group is formed with two vertices: *ld_1* and *st_2* (both having call stack *(call_3)*). Accounting for infrequent data dependences would result in an overly-large group.

data-flow analyses to find locations that satisfy such constraints to insert the signal operations, similar to the data-flow analyses used to synchronize scalar values [32]. The results of these data-flow analyses are propagated to the cloned procedures to allow signal instructions to be inserted as close as possible to where the value is produced.

2.4 Analysis of Data Dependence Patterns

The synchronization mechanism proposed in this section attempts to reduce failed speculation by targeting only frequently-occurring data dependences between consecutive epochs. We now demonstrate that the overall performance penalty due to failed speculation can be mostly attributed to such dependences, and that our decision to ignore infrequent dependences is justified. We performed a limit study using a model with perfect value prediction for loads of interest, which represents an upper bound on the possible performance of synchronizing those loads.

Although it is clear that a frequently-dependent load/store pair should be synchronized, we have yet to experimentally determine a threshold frequency at which synchronization is more beneficial than speculation. To answer this question we conducted the experiment shown in Figure 6. First, we identified load instructions that cause inter-epoch data dependences in more than 5%, 15% and 25% of all epochs. Then, we measure the impact of perfect prediction for each set of loads. Although perfect prediction of loads with highly-frequent dependences (eg., 25%) eliminates a significant amount of failed speculation, GZIP_COMP and BZIP2_COMP do not speed up with respect to sequential execution until we additionally predict loads with less-frequently occurring dependences. Only when all loads that cause inter-epoch data dependences in more than 5% of all epochs are perfectly predicted are we able to improve the performance of all benchmarks, suggesting a reasonably low threshold value of 5%.

The distance of a data dependence, in the context of TLS, is the number of epochs between the producer epoch and the consumer

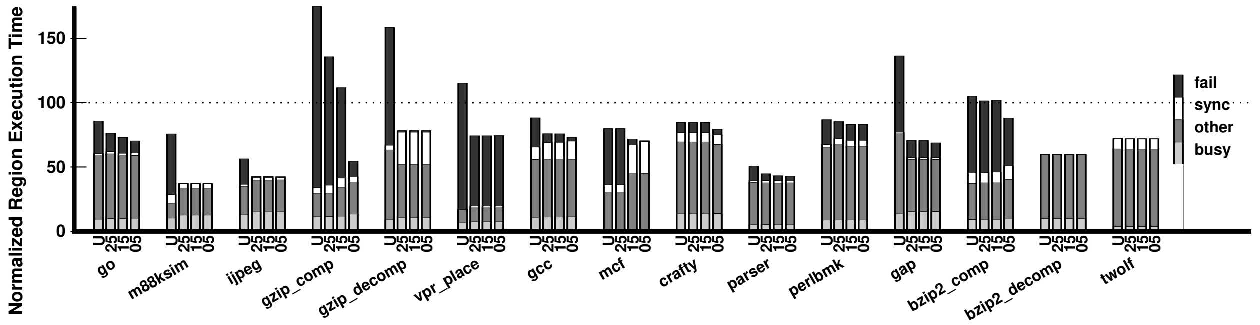


Figure 6: Impact of perfectly forwarding value for loads that depend on a store in the previous epoch (dependence distance one), broken down by frequency of the dependence: *U* is the unoptimized case with no forwarding; *25* shows the impact of perfectly forwarding all loads that depend on the previous epoch in more than 25% of all epochs; *15* shows the impact of perfectly forwarding all loads that depend on the previous epoch in more than 15% of all epochs; *05* shows the impact of perfectly forwarding all loads that depend on the previous epoch in more than 5% of all epochs.

epoch. For example, a data dependence between two consecutive epochs has distance of one. To determine the amount of failed speculation caused by dependences of different distances, we carry out an idealized simulation assuming that we can perfectly predict values for loads with dependences of varying distances, as shown in Figure 7. We observe that the performance impact for distance-one loads is significant; however, the impact of loads with larger dependence distances is small, and only relevant for one benchmark. Hence dependences of distance one should be the focus of any synchronization effort.

3. INFRASTRUCTURE FOR TLS

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

3.1 Compiler Infrastructure

We rely on the compiler to define where and how to parallelize. Our compiler infrastructure is based on the Stanford SUIF 1.3 compiler system [27], and performs the following phases when transforming an application to exploit TLS.

Deciding Where to Parallelize: A *speculative region* is a portion of a program that we speculatively parallelize. In this paper, we focus solely on loops. With the profile information automatically gathered, the compiler starts with a set of loops chosen 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 epochs per instance, as well as an average of at least 15 instructions per epoch. Loops satisfying these conditions are considered for parallelization. We want to identify the set of loops that are *likely* to minimize total execution time, given that the techniques described in this paper can improve the performance of value communication through memory. We do so by obtaining an optimistic upper bound on performance by identifying all loads that cause inter-epoch data dependences in more than 5% of all epochs and assume that we can perfectly predict values for these loads during execution. The set of loops that minimize the total execution time of the entire program under this ideal condition are selected for this study. Once loops are selected, the compiler automatically applies loop unrolling to small loops to help amortize the overheads of speculative parallelization. Note that the profiling described above is required only to select which loops are to be parallelized (and not to decide how to forward values). Deciding which regions of code to speculatively parallelize using a minimum amount of profiling information is the subject of ongoing research.

Transforming to Exploit TLS: Once speculative regions are chosen, the compiler inserts new TLS-specific instructions into the code that interact with the TLS hardware to create and manage epochs [23]. For each speculatively parallelized region the compiler inserts explicit synchronization to communicate scalar values between epochs. To avoid parallel epochs being serialized unnecessarily by such synchronization, the compiler schedules instructions within the epoch to reduce the *critical forwarding path* [32].

Inserting Synchronization for Memory-Resident Values: The final optimization step is for the compiler to identify frequently-occurring inter-epoch data dependences through memory resident values and to insert explicit synchronization (the subject of this paper). In our implementation, we identify these dependences with the help of detailed profile information. The details of this data dependence profiling, as well as the synchronization mechanisms and corresponding compiler support are discussed in section 2.3.

Code Generation: 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 containing these new TLS instructions.

3.2 Underlying Hardware Support

The hardware which supports TLS must implement two important features: buffering speculative modifications from regular memory, and detecting and recovering from failed speculation. Our underlying hardware support is based on the scheme proposed in our previous work [23, 24] which extends invalidation-based cache coherence to track data dependences and uses the first-level data caches to buffer speculative state from the rest of the memory system.

3.3 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 R14000 [31], but modernized to have a 128-entry reorder buffer. We simulate a system with four processing cores, where each 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 pre-

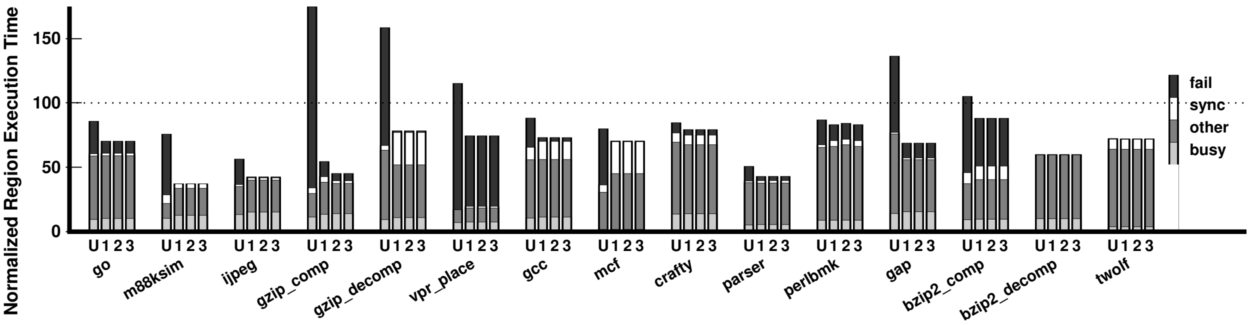


Figure 7: Impact of forwarding values for frequently occurring data dependences (5% of the time), broken down by dependence distance: *U* is the unoptimized case with forwarding; *1* shows the impact of perfectly forwarding values for all loads causing dependences of distance one; *2* builds on *1* by forwarding values for loads causing dependences of distance two; *3* builds on *2* by forwarding values for loads causing dependences of distance three.

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

diction, instruction fetching, branching penalties, and the memory hierarchy (including bandwidth and contention) are all modeled, and are parameterized as shown in Table 1. We report results for all of the SPECint95 and SPECint2000 benchmarks [22] 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; 129.COMPRESS, 130.LI, 134.PERL and 255.VORTEX each have low parallel coverage, and hence are not included in the performance graphs. For each benchmark, after skipping over the initialization phases, we simulate approximately a billion instructions using the first input in the `ref` input set.

4.. PERFORMANCE EVALUATION

We now present the results of our experiments to quantify the performance impact of our compiler-based technique, and we also compare it with related hardware-based techniques [8, 25].

4.1 Impact of Compiler-Inserted Synchronization

Figure 8 shows the performance impact of our compiler algorithm (described earlier in Section 2) for synchronizing frequently-occurring memory-resident data dependences. This figure shows the time spent in parallelized regions of the code (we will focus on overall program speedups later in Section 4.3), normalized to the time spent in those regions in the original sequential program. (Hence if a bar is less than 100, it means that the parallelized regions would be speeding up under TLS.) The **U** bars are the baseline ("unsynchronized") case that we are attempting to improve upon: they contain no synchronization for memory-resident values (but may contain synchronization for scalar register values [32]). The **T** and **C** bars show the impact of compiler-inserted synchronization for memory-resident values on region performance with the `ref` input sets, where profiling was done with the `train` (**T**) and `ref` (**C**) input sets, respectively.

Comparing **C** with **U** that compiler-inserted synchronization improves performance in half of the benchmarks (GO,GZIP_COMP, GZIP_DECOMP, VPR_PLACE, GCC, PARSER, PERLBMK, and GAP), and has no significant impact in the other seven cases. (Note that in two of the seven cases where our technique did not improve performance—BZIP2_DECOMP and TWOLF—failed speculation was not a problem to begin with.) Among the seven cases that do improve, the amount of execution time wasted on failed speculation (*"fail"*) is reduced by an average of 68%. Although some of this gain is offset by an increase in time stalled waiting for synchronization (*"sync"*), these seven applications still enjoy an average region speedup of 17%.

By comparing the **T** and **C** bars in Figure 8, we can get a sense of how the accuracy of the profiling information used by our compiler can affect the quality of its results. The **T** bars represent the more realistic scenario where profiling is done with a different input set (`train`) than the one used in the actual run (`ref`), and the more optimistic scenario for the **C** bars (where the profiling and actual input sets are the same) is included for the sake of comparison. Note that in all but one case (GZIP_COMP), the results are fairly insensitive to the choice of profiling input set. In GZIP_COMP, however, the flow of control is complex and sensitive to the input set, and this in turn determines which loads and stores are dependent; hence different profiling input sets can lead the compiler to synchronizing different pairs of loads and stores. Because the **T** and **C** cases behave differently for GZIP_COMP, we will present both cases throughout the remainder of this paper as "GZIP_COMP_T" (**T**) and "GZIP_COMP" (**C**).

By synchronizing data dependences, we trade time spent on failed speculation with that on synchronization. When synchronization is not properly placed, it could create a critical forwarding path which dominates execution time. Although reducing the cost of synchronization is not the goal of this paper, we attempt to evaluate the significance of synchronization with two experiments: in Figure 9, the

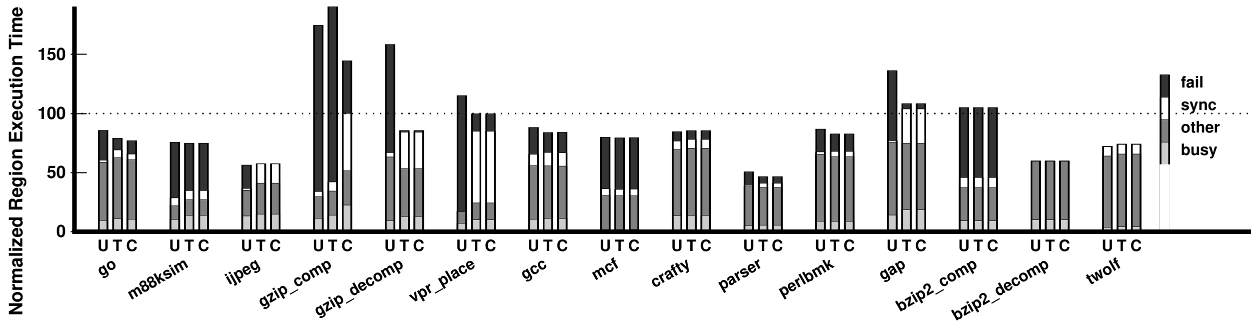


Figure 8: Impact of compiler-inserted synchronization using different profiling information. U has no synchronization; T profiles with the *train* input set and measures with the *ref* input set; C profiles with the *ref* input set and measures with the *ref* input set.

E bars correspond to an idealized experiment where the consumer is always able to perfectly predict any synchronized memory value. This eliminates all time spent on synchronization of memory values, but may increase violations since it increases parallel overlap. The **L** bars in Figure 9 correspond to a more conservative forwarding scheme where synchronized loads issued by the consumer are stalled until the previous epoch completes.

For M88KSIM, IJPEG, GZIP_COMP, GZIP_DECOMP and VPR_PLACE, execution time is positively correlated with the cost of synchronization. This indicates that stalling frequently violated loads until previous thread completes could serialize the execution unnecessarily and degrades performance. On the other hand, being able to forward the value early can reduce synchronization and improve performance.

4.2 Comparison with Hardware-Inserted Synchronization

Previous research [8, 25] proposed two *hardware* techniques to reduce the cost of failed speculation due to memory-resident values: *prediction* and *synchronization*. Neither of the proposed techniques require centralized structures to match dependence pairs; however, the they differ in complexity, from a 2KB violation prediction table [8] to two 32-entry tables that track loads which are exposed and loads which have caused speculation to fail [25]. We have implemented hardware-based *prediction* and *synchronization* as described *et. al* [25]. In Figure 10, we compare our compiler-inserted synchronization techniques with these two hardware mechanisms. The **P** bar shows that the *value prediction* technique that we have evaluated has insignificant effect on performance, indicating that forwarded memory-resident values are unpredictable. In the rest of this section, we focus on comparing hardware-inserted synchronization (**H**) with compiler-inserted synchronization (**C**). For the hardware inserted synchronization, the hardware identifies loads that frequently cause violations and stalls these loads until the previous epoch completes. To avoid over-synchronization of infrequently-dependent loads, we periodically reset the table that tracks the loads that have caused speculation to fail.

A comparison between compiler-inserted and hardware-inserted synchronization reveals that *each of the techniques wins in some cases but none of them wins for them all*. In eleven out of the fifteen benchmarks, at least one synchronization technique is able to improve the performance over the unoptimized case. Four benchmarks, GO, GZIP_DECOMP, PERLBMK and GAP, achieve the best performance with compiler-inserted synchronization; three benchmarks, M88KSIM, GZIP_COMP, VPR_PLACE, achieve the best performance with hardware-inserted synchronization. For the rest of the benchmarks, the two techniques are comparable. Here we attempt to explain why each benchmark responds differently to the two optimization techniques:

- In M88KSIM, violations are not caused by true data dependences, rather they are caused by false sharing. The compiler is attempting to synchronize true dependences, while the hardware is tracking dependences at a cache line granularity. Since violations are tracked at a cache line granularity, the hardware inserted synchronization yields the best results—we could track dependences at a cache line granularity in the compiler as well, but we believe that other techniques (such as memory layout optimizations or loop unrolling) are better for addressing false sharing in the compiler.
- In GZIP_DECOMPRESS, the compiler and the hardware both insert synchronization, however, the compiler is able to speculatively forward the desired value much earlier than our hardware can. This avoids over-synchronization, resulting in better performance.
- Software-inserted synchronization can be conservative—it synchronizes dependences which *may* or *may not* actually happen at runtime, depending on the timing of the epochs. If a load tends to be executed only when all prior epochs have completed, then it will rarely cause a violation. In such a case, the synchronization code just adds extra overhead—this is the cause of the small performance degradation in TWOLF.

Since the hardware and the compiler based synchronization can each benefit a different set of benchmarks, we conduct the following experiment to determine whether the two techniques are synchronizing the same set of memory-resident values: we invoke our synchronization scheme to mark each load instruction as *would be* synchronized by the compiler and/or by the hardware (depending on the execution mode, we may or may not stall for marked synchronization). When a violation does occur, we record whether the load that caused this violation *would* have been synchronized, and by which scheme. We execute the program under four different modes, and show the results in Figure 11: (i) do not stall for any synchronization, denoted by the **U** bars; (ii) only stall for compile-inserted synchronization, denoted by the **C** bars; (iii) only stall for hardware-inserted synchronization, denoted by the **H** bars; (iv) stall for both hardware-inserted and compiler-inserted synchronization, denoted by the **B** bars. We observe that a significant number of violating loads would only be synchronized by either the hardware or the compiler, but not both. Our existing compiler and hardware support can be complementary as follows: (i) the hardware

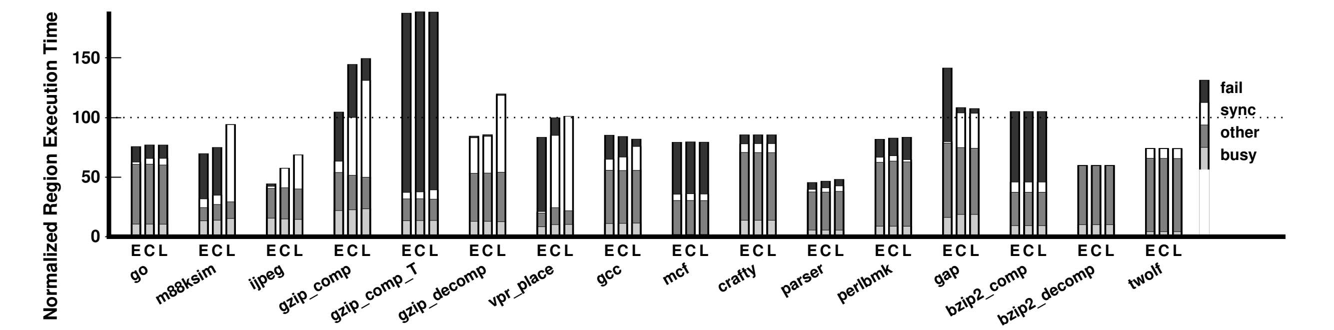


Figure 9: Potential impact of changing to a more aggressive or less aggressive synchronization scheme. E is the idealized case where the consumer can perfectly predict synchronized memory values; in C the consumer stalls any load of a synchronized memory value until the producer forwards the value; in L the consumer stalls any load of a synchronized memory value until the producer completes.

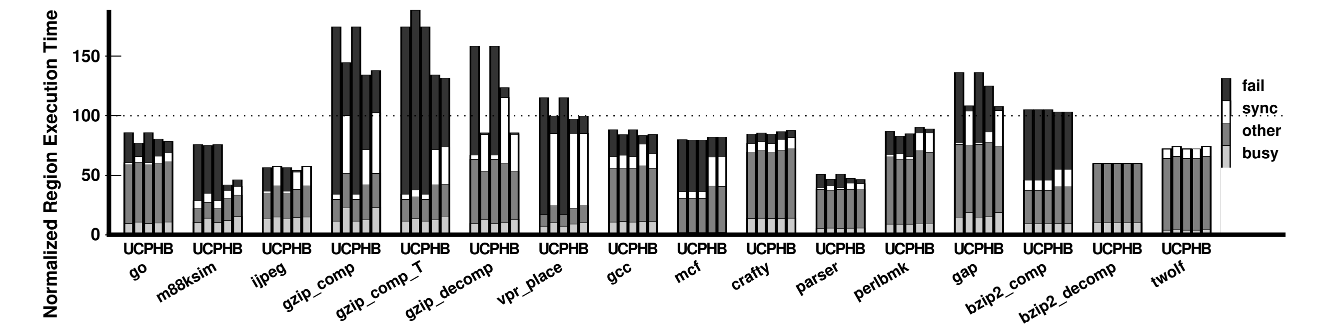


Figure 10: Comparison of compiler-inserted synchronization, hardware-inserted synchronization, and a hybrid scheme. U has no synchronization; C uses compiler-inserted synchronization; P uses hardware value prediction; H uses hardware-inserted synchronization with the violating load table periodically reset; B uses a hybrid of compiler-inserted and hardware-inserted synchronization with the violating load table periodically reset.

synchronizes violating loads that are not identified by profiling information; (ii) the compiler reduces the cost of synchronization by providing the forwarded value early. Two possible ways to further enhance complementary behavior are (iii) for the hardware to filter out compiler-inserted synchronization that rarely forward the correct values; and (iv) for the hardware to reset a violating load less frequently if the compiler hints that it will occur frequently.

To illustrate the feasibility of such a compiler-hardware hybrid, we enable both hardware synchronization with periodic reset and compiler-inserted synchronization. The results are shown in Figure 10 as bar **B**. In several benchmarks, the hybrid approach nearly captures the performance of the best of the two techniques: M88KSIM benefits from hardware-inserted synchronization and avoids the cost of false sharing, while GZIP_DECOMP benefits from having values be forwarded early by compiler-inserted synchronization. Therefore, it is possible for us to implement a hybrid that can improve the performance of a larger set of programs by taking advantage of both compiler and hardware inserted synchronization.

4.3 Program Performance

Since we are interested in studying the impact of synchronization on parallelized code, we so far have focused on region speedups. In Figure 12 we instead take the coverage of these loops into account and look at the performance impact on the whole program. We see that inserting synchronization of memory values has a significant positive impact for six of these benchmarks, and that the best results overall can be achieved with a hybrid of both software and hardware synchronization. Table 2 presents the speedups in detail, and we see that relatively large speedups in our parallel regions are sometimes offset by slowdowns in our sequential code. (Ideally, we should see a speedup of 1.0 in the sequential regions.) This is a side effect of our compiler infrastructure—the inline assembly we use to instrument parallelized loops can inhibit the optimization and register allocation of our `gcc` back-end, causing this measurement artifact. This overhead remains constant regardless of the hardware and/or compiler optimizations applied. We anticipate that with a proper compiler back-end (instead of using a source-to-source compiler followed by `gcc`) even better program performance would be observed.

5. CONCLUSIONS

TLS provides a mechanism for speculating that data dependences across optimistically-parallelized threads do not exist. Like most forms of speculation, however, when you speculate correctly you win, but when you speculate incorrectly, you can actually hurt performance. Hence an important question is *how frequently* do inter-thread data dependences occur? If they occur frequently enough for a given load-store pair, we may be better off explicitly synchronizing the threads so that the consumer waits for the value (or at least a good guess of what the value might be) from the producer. In previous work, we demonstrated that compiler-inserted synchronization for *scalar register* values was an important technique for improving TLS performance [32], and in this paper we tackled the question of whether the same is also true for *memory-resident* values.

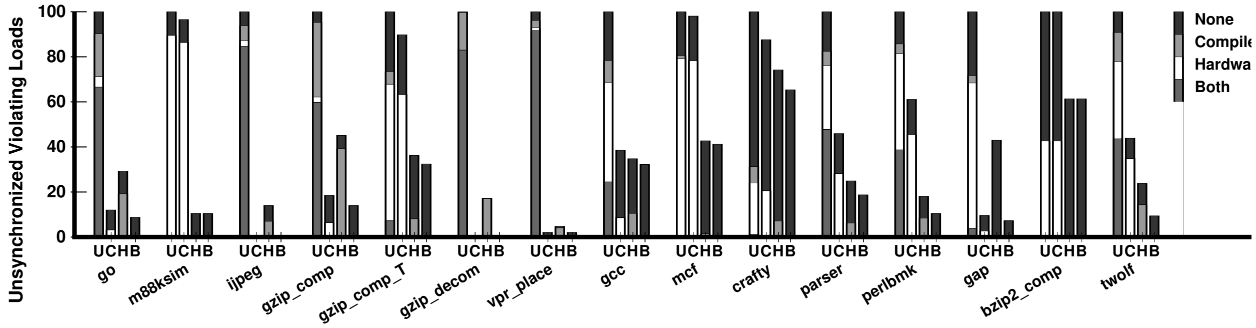


Figure 11: Breakdown of all loads that cause violations by whether they would be synchronized by hardware or compiler-inserted synchronization. U stalls on no synchronization; C stalls only on compiler-inserted synchronization; H stalls only on hardware-inserted synchronization; B stalls on both compiler-inserted and hardware-inserted synchronization. BZIP2_DECOMP **is omitted because it has no violations.**

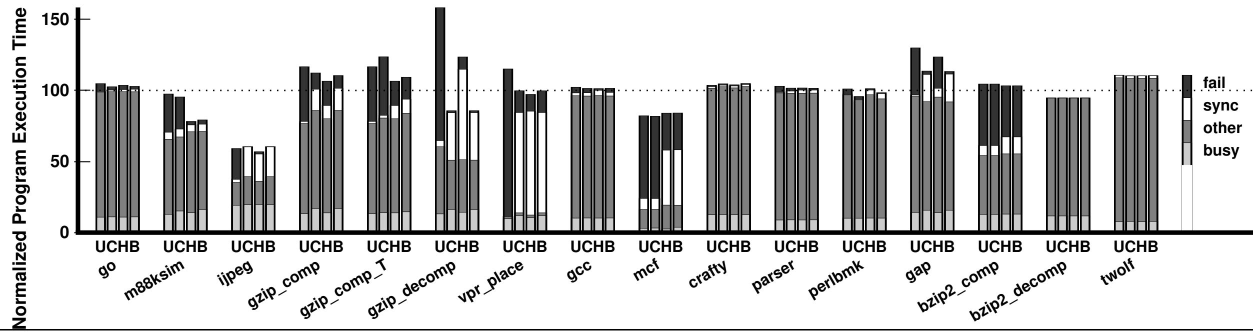


Figure 12: Program speedup. U is the case with no synchronization; C is compiler-inserted synchronization; H is hardware-inserted synchronization with the violating load table periodically reset; B is a hybrid of compiler-inserted and hardware-inserted synchronization.

We observe that for most benchmarks failed speculation is usually caused by load instructions that suffer dependence violations relatively *frequently* (e.g., at least 25% of the time), which makes them easy to spot given a mechanism for profiling data dependences. However, for some benchmarks we must be able to synchronize data dependences that only occur in 5% of the epochs to achieve a reasonable speedup. In addition, we also observe that the producer and consumer of inter-thread dependences are usually *consecutive* epochs, which simplifies the process of explicitly forwarding the data.

Our performance results demonstrate that applying compiler-inserted synchronization to *memory-resident* values that would otherwise cause frequent dependence violations does improve TLS performance in many cases: half of the applications enjoyed significant region speedups, while the other half were unaffected. The most dramatic case was GZIP_DECOMP, which went from a significant region slowdown to a significant region speedup using our technique. For GZIP_DECOMP and several other cases, we observe a significant benefit from the compiler's ability to forward data when it is produced (rather than waiting until the producer thread completes its execution).

Comparing our compiler-based approach with a hardware-based approach to synchronizing memory-resident values, we observe that both approaches are useful, and that neither approach consistently dominates the other: sometimes the compiler-based approach is much better than the hardware-based approach, and vice-versa. We observe that the two different approaches seem to behave differently because they often choose different sets of load instructions to synchronize. This suggests that a hybrid approach that combines the advantages of both approaches might be best. While the simple hybrid approach that we explored did not outperform the best of the two approaches for a given benchmark, it did a better job of tracking the best performance overall than either approach individually. In future work, it may be possible to design an even better hybrid approach.

6. REFERENCES

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

[2] BANERJEE, U. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, Mass., 1988.

[3] BHOWMIK, A., AND FRANKLIN, M. A fast approximate interprocedural analysis for speculative multithreading compiler. In *17th Annual ACM International Conference on Supercomputing* (2003).

[4] CHEN, D. K., AND YEW, P. C. Redundant synchronization elimination for doacross loops. *IEEE Transactions on Parallel and Distributed System 10*, 5 (1999), 459–470.

[5] CHEN, P.-S., HUNG, M.-Y., HWANG, Y.-S., JU, R., AND LEE, J. K. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming* (2003).

[6] CHEN, T., LIN, J., DAI, X., HSU, W.-C., AND P.-C.YEW. Data dependence profiling for speculative optimization. In *13th International Conference on Compiler Construction* (Barcelona, Spain, March 2004).

Table 2: Region coverage and program speedup (relative to sequential execution).

Benchmark	Coverage	Parallel Region Speedup		Sequential Region Speedup		Program Speedup	
		Compiler-only	Both	Compiler-only	Both	Compiler-only	Both
099.go	22%	1.29	1.27	0.90	0.90	0.97	0.97
124.m88ksim	56%	1.33	2.15	0.82	0.82	1.04	1.25
132.ijpeg	97%	1.73	1.73	0.52	0.52	1.64	1.64
164.gzip_comp	25%	0.69	0.72	0.98	0.98	0.88	0.90
164.gzip_comp_T	25%	0.52	0.75	0.98	0.98	0.80	0.91
164.gzip_decomp	99%	1.16	1.16	0.93	0.93	1.16	1.16
175.vpr_place	99%	1.00	1.00	0.97	0.97	1.00	1.00
176.gcc	18%	1.18	1.18	0.94	0.94	0.98	0.98
181.mcf	89%	1.25	1.21	0.99	0.99	1.21	1.18
186.crafty	14%	1.16	1.13	0.92	0.92	0.95	0.95
197.parser	37%	2.13	2.14	0.74	0.74	0.98	0.98
253.perlbmk	20%	1.20	1.12	1.00	0.98	1.04	1.01
254.gap	57%	0.92	0.92	0.82	0.82	0.87	0.88
256.bzip2_comp	63%	0.94	0.96	0.96	0.96	0.95	0.96
256.bzip2_decomp	13%	1.66	1.66	0.99	0.99	1.05	1.05
300.twolf	19%	1.34	1.34	0.84	0.83	0.90	0.90

[7] CHRYSOS, G., AND EMER, J. Memory dependence prediction using store sets. In *Proceedings of the 25th ISCA* (June 1998).

[8] CINTRA, M., AND TORRELLAS, J. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th HPCA* (Feb 2002).

[9] CYTRON, R. Doarcoss: Beyond vectorization for multiprocessors. In *Int'l. Conf. on Parallel Processing* (Aug. 1986).

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

[11] GOPAL, S., VIJAYKUMAR, T. N., SMITH, J. E., AND SOHI, G. S. Speculative Versioning Cache. Tech. Rep. 1334, Computer Sciences Department, University of Wisconsin-Madison, July 1997.

[12] GUPTA, M., AND NIM, R. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98* (November 1998).

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

[14] LIVSHITS, V. B., AND LAM, M. S. Tracking pointers with path and context sensitivity for bug detection in c programs. In *11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)* (September 2003).

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

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

[17] MOCK, M., DAS, M., CHAMBERS, C., AND EGGERS, S. J. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *SIGSOFT workshop on on Program analysis for software tools and engineering Snowbird* (June 2001).

[18] 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).

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

[20] RAUCHWERGER, L., AND PADUA, D. A. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed System 10*, 2 (1999), 160–172.

[21] ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. Trace processors. In *Proceedings of Micro 30* (1997).

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

[23] 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.

[24] 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).

[25] 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).

[26] SURA, Z., WONG, C.-L., FANG, X., MIDKIFF, J. L. S., AND PADUA, D. Automatic implementation of programming language consistency models. In *Sixth International Symposium on Parallel Architectures, Algorithms and Networks (LCPC'02)* (2002).

[27] 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.

[28] 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).

[29] WILSON, R. P., AND LAM, M. S. Efficient context-sensitive pointer analysis for c programs. In *Proc. ACM SIGPLAN 95 Conference on Programming Language Design and Implementation* (June 1995), pp. 1–12.

[30] WOLFE, M. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Massachusetts, 1989.

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

[32] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *Proceedings of the 10th ASPLOS* (Oct 2002).