Using Thread-Level Speculation to Simplify Manual Parallelization

Manohar K. Prabhu
Stanford University
Computer Systems Laboratory
Stanford, California 94305
mkprabhu@stanford.edu

Kunle Olukotun
Stanford University
Computer Systems Laboratory
Stanford, California 94305
kunle@stanford.edu

ABSTRACT
In this paper, we provide examples of how thread-level speculation (TLS) simplifies manual parallelization and enhances its performance. A number of techniques for manual parallelization using TLS are presented and results are provided that indicate the performance contribution of each technique on seven SPEC CPU2000 benchmark applications. We also provide indications of the programming effort required to parallelize each benchmark. TLS parallelization yielded a 110% speedup on our four floating point applications and a 70% speedup on our three integer applications, while requiring only approximately 80 programmer hours and 150 lines of non-template code per application. These results support the idea that manual parallelization using TLS is an efficient way to extract fine-grain thread-level parallelism.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) – *multiple-instruction-stream, multiple-data-stream processors (MIMD)*; C.1.4 [**Processor Architectures**]: Parallel Architectures.

General Terms
Design, Algorithms, Performance, Measurement.

Keywords
Chip multiprocessor, data speculation, manual parallel programming, multithreading, feedback-driven optimization.

1. BACKGROUND
As the number of transistors that can fit within a die has grown according to Moore's Law, computer hardware has grown in complexity. While hardware can rapidly evolve, operating systems and software must vary more slowly for a number of reasons. First, successful computer architectures support a large base of legacy applications, which are difficult to port to new platforms. Additionally, programmers become familiar with a few standard languages and hardware interfaces and seldom want to migrate to new ones. Finally, the efficient development of applications often requires complex code development tools, and the creation or purchase of these tools can be a significant barrier to entry for innovative hardware platforms.

As a result, often most applications do not utilize the newest features of a hardware platform. For example, multiprocessor computers are becoming increasingly common, with even some consumer desktop systems utilizing dual processors. However, few applications currently make efficient use of multiple processors to enhance single application performance. Rather, processors are assigned to separate applications, resulting in suboptimal single application performance and frequent processor idle cycles from having too few applications available to execute.

The primary problem is that creating parallelized versions of legacy code is difficult. Even with a good tool chain including profilers and parallelizing compilers [1][2][9][11], automated parallelization has proven to be a very difficult problem [19]. While successful for certain scientific applications, automated parallelization has typically provided poor parallel performance on general-purpose applications, especially integer ones. Manual parallelization can provide good performance on a much wider range of applications, but typically requires a different initial program design and programmers with additional skills. Parallel programs require data structures and algorithms intended for parallel processing. Data placement and data replication must be considered more extensively to minimize interprocessor access contention and latency. Likewise, algorithms must be designed to allow concurrent and synchronized accesses without data races or data access hot spots. For example, single counters may need to be replaced with distributed counters, and multiple entry points may need to be added to linked lists that would have performed well with only a single entry point in a uniprocessor algorithm.

In this paper we show that the use of thread-level speculation (TLS) for manual parallelization, along with only minimal additional support in the software development tool chain, can provide much of the ease of automated parallelization. At the same time, it achieves the high performance and broad applicability characteristic of conventional (non-TLS) manual parallelization. In conventional manual parallelization, software development is best targeted to a parallel platform from the start. Otherwise, extensive redesign of applications to use explicitly

PPoPP'03, June 11-13, 2003, San Diego, California, USA.

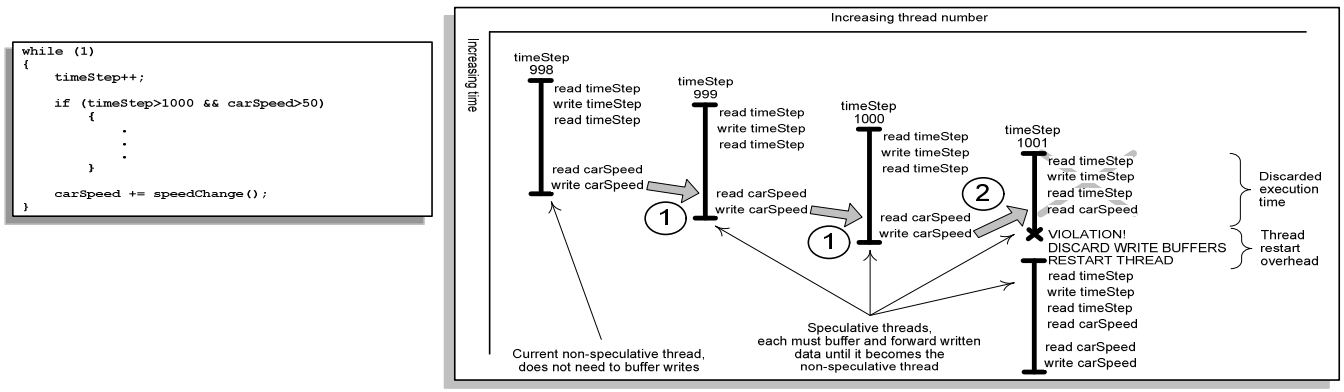

Figure 1. Thread-level speculation

parallel data structures and algorithms is often required. In contrast, we demonstrate large parallel speedups on a diverse set of uniprocessor applications, while changing or adding very few lines of code. With TLS-driven manual parallelization almost all of the application design is done for a uniprocessor target, and only a small effort is required at the end to transform the application into a high-performance parallel application. This simple transformation to parallel code is possible because TLS automatically detects and prevents data races, one of the main hazards that makes conventional manual parallelization so difficult. In this way, manual parallelization with TLS provides programmers with a new and incremental approach to incorporating parallel performance into legacy applications.

The rest of this paper is organized as follows. In the next section we will explain TLS, our architecture and the benchmarks we chose to parallelize. In Section 3, we discuss two simple examples of code containing thread-level parallelism (TLP), to show the simplicity and performance advantages of applying TLS to manual parallelization, rather than using conventional non-TLS manual parallelization. We also use these examples to introduce some useful source code transformations that allow TLS to extract TLP from many types of code that exhibit parallelism. Section 4 focuses upon the results of parallelizing the benchmarks. Section 5 frames the current research within the context of previous work, and Section 6 presents our conclusions.

2. TLS AND THE TEST PLATFORM

This section begins with an explanation of the theory and mechanisms of thread-level speculation (TLS). Next it discusses the specific TLS implementation simulated in our experiments. Finally, it describes our simulator and compilation environment.

TLS is the process of speculatively executing interdependent threads out-of-order, while appearing to have executed them in order. In this paper, the threads will be formed by splitting the dynamic execution path of a single-threaded application into multiple ordered threads. A loop in a single-threaded application is a common place to create TLS threads, one for each iteration of the loop body, as shown in Figure 1. Because each TLS thread is a portion of the dynamic execution path of the original single thread, the TLS threads are both interdependent and also ordered according to their order in the original single-threaded application. To enforce correctness, hardware must make each thread appear to have executed sequentially in the original order.

In a typical TLS system [7][13][17][21], several consecutive threads are executing at any moment. The first in order is nonspeculative; the rest are speculatively executing ahead in time. When the nonspeculative thread completes, the next thread in order becomes nonspeculative. Speculative reads and writes are handled specially. Each speculative thread must have its writes buffered until the time that it becomes nonspeculative, at which time it commits the writes. Additionally, speculative threads must ensure that any values that they read include the effects of writes stored in buffers of less speculative threads, but not writes from more speculative threads. For example, if a speculative thread creates and buffers a new value that a more speculative thread subsequently reads, the new value must be forwarded to the more speculative thread (Case ① in Figure 1). On the other hand, if a new value is created, but the old value has already been used by a more speculative thread, that thread and all threads following it must discard their results and restart execution with the new, correct value (Case ②). This is known as a data dependence violation, and, as shown in Figure 1, the cost of discarded execution time plus the thread restart overhead can waste time and cause large performance losses. Hence, reducing violations is important in any code with frequent inter-thread dependences.

The TLS hardware implementation used for this paper is the Stanford Hydra chip multiprocessor (CMP). This CMP comprises four pipelined MIPS-based R3000 processor cores, each with private L1 instruction and data caches (Figure 2). The four processors share an on-chip, unified L2 write-back cache, and each processor executes a single thread. Each processor's L1 data cache is write-through, and the other processors snoop the bus connecting the processors and the L2 cache to permit data dependence violation detection. Dependences are tracked on a

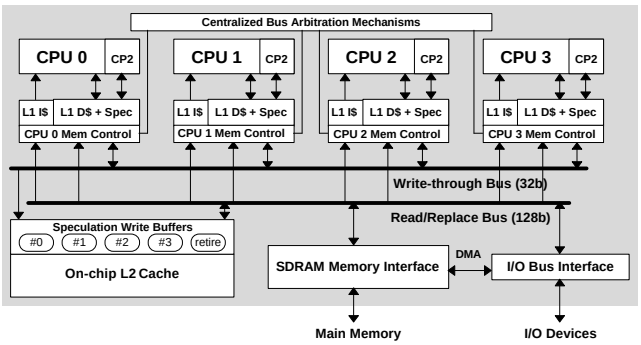

Figure 2. Hydra chip multiprocessor

Table 1. Memory system specifications

Characteristic	Memory system		
	L1 cache	L2 cache	Main memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16 KB each	2 MB	256 MB
Bus width	32-bit connection to CPU	256-bit read bus and 32-bit write bus	64-bit SDRAM at half of CPU speed
Access time	1 CPU cycle	5 CPU cycles	At least 50 cycles
Associativity	4-way	4-way	N/A
Line size	32 bytes	64 bytes	4 KB pages
Write policy	Writethrough, no write allocate	Writeback, allocate on writes	"Writeback" (virtual memory)
Inclusion	N/A	Inclusion enforced by L2 on L1 caches	Includes all cached data

per-word basis, thereby eliminating almost all violations due to false sharing. Speculative result buffering is achieved by buffering speculative writes to the L2 cache in a group of 32-cache-line buffers, one for each processor. These buffers also monitor read requests made to the L2 cache. This allows them to forward data created by writes from less speculative processors to satisfy the requests of more speculative processors. Other hardware in the L1 data caches enforces the TLS protocols, such as the detection and processing of dependence violations. While the CMP hardware follows a partial store ordering memory consistency model, the TLS system causes the CMP to appear to the programmer like a single out-of-order processor, so the programmer need not consider memory consistency issues. Further details of this design can be found in Table 1 and [13].

The support for TLS is implemented in a combination of hardware and software for ease of implementation and adaptability of the protocols, although this does increase the thread control overheads. Table 2 provides information on these software overheads. The first three can be statically analyzed, and occur whenever a speculative loop is started or ended and whenever iterations in a speculative loop are committed. The remaining four overheads occur dynamically due to violations detected on this CPU or on less speculative CPUs and due to stalls for speculative threads resulting from buffer constraints or the handling of an exception. The TLS system allows speculation

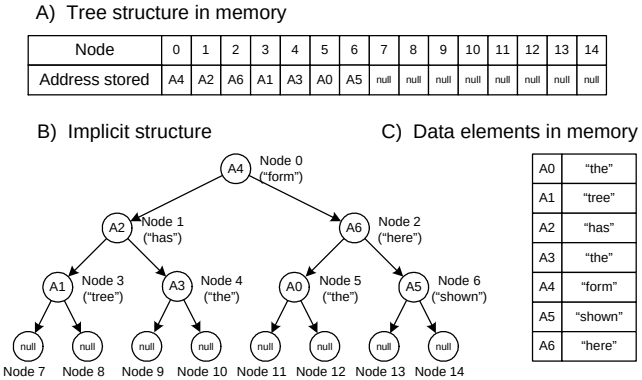

Figure 3. Organization of the heap array

Table 2. Loop-only TLS overheads

Overheads for loop-only TLS	Software handler	Instruction count
Regular events	Start loop	~30
	End of each loop iteration	12
	Finish loop	~22
Irregular events	Violation: local	7
	Violation: receive from another CPU	7
	Hold: buffer full	12
	Hold: exception	17 + OS

only on loops and at a single level, i.e. not speculation on a loop nested within another speculative loop. The system could have conducted procedural speculation via the use of different software handlers [13], but loop-only speculation was chosen for its lower overheads. As a result, the performance losses resulting from the speculation software handler overheads are typically quite small.

To simulate this TLS system, a cycle-accurate, execution-driven simulator was used to execute all application instructions, including the speculation software handlers. The simulator can accurately model a realistic memory system, including the effects of bus contention and memory access queuing. A perfect memory model was also simulated to gauge the performance losses due to not scaling the memory system with the number of processors running in parallel. All performance measurements presented here were done using applications compiled by GCC 2.7.2 with optimization level -O2 on an SGI workstation running IRIX 5.3.

3. METHODS FOR USING TLS

In this section, we will use two simple examples to illustrate many important points about how a programmer can use TLS to parallelize applications. First, we will show the ease of using TLS versus conventional (non-TLS) manual parallelization. Second, we will discuss the performance advantages of using even simple TLS parallelization versus a thorough redesign of applications using conventional parallelization. We will also show the performance advantages of manual over purely automatic TLS parallelization. Third, we will explain several types of source code transformations that can expose more of the TLP inherent in applications. Fourth, we will illustrate the very different code development cycle experienced by a manual TLS programmer.

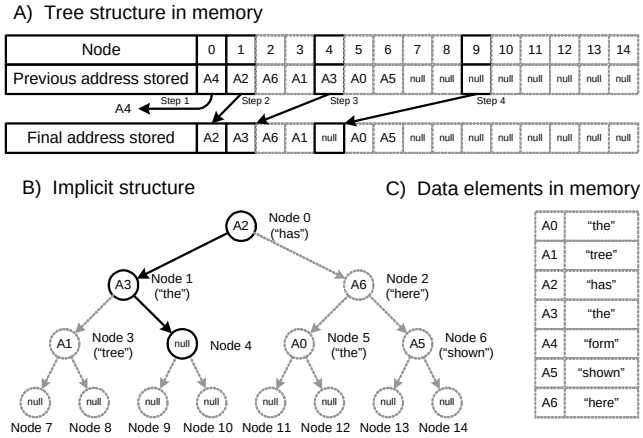

Figure 4. Top node removal and update of the heap

3.1 Heap Sort Example

The first example is C code that implements the main algorithm for a heap sort. In this algorithm, an array of pointers to data elements is used to sort the elements. Encoded in memory as a simple linear array (Figure 3A), the node array is actually interpreted as a balanced binary tree by the algorithm (Figure 3B). Tree sibling nodes are recorded consecutively in the array, while child nodes are stored at indices approximately twice that of their parents. For example, Node 2 is located directly after its sibling (Node 1) in the array, while Node 2's children (Nodes 5 and 6) are located adjacent to each other with indices approximately twice that of Node 2. This structure allows a complete binary tree to be recorded without requiring explicit pointers to connect parent and child nodes together, because the tree structure can always be determined arithmetically. In this example, each node of the tree consists of a single pointer to a variable-length data element located elsewhere in memory (Figure 3C).

The heap is partially sorted. The element pointed to by any parent is always less than the element pointed to by each of the children, so the first pointer always points to the smallest element. Nodes are added to the bottom of the tree (highest indices) and bubble upwards, switching places with parents that point to greater valued elements. Final sorting is conducted by removing the top node (first pointer) and iteratively filling the vacancy by selecting and moving up the child pointer that points to the lesser element (Figure 4). We will focus only on this final sorting, which typically dominates the execution time of heap sort.

```
1:  #define COLWID (30)
2:  char *result, *node[];
3:  void compileResults() {
4:   char *last, *inRes;
5:   long cmpPt, oldCmpPt, cnt;
6:   int sLen;
     //  INITIALIZATION
7:   inRes = result; last = node[0]; cnt = 0;
     //  OUTER LOOP - REMOVES ONE NODE EACH ITERATION
8:   while (node[0]) {
      //  IF NEW STRING, WRITE LAST STRING AND COUNT
      //  TO RESULT STRING AND RESET COUNT
9:    if (strcmp(node[cmpPt=0], last)) {
10:    strcpy(inRes, last);
11:    sLen = strlen(last);
12:    memset(inRes+sLen, ' ', COLWID-sLen);
13:    inRes += sprintf(inRes+=COLWID,"%5ld\n",cnt);
14:    cnt = 0;
15:    last = node[0];
16:   }
17:   cnt++;
      //INNER LOOP - UPDATE THE HEAP, REPLACE TOP NODE
18:   while (node[oldCmpPt=cmpPt] != NULL) {
19:    cmpPt = cmpPt*2 + 2;
20:    if (node[cmpPt-1] && !(node[cmpPt] &&
        strcmp(node[cmpPt-1], node[cmpPt]) >= 0))
21:     --cmpPt;
22:    node[oldCmpPt] = node[cmpPt];
23:   }
24:  }
     //  WRITE FINAL STRING AND COUNT TO RESULT STRING
25:  strcpy(inRes, last);
26:  sLen = strlen(last);
27:  memset(inRes+sLen, ' ', COLWID-sLen);
28:  sprintf(inRes+=COLWID, "%5ld\n", cnt);
29: }
```

Figure 5. Code for top node removal and heap update

The code is provided in Figure 5. It can be used to count the number of appearances of each (linguistic) word in a passage of text. It has been optimized for uniprocessor performance, so that parallelization with TLS can only derive speedups due to true parallelism and not due to more efficient code design. The code processes the pre-constructed heap `node[]`, where each node (e.g. `node[3]`) is a pointer to a string (line 2). As each top node is removed and replaced from the remaining heap, a count is kept of the number of instances of each string dereferenced by the nodes (line 17). Each string and its count are written into a (previously allocated) result string (line 2) at the position pointed to by `inRes` (lines 9-16). To do this, the top node of the heap (`node[0]`, which points to the alphabetically first string) is removed and compared to the string pointed to by the previous top node removed (lines 8 and 9). If they point to dissimilar strings, then all nodes pointing to the previous string have been removed and counted, so the string and its count are written to the result string and the count is reset (lines 9-16). In all cases, the count for the current string is incremented (line 17) and the heap is updated/sorted in the manner described above (lines 18-23). The heap is structured so that below the last valid child on any tree descent, the left and right child are always two NULL pointer nodes (line 18). This whole counting and sorting process is conducted until the heap is empty (line 8). Then the results for the last string are written to the result string (lines 25-28).

3.2 Parallelizing with TLS

When parallelizing with TLS, the programmer first looks for parts of the application with some or all of the following qualities. These parts should dominate the execution time of the application with that time concentrated in one or more loops, preferably with a number of iterations equal to or greater than the number of processors in the TLS CMP. These loops should contain fairly independent tasks (few inter-task data dependences), with each task requiring from 200 to 10,000 cycles to complete, and all tasks being approximately equal in length for good load balancing. For the example program, we see that the two loop levels where we can parallelize this code are either the inner loop or the outer loop, i.e. within a single event of removing `node[0]` and updating the heap (lines 8-24), or across multiple such events. The first is not good due to the small parallel task sizes involved, which are better targeted with techniques that exploit ILP. The second level is much better suited to the per-iteration overheads of the TLS system. But, parallelizing across multiple node removals and heap updates requires each thread to synchronize the reading of any node (lines 8, 9, 15, 18, 20, 22) with the possible updates of that node by the previous threads (line 22). The top node will always require synchronization, while nodes at lower levels will conflict across threads with a decreasing likelihood at each level.

This example can be parallelized using TLS simply by choosing and specifying the correct loop to parallelize. In this example, changing line 8 to use the special keyword `pwhile` rather than `while` can be used with a fairly simple source-to-source translator to trigger the automatic generation of TLS parallel code [13]. The translator performs several operations. First, it analyses the loop to determine loop-carried dependences, i.e. dependences that span iteration boundaries. In this example,

these can occur for the variables `node`, `last`, `inRes`, and `cnt`, and also for any access to data dereferenced from a pointer. All variables and accesses that can have loop-carried dependences appear in boldface type in Figure 5. Then, it transforms the code so that during every iteration the initial load from and the final store to these variables or to dereferenced pointers occur from or to memory, preventing the data from being register-allocated across iteration boundaries. By forcing data to memory, the transformed source code ensures that the TLS system can detect inter-thread data dependence violations. Meanwhile, all variables without loop-carried dependences are made private to each thread to prevent false sharing and violations. Additionally, for peak performance, the source code is transformed to register-allocate variables having loop-carried dependences in all places other than the first load and the final store in each iteration.

This parallelized TLS code was executed upon a heap comprising the approximately 7800 words in the U.S. Constitution and its amendments. The TLS CMP provides a speedup of 2.6 over a single-processor system with the same, unscaled, realistic memory system. Very little of the difference between the achieved speedup and a "perfect" speedup of 4 is due to not scaling the memory system, as the speedup when both have a perfect memory system is only 2.7. Likewise, the requirement that shared variables not be register-allocated causes only a 2% slowdown, if the code is executed sequentially. This is what we call the base TLS parallelization.

3.3 Ease of TLS Parallelization

The base case illustrates the simplicity of TLS programming and the efficiency of its resultant programs, in contrast to the complexity and overheads of conventional parallelization. Like TLS, conventional parallelization requires that loop-carried dependences be identified. However, once this has been done, the difficult part of conventional parallelization begins.

Accesses to any dereferenced pointer or variable with loop-carried dependences could cause data races between processors executing different iterations in parallel. While synchronization must be considered for each access, to avoid poor performance only accesses that could actually cause data races should be synchronized with each other. However, determining which accesses conflict requires either a good understanding of the algorithm and its use of pointers or a detailed understanding of the memory behavior of the algorithm. Pointer aliasing and control flow dependences can make these difficult. Finally, a method for synchronizing the accesses must be devised and implemented. This typically requires changes in the data structures or algorithms and must be carefully considered to provide good performance. None of this is necessary when parallelizing with TLS.

In this example, one set of accesses that must be explicitly synchronized when using conventional parallelization are the read accesses of the nodes (lines 8, 9, 15, 18, 20, 22) with the possible updates of those nodes by earlier iterations (line 22). To do this a new array of locks could be added, one for each node in the heap. However, this would introduce large overheads. Extra storage would be required to store the locks. Each time a comparison of child nodes and an update of the parent node were to occur, an additional locking and unlocking of the parent and testing of locks for each of the child nodes would need to be done. Furthermore, doing this correctly would require careful analysis. The ordering of these operations would be critical. For example, unlocking the parent before locking the child to be transferred to the parent node would allow for race conditions between processors. Worse yet, these races would be challenging to correct because they would be difficult to detect, to repeat and to understand.

One could attempt a different synchronization scheme to lower the overheads. For example, each processor could specify the level of the heap that it is currently modifying, and processors executing later iterations could be prevented from accessing nodes at or below this level of the heap. While this would reduce the storage requirements for the locks to just one per processor, it would introduce unnecessary serialization between accesses to nodes in different branches of the heap. Another alternative would be to have each processor specify only the node which is being updated, so processors executing later iterations would stall only on accesses to this node. But, locking overheads would still exist in either case, and care would still need to be taken to prevent data races. Alternatively, the choice could be made to completely replace the uniprocessor heap sort with a new algorithm designed for parallelism from the start. But, this would likely be more complex than any solution discussed so far, and the support for parallelism will still introduce overheads into any algorithm that has inter-thread dependences. As this example shows, parallelization without TLS can be much more complex and error-prone than parallelization with TLS. Because the complexity of redesign versus incremental modification becomes greater for larger, more complex programs, its simplicity is even more of a benefit for real-world applications.

3.4 Performance of TLS Parallelization

The base case also illustrates the second point of this section, that parallelization with TLS can often yield better performance than parallelization without TLS [14]. This occurs for two reasons. First, the hardware-assisted automatic detection and correction of dynamic dependence violations reduces communication overheads. Furthermore, it is often possible to speculate beyond potential dependences, eliminating all synchronization stall time when the potential violations do not actually occur. We call this optimistic parallelization. It can be much more efficient than the pessimistic static synchronization used in conventional parallelization, which synchronizes on all possible dependences, no matter how unlikely.

It is worth considering this point further. Very often, TLS can improve the performance of an application that has already been manually parallelized by allowing some optimistic parallelization [10]. Less apparent is that a single-threaded application only incrementally modified using manual TLS parallelization can sometimes provide better performance than an application that has been completely redesigned for optimal parallel performance using only conventional manual parallelization. This is because code optimized for non-TLS parallel performance introduces overhead over uniprocessor code to support low-contention parallel structures, algorithms and synchronization. The advantage that results from this redesign for conventional parallelism can be less than the combined advantages of using TLS and starting with more efficient, optimal uniprocessor code. Given the difficulty of redesigning legacy code and of parallel programming, this can make manual

parallelization with TLS a better alternative than application redesign using conventional manual parallelization.

3.5 Optimizing TLS Performance

We will now cover a variety of methods for achieving better TLS parallel performance. This will allow us to focus on three main points: 1) introducing the reader to the process of parallel programming using TLS, which is substantially different from conventional parallel programming; 2) demonstrating several categories of source code transformations that allow extraction of more of the inherent TLP from applications; and 3) indicating situations in which a minor manual adjustment can substantially outperform the automatic base parallelization. We will show how the programmer can detect and understand sources of performance loss and use this to conduct incremental changes to the original source code to improve performance. This process is repeated until no further TLP can be exposed to the TLS hardware.

First, a programmer conducts the base TLS parallelization, as described above, and then executes the resultant code against a representative data set. The TLS hardware is capable of reporting instances of dependence violations, including data on which processors were involved, the address of the violating data element, which load and store pairs triggered the violation, and how much speculative work was discarded. This data is then sorted by each load-store violation pair. By totaling the cycles discarded for each pair and sorting the pairs by these totals, the causes of the largest losses can be known. Using this ranking, a programmer can better understand the dynamic behavior of the parallel program and more easily reduce violation losses.

Compared to non-TLS parallel programming, parallelization with TLS allows the programmer to more quickly transform a portion of code. The key to this is that TLS provides the ability to easily test the dynamic behavior of speculatively parallel code (while it correctly executes in spite of dependence violations) and get specific information about the violations most affecting performance. The programmer can then focus only on those violations that most hamper performance, rather than being required to synchronize each potentially violating dependence to avoid introducing data races into the program.

Before discussing specific code transformations for performance enhancement, we will summarize the general approach to reducing performance-limiting violations. Typically parallel performance is most severely impacted by a small number of inter-thread data dependences. Moving the writes as early as possible within the less speculative thread and the reads as late as possible within the more speculative thread usually reduces the chance of experiencing a data dependence violation. For loop-based TLS, this corresponds to moving performance-limiting writes toward the top of the loop and delaying performance-limiting reads toward the end of the loop; in the limit, the first load of a dependent variable occurs just prior to the last store, forming a tiny critical region. Furthermore, moving this critical region as close as possible to the top of the loop minimizes the execution discarded when violations do occur. Finally, constructing the loop body to ensure that the critical region always occurs approximately the same number of cycles into the execution of the loop and requires a fairly constant time to complete allows the speculative threads to follow each other with a fixed inter-thread delay without experiencing violations. In contrast, critical sections that occur sometimes early and sometimes late increase violations due to late stores in less speculative threads violating early reads in more speculative ones.

3.6 Automatic Optimization

We will now consider optimizations that can be done automatically. More than three violations per committed thread occur while executing the base parallelization. The store of `last` in line 15 often violates the speculative read of `last` in line 9. The same occurs with `cnt` (the store in line 17 violates the load in line 13), `inRes`, and several other variables. To reduce these violations, we can minimize the length of the critical regions from first load to last store. For example, the store of `last` in line 15 can be moved right after the load in line 9. Because each thread optimally executes with a lag of one-quarter iteration from the previous thread on a four-processor CMP, this makes it unlikely that any other thread will be concurrently executing the same critical region. To hoist the store of `last`, the previous value must first be saved in a temporary variable for lines 10 and 11. Research shows that this transformation can be automated [19]. We can also move these critical regions as early in each thread as possible For example, line 17 (the increment of `cnt`) can be moved above the conditional block (lines 9-16). Automatically determining and conducting this is complex [19]. However, we will assume that automated parallelization can conduct all these transformations optimally to strengthen the argument that manual TLS programming can still further improve performance.

When these transformations have been completed for all variables that can benefit, surprisingly the performance remains virtually unchanged. Upon inspecting the violation report, we see that most of the lines which were causing violations before are no longer significant sources of losses, but now previously unimportant load-store violation pairs dominate performance by causing much larger losses than before. Threads now progress farther per violation, but nonetheless violate anyway before they can successfully commit. This results in a lower violation count, but more discarded execution time per violation. This is shown in Figure 6, which shows speedup results with real and perfect memory systems and the number of violations per committed thread, for each version of the example application.

Unfortunately, the performance at this point (a speedup of 2.6) represents an optimistic upper bound on the current capability of automated TLS parallelization. We have optimally used all the automated methods of which we are aware that can

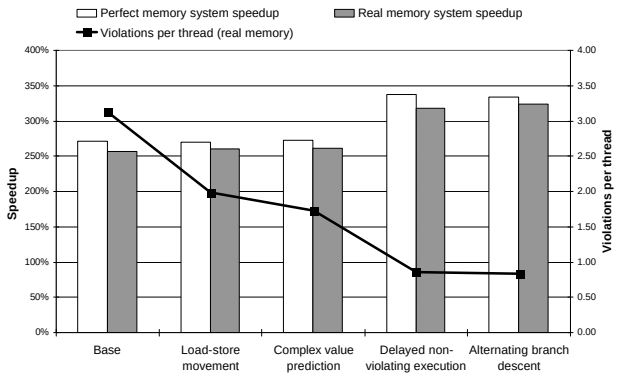

Figure 6. Performance of incremental optimizations

benefit this example. However, manual TLS parallelization can provide still more speedup (a final speedup of 3.4) with a minimum of code transformation. This is because a programmer can do more complex value prediction than an automated parallelizer. Also, automated parallelization is constrained to allow only transformations that appear to preserve the original execution ordering and data structures, even if a minor, obvious change in them could enhance performance. This arises because the original program was targeted to a uniprocessor, where data contention or value prediction was never an issue, so often a small and obvious change can lessen contention or reduce dependences.

The techniques to be discussed below require an increasingly detailed understanding of the application. However, it should be noted again that these performance optimizations are optional and for improved performance only. Unlike conventional manual parallelization, TLS programming does not require a thorough understanding of the application to ensure correctness.

3.7 Complex Value Prediction

In the current example, one of the main variables suffering violations is `inRes`. Complex value prediction can reduce these violations. Note that the result string is constructed out of fixed width columns. The first column is `COLWID` characters wide and contains the word (lines 10-12). The next column is five characters wide and contains the final count of the number of instances of the string, followed by a carriage return (line 13). From the code a programmer can determine that the final value of `inRes` will always be `COLWID+5+1` characters greater after line 13 than it was in line 10, but an automated parallelizer would have difficulty deciphering this. Using this prediction of the final value of `inRes`, the programmer is able to hoist the final update of `inRes` above the many function calls in lines 10-13, once again reducing both the chance of a violation occurring and the execution time discarded if a violation does occur.

This violation could perhaps be alleviated automatically using a combination of profiling, violation tracking and a stride predictor. A more challenging example would be if the count of instances were printed to a variable, rather than fixed, length field. Complex value prediction could quickly determine the final value of `inRes` based upon the number of digits used to print `cnt`, but this would be difficult to do automatically using a stride predictor.

Likewise, if the count had been printed to a variable-length field, the programmer could have chosen to change the format to a fixed length to allow for complex value prediction. This could occur if the output format was not critical and could tolerate a change. If so, this would also show how a small change in the algorithm and data structures can allow further optimization on a program exhibiting contention due to its having been designed without parallel execution in mind. This change would not generally be allowed for an automatic parallelizer.

3.8 Algorithm Adjustments

By this point almost all loads and stores to the same variable are placed close to each other and close to the top of each iteration, and yet the performance has not improved significantly. Upon closer examination, we see that many of the violations would never occur if each thread did not execute lines 9-16 and if it maintained a spacing of one quarter iteration from the threads immediately previous to and following it. The problem is that when lines 9-16 are executed, a large number of cycles are consumed to store a word and its count to the result string. Only after completing this, the thread updates the top of the heap (line 22). This violates all more speculative processors, due to the load in line 8, and causes them to discard all their execution during the time the result string was being updated. While conducting an early update on the top node of the heap could yield some benefit, nodes further down would still likely cause violations.

The optimization to alleviate this problem is to move as much of the execution in lines 9-16 to the position following line 23. By minimizing the work conducted before lines 17-23, we can reduce or eliminate many of the violations. In particular, only the updates of data locations with loop-carried dependences should occur before line 17, i.e. updates to `inRes`, `last` and `cnt`. The `strcpy`, `strlen`, `memset` and `sprintf` functions can be conducted later, after lines 17-23, without causing violations. This is similar to moving load-store pairs closer to the start of each iteration, but instead we are making algorithm changes to move non-violating work closer to the end of each iteration. Specifically, we are moving these four functions from before to after the heap update, which repeatedly dereferences dynamically determined pointers. It may be obvious to the programmer that `resultString` and the heap are never intended to have intersecting addresses; hence no violations should occur. However, it appears the compiler would need to conduct either an advanced general analysis or an analysis quite specific to this situation to assert this non-intersection in all cases. Therefore, this is a change in algorithms that may not be possible for an automated compiler to conduct. As Figure 6 shows, this optimization greatly improves performance, raising the speedup from 2.6 to 3.2 and also halving the number of violations.

After this optimization, we observe that the dominant remaining violations are the loads in line 20 with the store in line 22. We observe that this is in part due to the fact that when the two child nodes point to equal strings (a common occurrence at the top of the heap), the second (right, higher-index) node is always selected. This leads to frequent contention for all nodes near the top of the heap and resultant violations, as each thread descends down the same path through the heap.

We can easily change the algorithm so that each speculative thread chooses the opposite direction from the thread immediately before it. Consecutive threads will alternate between always selecting the left or always selecting the right node in cases of equality, thereby descending down the opposite path from the immediately previous thread. Again, a parallelizing compiler could not make this change, because it alters the behavior of the program, even though in this case the program will still produce exactly the same final result string. This final optimization results in slightly improved performance and less frequent violations. Note that including all the transformations so far would yield a 4% slowdown if the code were executed on a uniprocessor. Hence, a perfect, linear speedup on this code would correspond to a speedup of only 3.85 versus the original sequential program.

Further attempts at optimization were unsuccessful. Yet, violations do remain, because they occur infrequently enough that their losses are less than the overheads of reducing them. For example, attempts at synchronizing on the most frequent

violations, using locks similar to those used in conventional parallelization, generated excessive waiting times. This supports the assertion that TLS parallelization often performs better than manual parallelization without TLS due to its optimistic execution of code that only occasionally causes violations.

3.9 Additional Automatic Optimizations

Several additional automatic techniques exist for improving TLS parallel performance. These were not used in the heap sort example, but will be discussed briefly here for completeness; more details can be found in [3][5][6][8][12][14][15][17]. The techniques comprise loop chunking, loop slicing, parallel reductions and explicit synchronization. Loop chunking refers to unrolling multiple small loop iterations to form each TLS iteration, usually to amortize per-iteration overheads. Loop slicing is the opposite, i.e. splitting each large iteration into multiple, more manageable ones, a technique that represents a simple and automatically transformable case of speculative pipelining described below. Parallel reduction transformations allow certain iterative functions to be parallelized. For example, iterative accumulations into a single summation variable could be instead transformed into four parallel summations that are combined at the end of the loop. Explicit synchronization works much like locks in conventional parallelization to protect a variable and can be used on a frequently violated variable to reduce the violation frequency and the associated discarding of execution [3][5][12]. Unlike its use in conventional parallelization, it is used for performance and not correctness. If the violation data for a TLS parallel application indicates that a read is frequently violated by a write from a less speculative thread, then these two instructions can be explicitly synchronized. By eliminating frequent violations, it trades a large quantity of discarded execution time for a smaller quantity of waiting time.

```
for (x=0; x<1000; x++) {
   if ((x%10) == 0)
      for (y=0; y<10; y++)
         InnerLoopOneThousandCyclesOfWork();
   OuterLoopOneThousandCyclesOfWork();
}
```

Figure 7A. Original code with independent tasks

```
shared_threadChoice = shared_y = 0;
for (shared_x=0; shared_x<1000;) {
    threadChoice = shared_threadChoice;
    x = shared_x;
    if ((x%10) == 0) {
        y = shared_y++;
        if (y == 0) {
            threadChoice=shared_threadChoice=1;
        } else if (y >= 10) {
            threadChoice=shared_threadChoice=0;
            shared_x++;
            shared_y = 0;
        }
    } else
        shared_x++;

    switch (threadChoice) {
        case 0:
            OuterLoopOneThousandCyclesOfWork();
            break;
        case 1:
            InnerLoopOneThousandCyclesOfWork();
            break;
    }
}
```

Figure 7B. Speculative pipeline ready for loop-only TLS

Table 3. Benchmark characteristics

Benchmark		Application category	Lines of code
CFP 2000	177.mesa	3-D graphics library	61,343
	179.art	Image recognition/neural networks	1,270
	183.equake	Seismic wave propagation simulation	1,513
	188.ammp	Computational chemistry	14,657
CINT 2000	175.vpr	FPGA circuit placement and routing	17,729
	181.mcf	Combinatorial optimization	2,412
	300.twolf	Place and route simulator	20,459

3.10 Speculative Pipelining

Finally we will describe one other very important code transformation, speculative pipelining. Until now, we have focused on single-level, loop-based speculation, because loops are an obvious and easy form of parallelism to extract and because the TLS software speculation overheads for single-level loop-only speculation are low. However, TLS can also extract parallelism from tasks that are not associated with a single loop. For example, Figure 7A shows how parallelism can exist at multiple levels within a set of nested loops, making single-level parallelization suboptimal. Here we assume that each of the thousand-cycle routines is a fairly independent task. If only either the outer loop or the inner loop is parallelized using single-level parallelization, half the TLP that exists will not be extracted.

Similarly, TLP can exist between a procedure and the code following the procedure call. In the past this has been exploited with procedural speculation [13], but speculative pipelining can extract this parallelism with lower overhead. Finally, fairly independent, sequential tasks that are not part of a loop can be parallelized. This is similar to the TLS conducted by Multiscalar [12][21], but because the programmer explicitly selects the parallel tasks and the TLS hardware support is less closely coupled to the processor cores, speculative pipelining focuses on longer threads. In some cases, speculative pipelining can be automatically applied (loop slicing, procedural speculation), but in other cases the technique must be conducted manually

In speculative pipelining we break the dynamic execution path of a uniprocessor program between fairly independent tasks and make each task an iteration of a newly constructed loop. To do this, we create a loop shell that chooses between the tasks each iteration by using a switch-case statement directed by a dynamically updated thread-choice variable. Figure 7B demonstrates how multi-level speculation can be implemented. The outer loop body is represented by `case 0` and the inner loop body by `case 1`. The selection between them is made by the `threadChoice` variable, which is updated each time program flow switches between executing iterations of the outer loop and the inner loop. New shared variables allow each thread to update early the next thread's value of `x`, `y` and `threadChoice` while maintaining a private copy of the variables to be used for conducting the thread's remaining execution.

The overhead of speculative pipelining is very small; this example has less than 12 extra dynamic instructions per thread, or roughly 1% overhead. But, speculative pipelining allows great flexibility in constructing threads. Unlike regular loop-based speculation, it can create threads that start and end in different

Table 4. Code transformations

Transformation	SPEC CFP2000				SPEC CINT2000		
	177 mesa	179 art	183 equake	188 ammp	175 vpr	181 mcf	300 twolf
Loop chunking/slicing		X	X	X		X	
Parallel reductions		X			X	X	X
Explicit synchronization					X		X
Speculative pipelining				X	X		X
Adapt algorithms or data structures					X		X
Complex value prediction					X		X

functions or that are from portions of the program that do not iterate at all. As a result, speculative pipelining is one of the most powerful and difficult techniques for enhancing TLS performance.

4. SPEC CPU2000 BENCHMARKS

4.1 Benchmark Selection and Simulation

The SPEC2000 benchmark suite was chosen for this study, as it contains a selection of applications that are representative of CPU-intensive workloads executed on high-performance processors and memory systems. All four floating point applications coded in C were selected, as they were expected to be more challenging to parallelize than the Fortran benchmarks. Three integer benchmarks were selected on the basis of source code size and indications of amenability to manual parallelization, such as a concentration of execution time within a small number of functions. While a few of the other integer benchmarks look amenable to manual parallelization, it is clear that several would be very difficult or impossible to manually parallelize without an extensive understanding of the algorithms and data structures in use. Information on the selected benchmarks is given in Table 3.

We utilized the reference input data sets. Due to the long execution times of these data sets, complete execution was not possible for any of the benchmarks. Since previous research on SPEC benchmarks [16] has demonstrated both the difficulty and the importance of choosing carefully the portion of execution to simulate for applications that exhibit large-time-scale cyclic behavior, we followed the recommendation to simulate one or more whole application cycles. The total of all simulation samples was at least 100 million instructions from each original (non-parallelized) application. One should note that all speedup and coverage results presented below are based upon an extrapolation of these samples of whole application cycles back to the entire application. The extrapolation was conducted by first profiling the full application using similar real hardware and the same compiler as the Hydra CMP. Full application speedup was then calculated assuming the simulated speedup on the portion of execution time corresponding to the application cycles, and assuming no speedup on the portion of the original execution time that was not a part of the application cycles we sampled.

4.2 Results of Parallelization

In this section we will present results from simulations of our benchmarks once the transformations discussed above were performed. We will discuss the performance and the programmer effort required for parallelization. We will also characterize the threads and the reasons for performance losses in the TLS system.

Table 5. Speedup from each additional transformation

Application		Last transformation applied	Cumu-lative speedup	Incre-mental speedup
CFP 2000	177. mesa	Basic	115%	
	179. art	Basic	48%	
		Parallel reductions	104%	38%
		Loop chunking/slicing	135%	15%
	183. equake	Basic	134%	
		Loop chunking/slicing	143%	4%
	188. ammp	Basic	35%	
		Speculative pipelining	52%	13%
		Loop chunking/slicing	60%	5%
CINT 2000	175. vpr (place)	Basic	7%	
		Complex value prediction	55%	44%
		Parallel reductions, explicit synchronization	111%	36%
	175. vpr (route)	Basic	0%	
		Speculative pipelining	16%	16%
		Algorithm/data structure changes	48%	28%
		Complex value prediction	63%	10%
	181. mcf	Basic	38%	
		Loop chunking/slicing	45%	5%
		Parallel reductions	47%	1%
	300. twolf	Basic	0%	
		Speculative pipelining	18%	18%
		Parallel reductions, explicit synchronization, algorithm/data structure changes	39%	18%
		Complex value prediction	77%	27%

Each application was initially parallelized using base parallelization of loops and automatic load-store placement. Table 4 lists the additional transformations that were then used. The first three are simple and can be automated; the second three are complex, requiring manual programming.

The data demonstrate that the simple transformations are beneficial for both floating point and integer applications. However, the complex ones are beneficial mainly for the integer applications. This was because the execution times of the floating point applications were all dominated by easily parallelizable loops, except for `ammp`. Therefore, the complex transformations added little or no benefit. In contrast, all the integer applications benefited from the code transformations, and two of the three benefited from complex ones. Notably, explicit synchronization was not very valuable, enhancing performance for just two applications and both times only when used in combination with some other technique. This is for two reasons. First, it does not work well for infrequent violations, as discussed in Section 3.9. Second, many of the violations typically prevented by explicit synchronization are instead better eliminated through the use of complex methods that do not cause serialization.

Table 6. Speculative thread lengths, regions and coverage

Application		Dynamic thread length (instructions)	Number of speculative regions	Percent execution time coverage
CFP 2000	177.mesa	7,735	1	84%
	179.art	453	7	95%
	183.equake	1,055	6	100%
	188.ammp	140	1	86%
CINT 2000	175.vpr (place)	5,061	1	100%
	175.vpr (route)	1,309	1	97%
	181.mcf	238	5	91%
	300.twolf	784	1	100%
Column mean		2,097	3	94%

Table 5 details the speedups achieved for each application as the transformations were sequentially added. Ideally, the incremental speedup due to each transformation could be listed. However, the transformations interact with each other. For example, on `vpr (place)` explicit synchronization yielded no speedup after base parallelization with additional value prediction. However, applying it together with the parallel reduction transformation provided a sizeable advantage. Due to the interactions and the many permutations of transformations, we have instead listed the speedups along the single path of transformations we actually followed. Note that because `vpr` is a place and route application and the two portions of the application are very different, we have listed results for them separately.

Table 5 more clearly highlights that simple transformations parallelize floating point applications well, but that integer applications require complex transformations. In fact, most if not all of the speedup for each floating point application is already realized using only basic parallelization, while the opposite holds true for the integer applications, which often require complex transformations to get any significant speedup at all.

Figure 8 shows the speedups that were achieved using three TLS/memory systems. The first is a realistic system, the second assumes a perfect memory system and the third assumes a perfect memory system with a zero-overhead TLS implementation. The

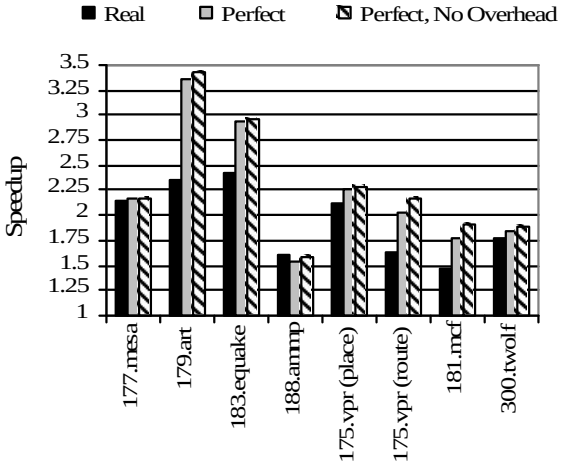

Figure 8. Speedups with real memory, perfect memory and perfect memory with no TLS overhead

Table 7. Breakdown of parallelized execution times

Application		Useful	Discarded	Waiting	Overhead
CFP 2000	177.mesa	70%	28%	2%	0%
	179.art	98%	0%	1%	1%
	183.equake	78%	16%	5%	1%
	188.ammp	50%	40%	6%	4%
CINT 2000	175.vpr (place)	63%	36%	0%	1%
	175.vpr (route)	47%	35%	10%	8%
	181.mcf	65%	24%	6%	5%
	300.twolf	55%	23%	20%	2%
Column mean		66%	25%	6%	3%

average (arithmetic mean) floating point speedup with the real memory system is 2.1, and the average integer speedup is 1.7. Comparison of these speedups with those generated by previous studies on automatic parallelization with TLS is difficult, due to the different architectures, compilers and execution segments utilized. Since TLP is mostly orthogonal to ILP, a rough comparison of speedups can be done using systems with different processor cores and compilers, but different memory systems will still affect the results. With these caveats, a comparison with results from [17][18][21] indicates that the manual parallelization has provided very good parallel performance, well in excess of automatic extraction of TLP at similar thread granularities.

The results for the realistic and perfect memory systems in Figure 8 indicate a sensitivity to memory system delay that varies, with some application speedups fairly insensitive to the characteristics of the memory system and others more strongly affected. The perfect memory system results usually provide an upper bound on the performance that can be achieved by scaling the memory system with the number of processors. An unusual exception occurs for `ammp`, because the faster memory system causes a large number of violations on a load-store pair that would otherwise have experienced far fewer violations. Likewise, the results for the perfect memory systems with and without speculation overheads indicate the performance losses caused by the use of a TLS system with speculation software handlers. These results indicate that fully hardware-based speculation would improve performance fairly little for these applications.

Table 6 characterizes the speculative threads created within each application. Thread sizes span almost two orders of magnitude. The number of distinct speculative regions is small, demonstrating that for many representative applications a large portion of the total execution time can be parallelized by selecting only a few locations in the code. The parallel coverage of the original sequential execution time is uniformly high, even for the integer applications. Parallel coverage typically increases as more sophisticated transformations are applied to the applications. Similarly, thread lengths can also increase safely as violations are reduced. Because longer threads expose more work to losses from violations, as violations become less frequent, thread lengths can be safely increased, for example, by loop chunking. This reduces speculation overheads and the serialization enforced by the in-order commit at the end of each thread. Correspondingly, applications for which the TLS thread lengths are large and the parallel coverage high tend to have good speedups. But, if either quality is absent, then the performance will usually be substantially diminished. Amdahl's Law explains why coverage

Table 8. Lines of code added to parallelize applications

Application		Original lines	Lines added	Percent added	Prog. hours required
CFP 2000	177.mesa	61,343	20	0%	33
	179.art	1,270	140	11%	24
	183.equake	1,513	130	9%	18
	188.ammp	14,657	130	1%	107
CINT 2000	175.vpr	17,729	160	1%	102
	181.mcf	2,412	120	5%	165
	300.twolf	20,459	320	2%	112
Column mean			146	4%	80

must be high, while the poorer performance for small thread lengths can be explained by their correlation with high violation rates and greater speculation overheads and commit serialization.

Table 7 provides the breakdown of execution times spent in the parallelized sections of code. The useful work done is generally quite high, the TLS system overhead is negligible, and violations (discarded time) waste over four times as many cycles as load imbalances (waiting time). The remaining violations tended to be due to variables that had frequent accesses distributed amongst the threads, where each access unpredictably caused a violation a small percentage of the times it dynamically occurred. This prevented any benefits from explicit synchronization, because it causes too much execution serialization. Likewise, these qualities would prevent the dynamic dependence detector described in [3] from providing any benefit, although the one proposed in [12] could work, but only if the dependence distances defined in their paper could be used to develop a reliable dependence predictor for these specific dependences. Dynamic load imbalance, due to size-mismatched threads that must wait to be committed in order, is the source of the waiting losses. Both `vpr (route)` and `twolf` show large losses due to load imbalances. This is especially a problem for applications that have been parallelized with small thread sizes.

Part of the useful work done includes the execution of the additional instructions required for the parallel transformations and to support the interprocessor communication and control. In general, the programmer must make a choice between the cost of supporting each additional transformation and the cost of the violations that occur from not using it, instead. These extra instructions limit the maximum speedup, even though for these benchmarks the losses due to the extra work were fairly small.

Table 8 provides an indication of the programmer effort required for the parallelization of these benchmarks. It lists the number of lines of code added and the total number of hours spent analyzing, parallelizing and debugging each application. While the hours required are highly dependent on the capabilities of the programmer, these data provide at least an order-of-magnitude gauge of programmer effort involved, and no better metric is apparent. We only counted lines of code that were new and unique, or at least substantially changed. We did not count lines of code that were changed or added to implement the automatic base parallelization, i.e. we did not count lines of code that were effectively replicated from the original application or lines of template code that were inserted purely to support the simulator.

The number of lines of code added remains fairly small and constant across applications, almost always less than two hundred. For these applications, the number of lines required has little correlation with the size of the application. However, this may not hold true for larger, more complex applications, which may require the parallelization of more speculative regions, i.e. loops. The number of hours required to parallelize each application was also quite small, in comparison to the number of hours that were originally required to develop them. This strongly supports our claim that manual parallelization with TLS allows programmers to code for a uniprocessor target in a straightforward way, and then with minimal effort port the entire optimized application to a TLS CMP platform to realize good parallel performance.

TLS parallelization depends primarily on an application's algorithms and source code and the TLS and memory systems, rather than the processor architecture. Therefore, the final parallelized application should port easily to other CMP systems that support loop-based TLS and explicit synchronization. For equivalent speedups, they should have low interprocessor communication delays, low speculation overheads and similarly sized caches and write buffers. A CMP with fewer processors will generally not require significant code changes in order to run efficiently, but more processors may necessitate modifications to use all the processors. This will be examined in future research.

5. RELATED WORK

Research on automatic parallelization [2][9] and speculation [4][13][14][17][21] is underway at various universities. Several projects share our focus on general purpose applications. However, they primarily investigate parallelization that can be automated, while in this paper we use techniques that cannot be easily automated to explore the full potential of TLS. The Wisconsin Multiscalar team achieves excellent speedups on general purpose applications, including integer applications [12][21]. However, Multiscalar allows register-to-register communication between the processors at the cost of more complex and high-speed hardware. So, their research explores a different hardware/software design space, generally utilizing finer-granularity threads. Research by the CMU STAMPede team [17][18][19] and at the University of Illinois at Urbana-Champaign [4][5][20] explores different design points with less closely coupled processors, more similar to our TLS CMP.

Relevant research done by Rauchwerger, Padua and Amato considered software-based schemes of speculation and parallelization [15], while later work in conjunction with Zhang and Torrellas utilized hardware support, as well [20]. Other studies [4][18] have focused on achieving highly scalable parallelization. These studies differ from ours in that they either focus on using software only, or on using hardware support specific to the code transformation applied, i.e. hardware for conducting reductions or for achieving scalable speedups. Also, much of their research has centered on scientific, floating-point-intensive Fortran applications, while our research considers both floating point and integer programs that are all written in C.

Finally, substantial work on exploiting value prediction and dynamic synchronization has been conducted in [3][5][6][8][12][17]. We incorporate the benefits of these studies where possible and extend upon them. For example, the earlier value prediction studies explore only predictions of values that do not change or are in a simple stride. In this study, we explore predictions of values that evolve in a more complex manner.

6. CONCLUSION

In this paper, we described the way in which TLS manual programming is done and three of the most useful manual TLS code transformations: complex value prediction, data structure /algorithm changes and speculative pipelining. These techniques were applied to several applications in SPEC CPU2000 to assess the performance and difficulty of using TLS on well-known processor benchmarks. While simple (automatic) transformations were useful for many applications, complex transformations were able to provide further large performance benefits. This was especially true for integer applications, some of which would have experienced no significant speedup with only automatic parallelization. We also show that real-world applications can be parallelized with very little effort with manual TLS programming.

Our experience shows that TLS can dramatically reduce the programmer effort required for application parallelization, while yielding performance gains similar to, if not exceeding, those obtainable using conventional manual parallelization. This enables a new approach to parallel programming. In this paradigm, the majority of the programming effort can focus on conventional single-threaded application design, with a relatively small effort at the end to port the application to a multiprocessor platform using manual parallelization with TLS.

7. ACKNOWLEDGMENTS

This work was supported by Air Force contract F29601-01-2-0085, NSF contract CCR-0220138, the Intel graduate fellowship program and the Alliance for Innovative Manufacturing at Stanford. The authors would also like to thank Lance Hammond for extensive discussions and key insights on this paper and support of the Hydra CMP simulator.

8. REFERENCES

[1] V.S. Adve, et al., "An integrated compilation and performance analysis environment for data parallel programs," *Supercomputing 1995*, San Diego, California, pp. 1370-1404, Nov. 1995.

[2] B. Blume, et. al, "Restructuring programs for high-speed computers with Polaris," *Proc. 1996 ICPP Workshop on. Challenges for Parallel Processing*, pp. 149-161, Aug. 1996.

[3] G.Z. Chrysos and J.S. Emer, "Memory dependence prediction using store sets," *Proc. 25th Annual Intl. Sym. on Computer Architecture (ISCA)*, Barcelona, Spain, pp. 142-153, June 1998.

[4] M. Cintra, J. Martínez and J. Torrellas, "Architectural support for scalable speculative parallelization in shared-memory multiprocessors," *ISCA-27*, Vancouver, Canada, pp. 13-24, June 2000.

[5] M. Cintra and J. Torrellas, "Eliminating squashes through learning cross-thread violations in speculative parallelization for Multiprocessors," *Proc. 8th Intl. Sym. on High-Performance Computer Architecture (HPCA)*, Cambridge, Massachusetts, Feb. 2002.

[6] F. Gabbay and A. Mendelson, "Using value prediction to increase the power of speculative execution hardware," *ACM Transactions on Computer Systems*, vol. 16, pp. 234-270, Aug. 1998.

[7] S.W. Keckler et al., "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," *ISCA-25*, Barcelona, Spain, pp. 306-317, June 1998.

[8] K.M. Lepak, G.B. Bell, and M.H. Lipasti, "Silent stores and store value locality," *IEEE Transactions on Computers*, vol. 50, pp. 1174-1190, Nov. 2001.

[9] S.W. Liao, et al., "SUIF Explorer: An Interactive and Interprocedural Parallelizer," *Proc. Sym. PPOPP 1999*, pp. 37-48, Atlanta, Georgia, Aug. 1999.

[10] J.F. Martinez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," *Proc. 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, Oct. 2002.

[11] B.P. Miller, et al., "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer*, 28(11):37-46, Nov. 1995.

[12] A. Moshovos, S.E. Breach, T.N. Vijaykumar, G.S. Sohi, "Dynamic speculation and synchronization of data dependences," *ISCA-24*, Denver, Colorado, pp. 181-193, June 1997.

[13] K. Olukotun, L. Hammond, and M. Willey, "Improving the performance of speculatively parallel applications on the Hydra CMP," *Proc. 13th ACM International Conference on Supercomputing (ICS)*, Rhodes, Greece, pp. 21-30, June 1999.

[14] C.-L. Ooi, et al., "Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor," *ICS-15*, June 2001.

[15] L. Rauchwerger, N. Amato, and D. Padua, "Run–time methods for parallelizing partially parallel loops," *ICS-9*, Barcelona, Spain, pp. 137-146, July 1995.

[16] T. Sherwood and B. Calder, "Time varying behavior of programs," *Tech. Rep. No. CS99-630*, Dept. of Computer Science and Eng., UCSD, Aug. 1999.

[17] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "Improving value communication for thread-level speculation," *HPCA-8*, Cambridge, Massachusetts, Feb. 2002.

[18] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A scalable approach to thread-level speculation," *ISCA-27*, Vancouver, Canada, pp. 1-12, June 2000.

[19] A. Zhai, C.B. Colohan, J.G. Steffan, and T.C. Mowry, "Compiler optimization of scalar value communication between speculative threads," *ASPLOS-10*, San Jose, California, Oct. 2002.

[20] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors," *HPCA-5.*, Orlando, Florida, pp. 135-141, Jan. 1999.

[21] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," *ISCA-28*, Goteborg, Sweden, pp. 2-13, July 2001.