Loop Selection for Thread-Level Speculation

Shengyue Wang, Xiaoru Dai, Kiran S. Yellajyosula,
Antonia Zhai, Pen-Chung Yew

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

Abstract. Thread-level speculation (TLS) allows potentially dependent threads to speculatively execute in parallel, thus making it easier for the compiler to extract parallel threads. However, the high cost associated with unbalanced load, failed speculation, and inter-thread value communication makes it difficult to obtain the desired performance unless the speculative threads are carefully chosen.

In this paper, we focus on extracting parallel threads from loops in general-purpose applications because loops, with their regular structures and significant coverage on execution time, are ideal candidates for extracting parallel threads. General-purpose applications, however, usually contain a large number of nested loops with unpredictable parallel performance and dynamic behavior, thus making it difficult to decide which set of loops should be parallelized to improve overall program performance. Our proposed loop selection algorithm addresses all these difficulties. We have found that (i) with the aid of profiling information, compiler analyses can achieve a reasonably accurate estimation of the performance of parallel execution, and that (ii) different invocations of a loop may behave differently, and exploiting this dynamic behavior can further improve performance. With a judicious choice of loops, we can improve the overall program performance of SPEC2000 integer benchmarks by as much as 20%.

1 Introduction

Microprocessors that support multiple threads of execution are becoming increasingly common [1, 13, 14]. Yet how to make the most effective use of such processors is still unclear. One attractive method of fully utilizing such resources is to automatically extract parallel threads from existing programs. However, automatic parallelization [4, 10] for general-purpose applications (e.g., compilers, spreadsheets, games, etc.) is difficult because of pointer aliasing, irregular array accesses, and complex control flow. Thread-level speculation (TLS) [3, 6, 9, 11, 16, 22, 24, 26] facilitates the parallelization of such applications by allowing potentially dependent threads to execute in parallel while maintaining the original sequential semantics of the programs through runtime checking. Although researchers have proposed numerous techniques for providing the proper hardware [17, 18, 23, 25] and compiler [27-29] support for improving the efficiency of TLS, how to provide adequate compiler support for decomposing sequential programs into parallel threads that can deliver the desired performance has not yet been explored with the proper depth. In this paper, we present a detailed investigation of extracting speculative threads from loops for general-purpose applications.

Loops are attractive candidates for extracting thread-level parallelism, as programs spend significant amounts of time executing instructions within loops, and the regular structure of loops makes it relatively easy to determine (i) the beginning and the end of a thread (i.e., each iteration corresponds to a single thread of execution) and (ii) the inter-thread data dependences. Thus it is not surprising that most previous research on TLS has focused on exploiting loop-level parallelism. However, general-purpose applications typically contain a large number of potentially nested loops, and thus deciding which loops should be parallelized for the best program performance is not always clear. We have found 7800 loops from 11 benchmarks in the SPEC2000 integer benchmarks; among these, *gcc* contains more than 2600 loops. Thus it is necessary to derive a systematic approach to automatically select loops to parallelize for these applications.

It is difficult for a compiler to determine whether a loop can speed up under TLS, as the performance of the loop depends on (i) the characteristics of the underlying hardware, such as thread creation overhead, inter-thread value communication latency, and mis-speculation penalties, and (ii) the characteristics of the parallelized loops, such as the size of iterations, the number of iterations, and the inter-thread data dependence. While detailed profiling information and complex estimations can potentially improve the accuracy of estimation, it is not clear whether these techniques will lead to an overall better selection of loops.

When loops are nested, we can parallelize at only one loop nest level. We say that loop B is nested within loop A when loop B is syntactically nested within loop A or when A invokes a procedure that contains loop B. On average, we observe that the SPEC2000 integer benchmarks have a nesting depth of 8. Figure 1 shows that straightforward solutions that always parallelize the innermost or the outermost loops do not always deliver the desired performance. Therefore a judicious decision must be made to select the proper nest level to parallelize.

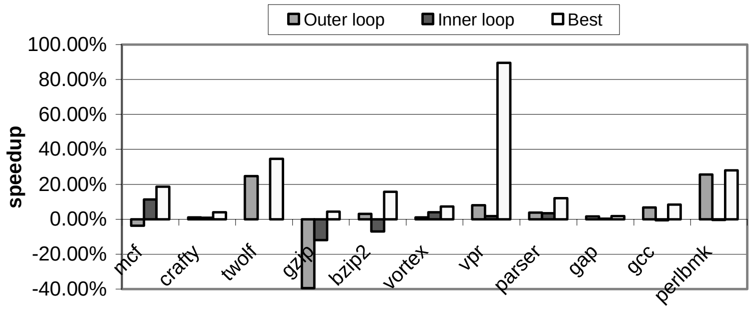


Fig. 1. Performance comparison of simple loop selection techniques

Furthermore, different invocations of the same static loop may have different behaviors. For instance, a parallelized loop may speed up relative to the sequential execution in some invocations but slow down in others. We refer to this behavior as *context sensitivity*. Exploiting this behavior and parallelizing a loop invocation only if that particular invocation is likely to speed up can potentially offer additional performance benefit.

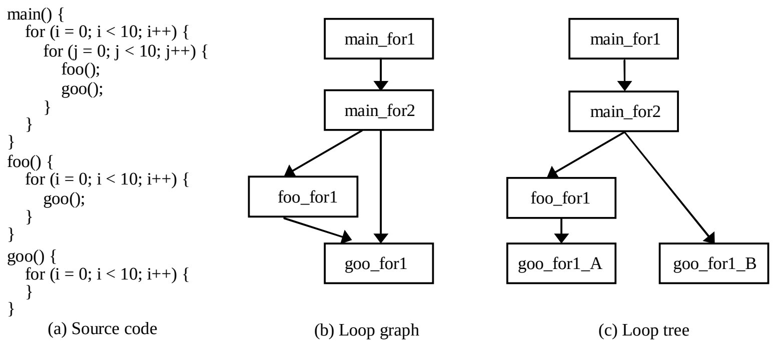
This paper makes the following contributions. First, we propose a loop selection algorithm that decides which loops should be parallelized to improve overall performance for a program with a large number of nested loops. Second, we find that compiler analyses can achieve a reasonably accurate performance prediction of parallel execution. And third, we observe that exploiting dynamic loop behavior can further improve this performance. Overall, by making a judicious choice in selecting loops, we can improve the performance of SPEC2000 integer benchmarks by 20%.

The rest of this paper is organized as follows. In Section 2, we describe a loop selection algorithm that selects the optimal set of loops if the parallel performance of a loop can be accurately predicted. In Section 3, we describe our experimental framework. Three performance estimation techniques are discussed and evaluated in Section 4. We investigate the impact of context sensitivity in Section 5. We discuss related work in Section 6 and present our conclusions in Section 7.

2 Loop Selection Algorithm

In this section, we present a loop selection algorithm that chooses a set of loops to parallelize while maximizing overall program performance. The algorithm takes as input the speedup and coverage of all the loops in a program and outputs an optimal set of loops for parallelization.

2.1 Loop Graph



```
main() {
   for (i = 0; i < 10; i++) {
      for (j = 0; j < 10; j++) {
         foo();
         goo();
      }
   }
}
foo() {
   for (i = 0; i < 10; i++) {
      goo();
   }
}
goo() {
   for (i = 0; i < 10; i++) {
   }
}
```

(a) Source code

(b) Loop graph

(c) Loop tree

Fig. 2. Examples of loop graph and loop tree

The main constraint in loop selection is that there should be no nesting relation between any two selected loops. To capture the nesting relation between loops, we construct a directed acyclic graph (DAG) called a *loop graph*. As shown in Figure 2(b), each node in the graph represents a static loop in the original program, and a directed edge represents the nesting relation between two loops. Loops could have a direct nesting relation or an indirect nesting relation through procedure calls. In this example, the edge from *main_for1* to *main_for2* indicates direct nesting, while the edge from *main_for2* to *foo_for1* indicates indirect nesting.

A recursive call introduces a cycle in the loop graph that violates the acyclic property. But cycles can be broken if we can identify backward edges. An edge from node *s* to node *t* is a backward edge if every path that reaches *s* from the root passes through *t*. All backward edges are removed once they are detected. If no backward edge is detected, we arbitrarily select an edge and remove it to break the cycle.

A loop graph, like a call graph, can be constructed through runtime profiling or compiler static inter-procedure analysis. In this study, it is built upon efficiently collected runtime profiles.

2.2 Selection Criterion

We cannot simultaneously select any two loops that have nesting relations. To decide which loop to select, we use a criterion called *benefit* that considers both speedup and coverage of a loop. It is defined as follows:

$$benefit = coverage \times (1 - 1 / speedup) \qquad (1)$$

The benefit value indicates the overall performance gain that can be obtained by parallelizing that loop. A loop with a larger benefit value is more likely to be selected. The benefit value is additive, as there is no nesting relationship between the selected loops. The speedup for the whole program can be computed directly from the benefit value as follows:

$$program\ speedup = 1 / (1 - benefit) \qquad (2)$$

2.3 Loop Selection Problem

The general loop selection problem can be stated as follows: given a loop graph with benefit value attached to each node, find a set of nodes that maximizes the overall benefits such that there is no path between any two selected nodes.

We transform this loop selection problem into a well-known NP-complete problem, weighted maximum independent set problem [8], by computing the transitive closure of the loop graph. A set of nodes is called an independent set if there is no edge between any two of them.

2.4 Graph Pruning

The general loop selection problem is NP-complete, so that an exhaustive search algorithm only works for a graph with few nodes. For a graph with hundreds or thousands of nodes, which is common for most of the benchmarks that we are studying, a more efficient heuristic has to be used. Because a heuristic-based algorithm only gives a sub-optimal solution, we must use it wisely. By applying a technique called *graph pruning*, we can find a reasonable approximation more efficiently. Graph pruning simplifies the loop graph by eliminating those loops that will not be selected as speculative threads. These would include such loops as: (i) loops that have less than 100 dynamic instructions on average, as they are more appropriate for instruction-level parallelism (ILP); (ii) loops that have no more than 2 iterations on average, as they are more likely to underutilize multiple processor resources; and (iii) loops that are predicted to slow down the program execution if parallelized.

Graph pruning reduces the size of a loop graph by eliminating unsuitable loops. After we delete unnecessary nodes, one single connected graph is split into multiple small disjointed sub-graphs. Then we can apply selection algorithm to each sub-graph independently. It is efficient to use exhaustive searching algorithm for small sub-graphs. For larger sub-graphs, heuristic-based searching algorithm usually gives a reasonable approximation.

2.5 Exhaustive Searching Algorithm

In this simple algorithm, we exhaustively try every set of independent loops to find the one that provides the maximum benefit. For each computed independent loop set, we track all loops that have nesting relations to any loop within this independent set and record them in a vector called a *conflict vector*. By using a conflict vector, it is easy to find a new independent loop to add into the current independent set. After a new loop is added, the conflict vector is updated as well.

An exhaustive searching algorithm gives an accurate solution for the loop selection problem, but is very inefficient. Graph pruning creates smaller sub-graphs that are suitable for exhaustive searching that works efficiently for sub-graphs with fewer than 50 nodes in our experiments.

2.6 Heuristic-based Searching Algorithm

Even after graph pruning, some sub-graphs are still very big. For those, we use a heuristic-based algorithm. We first sort all the nodes in a sub-graph according to their benefit values. Then we pick one node at a time and add it into the independent set such that the node has the maximal benefit value and it does not conflict with already selected nodes. Similarly to the exhaustive searching algorithm, we maintain a conflict vector for the selected independent set and update it whenever a new node is added.

Although this simple greedy algorithm gives a sub-optimal solution, it can select a set of independent loops from a large graph in polynomial time. In our experiments, the size of sub-graph is less than 200 nodes after graph pruning, so the inaccuracy introduced by this algorithm is negligible.

3 Experimental Framework

We implement the loop selection algorithm in the Code Generation phase of the ORC compiler [2], which is an industrial-strength open-source compiler based on the Pro64 compiler and targeting on Intel’s Itanium Processor Family (IPF).

For each selected loop, the compiler inserts special instructions to mark the beginning and the end of parallel loops. Fork instruction is inserted at the beginning of the loop body. We optimize inter-thread value communication using the techniques described in [28, 29]. The compiler synchronizes all inter-thread register dependences and memory dependences with a probability greater than 20%. Both intra-thread control and data speculation are used for more aggressive instruction scheduling so as to increase the overlap between threads.

Our execution-driven simulator is built upon Pin [15]. The configuration of our simulated machine model is listed in Table 1. We simulate four single-issue in-order processors. Each of them 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. The address buffer keeps all 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.

Table 1. Machine configuration.

Issue Width	1
L1-D Cache	32K, 2-way, 1 cycle
L2-D Cache	2M, 4-way, 10 cycles
Write Buffer	32K, 2-way, 1 cycle
Address Buffer	32K, 2-way, 1 cycle
Communication Buffer	128 entries, 1 cycle
Communication Delay	10 cycles
Thread Spawning Overhead	10 cycles
Thread Squashing Overhead	10 cycles
Main Memory	50 cycles

Table 2. Benchmark statistics.

Program	Number of Loops	Average Loop Iteration Size	Maximal Nest Depth
mcf	51	29,605	4
crafty	420	59,775	10
twolf	899	12,437	7
gzip	178	206,755	6
bzip2	163	109,227	9
vortex	212	45,179	7
vpr	401	1,500	5
parser	532	8,820	10
gap	1,655	53,721	10
gcc	2,619	5,394	10
perlbmk	729	2,826	10

3.1 Benchmarks

We study all the SPEC2000 integer benchmarks except for eon, which is written in C++. The statistics for each benchmark are listed in Table 2. The average loop iteration size is measured by using the *ref* input set and counting dynamic instructions. Most of the benchmarks have a large set of loops with complex loop nesting, which makes it difficult, if not impossible, to select loops without a systematic approach.

3.2 Simulation Methodology

All simulation is performed using the *ref* input set. To save simulation time, we parallelize and simulate each loop once. After applying a selection technique, we directly use the simulation result to calculate the overall program performance. In this way, we avoid simulating the same loop multiple times if it is selected by different techniques.

Moreover, we use a simple sampling method to further speed up the simulation. For each loop, we select the first 50 invocations for simulation. For each invocation, we simulate the first 50 iterations. This simple sampling method allows us to simulate up to 6 billion dynamic instructions while covering all loops.

4 Loop Speedup Estimation

Our goal in loop selection is to maximize the overall program performance, which is represented as the benefit value of the selected loops. In order to calculate the benefit value for each loop, we have to estimate both the coverage and speedup of each loop.

Coverage can be estimated using a runtime profile. To estimate speedup, we have to estimate both sequential and parallel execution time.

We assume that each processor executes one instruction per cycle, i.e., each instruction takes one cycle to finish. It is relatively easy to estimate sequential execution time T_{seq} of a loop. We can determine the average size of a thread (average number of instructions executed per iteration) and the average number of parallel threads (average number of times a loop iterates) by using a profile. T_{seq} can be approximated by using equation (3), where S is the average thread size and N is the average number of threads.

$$T_{seq} = S \times N \tag{3}$$

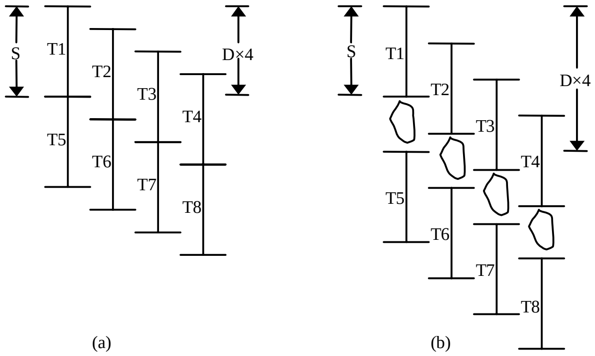


Fig. 3. Impact of delay D assuming 4 processors

On the other hand, the parallel execution time depends on other factors such as the thread creation overhead, the cost of inter-thread value communication, and the cost of mis-speculation. We simplify the calculation by dividing the total parallel execution time T_{par} into two parts: perfect execution time $T_{perfect}$ and mis-speculation time $T_{misspec}$. $T_{perfect}$ is the parallel execution time on p processors assuming that there is no mis-speculation. $T_{misspec}$ is the wasted execution time due to mis-speculation.

$$T_{par} = T_{perfect} + T_{misspec} \tag{4}$$

We also define delay D as the delay between two consecutive threads caused by inter-thread value communication T_{comm} and thread creation overhead O.

$$D = max(T_{comm}, O) \tag{5}$$

Depending on the delay D, we use different equations to estimate $T_{perfect}$. If $D \leq S / p$, we can have a perfect pipelined execution of threads, as shown in Figure 3(a), and use equation (6) for estimation.

$$T_{perfect} = ((N - 1) / p + 1) \times S + ((N - 1) \bmod p) \times D \tag{6}$$

If $D > S / p$, delay D causes bubbles in the pipelined execution of threads and has a higher impact on the overall execution time, as shown in Figure 3(b). In this case, we use equation (7) for estimation.

$$T_{perfect} = (N - 1) \times D + S \tag{7}$$

The key to accurately predicting speedup is how to estimate T_{comm} and $T_{misspec}$. T_{comm} is caused by the synchronization of frequently occurring data dependences, while $T_{misspec}$ is caused by the mis-speculation of unlikely occurring data dependences. We describe techniques to estimate $T_{misspec}$ and T_{comm} in the following sections.

4.1 $\mathbf{T_{misspec}}$ Estimation

When a mis-speculation is detected, the violating thread will be squashed and all the work done by this thread becomes useless. We use the amount of work thrown away in a mis-speculation to quantify the impact of the mis-speculation on the overall parallel execution. The amount of work wasted depends on when a mis-speculation is detected. For instance, if a thread starts at cycle *c1* and mis-speculation is detected at cycle *c2*, we have *(c2 – c1)* wasted cycles.

In our machine model, a mis-speculation in the current thread is detected at the end of the previous thread, so we could waste *(S – D)* cycles for a mis-speculation. The overall execution time wasted due to mis-speculation is calculated in equation (8), where P_{misspec} is the probability that a thread will violate inter-thread dependences and is obtained through a profile.

$$T_{misspec} = (S - D) \times P_{misspec} \tag{8}$$

4.2 $\mathbf{T_{comm}}$ Estimation I

One way to estimate the amount of time that parallel threads spend on value communication is to identify all the instructions that are either the producers or the consumers of inter-thread data dependences and estimate the cost of value communication as the total cost of executing all such instructions.

Although this estimation technique is simple, it assumes that the value required by a consumer instruction is immediately available when it is needed. Unfortunately, this assumption is not always realistic, since it is often the case that the instruction that consumes the value is issued earlier than the instruction that produces the value, as shown in Figure 4(a). Thus the consumer thread T2 has to stall and wait until the producer thread T1 is able to forward it the correct value, as shown in Figure 4(b). The flow of the value between the two threads serializes the parallel execution, so we refer to it as a *critical forwarding path.*

4.3 $\mathbf{T_{comm}}$ Estimation II

To take into consideration the impact of the critical forwarding path, we propose estimation technique II. Assuming that *load1*, the consumer instruction in thread T2, is executed at cycle *c2* and that *store1*, the producer instruction in thread T1, is executed at cycle *c1*, the cost of value communication between these two instructions is estimated as *(c1 – c2)*.

If the data dependence does not occur between two consecutive threads but rather has a dependence distance of *d*, the impact on the execution time of a particular thread should be averaged out over the dependence distance. Thus the impact of communicating a value between two threads is estimated as follows:

$$criticalness = (c1 - c2) / d \qquad (9)$$

There is one more mission piece if this estimation technique is to be successful, which is how to determine which cycle of a particular instruction should be executed. Since it is not possible to perfectly predict the dynamic execution of a thread, we made a simplification assuming each instruction will take one cycle to execute; thus the start cycle is simply an instruction count of the total number of instructions between the beginning of the thread and the instruction in question. However, due to complex control flows that are inherent to general-purpose applications, there can be multiple execution paths, each with different path length, that reach the same instruction. Thus the start time of a particular instruction is the average path length weighted by path taken probability, as shown in equation (10).

$$c = \Sigma\, p_i \in {}_{all\_paths}(length(p_i) \times prob(p_i)) \qquad (10)$$

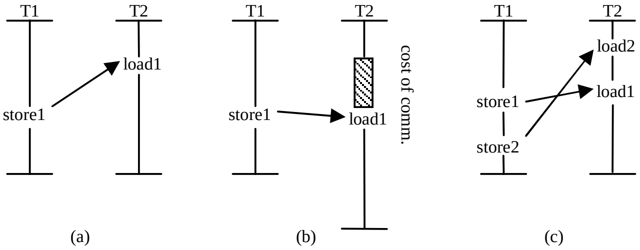


Fig. 4. The data dependence patterns between two speculative threads

For many loops, multiple data dependences exist between two threads, as shown in Figure 4(c). In such cases, the cost of value communication is determined by the most costly one, since the cost of other synchronizations can be hidden.

4.4 T_{comm} Estimation III

Previous work has shown that the compiler can effectively reduce the cost of synchronization through instruction scheduling and that such optimizations are particularly useful for improving the efficiency of communicating register-resident scalars [28, 29]. Unfortunately, the estimation technique described in the previous section does not take such optimization into consideration and tends to overestimate the cost of inter-thread value communication.

It is desirable to find an estimation technique that considers the impact of instruction scheduling on reducing the critical forward path length. Thus, we use a third technique, in which the start time of an instruction is computed from the data dependence graph. When there are multiple paths that can reach an instruction in the data dependence graph, the average start time of this instruction can be measured by equation (11), assuming that the average length of a path p_i that reaches this instruction in the data dependence graph is *length(p_i)*.

$$c = max(length(p_i)) \qquad (11)$$

4.5 Evaluation

The three speedup estimation techniques described above have been implemented in our loop selection algorithm and three sets of loops are selected for parallelization respectively. The performance improvement of the parallel execution is evaluated against sequential execution and the results are illustrated in Figure 5. For comparison, we also select loops using speedup value calculated from simulation results and use this perfect estimation as the upper bound.

We make several observations. First, for estimation I, the performance improvement obtained by most benchmarks is close to the perfect performance improvement obtained through simulation. However, for *gzip*, the loops selected using this estimation is completely wrong and results in a 40% performance degradation. Second, the set of loops selected using estimation II is able to achieve only a fraction of the performance obtained by the set of loops selected using simulation results. This estimation technique tends to be conservative in selecting loops. Third, the set of loops selected with estimation III always performs at least as well as the set of loops selected by estimation I and estimation II.

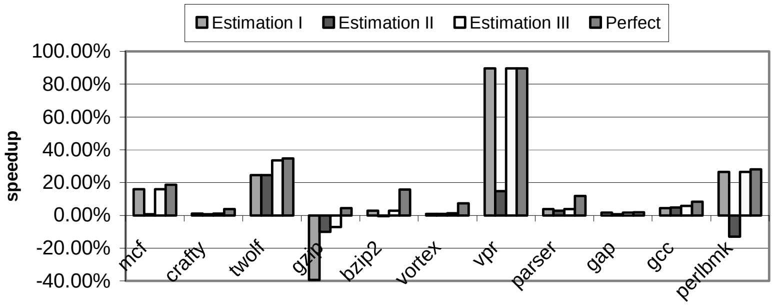


Fig. 5. Performance comparison of different speedup estimation techniques

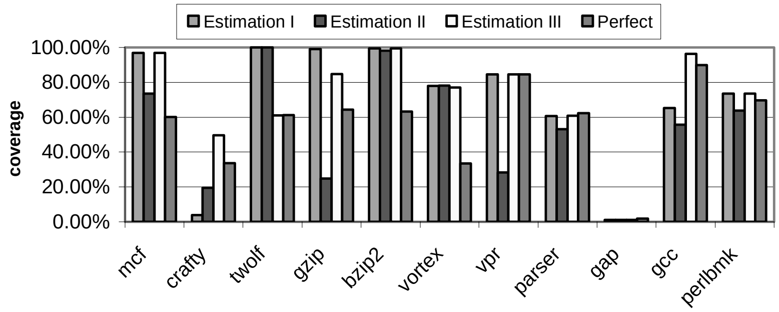


Fig. 6. Coverage comparison of different speedup estimation techniques

Figure 6 illustrates the coverage of parallel execution on the total execution time. We have found that although the set of loops selected using simulation results demonstrate the most performance improvement, these loops do not always have the large coverage on execution time. In *mcf*, the set of loops selected using estimation III has the similar performance as the set of loops selected using simulation results, however, the coverage of the perfect loop set is significantly smaller. This phenomenon suggests that our estimation method may not be very accurate but is useful in selecting a set of loops that have good performance potential.

5 The Impact of Dynamic Loop Behavior on Loop Selection

Once a loop is selected by our current loop selection algorithm, every invocation of this loop is parallelized. The underlying assumption is that the parallel execution of a loop behaves the same across different invocations. However, some loops exhibit different behaviors when they are invoked multiple times. Different invocations of a loop may differ in the number of iterations, the size of iterations, and the data dependence patterns, and thus demonstrate different parallel execution efficiency. Consequently, it might be desirable to parallelize only certain invocations of a loop. In this section, we address this phenomenon. In particular, we examine whether exploiting such behavior can help us select a better set of loops and improve the overall program performance.

5.1 Calling Context of a Loop

In the loop graph, as described in Section 2, we refer to the path from the root node to a particular loop node as the *calling context* of that loop. It is possible for a particular loop to have several distinct calling contexts, and it is also possible for loops with different calling contexts to behave differently. To study this behavior, we replicate the loop nodes for each distinct calling context. An example is shown in Figure 2(c), where the loop node *goo_for1* has two distinct calling contexts and is thus replicated into *goo_for1_A* and *goo_for1_B*. After the replication, the original loop graph is converted into a tree, which we refer to as the *loop tree*.

We parallelize a loop under a certain calling context if the parallel execution speeds up under that calling context. Loop selection on the loop tree is straightforward. The algorithm is as follows. We first traverse the loop tree bottom-up. For each node in the tree, we evaluate its benefit value as $B_{current}$. We sum up the benefit values if we parallelize its descendants, and refer to this number as $B_{subtree}$. If $B_{current}$ is greater than $B_{subtree}$, we mark this node as a potential candidate for parallelization. We also record the larger of these two numbers as $B_{perfect}$, which is used to calculate $B_{subtree}$ of its parent. Next we traverse the loop tree top-down. Once we have encountered a loop node that is marked as a potential candidate from the previous step, we prune its children. The leaf nodes of the remaining loop tree correspond to the loops that should be parallelized. The accurate solution for selecting loops from a loop tree can be found in polynomial time.

5.2 Dynamic Behavior of a Loop

It is possible for two different invocations of a loop to behave differently even if they have the same calling context. To study this behavior further, we assume an oracle that can perfectly predict the performance of a particular invocation of a loop and parallelize this invocation only when it speeds up. A different set of loops are selected and evaluated assuming that such an oracle is in place.

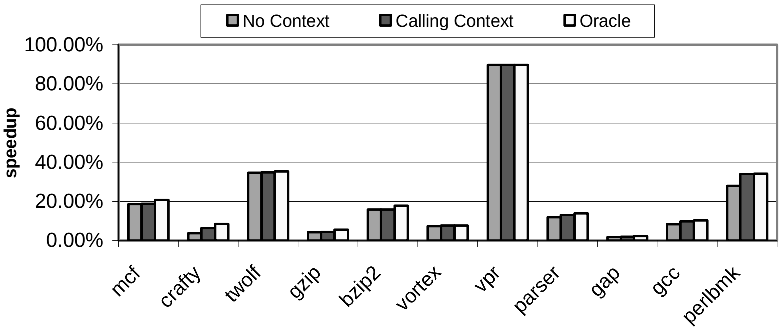


Fig. 7. Performance comparison of loop selection based on different contexts

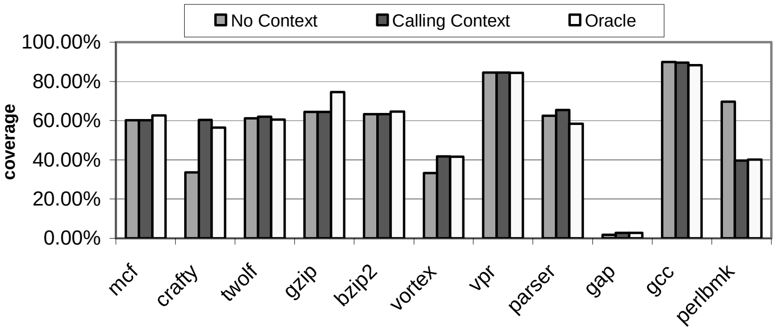


Fig. 8. Coverage comparison of loop selection based on different contexts

5.3 Evaluation

In this section, we evaluate the impact of considering the calling context of a loop (as described in Section 5.1) and the impact of parallelizing only selected invocations of a loop (as described in Section 5.2). The impact of such behavior on overall program performance is shown in Figure 7. We have observed that by differentiating loops with different calling contexts, some benchmarks are able to obtain better program performance. Among them, *crafty* has an additional speed up of 2% and *perlbmk* speeds up by 7%. The performance of *mcf*, *crafty*, and *bzip2* improves an additional 2% by having an oracle that parallelizes only invocations of loops that speed up. Thus, we found that the dynamic behavior of loops has performance impact for some benchmarks. We believe that a dynamic or static loop selection strategy that can predict whether a particular invocation of a loop speeds up can help us achieve additional performance improvement.

Figure 8 shows the coverage for the selected loops. For some benchmarks, such as *perlbmk*, we observe that the overall program performance improves although the coverage of parallelized loops decreases when we take context information into consideration. Close examination reveals that *perlbmk* contains a loop that only speeds up under certain circumstances, and by parallelizing only such invocations, we can achieve better performance. For some other benchmarks, such as *crafty* and *vortex*, the coverage of parallel loops increased due to the selection of a different set of loops.

6 Related Work

Colohan et al. [7] have empirically studied the impact of thread size on the performance of loops, and derived several techniques to determine the unrolling factor of each loop. Their goal is to find the optimal thread size for parallel execution. Our estimation techniques can be employed to determine the candidate loops to unroll. They also propose a runtime system to measure the performance and select loops dynamically.

Oplinger et al. [19] have proposed and evaluated a static loop selection algorithm in their study of the potential of TLS. In their algorithm, they select the best loops in each level of a dynamic loop nest as possible candidates to be parallelized and compute the frequency with which each loop is selected as the best loop. Then they select loops for parallelization based on the computed frequencies. Their concept of a dynamic loop nest is similar to the loop tree proposed in this paper, but is used only to guide the heuristic in context-insensitive loop selection. Their performance estimation is obtained directly from simulation and does not consider the effect of compiler optimization.

Chen et al. [5] have proposed a dynamic loop selection framework for the Java program. They use hardware to extract useful information (such as dependence timing and speculative state requirements) and then estimate the speedup for a loop. Their technique is similar to the runtime system proposed by Colohan et. al. [7] and can only select loops within a simple loop nest. Considering the global loop nesting relations and selecting loops globally introduces significant overhead for a runtime system.

Several papers [12, 21] have studied thread generation techniques that extract speculative parallel threads from consecutive basic blocks. Threads generated using these techniques are fine-grained and usually contain neither procedure calls nor inner loops. These thread generation techniques can complement loop-based threads by exploiting parallelism in the non-loop potion of the program or in loops that are not selected for parallel execution by our algorithm.

Prabhu et al. [20] manually parallelize several SPEC2000 applications using techniques beyond the capabilities of current parallelizing compilers. However, only a few loops are evaluated due to the time-consuming and error-prone nature of this process.

7 Conclusions

Loops, with their regular structures and significant coverage on execution time, are ideal candidates for extracting parallel threads. However, typical general-purpose applications contain a large number of nested loops with complex control flow and ambiguous data dependences. Without an effective loop selection algorithm, determining which loops to parallelize can be a daunting task. In this paper, we propose a loop selection algorithm that takes the coverage and speedup achieved by each loop as inputs and produces the set of loops that should be parallelized to maximize program performance as the output. One of the key components of this algorithm is the ability to accurately estimate the speedup that can be achieved when a particular loop is parallelized. This paper evaluates three different estimation techniques and finds that with the aid of profiling information, compiler analyses are able to come up with reasonably accurate estimates that allow us to select a set of loops to achieve good overall program performance. Furthermore, we have observed that some loops behave differently across different invocations. By exploiting this behavior and parallelizing only invocations of a loop when it actually speeds up, we can potentially achieve better overall program performance for some benchmarks.

References

1. Intel Pentium Processor Extreme Edition. http://www.intel.com/products/processor/pentiumXE/prodbrief.pdf.
2. Open Research Compiler for Itanium Processor Family. http://ipf-orc.sourceforge.net/.
3. Akkary, H. and Driscoll, M., A Dynamic Multithreading Processor. in Proceedings of Micro-31, (December 1998).
4. Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P. and Weatherford, S., Polaris: Improving the Effectiveness of Parallelizing Compilers. in Proceedings of the 7th LCPC, (1994).
5. Chen, M. and Olukotun, K., TEST: A Tracer for Extracting Speculative Threads. in Proceedings of 2003 International Symposium on CGO, (March 2003).
6. Cintra, M.H., Martínez, J.F. and Torrellas, J., Architectural support for scalable speculative parallelization in shared-memory multiprocessors. in Proceedings of the ISCA, (2000).
7. Colohan, C.B., Zhai, A., G., S.J. and Mowry, T.C., The Impact of Thread Size and Selection on the Performance of Thread-Level Speculation. in progress.
8. Du, D.Z. and Pardalos, P.M., Handbook of Combinatorial Optimization. Kluwer Academic Publishers., 1999.
9. Gopal, S., Vijaykumar, T., Smith, J. and Sohi, G., Speculative Versioning Cache. in Proceedings of the 4th HPCA, (February 1998).
10. Hall, M.W., Anderson, J.M., Amarasinghe, S.P., Murphy, B.R., Liao, S.-W., Bugnion, E. and Lam, M.S., Maximizing Multiprocessor Performance with the SUIF Compiler. IEEE Computer, 1999 (12).
11. Hammond, L., Willey, M. and Olukotun, K., Data Speculation Support for A Chip Multiprocessor. in Proceedings of ASPLOS-8, (October 1998).
12. Johnson, T.A., Eigenmann, R. and Vijaykumar, T.N., Min-Cut Program Decomposition for Thread-Level Speculation. in Proceedings of PLDI, (2004).
13. Kalla, R., Sinharoy;, B. and Tendler, J.M., IBM Power5 Chip: a Dual-Core Multithreaded Processor. IEEE MICRO, 2004 (2).
14. Kongetira, P., Aingaran, K. and Olukotun, K., Niagara: A 32-Way Multithreaded Sparc Processor. IEEE MICRO, 2005 (2).
15. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J. and Hazelwood, K., Pin: Building Customized Program Analysis Tools with Dynamic

Instrumentation. in Proceedings of the ACM Intl. Conf. on Programming Language Design and Implementation, (June 2005).

16. Marcuello, P. and Gonzlez, A., Clustered Speculative Multithreaded Processors. in Proceedings of MICRO-32, (November 1999).
17. Moshovos, A.I., Breach, S.E., Vijaykumar, T. and Sohi, G.S., Dynamic Speculation and Synchronization of Data Dependences. in the proceedings of the 24th ISCA, (June 1997).
18. Olukotun, K., Hammond, L. and Willey, M., Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. in Proceedings of the ACM Int. Conf. on Supercomputing, (June 1999).
19. Oplinger, J., Heine, D. and Lam, M.S., In Search of Speculative Thread-Level Parallelism. in Proceedings of PACT, (October 1999).
20. Prabhu, M. and Olukotun, K., Exposing Speculative Thread Parallelism in SPEC2000. in Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming, (2005).
21. Quinones, C.G., Madriles, C., Sanchez, J., Marcuello, P., González, A. and Tullsen, D.M., Mitosis Compiler: An Infrastructure for Speculative Threading Based on Pre-Computation Slices. in Proceedings of the ACM Intl. Conf. on Programming Language Design and Implementation, (June 2005).
22. Rauchwerger, L. and Padua, D.A., The LRPD Test: Speculative RunTime Parallelization of Loops with Privatization and Reduction Parallelization. IEEE Transactions on Parallel Distributed Systems, 1999 (2). 160-180.
23. Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K. and Torrellas, J., Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation. in Proceeding of the 19th ACM International Conference on Supercomputing, (2005).
24. Sohi, G.S., Breach, S.E. and Vijaykumar, T.N., Multiscalar Processors. in Proceedings of the 22nd ISCA, (June 1995).
25. Steffan, J.G., Colohan, C.B., Zhai, A. and Mowry, T.C., Improving Value Communication for Thread-Level Speculation. in Proceedings of the 8th HPCA, (February 2002).
26. Tsai, J.-Y., Huang, J., Amlo, C., Lilja, D. and Yew, P.-C., The Superthreaded Processor Architecture. IEEE Transactions on Computers, 1999 (9).
27. Vijaykumar, T.N. and Sohi, G.S., Task Selection for a Multiscalar Processor. in Proceeding of the 31st International Symposium on Microarchitecture, (December 1998).
28. Zhai, A., Colohan, C.B., Steffan, J.G. and Mowry, T.C., Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. in Proceedings of 2004 International Symposium on CGO, (March 2004).
29. Zhai, A., Colohan, C.B., Steffan, J.G. and Mowry, T.C., Compiler Optimization of Scalar Value Communication Between Speculative Threads. in Proceedings of the 10th ASPLOS, (October 2002).