# Exploiting Speculative Thread-Level Parallelism in Data Compression Applications

Shengyue Wang, Antonia Zhai, and Pen-Chung Yew

Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, USA
{shengyue, zhai, yew}@cs.umn.edu

**Abstract.** Although hardware support for Thread-Level Speculation (TLS) can ease the compiler's tasks in creating parallel programs by allowing the compiler to create potentially dependent parallel threads, advanced compiler optimization techniques must be developed and judiciously applied to achieve the desired performance. In this paper, we take a close examination on two data compression benchmarks, GZIP and BZIP2, propose, implement and evaluate new compiler optimization techniques to eliminate performance bottlenecks in their parallel execution and improve their performance. The proposed techniques (i) remove the critical forwarding path created by synchronizing memory-resident values; (ii) identify and categorize reduction-like variables whose intermediate results are used within loops, and propose code transformation to remove the inter-thread data dependences caused by these variables; and (iii) transform the program to eliminate stalls caused by variations in thread size. While no previous work has reported significant performance improvement on parallelizing these two benchmarks, we are able to achieve up to 36% performance improvement for GZIP and 37% for BZIP2.

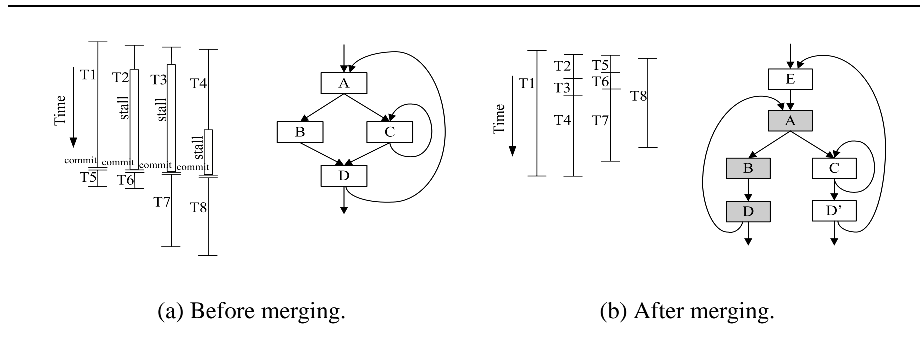
## 1 Introduction

Chip Multiprocessors (CMP) have become nearly commonplace [2, 14, 17, 32]. It is relatively straightforward for explicitly multithreaded workloads to benefit from the increasing computing resources, but how would sequential programs take advantage of such resources? One natural way to speedup a sequential program is to exploit parallelism to utilize multiple processing units. Traditionally, compilers create parallel programs by identifying *independent* threads [4, 13, 33]—but this is extremely difficult, if not impossible, for many general purpose programs due to their complex data structures and control flow, as well as their use of ambiguous pointers. One promising alternative to overcome this problem is *Thread-Level Speculation* (**TLS**), which allows the compiler to create parallel threads without insisting on their independence. The underlying hardware ensures that inter-thread dependences through memory are satisfied, and re-executes any thread for which they are not. Unfortunately, despite numerous proposals on efficient hardware support [1, 7, 9–12, 19, 20, 24, 25, 29, 31, 34] and compiler optimizations [22, 38–40], only moderate performance improvements have been reported from parallelizing general purpose programs. This calls for the development of more

aggressive compiler optimizations that take full advantage of profiling information to perform novel program transformations.

In this paper, we focus on one important class of general purpose applications, data compression, for which most of the previously proposed techniques are unable to claim significant performance improvement. Starting with a parallelized and optimized version of BZIP2 and GZIP—the parallel loop selection and value communication optimization algorithms are described in our previous work [38–40], we carefully studied the program behaviors. Three performance bottlenecks are identified and the corresponding optimization techniques are proposed.

1. The first performance bottleneck that we observed is the critical forwarding path introduced by forwarding *memory*-resident values between threads. In our previous work [40], we have demonstrated the importance of forwarding values between speculative threads to statisfy inter-thread data dependence of *memory*-resident values and to avoid speculation failure. However, the critical forwarding path introduced by such synchronization can serialize paralle threads. In our previous work, we have proposed a compiler optimization technique to address a similar issue—reducing the critical forwarding path introduced by communicating *register*-resident values. In this paper, we apply the same instruction scheduling technique to reduce the critical forwarding path introduced by communicating *memory*-resident values. We observe that, to reduce the critical forwarding path of a *memory*-resident value, instructions *must* be scheduled aggressively—across both control and data dependences to achieve performance improvement. On the contrary, in the case of *register*-resident value, conservative instruction scheduling provides most of the performance benefit. Details are described in Section 3.1.
2. The second performance bottleneck is also due to inter-thread value communication. This bottleneck is caused by a class of reduction-like variables: where the variable is defined in the loop body through reduction operations, but there also exist *use*s of the intermediate result of this variable, thus it is impossible to apply traditional reduction optimizations [18] to eliminate the inter-thread data dependence. We propose an aggressive speculative reduction transformation to reduce the critical forwarding path caused by reduction-like variables. Details of this technique is described in Section 3.2.
3. The third performance bottleneck is caused to complex control flow—threads can take any execution path through an iteration, and thus vary in execution time. In the example shown in Figure 1(a), there are 8 parallel threads with the following execution order $T1, T2, T3, T4, T5, T6, T7$, and $T8$. The size of long threads $T1, T4, T7$, and $T8$ contains thousands of dynamic instructions and short threads $T2, T3, T5$, and $T6$ contain only a few instructions. Assuming 4 threads are executed in parallel, little parallel overlap is possible. One way to get parallel overlap is to merge short threads with long threads and execute consecutive long threads in parallel, as shown in Figure 1(b). Unfortunately, it is impossible to statically determine the number of iterations to merge since thread sizes are not known until runtime. We propose a program transformation to *dynamically* merge multiple short threads with a long thread. The details are described in Section 3.3.




(a) Before merging.

(b) After merging.

**Fig. 1.** Iteration merging.

### 1.1 The TLS Execution Model

In TLS, the compiler partitions a program into parallel speculative threads without having to prove that they are independent, while at runtime the underlying hardware checks whether inter-thread data dependences are preserved and re-executes any thread for which they are not. This TLS execution model allows the parallelization of programs that were previously non-parallelizable as demonstrated by the following example.

In this paper, we will only experiment with parallel threads that are created by executing multiple iterations of the same loop simultaneously . However, we expect the techniques developed for improving value communication applicable to other parallel threads. The most straightforward way to parallelize a loop is to execute multiple iterations of the same loop in parallel. With TLS, loops with potential loop-carried data dependences are speculatively parallelized. A thread is allowed to commit if no inter-thread data dependence is violated. In case of a data dependence violation, the thread that contains the consumer instruction is re-executed.

**Inter-Thread Value Communication in TLS** From the compiler's perspective, TLS supports two forms of communication and the compiler can decide which mechanism is appropriate for a particular data dependence to obtain maximum parallel overlap:

**Synchronization** explicitly forwards a value between the source and the destination of a data dependence. It allows for partial parallel overlap and is thus suitable for frequently occurring data dependences that can be clearly identified. However, if the instructions that compute the communicating value are sparsely located in a thread, explicit synchronization could also limit performance by stalling the consumer threads more than necessary.

**Speculation** relies on the underlying hardware to detect data dependence violations at runtime and trigger re-execution when necessary. It allows for maximum parallel overlap when speculation always succeeds, however, if speculation always fails, this mechanism would introduce a significant performance penalty. Thus, this form of value communication is suitable for data dependences that are difficult to analyze and occur rarely.

### 1.2 Related Work

Researchers have developed various compiler [6, 8, 16, 22, 39, 40] and manual [26, 27] optimization techniques to fully utilize the hardware support for TLS to parallelize general purpose applications. This paper extends our previous work on improving inter-thread value communication [39, 40] and integrates the recover code generation mechanism to enable both inter-thread and intra-thread speculation to avoid processor stalls caused by data dependences from memory-resident values.

Existing research in parallel compilers has mainly focused on two critical performance problems: how to divide a sequential program into parallel threads [3, 8, 16, 22, 37, 38] and how to improve inter-thread value communication [12, 21, 22, 28, 35, 36, 39–41]. These compiler optimization techniques typically start by building a probabilistic model of speculative execution first, and then estimating the amount of parallel overlap that can be achieved. However, few recognized that in real applications dependences are often inter-related and intelligent code transformations could be used to speculate on predictable dependence patterns for performance gain.

Prabhu *et al.* [26, 27] have developed several advanced *manual* code transformations to improve the performance of TLS, and they expect the programmers to apply these techniques. Although some of the techniques described by Prabhu *et al.* resemble the techniques in this paper at a first glance; detailed examinations reveal significant differences: (i) both papers have observed that reduction variables can serialize program execution, but Prabhu *et al.* only applied traditional reduction variable elimination technique to remove them, while we studied the existence of a large class of reduction-like variables with complex usage patterns and develop new code transformations to prevent them from serializing execution; (ii) although both paper has proposed techniques to balance the workload that are assigned to each thread, our iteration merging technique is proposed in the context of automatic compilation and thus can be integrated in an optimizing compiler.

## 2 Compression Algorithms

The compression applications are commonly used general-purpose applications. TLS typically achieves modest speedup for those applications. In order to gain insight on the performance bottleneck and exploit more potential speculative TLP, we select two compression benchmarks BZIP2 and GZIP from SPEC2000 benchmark suite for an extensive study.

**BZIP2** BZIP2 [5] represents one class of compression applications that uses a block-based algorithm. It divides the input data into blocks of the size $N$ ranging from 100k to 900k bytes, and processes the blocks sequentially. While it is possible to process different blocks in parallel, the huge size of the speculative data modified by each thread often exceeds the capacity of speculative buffer provided by TLS, which is typically from 16k to 32k bytes. The frequent stalls due to the buffer overflow inhibit most of the performance gains from TLS.

During the compression of each block $S$ of size $N$, the most time consuming part is Burrows-Wheeler Transform (BWT). It forms $N$ rotations of a block by cyclically shift $S$, and sorts these rotations lexicographically. Bucket sort is used in the main sorting phase. The buckets are organized as a two-level hierarchical structure. The big bucket in the outer level contains all rotations starting with the same character, while the small bucket in the inner level contains all rotations starting with the same two characters.

Consequently, a two-level nested loop is used to traverse each bucket to sort all rotations inside. The outer loop seems an ideal target for parallel execution since sorting of big buckets can be done independently. However, in order to speedup the sequential algorithm, the information about the sorting of the current bucket is kept in the global data structures such as *quadrant* and used in the sorting of following buckets to avoid redundant computations. Also, the results of sorting the current big bucket are used to update other unsorted buckets. As a result, those optimizations for sequential algorithm introduces inter-thread dependences that are undesirable for parallel execution. On the other hand, the performance of the inner loop is mainly limited by the reduction-like variable *workDone*. Reduction elimination cannot be applied here since *workDone* is also used somewhere in the loop besides the reduction operations. The sorting of each small bucket is done by calling *qSort*. Since *qSort* is not always called in every inner loop iteration, it introduces unbalanced load among threads.

The compression algorithm also include other phases such as run-length encoding, move-to-front encoding, and Huffman encoding. The performance of the main loops in those phases are typically limited by long critical forwarding paths that are hard to optimize.

The decompression phase in BZIP2 has much lower coverage than in the compression phase. Similar to compression, decompression is performed for one block at a time. Decompression of multiple blocks cannot run in parallel due to the size limitation of the speculative buffer. Most loops in the decompression phase is sequential due to the fact that the decoding of a character is completely dependent on the previous characters.

**GZIP** GZIP [42] represents another class of compression applications that uses a dictionary-based algorithm. The input data is scanned sequentially, once a repeated string is detected, it is replaced by a pointer to the previous string. A hash table is used for detecting a repeated string. All input strings of length three are inserted in the hash table.

Two versions of the algorithm are implemented. *Deflate_fast* is a simplified version, which is fast but with low compression ratio. The main loop iterates through all input characters. Each time a match is found, it is selected immediately. The main performance limitation is caused by the use of global variables such as *lookahead* and *strstart*. *Deflate*, a more complex and time consuming version, uses a technique called lazy evaluation in order to find a longer match. With lazy evaluation, the match is not selected immediately. Instead, it is kept and compared with the matches for the next input string for a better choice. However, the use of current match in the next matching step causes additional data dependences. Both *deflate* and *deflate_fast* call *longest_match* to find the longest match among all string with the same hash index. The average iteration size of the main loop is typically small due to the facts that most of strings do not match with the current string and a fast check is used to avoid unnecessary comparison.

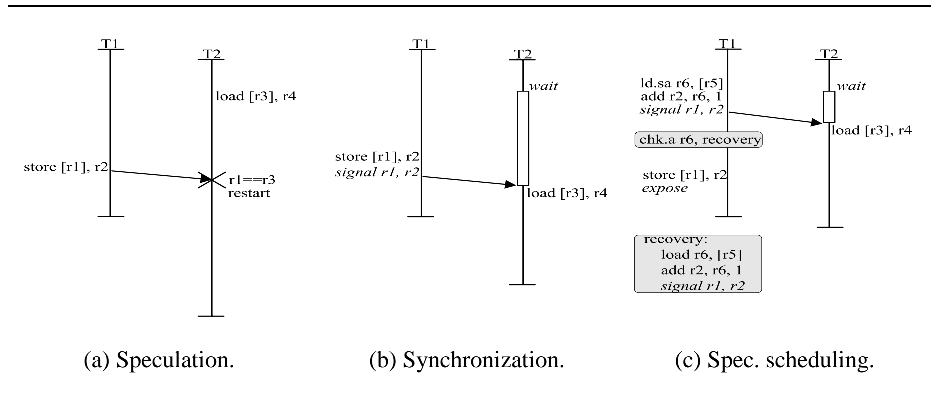
Similar to BZIP2, the decompression phase in GZIP has a much lower coverage than in the compression phase. The decompression is performed sequentially since the decoding of the current character depends on the characters decoded previously. As a result, it is hard to extract TLP in the decompression phase.

## 3 TLS Optimizations

For both BZIP2 and GZIP, the main hurdle to create efficient parallel programs under TLS is data and control dependences. In this session, we propose several compiler optimization techniques to overcome these limitations.

### 3.1 Speculative Scheduling for Memory-Resident Value Communication

In order to avoid excessive failures under TLS, synchronizations are required for frequently occurring *memory* dependences. While scheduling for *register*-resident value communication has been shown effective for many benchmarks [39], the benefit of scheduling for *memory*-resident value communication is still unknown. Due to the facts that *memory* dependences are prevalent in both BZIP2 and GZIP, it is important to investigate the performance impact of scheduling techniques for *memory*-resident value. Unlike the scheduling for *register*-resident value, the scheduling of *memory*-resident value may interact with the underlying TLS support. The details of how scheduling and TLS work together need a closer examination.




(a) Speculation. (b) Synchronization. (c) Spec. scheduling.

**Fig. 2.** Scheduling for memory-resident value communication.

Figure 2(a) shows two threads T1 and T2 with a frequently occurring dependence between *store* and *load*. To avoid mis-speculation, synchronization is used to delay the execution of *load* until *store* finishes its execution, as shown in Figure 2(b). A *signal* instruction is inserted after *store* to explicitly forward both the address (stored in register *r1*) and the value (stored in register *r2*) to T2.

In order to reduce the waiting time of *load*, speculative scheduling is applied so that both the address (stored in register *r1*) and the value (stored in register *r2*) of *store* are

computed earlier. Control and data speculation are used to overcome the dependence limitation during aggressive scheduling, and recovery code needs to be generated for possible mis-speculation as well. In our study, we support both control and data speculation similar to those on IA-64 architecture [15]. As shown in Figure 2(c), in order to compute the value of *r2* earlier, the instructions it depends on have to be scheduled. If *load r6, [r5]* in the dependence chain is scheduled across an aliasing store, it will be changed into a data speculative load (*ld.a*). If it crosses a branch, it will be changed into a control speculative load (*ld.s*). In this example, *ld.sa* is used for both data and control speculation. A check instruction *chk.a* is inserted to the home location of the speculatively scheduled load to detect possible mis-speculations. In case of a mis-speculation, the corresponding recovery code is invoked to re-compute and re-forward the value, as shown in Figure 2(c). When instructions are speculatively scheduled across a branch, a signal with NULL address is inserted to the alternative path. Execution of such signal indicates that a wrong signal from the other path has already been forwarded.

The consumer thread keeps both addresses and values in a special forwarding buffer. It is accessed by each exposed load that is not proceeded by a store to the same location in the same thread. If the address matches, the value will be used. When the consumer thread receives the same signal *twice*, it indicates a mis-speculation is detected by the producer thread, and either the address or value has been wrongly forwarded. The old address and value are replaced by the new ones. The consumer thread has to be squashed if the old value has already been consumed.

The data stored in the forwarding buffer will not be checked for inter-thread dependence violation. In Figure 2(c), if there is another store instruction *store 1* between *signal* and *store*, and it accesses the same address as *store*, it will not cause an inter-thread dependence violation since the forwarded data is invisible to *store 1*. However, if another store instruction *store 2* after *store* accesses the same address, an inter-thread dependence violation should be detected since *store 2* produces a newer value that should be used by *load*. For this purpose, we insert a new instruction *expose* immediately after *store* to inform the consumer thread to make the forwarded data exposed for the dependence checking.

### 3.2 Aggressive Reduction Transformation

A reduction operation iteratively summarizes information into a single variable called the reduction variable. The presence of reduction variables causes inter-thread dependences, and serializes parallel execution. Such serialization can become performance bottlenecks when nested loops are involved. The example in Figure 3(a)-i shows a reduction variable $sum$ defined in a nested loop. During the parallel execution of the outer loop, in thread $i$, the definition in the last iteration of the inner loop is used by thread $i+1$ in the first iteration of the inner loop. This creates an inter-thread data dependence that must be synchronized as shown in Figure 3(a)-ii. However, such synchronization can potentially serialize parallel execution.

In traditional parallelizing compilers [18], reduction variables are eliminated through a process in which multiple independent variables are created and store in an array as shown in Figure 3(a)-iii. Because each thread stores reduction variable in a different

location, inter-thread data dependences are eliminated, thus the threads can be parallelized. The final result of the reduction operation is computed after parallel execution ends.

Although reduction elimination is effective in removing inter-thread dependences, the application of this technique is limited due to one important constraint—intermediate results of the reduction operation cannot be used anywhere in the parallelized loop. In the example shown in Figure 3(b)-i, the intermediate result of the reduction variable $sum$ is used in the outer loop. In order to retrieve the intermediate result, the reduction valuable must be communicated between the parallel threads. Fortunately, we can perform partial reduction elimination and update the value of $sum$ only once in the outer loop, as shown in Figure 3(b)-ii, where all uses of the reduction variable $sum$ in the outer loop is replaced with $sum + sum[i]$. Although the reduction variable is not eliminated, its impact on the parallel performance is greatly reduced, as we can see that the distance between the update of $sum$ in a thread and use of it in the successor thread becomes relatively small. Another benefit of this transformation is that the summation step can be eliminated. Under TLS, because all local scalars are thread-private, we can avoid the creation of the array $sum[]$ and use a scalar to hold the partial results of the reduction operation, as shown in Figure 3(b)-iii. Just like any other values that are communicated through synchronization, the critical forwarding path of this communication can be reduced with instruction scheduling.

Unfortunately, not all usage patterns of reduction variables can be optimized as described above. In the example shown in Figure 3(c)-i, the *signal* instruction cannot be scheduled before the inner loop because it depends on the value of $sum0$ which is computed by the inner loop; and *wait* instruction cannot be scheduled after the inner loop, because it is used to guard a branch instruction. As a result, the critical forwarding path introduced by the reduction variable is very long. Fortunately, the outcome of the branch instruction guarded by the reduction variable is often predictable; and we can exploit this predictability to postpone the use of the reduction variable till after the completion of the inner loop. In the example shown in Figure 3(b)-ii, the branch is predicted as not-taken; moved across the inner loop and executed as a verification. In the original location of this branch, both $sum0$ and $x$ are saved, so that they can be used later in the verification. The use of $sum$ is delayed so that the critical forwarding path is reduced. When the value of $sum$ becomes available, and the branch is proved to be mis-predicted, the thread must be squashed and an un-optimized version of code must be executed [30]. The squash/recovery mechanism that enables this aggressive optimization is already available in TLS, thus no extra hardware support is needed.

However, this aggressive transformation does not handle all usage patterns of $sum$ within the loop: the reduction variable can be used in the inner loop, as shown in Figure 3(c)-iii. In order to reduce the critical forwarding path introduced by such usage, the branch in the inner loop has to be moved to the outer loop. Is it possible to make such a code transformation and to guarantee that all mis-predictions are detected? The answer is yes, and the key to this transformation is that most reduction operations are monotonic. If the reduction variable is monotonically increasing or decreasing and the branch is to test whether it is greater or less than a certain loop invariant, the verification can be delayed till after the inner loop is complete. In our example, if the condition

$sum + sum0 > 100$ is true in the inner loop, it must also be true for the test in the outer loop. Mis-predictions can always be detected by the delayed verification in the outer loop.

---

```
while(cond1) {
   while(cond2) {
      sum++;
   }
}
(i)A reduction variable
```

```
while(cond1) {
   wait(sum);
   while(cond2) {
      sum++;
   }
   signal(sum);
}
(ii)Synchronizing the
reduction variable
```

```
while(cond1) {
   while(cond2) {
      sum[i]++;
   }
}
while(cond1) {
   sum+=sum[i];
}
(iii)Traditional
reduction elimination
```

(a) Reduction elimination.

```
while(cond1) {
   while(cond2) {
      sum++;
   }
   ...
   =sum;
   ...
} (i)Using intermediate
result of reduction
variable
```

```
while(cond1) {
   while(cond2) {
      sum[i]++;
   }
   ...
   wait(sum);
   =sum+sum[i];
   ...
   sum+=sum[i];
   signal(sum);
}
(ii)Reduction transformation
with explicit forwarding
```

```
while(cond1) {
   while(cond2) {
      sum0++;
   }
   ...
   wait(sum);
   =sum+sum0;
   ...
   sum+=sum0;
   signal(sum);
}
(iii)Replace sum[]
with sum0
```

(b) Reduction-like variable with a single use and short critical forwarding path.

```
while(cond1) {
   wait(sum);
   if(sum+sum0>x)
      work1;
   else
      work2;
   ...
   while(cond2) {
      sum0++;
   }
   ...
   sum+=sum0;
   signal(sum);
}
(i)Used to determine
a branch outcome
```

```
while(cond1) {
   sum0'=sum0;
   x'=x;
   work2;
   ...
   while(cond2) {
      sum0++;
   }
   ...
   wait(sum);
   if(sum+sum0'>x')
      recovery;
   sum+=sum0;
   signal(sum);
}
(ii) Predicting
branch outcome
then verifying
```

```
while(cond1) {
   wait(sum);
   while(cond2) {
      sum0++;
      if(sum+sum0>100)
         return;
      work1;
   }
   ...
   sum+=sum0;
   signal(sum);
}
(iii) Used in the
inner loop
```

```
while(cond1) {
   while(cond2) {
      sum0++;
      work1;
   }
   ...
   wait(sum);
   if(sum+sum0>100)
      recovery;
   sum+=sum0;
   signal(sum);
}
(iv) Predicting
branch outcome
then verifying
in the outer loop
```

(c) Using the intermediate result of a reduction variable to determine a branch outcome.

**Fig. 3.** Transformation for reduction-like variable.

### 3.3 Iteration Merging for Load Balancing

In TLS, to preserve the sequential semantics, speculative threads must be committed in order. Thus, if a short thread that follows a long thread completes before the long thread, it must stall till the long thread completes. When workload is not balanced between parallel threads, the waiting time can be significant. One way to achieve more balanced workloads is to merge multiple short iterations with a long iteration, so that multiple iterations of a loop are aggregated into a single thread.

Figure 1(a) shows the Control Flow Graph (CFG) of a nested loop. Each node in the graph represents a basic block. The outer loop is selected for parallelization. The path A->B->D on the left is more likely to be taken than the path A->C->...->C->D on the right. However, the right path is much longer than the left path since an inner loop is involved. This causes the load imbalance problem. A shorter thread finishes its execution much earlier than a longer thread, but it has to wait until the previous longer thread commits its results.

The idea of using iteration merging to solve this problem is to combine multiple consecutive loop iterations to make the workload more balanced. When a short but frequent path is identified, a new inner loop is formed which only contains the part from this short path. As shown in Figure 1(b), the newly formed inner loop, which contains A, B and D from the short path, is marked by the shadowed blocks. For all basic blocks that are reached from the outside of this inner loop, tail duplications are needed in order to eliminate side entries. In this example, block $D'$ is tail duplicated and inserted in the outer loop. A new block E is also inserted to the beginning of the outer loop, and only contains a trivial unconditional branch that transfers the control flow to A. Later a $fork$ instruction will be inserted to E in order to spawn a new thread at runtime. After this transformation, multiple short iterations are combined together with a long iteration, resulting in more balanced workloads among threads.

## 4 Evaluation

We have developed our TLS compiler based on the ORC compiler, which is an industrial-strength open-source compiler targeting Intel's Itanium Processor Family (IPF). Three phases are added to support TLS compilation. The first phase performs data dependence and edge profiling, and feeds profile information back to the compiler. The second phase selects loops that are suitable for TLS [38]. Optimizations for TLS are applied in the third phase. Both the loop selection and the optimization phases extensively use profiles obtained by using `train` input set.

For the optimization techniques proposed in this paper, we have implemented scheduling for memory-resident value communication. Both aggressive reduction transformation and iteration merging are still under development and performed manually for this study.

### 4.1 Simulation Methodology

The compiled multithreaded binary is running on a simulator that is built upon Pin [23] and models a CMP with four single-issue in-order processors. The configuration of our

**Table 1.** Simulation parameters.

| Issue Width | 1 | | |
|---|---|---|---|
| L1-Data Cache | 32KB, 2-way, 1 cycle | Commu. Buffer | 128 entries, 1 cycle |
| L2-Data Cache | 2MB, 4-way, 10 cycle | Commu. Delay | 10 cycles |
| Cache Line Size | 32B | Thread Spawning | 10 cycles |
| Write Buffer | 32KB, 2-way, 1 cycle | Thread Squashing | 10 cycles |
| Addr. Buffer | 32KB, 2-way, 1 cycle | Main Memory Latency | 50 cycles |

**Table 2.** Benchmark statistics.

| Application Name | Input Set | Number of Parallelized Loop | Average Thread Size | Average Num of Threads/Invocation | Coverage |
|---|---|---|---|---|---|
| BZIP2 | program | 11 | 147 | 3692 | 61% |
| | graphic | 11 | 153 | 3929 | 58% |
| | source | 11 | 137 | 4451 | 65% |
| GZIP | program | 5 | 865 | 1519 | 89% |
| | graphic | 5 | 231 | 2532 | 80% |
| | source | 5 | 923 | 2549 | 84% |
| | random | 5 | 180 | 61921 | 81% |
| | log | 5 | 1378 | 1303 | 79% |

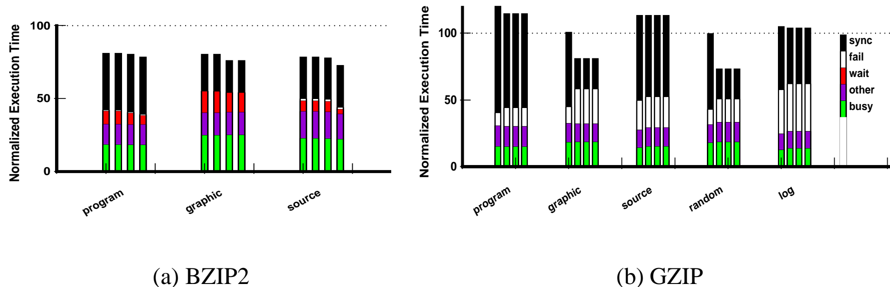
simulated machine model is listed in Table 1. Each of processor has a private L1 data cache, a write buffer, an address buffer, and a communication buffer. The write buffer holds the speculatively modified data within a thread [34]. The address buffer keeps all exposed memory addresses accessed by a speculative thread. The communication buffer stores data forwarded by the previous thread. All four processors share a L2 data cache.

All simulations are performed using the `ref` input set. To save the simulation time, each parallelized loop is simulated up to 1 thousand invocations, and each invocation is simulated up to 0.1 million iterations. Overall, it allows us to simulate up to 4 billion instructions while covering all parallel loops.

### 4.2 Performance Impact

We have evaluated the proposed compiler optimizations using the simulation infrastructure described in the last section. The performance of the parallelized code is measured against a fully optimized sequential version running on a single processor. Since both GZIP and BZIP2 have multiple `ref` input sets, we evaluate both two benchmarks on all input sets. Benchmark statistics are listed in the Table 2 and the program speedup is shown in Figure 4.

Each bar in Figure 4 is broken down into five segments explaining what happens during all cycles. The *sync* segment represents time spent waiting for forwarded values; the *fail* segment represents time wasted executing failed threads; the *wait* segment correspond to amount of time the processor has completed execution and is waiting for previous thread to commit; the *busy* segment corresponds to time spent performing useful work; and the *other* segment corresponds to stalls due to other constraints.




(a) BZIP2 (b) GZIP

**Fig. 4.** Program speedup compared to sequential execution. This figure shows the performance impact of the proposed optimizations on all `ref` input sets.

The baseline performance is obtained when all existing TLS optimization techniques are applied as described in [38]. It is shown in the first bar. Three optimization techniques proposed in this paper are added on top of the baseline in the following order: scheduling for memory-resident value communication, aggressive reduction transformation, and iteration merging. Each time a new technique is applied, the *cumulative* performance is reported. They are shown in the second, third, and fourth bar respectively.

1. Scheduling for memory-resident values has a significant performance impact on GZIP. For `graphic` and `random` input sets, the program performance is improved by 24% and 36% respectively.
2. Reduction transformation that removes the critical forwarding path introduced by reduction variables benefits BZIP2 significantly. For `graphic` input, we saw a 7% performance improvement. This performance improvement mainly comes from the inner loop of bucket sort, as described in Section 2, whose performance is limited by a reduction-like variable $workDone$.
3. Iteration merging can further improve the performance for loops that have unbalanced workloads. The performance of BZIP2 on `source` input is greatly improved by 9% after this technique is applied.

### 4.3 A Sensitivity Study

The statistics for different inputs can be found in the Table 2. In the table, coverage is defined as the fraction of execution parallelized in the original sequential program. The coverage for both benchmarks is high: around 60% for BZIP2 and 80% for GZIP. Thread size is defined as the number of dynamic instructions. In BZIP2, the average thread size and the average number of threads per invocation are consistent across different input sets. However, in GZIP, `graphic` and `random` input sets have much shorter threads than others. The difference in the thread size indicates that different execution paths may be taken under different input sets.

Instruction scheduling is effective for GZIP on both `graphic` and `random` input sets. However, it is ineffective on `source` input set. In order to better understand this

phenomenon, we examine the most time-consuming loop in $deflate$ for an examination (see Section 2). There are three major paths in that loop: `Path 1` is taken if no matched string is found. `Path 2` is taken if a matched string is found, and it is better than the previous match. `Path 3` is taken if a matched string is found, and it is not better than the previous match. The generated parallel threads are aggressively optimized along `path 2`, since `path 2` is identified as the most frequent path based on the `train` input set. For the `ref` input set, `path 2` is the frequent path for both `graphic` and `random` input sets, but it is taken less frequently for `source` input set. As a result, performance improvement on `source` input set is less significant than other two.

Reduction transformation achieves significant performance improvement for BZIP2 on the `graphic` input set, however, only moderate performance improvement for `source` and `program` input sets. After a close look at the loops, we find that they have more balanced workloads on `graphic` input set than on other two after reduction transformation. For other two input sets, load imbalance becomes the new bottleneck after reduction transformation, and their performance is greatly improved after iteration merging is applied.

## 5 Conclusions and Future Work

Researchers have found it difficult to exploit parallelism in data compression applications, even with the help of TLS. In this paper, we report the results of an extensive study on parallelizing these benchmarks, under the context of TLS. We have identified several performance bottlenecks caused by data and control dependences. To address these problems, we propose several effective compiler optimization techniques that take advantage of profiling information to remove stalls caused by such dependences. Careful evaluation of these technique reveals that, we can achieve up to 37% program speedup for BZIP2, and 36% for GZIP.

Although our techniques have only been applied to BZIP2 and GZIP, in our experience, the data and control access patterns we studied in this paper have been observed in many other integer benchmarks. We are currently integrate these applications in our compiler infrastructure so that we can evaluate the impact of the proposed techniques on a wide-range of applications. We believe, although no single optimization will enable the creation of efficient parallel programs for TLS, a compiler infrastructure that supports a general set of compiler optimization techniques, each designed to optimally manage a specific situation, can be built to create efficient parallel programs.

## References


1. AKKARY, H., AND DRISCOLL, M. A Dynamic Multithreading Processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)* (December 1998).
2. AMD CORPORATION. Leading the industry: Multi-core technology & dual-core processors from amd. http://multicore.amd.com/en/Technology/, 2005.
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. BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAEK, Y., POTTENGER, B., RAUCHWERGER, L., AND TU, P. Parallel programming with polaris. *IEEE Computer 29*, 12 (1996), 78–82.
5. BURROW, M., AND WHEELER, D. A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Systems Research Center, May 1994.
6. 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).
7. CINTRA, M., AND TORRELLAS, J. Learning cross-thread violations in speculative parallelization for multiprocessors. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)* (2002).
8. DU, Z.-H., LIM, C.-C., LI, X.-F., YANG, C., ZHAO, Q., AND NGAI, T.-F. A cost-driven compilation framework for speculative parallelization of sequential programs. In *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)* (June 2004).
9. DUBEY, P., O'BRIEN, K., O'BRIEN, K., AND BARTON, C. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1995)* (June 1995).
10. FRANKLIN, M., AND SOHI, G. S. The expandable split window paradigm for exploiting fine-grain parallelsim. In *19th Annual International Symposium on Computer Architecture (ISCA '92)* (May 1992), pp. 58–67.
11. GUPTA, M., AND NIM, R. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98* (November 1998).
12. HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IIX)* (October 1998).
13. HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. Preliminary experiences with the Fortran D compiler. In *Supercomputing '93* (1993).
14. INTEL CORPORATION. Intel's dual-core processor for desktop PCs. http://www.intel.com/personal/desktopcomputer/dual_core/index.htm, 2005.
15. INTEL CORPORATION. Intel itanium architecture software developer's manual, revision 2.2. http://www.intel.com/design/itanium/manuals/iiasdmanual.htm, 2006.
16. JOHNSON, T., EIGENMANN, R., AND VIJAYKUMAR, T. Min-cut program decomposition for thread-level speculation. In *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)* (June 2004).
17. KALLA, R., SINHAROY, B., AND TENDLER, J. M. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *Microprocessor Forum '99* (October 1999).
18. KENNEDY, K., AND ALLEN, R. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Academic Press, 2002.
19. KNIGHT, T. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Lisp and Functional Programming Conference* (August 1986), pp. 500–519.
20. KRISHNAN, V., AND TORRELLAS, J. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)* (October 1999).
21. LI, X.-F., DU, Z.-H., ZHAO, Q.-Y., AND NGAI, T.-F. Software value prediction for speculative parallel threaded computations. In *1st Value-Prediction Workshop (VPW 2003)* (Jun 2003).
22. LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. Posh: A tls compiler that exploits program structure. In *ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming* (March 2006).
23. LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN 05 Conference on Programming Language Design and Implementation (PLDI'05)* (June 2005).

24. MARCUELLO, P., AND GONZALEZ, A. Clustered speculative multithreaded processors. In *13th Annual ACM International Conference on Supercomputing* (Rhodes, Greece, June 1999).
25. OPLINGER, J., HEINE, D., AND LAM, M. In search of speculative thread-level parallelism. In *Proceedings PACT 99* (October 1999).
26. PRABHU, M., AND OLUKOTUN, K. Using thread-level speculation to simplify manual parallelization. In *ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming* (2003).
27. PRABHU, M., AND OLUKOTUN, K. Exposing speculative thread parallelism in spec2000. In *ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming* (2005).
28. QUINONES, C. G., MADRILES, C., SANCHEZ, J., GONZALES, P. M. A., AND TULLSEN, D. M. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *ACM SIGPLAN 05 Conference on Programming Language Design and Implementation (PLDI'05)* (June 2005).
29. SOHI, G. S., BREACH, S., AND VIJAYKUMAR, T. N. Multiscalar Processors. In *22nd Annual International Symposium on Computer Architecture (ISCA '95)* (June 1995), pp. 414–425.
30. 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.
31. STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. A Scalable Approach to Thread-Level Speculation. In *27th Annual International Symposium on Computer Architecture (ISCA '00)* (June 2000).
32. SUN CORPORATION. Throughput computing—niagara. http://www.sun.com/processors/throughput/, 2005.
33. 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.
34. TSAI, J.-Y., HUANG, J., AMLO, C., LILJA, D., AND YEW, P.-C. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures 48*, 9 (September 1999).
35. TSAI, J.-Y., JIANG, Z., AND YEW, P.-C. Compiler techniques for the superthreaded architectures. *International Journal of Parallel Programming - Special Issue on Languages and Compilers for Parallel Computing* (June 1998).
36. VIJAYKUMAR, T. N., BREACH, S. E., AND SOHI, G. S. Register communication strategies for the multiscalar architecture. Tech. Rep. Technical Report 1333, Department of Computer Science, University of Wisconsin-Madison, Feb. 1997.
37. VIJAYKUMAR, T. N., AND SOHI, G. S. Task selection for a multiscalar processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)* (Nov. 1998).
38. WANG, S., YELLAJYOSULA, K. S., ZHAI, A., AND YEW, P.-C. Loop selection for thread-level speculation. In *The 18th International Workshop on Languages and Compilers for Parallel Computing* (Oct 2005).
39. ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler optimization of scalar value communication between speculative threads. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)* (Oct 2002).
40. ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler optimization of memory-resident value communication between speculative threads. In *The 2004 International Symposium on Code Generation and Optimization* (Mar 2004).
41. ZILLES, C., AND SOHI, G. Master/slave speculative parallelization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-35)* (Nov 2002).
42. ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. 337–343.