Compiler Optimizations for Parallelizing General-Purpose Applications under Thread-Level Speculation

Antonia Zhai, Shengyue Wang, Pen-Chung Yew, and Guojin He
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455, USA
{zhai, shengyue, yew, guojinhe}@cs.umn.edu

Categories and Subject Descriptors C.1.2 [*PROCESSOR ARCHITECTURES*]: Multiple Data Stream Architectures (Multiprocessors)

General Terms Languages, Performance

Keywords Thread-Level Speculation, Multicore systems, Compiler Optimizations, Parallelizing Compiler

As technology advances, microprocessors that integrate multiple cores on a single chip are becoming increasingly common. However, extracting independent threads for parallel execution is an extremely challenging task. Thread-Level Speculation (TLS) (4; 1; 2; 3; 5) provides hardware mechanisms necessary for optimistically parallelizing potentially dependent threads. However, the ubiquitous existence of complex, input-dependent control and data dependence patterns in general-purpose applications requires a collection of compiler optimizations that target a variety of code patterns to generate efficient parallel threads. In this paper, we propose, implement and evaluate three optimization techniques to improve TLS performance.

1. Compiler Optimizations for TLS

General-purpose applications often have complex behaviors, and thus are unable to take full advantage of the TLS support. A close examination of the code sequence in Figure 1(a) reviews that the loop can be parallelized by speculating on the dependence between *LoadAddr*, *LoadAddr_1* and *StoreAddr_1*, *StoreAddr_2*, *StoreAddr*. However, if the load through *LoadAddr* often depends the store through *StoreAddr*, speculation will often fail. In this case, it is desirable to synchronize these two memory accesses by forwarding the stored value between the two threads (with explicit *signal* and *wait* instructions. However, synchronization can create a critical forwarding path between these threads and serialize parallel execution. To overcome such serialization, the signal instructions, as well as the instructions it depends on (`v ← *LoadAddr_1`) can be moved as early as possible within the thread, as shown in Figure 1(b). It is important to point out that, in TLS, since inter-thread dependences of memory-resident values are verified by the underlying TLS hardware at runtime, moving store instructions with respect to other store instructions can interfere the underlying TLS mechanism and cause error. Thus, this paper makes:
Contribution 1: Propose hardware/software support necessary to enable aggressive instruction scheduling involving store instructions.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

```
do {
   wait;
   ...← *LoadAddr;
   work();
   *StoreAddr_1 ← ...;
   if (cond1)
      v ← *LoadAddr_1;
   *StoreAddr ← v;
   signal v;
   *StoreAddr_2 ← ...;
} while (cond)
```

(a) Synchronizing an inter-thread data dependence.

```
do {
   wait;
   v_temp ← *LoadAddr_1; ◇
   signal (v_temp); ◇
   ...← *LoadAddr;
   work();
   *StoreAddr_1 ← ...;
   if (cond1)
      v ← *LoadAddr_1;
   *StoreAddr ← v;
   *StoreAddr_2 ← ...;
} while (cond)
```

(b) Scheduling instructions.

```
do {
   if(count>500)
      break;
   do {
      count++;
   } while(cond1)
} while(cond1)
```

(c) Reduction-like variable.

```
do {
   ...;
   if (rand()>0.9) {
      do {
         ...;
      }
   }
} while (cond)
```

(d) Unbalanced workload.

Figure 1. Potentially parallelizable loops under TLS.

In the above example, scheduling `v ← *LoadAddr_1` leads to two intra-thread speculations: `if_cond` is always true, and the load instruction is independent of `StoreAddr_2`. When these speculations fail, we can either squash the entire thread (6; 7), or have the compiler create a small piece of recovery codes, and only execute the recovery codes. The later can be more cost effective than the former. Thus, this paper makes:
Contribution 2: Couple inter-thread and intra-thread dependence speculation to exploit thread-level parallelism, and develop recovery codes for handling intra-thread speculation failures.

Traditional compilers have a very strict definition on what reduction variables are. For the code sequence in Figure 1(c), variable `count` is not a reduction variable, since the intermediate value of `count` is used to evaluate the `if` statement. Thus, traditional compiler will not optimize this operation. Thus, this paper makes:
Contribution 3: Identify a set of common reduction-like variables, where the intermediate values of these variables are used either to computer another value or to determine the outcome of a branch; and propose code transformations for each usage pattern.

In general-purpose applications, consecutive threads often differ significantly in the number of dynamic instructions executed, due to complex control flow, inner loops, as well as procedures calls. For the loop in Figure 1(d), when the `if` condition evaluates to true, the iteration takes much longer to complete. Since the condition is based on a random value, it is impossible to predict which iterations are long. In TLS, speculative threads must commit in sequential order, thus short threads that are executed immediately after a long thread must stall upon completion and wait for the long thread

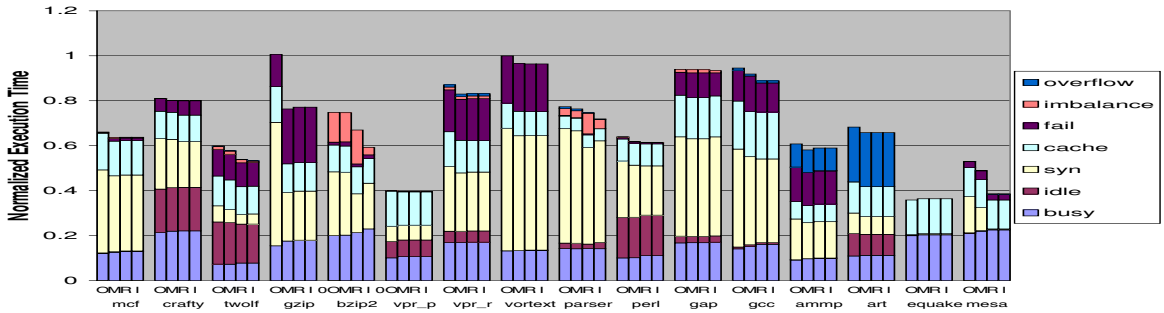

Figure 2. The performance impact of the proposed optimizations on parallel regions

to complete. This behavior can serialize parallel execution and degrade performance. Thus, this paper makes:

Contribution 4: Identify program execution patterns that leads to unbalanced workload between speculative threads, and propose code transformations to merge consecutive iterations to create balance threads.

2. Performance Evaluation

Our compiler infrastructure builds on Intel's Open Resource Compiler(ORC), extended to perform trade-off analysis to determine which loops to parallelize. In our study, a significant percent of the sequential execution can be parallelized: mcf (74%), crafty (11%), twolf (60%), gzip (81%),bzip2 (65%), vpr-place (69%), vpr-route (55%), vortex (28%), parser (47%), perl (23%), gap (12%), gcc (59%), ammp (97%), art (95%), equake (93%), mesa (65%). The parallelized programs are evaluated on a CMP simulator with four single-issue in-order processors, each with a 1-cycle, 32KB, 2-way set-associative L1 cache. All processors share a 2MB, 4-way set-associative L2 cache that has a 10 cycle access time.

Figure 2 shows the performance impact of the proposed optimization on the parallelized loops. The bars show the parallelized execution time with various levels of optimization of the parallelized loops normalized to that of sequential execution. The **O** bars show the normalized execution time without the optimizations proposed in this paper. Register-resident value and frequently dependent memory-resident values are synchronized. The **M** bars show the normalized execution time of the same loops, except for instructions are aggressively scheduled across intra-thread control and data dependences; and intra-thread data dependences violations are handled with a small piece of recovery code. The **R** bars integrates reduction variable optimizations on top of **M** bars. The **I** bars integrates iteration merging on top of the **R** bars.

Each bar is divided into seven segments: *busy* is the time spent executing useful instructions; *idle* is the time wasted due to the lack of parallel threads; *syn* is time spent on synchronization; *cache* is time spent on cache misses; *fail* is the time wasted executing instruction that are eventually thrown away due to failed speculation;*imbalance* is the time wasted waiting for commit; and *overflow* is the time wasted since the buffering space for speculative execution has overflowed.

Memory-Resident Values Communication: The difference between the **O** and the **M** bars is the benefit of scheduling instructions for synchronized memory-resident values. we observe that over half of the benchmarks benefit from this optimization, and the time they spend stalling on synchronization has reduced significantly. GZIP achieves the most performance improvement, a 32% speedup. Nine benchmarks, MCF, TWOLF, VPR_ROUTE, VORTEX, PERL, GCC, AMMP, ART, and MESA also achieve 3% to 16% performance improvement. Reduction in synchronization does not always translate into performance improvement, since it can increases the cost of mis-speculation, as in the case of MCF, GZIP, BZIP2, GAP, GCC and MESA. Generating recovery codes for handling intra-thread mis-speculation is important, without this optimization the instruction scheduling has no performance benefit for synchronized memory-resident values.

Reduction Variable Optimization: The performance impact of reduction transformation is shown as the difference between the **R** and the **M** bars in Figure 2. Without the proposed optimization, reduction variables are synchronized, and scheduled to reduce the critical forwarding path. The proposed optimization can significantly reduce or even eliminate the critical forwarding path associated with reduction variables, thus the performance improvement observed here is mainly due to the reduced in the cost of synchronizing associated with reduction-like variables. Three benchmarks benefit significantly from this transformation: TWOLF speedup by 7%; BZIP2 by 12%; MESA by 27%. Two other benchmarks, PARSER and GCC achieve moderate performance improvement.

Iteration Merging: The performance impact of reduction transformation is shown as the difference between the **I** and the **R** bars, in Figure 2. The *imbalance* segments of the **R** bars correspond to the amount of time processor stall and wait for previous threads to commit. This is the segment the proposed optimization aims to reduce. Without the proposed transformation, reduction variables are simply synchronized, and instructions are scheduled to reduce the critical forwarding path. Iteration merging is most effective for two benchmarks: BZIP2 speed up by 14% and PARSER by 5%, since these benchmarks each has a significant *imbalance* segment. Note that, for PARSER, the time save by improving the load balancing does not translate completely into performance improvement, rather it is translated to time spent waiting for forwarded variables.

References

[1] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data Speculation Support for a Chip Multiprocessor. In *ASPLOS'98* (October 1998).

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

[3] MARCUELLO, P., AND GONZALEZ, A. Clustered Speculative Multithreaded Processors. In *13th Annual ACM International Conference on Supercomputing* (Rhodes, Greece, June 1999).

[4] 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.

[5] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. A Scalable Approach to Thread-Level Speculation. In *ISCA'00* (June 2000).

[6] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS' 02* (Oct 2002).

[7] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *CGO'04* (Mar 2004).