Loop Selection for Thread-Level Speculation

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

Department of Computer Science
University of Minnesota
{shengyue, dai, kiran, zhai, yew}@cs.umn.edu

Abstract. Thread-level speculation (TLS) allows potentially dependent threads to execute in parallel, and thus, makes it easier for the compiler to extract parallel threads in the presence of complex control dependences and ambiguous data dependences. However, the high cost associate with unbalanced loads, failed speculation, and inter-thread value communication makes it impossible to obtain the desired performance unless the speculative threads are carefully chosen. Unfortunately, there has been relatively little work on automatically dividing programs into speculative threads. In this paper, we investigate speculative thread decomposition of loops in general-purpose applications since loops, with their regular structures and significant coverage on execution time, are ideal candidates for extracting parallel threads. However, typical general-purpose applications have a large number of loops, and it is difficult for the compiler to decide which loops to parallelize in order to maximize program performance due to the following reasons: (i) the unpredictability of control dependences and the ambiguity of inter-thread data dependences makes it difficult for the compiler to estimate the performance of a particular loop and (ii) the complex nesting structures of these loops makes it difficult to choose which loops to parallelize and the context sensitivity of dynamic loop behavior further exasperates this situation. In this paper, we evaluate the effectiveness of various compiler techniques that estimate the performance of parallel loops and study the context sensitivity of each loop. We have found, (i) with the aid of profiling information, compiler analyses can achieve a reasonably accurate estimation of the effectiveness of parallel execution and (ii) different invocations of a loop may behave differently, and exploiting this behavior can help improve the performance. With judicious choice of loops, we can improve the performance of SPEC2000 integer benchmarks by as much as 20%.

1 Introduction

Microprocessors that support multiple threads of execution are becoming increasingly common [1-8], however, how we can effectively make use of such processors is still unclear. One attractive method to fully utilize such resources is to automatically extract parallel threads from existing programs. However, automatic parallelization [9-11] for general-purpose applications (e.g., compilers, spreadsheets, games, etc.) is difficult due to pointer and indirect references, complex data structures and control flow, and input-dependent program behaviors. Thread-Level Speculation (TLS) [12-

28] 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 there are numerous proposals on providing the proper hardware [29-33] and compiler [34, 35] support for improving the efficiency of TLS; proper compiler support for decomposing sequential programs into parallel threads[16, 36, 37] that can deliver the desired performance has not been explored with the proper depth. In this paper, we present a detailed investigation on how to extract speculative threads from loops.

Loops are attractive candidates for extracting thread-level parallelism, since programs spend significant amount of time executing instructions within loops, and the regular structures of loops make it relatively easy to determine (i) the beginning and the end of a speculative thread (each iteration corresponds to a single thread of execution) and (ii) data dependences that occur between different iterations of the loop, a.k.a., inter-thread data dependences. Thus, it is not surprising that previous researches have focused mostly on exploiting loop-level parallelism[11-16, 19-24, 27-29, 31-35, 38, 39]. General-purpose applications typically contain a large number of potentially nested loops, and thus deciding which loops are best for parallelization is not always clear. We found 7800 loops from 11 benchmarks in SPEC2000 integer benchmarks, among them, *gcc* contains around 2600 loops. Such a large number of loops calls for a systematic approach to select the set of loops to parallelize in applications.

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

When loops are nested, we can only parallelize at one loop nest level. We say loop B is nested within loop A when loop B is syntactically nested with in loop A or when A invokes a procedures that contains loop B. On average, we observe that SPEC2000 integer benchmarks have a mean nesting depth of 8. In particular, gcc has a nesting depth of 10 and *gap* has a maximum depth of 12. Straightforward solutions that always parallelize the inner-most or the outer-most loops do not always deliver the desired performance. A judicious decision must be made to select the proper nest level to parallelize.

Furthermore, different invocations of the same static loop may have different behaviors. For instance, a parallelized loop may speed up relative to sequential execution in some invocations, while slow down in others. We refer to this behavior as context sensitivity. It is not clear whether this phenomenon is common among general purpose applications and whether it can lead to better performance when exploited.

This paper makes the following contributions: by evaluating the impact of three different loop performance estimation techniques and studying the context sensitivity of more than 7800 loops from 11 SPEC2000 integer benchmarks that we have found

(i) with the aid of profiling information, compiler analyses can achieve a reasonably accurate estimation of the effectiveness of parallel execution and (ii) different invocations of a loop may behave differently, and exploiting this behavior can help improve the performance. With a judicious choice of selecting loops, we can improve the performance of SPEC Integer benchmarks by 20%.

The rest of paper is organized as follows: In Section 2, we describe a loop selection algorithm that select the optimal set of loops to parallelize if we are able to perfectly predict the performance of all loops under TLS and loops are not context sensitive. Since such assumption is not realistic, we discuss and evaluate several realistic speedup estimation techniques. We investigate the impact of context sensitivity in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 Loop Selection Algorithm

In this section we present a loop selection algorithm that chooses a set of loop to parallelize while maximizing overall program performance. The input to this algorithm is the coverage and speedup for each individual loop in a program. The output is a set of selected loops.

2.1 Loop Graph

For single-level loop selection, the main constraint is that there should be no loop nesting relation between any two selected loops. Before we select loops, we first construct a Directed Acyclic Graph (DAG) called loop graph to represent nesting relation between loops. As shown in Figure 1, each node in the graph is a static loop in the original program, and a directed edge represents loop nesting relation between two loops. Loops could have direct nesting relation or indirect nesting relation through procedure calls. In this example, the edge from main_for1 to main_for2 indicates direct loop nesting, and the edge from main_for2 to foo_for1 indicates indirect loop nesting.

For efficiency, transitive nesting relation from main_for1 to foo_for1 is not represented in loop graph. We also add a pseudo root node to the graph, and create edges from this root node to each node that has no predecessor.

A recursive call introduces cycle in the call graph that violates the acyclic property. Cycles can be broken if we can identify backward edges. A backward edge in the loop graph is defined as the same as the one in the Control-Flow Graph (CFG) in Figure 1(b). After a backward edge is identified, it is simply deleted. If no backward edge is detected, we arbitrarily select an edge and remove it to break the cycle.

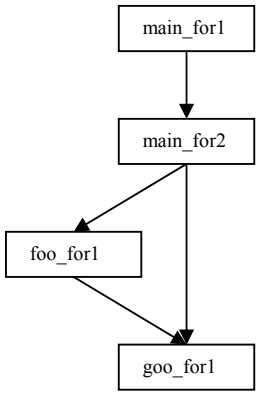
Loop graph, like call graph, can be constructed by using runtime profile or compiler static inter-procedure analysis. In this study, it is built upon efficient runtime profiling.

```
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

Fig. 1. Loop graph example

2.2 Selection Criterion

We cannot select two loops that have nesting relations. If both loops have good performance, we have to decide which one to select. We use a criterion called benefit that considers speedup and coverage simultaneously. It is defined as follows:

$$\text{benefit} = \text{coverage} \times (1 - 1 / \text{speedup}) \tag{1}$$

The benefit value is used to measure the overall program performance gain if we parallelize this loop. The bigger this value is, the more likely we will select this loop. It is additive such that we can compute the benefit value for a set of loops by simply adding up all benefit values of loops in this set. The speedup for the whole program can be computed directly from benefit value as follows:

$$\text{program speedup} = 1 / (1 - \text{benefit}) \tag{2}$$

2.3 Loop Selection Problem

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

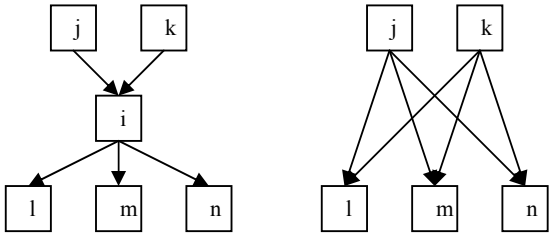
If we compute the transitive closure of the loop graph, the selection constraints requires that there is no edge between any two nodes. Now the problem is equivalent to the Weighted Maximum Independent Set Problem [40], which is a well-known NP-

complete problem. A set of nodes is called an independent set if there is no edge between any two of them.

2.4 Loop Selection Algorithms

Since the general loop selection problem is NP-complete, 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 benchmarks that we are studying, an efficient heuristic has to be used. Since a heuristic-based algorithm only gives sub-optimal solution, we should use it wisely. By applying a technique called graph pruning, we can find a reasonable approximation efficiently.

2.4.1 Graph Pruning



(a) Before node i is deleted (b) After node i is deleted

Fig. 2. Maintaining nesting relations in graph pruning

Before applying selection algorithms, we can simplify loop graph by pruning out those loops that will not be selected as speculative threads, such as: (i) loops that have less than 100 dynamic instructions on average are more appropriate for Instruction-Level Parallelism (ILP); (ii) loops that have no more than 2 iterations on average are more likely to underutilize multiple processor resources; (iii) loops with estimated speedup less than 1 could slow down the program execution if parallelized.

Each time we delete a node from the loop graph, we have to maintain the loop nesting relation between the remaining nodes. In Figure 2, after we remove node i, we have to create new edges from node j, k to node l, m, n respectively.

Graph pruning can reduce the size of loop graph by eliminating unsuitable loops. Moreover, after we delete some nodes, one single connected graph is split into several small disjoint sub-graphs. We can apply different loop selection algorithms to those small sub-graphs independently according to their sizes. It is efficient to use exhaustive searching algorithm for small sub-graphs. For larger sub-graphs, heuristic-based searching algorithm usually gives reasonable approximation.

2.4.2 Exhaustive Searching Algorithm

In this simple algorithm, we exhaustively try each independent loop set to find the one with maximum benefits. For each computed independent loop set, we maintain a vector to record all loops that have nesting relation to any loop within this independent set, and we call it conflict vector. By using conflict vector, it is easy to find a new independent loop to add into current independent set. After a new loop is added, the conflict vector is updated as well.

Exhaustive searching algorithm gives accurate solution for loop selection problem. However, its efficiency is the major concern in practice. Graph pruning creates small sub-graphs that are suitable for exhaustive searching. It works efficiently for sub-graphs with less than 50 nodes in our experiments.

2.4.3 Heuristic-based Searching Algorithm

Even after graph pruning, some sub-graphs are still very big. For those big sub-graphs, we use heuristic-based algorithms. We first sort all nodes in a sub-graph according to their benefit values. For all nodes that are independent of the selected independent set, we pick up the one with maximal benefit value and add it into the independent set. Similar to exhaustive searching algorithm, we create a conflict vector for the selected independent set and update it whenever a new node is added.

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

3 Experimental Framework

We implement loop selection algorithm on ORC compiler [41]. ORC compiler is an industrial strength open source compiler based on Pro64 compiler and targeting on Intel's Itanium Processor Family (IPF). We implemented our algorithm primarily in Code Generator (CG) phase of ORC.

For each selected loop, compiler inserts special instruction to mark the beginning and end of parallel loops. Fork instruction is inserted at the beginning of loop body. We optimize inter-thread value communication in CG using the techniques described in [34, 35]. Compiler synchronizes all inter-thread register dependences and memory dependences with probability greater than 20%. Both intra-thread control and data speculation are used for more aggressive instruction scheduling that increase the overlap between threads

Our execution-driven simulator is build upon Pin [42]. We simulate four single-issue in-order processors. Each of them has private L1 data cache, write buffer, address buffer and communication buffer. Write buffer holds the speculatively modified data. Address buffer keeps all memory addresses accessed by speculative loads. Communication buffer is for the data communicated between threads. The four processors share L2 data cache. The configuration of our simulated machine model is listed in Table 1.

3.1 Benchmarks

We study all the benchmarks from SPEC2000 integer suite with the exception of eon, which is written in C++. All simulation results are obtained with the ref input set, and all profiling information are obtained with the test input set. The statistics that we collect for each benchmark are listed in Table 2.

Most of benchmarks provide a large set of loops to select. It is extremely difficult to do loop selection without an automatic and systematic approach. The average loop iteration size is measured by using dynamic instruction count.

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

3.2 Simulation Methodology

To save the simulation time, we parallelize and simulate each loop once. After that, each time a selection technique is applied and a set of loops is selected, 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.

Since it is impossible to do full simulation for all loops, we use simple sampling method in our simulation. For each loop, we select the first 50 invocations for simulation. For each invocation, we simulate the first 50 iterations.

Table 2. Benchmark statistics.

Program	Number of Loops	Average Loop Iteration Size
Mcf	51	29605
Crafty	420	803553
Twolf	899	12437
Gzip	178	206755
Bzip2	163	109227
Vortex	212	45179
Vpr	401	1500
Parser	532	8820
Gap	1655	53721
Gcc	2619	5394

Perlbmk	729	1350

4 Loop Speedup Estimation

One of the key factors in our loop selection algorithm is being able to accurately predict the performance of parallel execution of all loops *through static analyses*. Our goal is to maximize the overall program performance represented as the benefit value of the selected parallel loops. The efficiency of a parallel loop is determined by both the coverage and speedup of this loop. In other words, given two completely nested loops, if we can only parallelize one of them, we will always choose to parallelize the loop with a larger benefit value.

In order to calculate the benefit value for each loop, we have to estimate both coverage and speedup of each loop. Coverage can be estimated using runtime profile. For speedup estimation, 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 (the number of time a loop iterates) by using profile. T_{seq} can be approximated using equation (3), where S is the average thread size and N is the average number of threads.

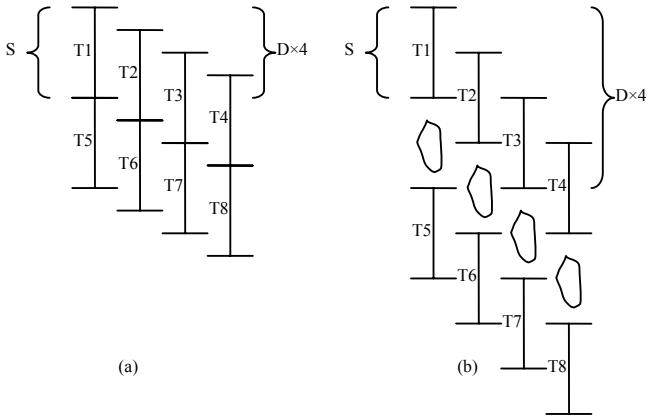


Fig. 3 Impact of delay D assuming 4 processors

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

On the other hand, the performance of parallel execution is determined by more factors, including the number of processors, the thread creation overhead, the cost of inter-thread value communication and the cost of mis-speculation. We divide 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 assuming that there is no mis-speculation. $T_{misspec}$ is the wasted execution time due to mis-speculation.

$$T_{par} = T_{perfect} + T_{misspec} \quad (4)$$

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.

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

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

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

If $D > S / p$, there are bubbles in pipelined execution of threads and the delay D has much high impact on the overall execution time as shown in Figure 3(b). We use equation (7) for estimation.

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

The key to accurately predict 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 mis-speculation of unlikely occurring data dependences. We will describe how we estimate $T_{misspec}$ first, and then three T_{comm} estimation techniques in the following sections.

4.1 $T_{misspec}$ Estimation

When a mis-speculation is detected, the violating thread will be squashed and all work that have been done by this thread become useless. We use the amount of work thrown away in a mis-speculation to quantify the impact of this mis-speculation on the overall parallel execution. Depending on when a mis-speculation is detected, the amount of wasted work could be different. 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 is detected at the end of previous thread. So that we could waste (S – D) cycles for one 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. This probability can be obtained through runtime profile.

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

4.2 T_{comm} Estimation I

One way to estimate the amount of time the parallel threads spent on value communication is to identify all the instructions that are either the producers or the consumers of some inter-thread data dependences and estimate the cost of value communication as the total cost of executing all such instructions. Although this estimation 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, consumer thread has to stall and wait until the producer thread 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, and so we refer to it as a critical forwarding path. In some cases, the critical forwarding path can become the dominating factor in parallel execution.

4.3 T_{comm} Estimation II

To take into consideration the impact of the critical forwarding path, we propose estimation technique II. Assuming load1, the consumer instruction in T2 is executed at cycle c2 and store1, the producer instruction in 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, rather it 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:

$$\text{criticalness} = (c1 - c2) / d \tag{9}$$

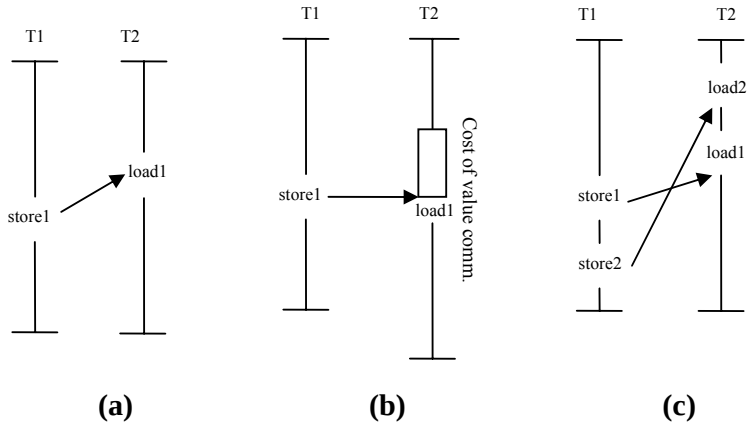


Fig. 4. The data dependence patterns between two speculative threads

There is one more mission piece for this estimation technique to be successful that is how to determine which cycle 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 the thread and the instruction in question. However, due to complex con-

trol flows that are inherit 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 path taken probability, as shown in equation (10).

$$c = \Sigma_{P \in all\_paths}(length(p) \times prob(p)) \quad (10)$$

For many loops, there are multiple data dependences exist between two speculative 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 the other synchronization can be complicated hidden.

Previous work has shown that the compiler can effective reduce the cost of synchronization through instruction scheduling and such optimizations are particularly useful for improving the efficiency of communicating register-resident scalars[34, 35]. Unfortunately, the estimation technique described in this section does not take such optimization into consideration, and it tends to be over estimate the cost of inter-thread value communication.

4.4 T_{comm} Estimation III

With T_{comm} estimation I underestimates the cost of value communication and estimation II overestimates the cost of value communication, it is desirable to find an estimation technique that considers both the critical forwarding path and the impact of instruction scheduling the reduce the critical forward path length. Thus, we have the third technique, in which the start time of an instruction based on data dependence graph. When there are multiple paths, in the data dependence graph, that can reach an instruction, we use $length(p_i)$ to represent the average length of path p_i. Average start time of this instruction can be measured as follows:

$$t = max(length(p_i)) \quad (11)$$

4.5 Evaluation

All three speedup techniques described above have been implemented in our loop selection algorithm to estimate T_{comm}. Three different sets of loops are selected, respectively. The impact of parallelizing each set of loops on the overall program performance is 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 for different estimations techniques.

We have observed: (i) for estimation I, the performance improvement obtained by most benchmarks are closed to the performance improvement obtained through simulation. However, for gzip, the loops selected using this optimistic estimation is completely wrong and results in a 40% performance degradation; (ii) the set of loops selected using estimation II is only able to achieve a fraction of the performance obtained by the set of loops selected using simulation results. This pessimistic estima-

tion technique tends to be conservative in selecting loops; (iii) the set of loops selected with estimation III always performs at least as well as the better set of loops selected using estimation I and estimation II.

Figure 6 illustrate the coverage of parallel execution on the total execution time. We have found that although the set of loops selected using simulation results demonstrated the most performance improvement, it does not always correspond to the large coverage. In mcf, although set of loops selected using estimation III has simulation performance as the set of loops selected using simulation results, the coverage of the simulation set is significantly smaller. This phenomenon suggests that, our estimation method may not be very accurate, but it is useful in helping use judiciously select a set of loops that have performance potential.

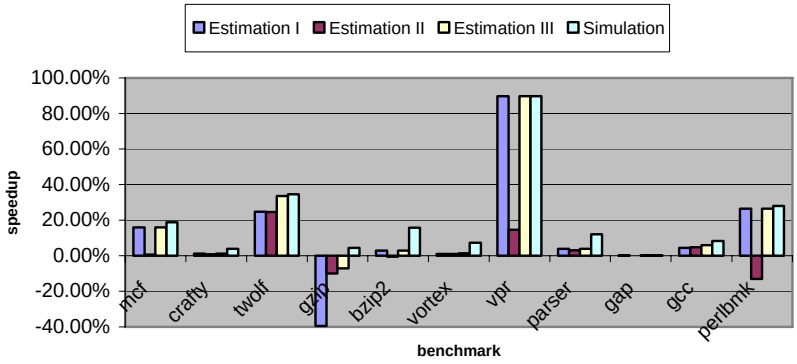


Fig. 5. Performance comparison

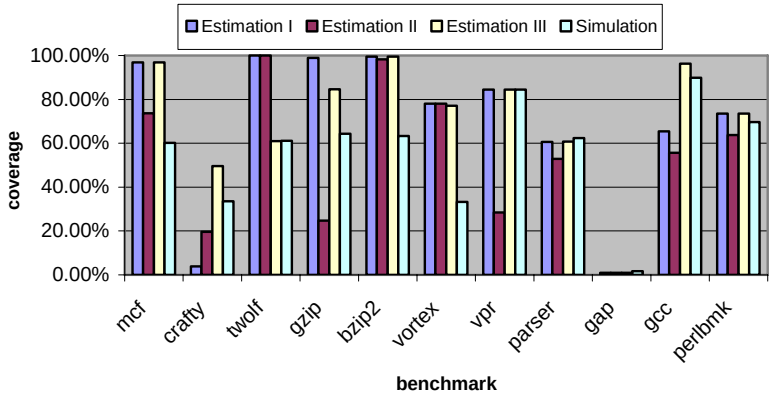


Fig. 6. Coverage comparison

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 times the loop iterates, the number of instructions executed per iteration and the data dependence patterns, and thus, demonstrate different parallel execution efficiency. Consequently, it might be desirable to only parallelize certain invocations of a loop. In this section, we concentrate on this phenomenon. In particular, we study 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 the path from the root node to a particular loop node as the calling context of that loop node. It is possible for a particular loop node to have several distinct calling contexts and it is also possible for loops with different calling context to behave differently. To study this behavior, we replicate the loop nodes for each distinct calling context. An example is shown in Figure 7, where the loop node goo_for1 has two distinct calling contexts, thus it is replicated into goo_for1_A and goo_for1_B. After the replication, the loop graph described in Section 2 is converted into a tree, and we refer to it as the loop tree. Figure 7 shows the loop tree derived from the example shown in Figure 1.

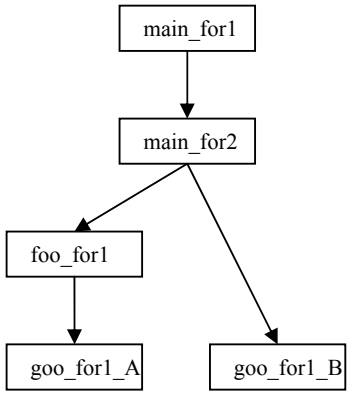


Fig. 7: Loop tree example

We perform loop selection under this context assuming different loop nodes correspond to completely different loops. Thus, a loop is parallelized under a certain calling context if the parallel execution speeds up under that calling context. Loop selection on the loop tree is relatively straightforward. The algorithm is as follows. We first traverse the loop tree bottom-up. For each node in the tree, we evaluate how

much execution time we can save if this particular loop is parallelized, and we refer to this number as $B_{current}$. We sum up the execution time we can save if we parallelize its descendants, we refer to this number as $B_{subtree}$. We record the larger of these two numbers as $B_{perfect}$ for this loop node. If $B_{perfect}$ equals to $B_{current}$, we mark this node as a potential candidate for parallelization. When we sum up the execution time saved if parallelizing the children of a loop node, we use $B_{perfect}$. We then traverse the loop tree top-down. Once we have encountered a loop node that is marked as a potential candidate for parallelization from previous step, we prune its children. The leaf nodes of the remaining loop tree correspond to loops that should be parallelized. The accurate solution can be found in polynomial time for a loop tree.

5.2 Dynamic Behavior of a Loop

It is possible for two different invocations of a loop to behave different even if they have the same calling context. To further study this behavior, we assume an oracle that can perfectly predict whether a particular invocation of a loop speeds up or not and only parallelize this invocation. A different set of loops are selected and evaluated assuming when such oracle is in place.

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 only parallelizing selected invocations of a loop as described in Section 5.2. The impact of such behavior on overall program performance is shown in Figure 8. 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 by an additional 2% by having an oracle that only parallelize invocations of loops that speed up. Thus, we found that the dynamic behavior of a loop is sensitive to its calling context. We also believe that a dynamic or static loop selection strategy that can predict whether a particular invocation of a loop speeds up or not can help us achieve additional program performance.

Figure 9 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 only parallelizing 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.

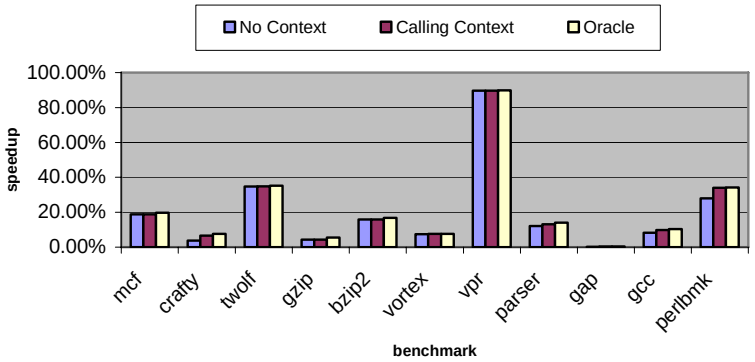


Fig. 8. Performance comparison

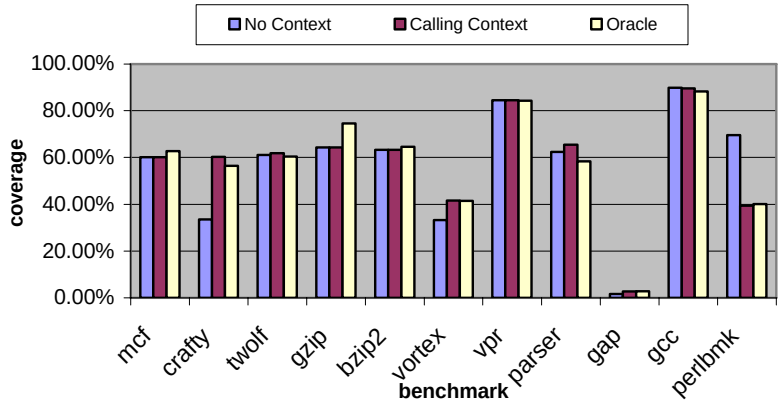


Fig. 9. Coverage comparison

6 Related Work

Colohan et al. [16] empirically studies the impact of thread size on the performance of loops. They employ different techniques to unroll loops in order to determine the best thread size per loop. 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 each loop dynamically. Due to the runtime overhead, the system can only select loops locally without considering loop nesting.

Oplinger et al. [24] proposes and evaluates a static loop selection algorithm using simulation. In their algorithm, they select the best loops in each level of dynamic loop nest as possible candidates to be parallelized and then compute the frequency with

which each loop is selected as best loop. Finally, they select the parallelized loops based on the computed frequencies. The concept of dynamic loop nest is similar to the loop tree proposed in our study. However, this technique is only used 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. [43] proposes a dynamic loop selection framework for 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. [16] and can only select loops within a simple loop nested program. Considering the global loop nesting relations and selecting the loops globally introduces significant overhead for the runtime system.

Johnson et al. [37] studies thread selection from consecutive basic blocks considering important factors such as thread predictability, data dependence and load imbalance. These threads are fine-grained and usually do not contain procedure calls or inner loops and complement threads based on loops.

Marcuello et al. [21] proposes a thread spawning scheme that supports spawning threads from any point in the program. They use profile to identify appropriate thread spawning points with more emphasis on thread predictability.

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 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 proposed a loop selection algorithm that takes the coverage of all loops and speedup achieved by parallelizing of these loop as inputs, then output the set of loops that should be parallelized to maximize program performance. 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 found that with the aid of profiling information, compiler analyses are able to come up with reasonably accurate estimate that allows our loop selection algorithm to select a set of loops to achieve good overall program performance. Furthermore, we have observed that some loops behave differently across different invocations. We study this phenomenon, and found that by exploiting this behavior and only parallelized invocations of a loop that actually speedup, we can potentially improve overall program performance even further for some benchmarks.

References

1. Tremblay, M. *MAJC: Microprocessor Architecture for Java Computing*. in *Hotchips '99*. August 1999.
2. *BroadCast Corporation: The Sibyte SB-1250 Processor, http://www.sibyte.com/mercurian*.
3. *Intel Pentium Processor Extreme Edition, "Delivering Dual-Core Four Thread Capability to the Desktop!"* http://www.intel.com/products/processor/pentiumXE/prodbrief.pdf, Editor.
4. *Sun Microsystem. "Throughput Computing"*, http://www.sun.com/processors/throughput/, Editor.
5. *IBM Corporation. "Planned 16-way 1.5 GHz IBM p5 575 Ultra-dense, Modular Cluster Node for High Performance Computing"*, http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/hpc_575_16w.pdf, Editor.
6. Emer, J. *Ev8: The post ultimate alpha. (Keynote address)*. in *the International Conference on Parallel Architectures and Compilation Techniques*. 2001.
7. Franklin, M. and G. Sohi, *ARB: A hardware Mechanism for Dynamic Reordering of Memory References.* IEEE Transactions on Computers, May 1996. **45**(5).
8. Kahle, J. *Power4: A Dual-CPU Processor Chip*. Microprocessor Forum October 1999
9. *Rice University. The Massively Scalar Compiler Group, http://softlib.rice.edu/MSCP/MSCP.html*.
10. *The SUIF Compiler System. http://suif.stanford.edu/suif/suif.html*.
11. Padua, D.A., J. Torrellas, and R. Eigenmann, *Automatic Parallelization of Conventional Fortran Programs. http://polaris.cs.uiuc.edu/polaris/polaris-old.html*.
12. Akkary, H. and M. Driscoll. *A Dynamic Multithreading Processor*. in *the proceedings of Micro-31*. December, 1998.
13. Chen, M. and K. Olukotun. *TEST: A Tracer for Extracting Speculative Threads*. in *the proceedings of 2003 International Symposium on CGO*. March 2003.
14. Cintra, M., J.F. Martinez, and J. Torrellas. *Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors.* . in *the proceedings of the 8th HPCA.* February 2002.
15. Cintra, M.H., J.F. Martínez, and J. Torrellas. *Architectural support for scalable speculative parallelization in shared-memory multiprocessors*. in *the proceedings of the ISCA*. 2000.
16. Colohan, C.B., et al., *The Impact of Thread Size and Selection on the Performance of Thread-Level Speculation.*
17. Frank, M., et al., *SUDS: Primitive Mechanisms for Memory Dependence Speculation. Technical Report. LCS-TM-591*. October 1998.
18. Gopal, S., et al. *Speculative Versioning Cache.* . in *the proceedings of the 4th HPCA*. February 1998.
19. Gupta, M. and R. Nim. *Techniques for Speculative Run-Time Parallelization of Loops.* in *the proceedings of Supercomputing'98.* November 1998.
20. Hammond, L., M. Willey, and K. Olukotun. *Data Speculation Support for A Chip Multi-processor*. in *the proceedings of ASPLOS-8*. October 1998.
21. Marcuello, P., A. González, and J. Tubella. *Speculative Multithreaded Processors*. in *the proceedings of International Conference on Supercomputing* 1998.
22. Marcuello, P. and A. Gonzlez. *Clustered Speculative Multithreaded Processors.* in *the proceedings of MICRO-32* November 1999.

23. Olukotun, K., L. Hammond, and M. Willey. *Improving the Performance of Speculatively Parallel Applications on the Hydra CMP*. in *the proceedings of the ACM Int. Conf. on Supercomputing*. June 1999.
24. Oplinger, J., D. Heine, and M. Lam. *In Search of Speculative Thread-Level Parallelism.* in *the proceedings of PACT*. October 1999.
25. Rauchwerger, L. and D.A. Padua, *The LRPD Test: Speculative RunTime Parallelization of Loops with Privatization and Reduction Parallelization.* IEEE Transactions on Parallel Distributed Systems, 1999. **10**(2): p. 160-180.
26. Rotenberg, E., et al. *Trace Processors*. in *the proceedings of International Symposium on Microarchitecture*. 1997.
27. Sohi, G.S., S.E. Breach, and T.N. Vijaykumar. *Multiscalar Processors.* . in *the proceedings of the 22nd ISCA*. June 1995.
28. Tsai, J.-Y., et al., *The Superthreaded Processor Architecture.* IEEE Transactions on Computers, September 1999.
29. Steffan, J.G., et al. *Improving Value Communication for Thread-Level Speculation*. in *the proceedings of the 8th HPCA*. February 2002.
30. Krishnan, V. and J. Torrells. *The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors.* in *the proceedings of the ACM Intl. Conf. on Supercomputing*. June 1999.
31. Marcuello, P., J. Tubella, and A. Gonzlez. *Value Prediction for Speculative Multithreaded Architectures.* in *the proceedings of Micro-32*. November 1999.
32. Moshovos, A.I., et al. *Dynamic Speculation and Synchronization of Data Dependences.* in *the proceedings of the 24th ISCA*. June 1997.
33. Steffan, J.G. and T.C. Mowry. *The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization*. in *the proceedings of the 4th HPCA*. February 1998.
34. Zhai, A., et al. *Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads*. in *the proceedings of the 2004 CGO*. March 2004.
35. Zhai, A., et al. *Compiler Optimization of Scalar Value Communication Between Speculative Threads*. in *the proceedings of the 10th ASPLOS*. October 2002.
36. Vijaykumar, T.N., *Compiling for the Multiscalar Architecture*, in *Computer Science Department*. January 1998, University of Wisconsin at Madison.
37. Johnson, T.A., R. Eigenmann, and T.N. Vijaykumar. *Min-cut program decomposition for thread-level speculation*. in *the proceedings of PLDI*. 2004.
38. Steffan, J.G., et al. *A Scalable Approach to Thread-Level Data Speculation*. in *the proceedings of the 27th ISCA*. June 2000.
39. Chen, T., et al. *Data Dependence Profiling for Specu-lative Optimization*. in *the proceedings of the 13th Intl. Conf. on Compiler Construction*. March 2004.
40. Du, D.Z. and P.M. Pardalos, *Handbook of Combinatorial Optimization.* Vol. 4. 1999: Kluwer Academic Publishers.
41. *Open Research Compiler for Itanium Processor Family.* http://ipf-orc.sourceforge.net/.
42. Luk, C.-K., et al. *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation.* . in *the proceedings of the ACM Intl. Conf. on Programming Language Design and Implementation*. June 2005.
43. Chen, M. and K. Olukotun. *The Jrpm System for Dynamically Parallelizing Java Programs.* in *the proceedings of the 30th ISCA*. June 2003.