A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs

Zhao-Hui Du
Intel China Research Center
Intel China Ltd.
Beijing, China
zhao.hui.du@intel.com

Chu-Cheow Lim
Programming Systems Lab
Intel Corporation
Santa Clara, California, USA
chu-cheow.lim@intel.com

Xiao-Feng Li
Intel China Research Center
Intel China Ltd.
Beijing, China
xiao.feng.li@intel.com

Chen Yang
Intel China Research Center
Intel China Ltd.
Beijing, China
chen.yang@intel.com

Qingyu Zhao
Intel China Research Center
Intel China Ltd.
Beijing, China
qingyu.zhao@intel.com

Tin-Fook Ngai
Programming Systems Lab
Intel Corporation
Santa Clara, California, USA
tin-fook.ngai@intel.com

ABSTRACT
The emerging hardware support for thread-level speculation opens new opportunities to parallelize sequential programs beyond the traditional limits. By speculating that many data dependences are unlikely during runtime, consecutive iterations of a sequential loop can be executed speculatively in parallel. Runtime parallelism is obtained when the speculation is correct. To take full advantage of this new execution model, a program needs to be programmed or compiled in such a way that it exhibits high degree of speculative thread-level parallelism. We propose a comprehensive cost-driven compilation framework to perform speculative parallelization. Based on a misspeculation cost model, the compiler aggressively transforms loops into optimal speculative parallel loops and selects only those loops whose speculative parallel execution is likely to improve program performance. The framework also supports and uses enabling techniques such as loop unrolling, software value prediction and dependence profiling to expose more speculative parallelism. The proposed framework was implemented on the ORC compiler. Our evaluation showed that the cost-driven speculative parallelization was effective. Our compiler was able to generate good speculative parallel loops in ten Spec2000Int benchmarks, which currently achieve an average 8% speedup. We anticipate an average 15.6% speedup when all enabling techniques are in place.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *compilers, optimization, code-generation*.

General Terms
Languages, Design, Algorithms, Performance.

Keywords
Speculative multithreading, speculative parallel threading, thread-level speculation, speculative parallelization, cost-driven compilation, loop transformation.

1. INTRODUCTION
Hardware support for speculative multithreading has been extensively investigated as a way to speed up hard-to-parallelize sequential programs [3][4][10]. It is generally recognized that to take full advantage of the hardware thread-level speculation support, software support is required to identify and create likely successful speculative threads and to avoid poor speculations.

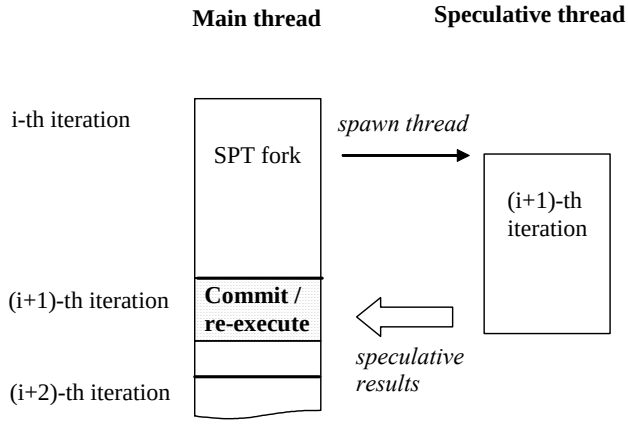

Figure 1: An SPT Execution Model

We propose a comprehensive cost-driven compiler framework to speculatively parallelize loops of sequential programs. We developed a misspeculation cost model and used it to drive the

speculative parallelization. The compiler aggressively looks for loop-level speculative execution opportunities, selects and transforms candidate loops into speculative parallel loops that likely can gain from the speculative parallel threading (SPT) execution. The proposed framework also allows the use of enabling techniques such as loop unrolling, value prediction and dependence profiling to expose more speculative parallelism.

Figure 1 depicts a general speculative parallel threading (SPT) execution model. The main thread is the only non-speculative thread that can commit and change the program state. It executes the main program. The threads share the same memory space but have their own context and execution states. During the parallel execution, the threads do not communicate. For the speculative threads, all speculative results (including memory writes) are buffered and are not part of the program state.

Suppose the main thread is executing the i-th iteration of a speculative parallel loop. When it executes an SPT fork instruction, it spawns a speculative thread to execute the next iteration (i.e., the (i+1)-th iteration) speculatively. To start the speculative execution, the context of the main thread is copied to the speculative thread as its initial context. Besides, the start address of the next iteration where the speculative thread starts execution is encoded in the SPT fork instruction and passed to the speculative thread.

After the speculative thread is created, the main thread continues its normal execution with the current iteration. When the main thread comes to the point where the speculative thread started execution (i.e., beginning of the next iteration), it will check the speculative execution results of the speculative thread and perform any of the following actions. If there is no dependence violation, the main thread commits the entire speculative results (i.e., the context of the speculative thread) at once. Otherwise, the main thread will commit those correct speculative results and for those incorrect results it re-executes the corresponding misspeculated instructions. The main thread then resumes normal program execution when it catches up with the speculative thread (i.e., when there is no more speculative results.)

When the main thread executes an SPT kill instruction (usually at the exit of an SPT loop), it kills any running speculative threads.

```
while (i<n) {
 cost0=0;
 for (j=0;j<i;j++) {
   cost0+=fabs(error[i][j]- p[j]);
 }
 cost+=cost0;
 i++;
}
```

(a) Original loop

```
temp_i=i;
while (i<n) {
  i=temp_i;
  temp_i=temp_i+1;
  SPT_FORK(loop_id);
  cost0=0;
  for (j=0;j<i;j++) {
    cost0+=fabs(error[i][j]-p[j]);
  }
  cost+=cost0;
  i=temp_i;
}
SPT_KILL(loop_id);
```

(b) SPT-transformed loop

Figure 2: An SPT Loop Transformation Example

Figure 2 shows how a loop can be transformed into a speculative parallel loop or SPT loop. In the SPT loop (Figure 2(b)), the SPT_FORK statement represents the SPT fork instruction. The code before the fork instruction is called the **pre-fork region** of the SPT loop. The code after the fork instruction is called the **post-fork region**. The partitioning of a loop body into pre-fork region and post-fork region has special meaning in the speculative parallelization. (Hereafter we refer such a partition **an SPT loop partition**.) Because all statements in the pre-fork region are executed before the speculative thread is spawned, their results are visible to the speculative thread. Their uses in the speculative thread do not violate any data dependences and are always correct. On the contrary, a statement in the post-fork region is executed after the speculative thread starts. If its result is supposed to be used in the next iteration, the speculative thread may use the old value in its speculative execution and violates the true data dependence. The corresponding speculative instruction and all its dependent instructions need to be re-executed.

One key speculative parallelization transformation in our framework is code reordering. If a define statement in the post-fork region can be moved into the pre-fork region, all re-executions due to its data dependence violation can be avoided. The example in Figure 2 illustrates this transformation. The increment of the induction variable i at the end of the original loop (Figure 2(a)) will cause many re-executions if it remains in the post-fork region. The SPT loop transformation moves it from the end of the loop body to the pre-fork region (Figure 2(b)) and avoids the corresponding re-executions. Note that the temporary variable temp_i is introduced to allow overlapping of the life ranges of the old and new values of the variable i.

We have implemented the proposed compilation framework on the Open Research Compiler (ORC) [8] and used it to generate speculative parallel loops. ORC is an open-source C/C++/Fortran compiler targeting Intel's Itanium™ processor family (IPF) processors. The performance of ORC 2.1 is on par with the Intel IPF product compiler ecc 7.0 and 30% ahead of gcc 3.1 on an Itanium system for the spec2000 integer benchmarks. We implemented our speculative parallelization framework primarily in the machine-independent scalar global optimization (WOPT) phase of ORC, right after its loop-nested optimization (LNO) phase. Most analyses and transformations were done in ORC's SSA form [14].

The rest of this paper is organized as follows. Section 2 discusses related work. In Section 3, we give an overview of our compilation framework. Section 4 describes the misspeculation cost model that we use to drive the speculative parallelization. Section 5 describes how we obtain the optimal loop partition that generates the least amount of re-executions for a particular loop candidate. Section 6 describes the final SPT loop selection and transformation. This framework was designed to support a number of important techniques that enables speculative parallelization. Section 7 describes the enabling techniques being supported and used. In Section 8, we present results to evaluate our framework and the generated SPT code. Section 9 concludes the paper.

2. RELATED WORK

The Multiscalar project was the first comprehensive study of both hardware and software supports for speculative multithreading

[3][12]. In particular, Vijaykumar et. al. described the use of compiler techniques to partition a sequential program into tasks for the Multiscalar architecture [12]. They showed that good task selection is crucial to the performance achieved by the Multiscalar architecture, as it pertains to performance issues such as control flow/data speculation, register communication and load imbalance. Various compiler heuristics were developed for task selection and register communication.

Tsai et. al. described how basic techniques such as alias analysis, function inlining and data dependence analysis could be used to partition a program into threads which are then pipelined in the superthread architecture [11]. For their experiments, the benchmark programs were manually transformed at the source level.

Zhai et.al. studied compilation techniques to improve thread-level speculation performance [13]. They focused on code scheduling and inter-thread communication optimization. They showed that the compiler could reduce value communication delay between the threads significantly by forwarding scalar values to the speculative threads and by inserting synchronization instructions into the threads.

More recently, Chen et. al. described a dynamic parallelization system that transformed sequential Java programs to run on a chip multiprocessor that supports speculative threads [1][2]. They used hardware profiling support to guide speculative thread decomposition. They estimated the execution time of a speculative loop from the profiling data (such as time stamps collected during profile runs) and based on the estimation selected which nested level of a loop to be run speculatively. While our speculative parallelization also does profiling and loop selection, our work provides a more general speculative parallelization framework and applies more aggressive cost-directed loop transformation and selection at compilation time.

3. SPT COMPILATION FRAMEWORK

This section gives an overview of our cost-driven compilation framework. There are two key elements in our speculative parallelization approach. First it is *cost-driven*. Because speculation does not always gain performance, it is essential to estimate and reduce the cost associated with misspeculation. We developed a misspeculation cost model to drive our speculative parallelization. The second element is *aggressive but careful selection*. We use a two-pass compilation process to explore every speculative parallelization opportunity, obtain the best transformation and select only loops that are likely to deliver performance gain.

3.1 Cost-driven transformations

The notion of **misspeculation cost** is central to our speculative parallelization. Given a particular SPT loop partition, its misspeculation cost is defined as the expected amount of misspeculated computation that needs to be re-executed within a speculative executed loop iteration.

Figure 3 shows the core of our compilation framework.

The central service component is the misspeculation cost computation. We developed a misspeculation cost model to estimate misspeculation cost of an SPT loop partition. The cost model is built using annotated control-flow and data dependence graphs. The following section will describe in detail the cost model construction and how it is used to compute misspeculation cost.

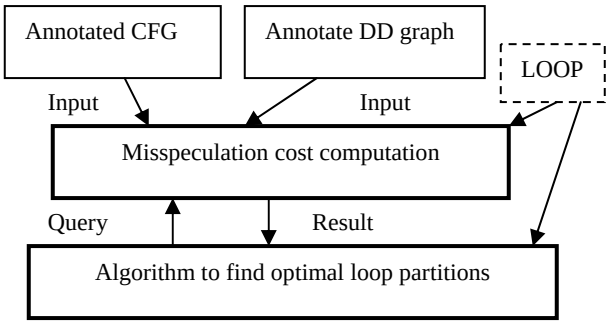

Figure 3: Core of compilation framework

The bottom component is the algorithm to find an optimal SPT loop partition for a given loop. The purpose of this component is to determine if a loop can become a good SPT loop. It searches for all possible SPT loop partitions of the loop, evaluate their misspeculation costs, and determine the best SPT loop partition that generates the least misspeculation cost. We will describe this algorithm in detail in Section 5.

3.2 Two-pass compilation

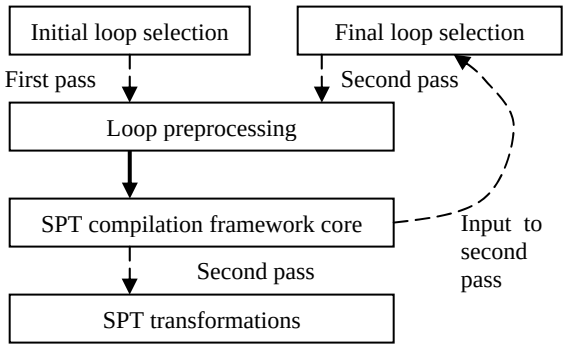

Figure 4: Overall framework -- Combination of core with compiler transformation techniques

Figure 4 shows our overall framework with a two-pass compilation process. The purpose of the two-pass compilation process is to select and transform only all good SPT loops in a program.

In the first pass compilation, the initial loop selection selects all loops that meet simple selection criteria (such as the loop body size requirement) as SPT loop candidates. Loop preprocessing such as loop unrolling and privatization is then applied to transform the loops into better forms for speculative parallelization. For each loop candidate, the SPT compilation

framework core is invoked to determine its best SPT loop partition. The result of the first pass is a list of SPT loop candidates with their optimal SPT loop partition results. All loop transformations are performed in a tentative manner and do not alter the compiled program. No actual SPT code is generated. This pass allows us to measure the amount of speculative parallelism in all loop candidates (including each nested level of a loop nest) and evaluate their potential benefits/costs.

The second pass takes the result list from the first pass and performs the final SPT loop selection. It evaluates all loop candidates together (not individually as in the case in the first pass) and selects only those good SPT loops. Then the selected loops are again preprocessed and partitioned. Finally the compiler applies the SPT transformations to generate the SPT loop code.

In Section 6, we will describe the SPT loop selection and transformation in detail.

4. MISSPECULATION COST MODEL

The misspeculation cost model is the central component in this cost-driven framework. For a given loop, we build a simplified control-flow graph and a dependence graph annotated with dependence probabilities. A misspeculation cost model is constructed based on these graphs and is used to compute the misspeculation cost of a loop partition.

4.1 Data-dependence graph

A dependence graph is built for each loop body. In order to indicate the likelihood of a particular data dependence, each true data-dependence edge in the graph is annotated with a probability value. A probability value of p on an edge $W \rightarrow R$ means for every N writes at W, only pN reads will access the same memory location at R during program execution.

These data dependence probabilities form the base reason for speculation. If we have a cross-iteration $W \rightarrow R$ edge with a low p value, we know that that it is unlikely for the write in the main thread to cause a misspeculated read in the speculative thread so that we could speculate it. A high p value on the other hand suggests a high probability of a misspeculated read when W is in the post-fork region. In such a case, to reduce the misspeculation cost, the write W should be moved to the pre-fork region.

4.2 Misspeculation cost computation

This subsection describes the details of the misspeculation cost model and how it is used to compute misspeculation cost. Misspeculation cost is a useful metric to evaluate the potential performance gains of an SPT partition. It gives, for a given loop partition, the expected amount of computation (typically in number of elementary operations) within a speculative loop iteration that needs to be re-executed. A higher cost means that the main thread needs more time to re-execute the misspeculated instructions, and hence such a partition is less desirable.

We build a cost graph to represent the code re-execution dependency due to misspeculation. Suppose we have an expression E, and it has a read R in a sub-expression. If R is misspeculated and has to be re-executed, this will cause E to be re-executed as well.

4.2.1 Violation candidate

The source of a cross-iteration true data-dependence edge is called the **violation candidate**[1]. A violation candidate will introduce some misspeculation cost if it is not in the pre-fork region. On the other hand, if the violation candidate is in the pre-fork region, it is executed sequentially before the speculative thread starts. It is not part of the parallel execution and the amount of parallelism is reduced.

4.2.2 Cost graph construction

The control-flow and data-dependence graphs are used to construct the cost graph. The cost graph is initialized with the set of violation candidates and their cross-iteration dependence edges. Then nodes in the dependence graph that can be reached by the dependence edges and their intra-iteration dependence edges are added to the cost graph recursively. To estimate the misspeculation cost more accurately, the head of a cost graph edge is an operation, rather than a statement[2]. Except the initial set of cost graph nodes, each subsequent node added to the cost graph represents an operation.

Conceptually the initial set of cost graph nodes represents the statements in the main thread that may cause misspeculation in the speculative thread. The subsequent portion of the graph represents the propagation effect of any misspeculation and re-execution within the speculative thread iteration.

Each edge $X \rightarrow Y$ in the cost graph is annotated with a probability p. This is the conditional probability that a re-execution at X will cause Y to be misspeculated and re-executed, given that X is misspeculated.

4.2.3 Computation of re-execution probability

The misspeculation cost *of a given partition* is calculated based on an estimate of the re-execution probability of each operation in the speculative thread. This value is updated during the misspeculation cost computation.

The following is an algorithm to estimate the re-execution probability of each node in the cost graph.

Steps 1 and 2 are preparation steps which are performed once for a given loop candidate.

1. We calculate the violation probability of each violation candidate node in the cost graph. The violation probability is the probability that the result of the corresponding statement is modified within an iteration. For a violation candidate, its violation probability means how often the main thread will reach it and modifies its results, thus initiating re-executions in the next but speculatively executed iteration.
2. Topologically sort all nodes in the cost graph.
3. For each node corresponding to a violation candidate not in the pre-fork region, initialize its re-execution probability to be its violation ratio.

[1] Each violation candidate is an SSA statement in our representation.

[2] In terms of actual implementation, each cost graph node corresponds to an operation (Coderep) rather than a statement (Stmtrep) in ORC's SSA representation.

4. The remaining nodes in the cost graph are visited in the topologically sorted order. We compute the probability that a node c is re-executed due to re-execution of ***any*** of its predecessors as follows:
    a. Initialize x = 0;
    b. For each predecessor node p of c,

       x = 1 - (1 - x)*(1 - r*v(p)), where v(p) is the re-execution probability of node p, and r is the dependency probability on edge p->c (as described in Section 4.2.2).
    c. Set the re-execution probability of c to be x

For a node dependent on multiple predecessors, this algorithm assumes that the re-execution probabilities of different nodes are independent of one another. So this algorithm only approximates the misspeculation cost.

4.2.4 Computation of misspeculation cost

The misspeculation cost of the given SPT partition is computed as:

$$\sum_{c} v(c) * Cost(c)$$

The summation is for all nodes c in the cost graph (excluding those for the violation candidates), v(c) is the re-execution probability of node c and *Cost(c)* is the amount of computation in node *c*.

4.2.5 An example

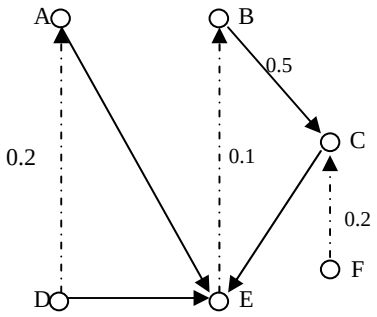

Figure 5: A dependence graph example

Figure 5 shows a data dependence graph of a loop body. For clearer illustration, we assume there is no branch statement in the loop body. All solid lines represent intra-iteration dependences while the dashed lines represent cross-iteration true data dependences. The dependence probability for the edges without annotation is 1.

The cost graph is shown in Figure 6. D', E' and F' are the pseudo nodes for violation candidates D, E and F respectively.

Let us consider an SPT partition in which only D is in the pre-fork region.

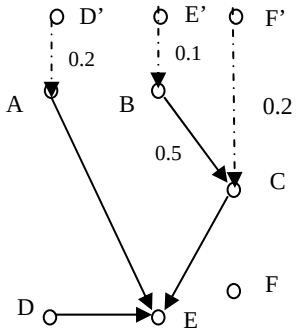

Figure 6: Cost graph for the dependence graph in Figure 5.

The re-execution probabilities of the pseudo nodes for the violation candidates are initialized.

v(D') = 0 (since D' is in pre-fork region)

v(E') = v(F') = 1 (since there is no branch)

The re-execution probability algorithm then calculates the re-execution probabilities of other nodes in topological order.

v(A) = 1 – (1– 0.2*v(D')) = 0

v(B) = 1 – (1 – 0.1*v(E')) = 0.1

v(C) = 1 – (1 – 0.5*v(B))(1– 0.2*v(F')) = 0.24

v(D) = v(F) = 0

v(E) = 0.24

Assuming all nodes have cost of one, the misspeculation cost of the SPT partition is 0.58.

5. OPTIMAL LOOP PARTITION

As described in Section 3, one key algorithm in our cost-driven framework is to determine an optimal SPT loop partition for a given loop. In this section, we describe how we formulate the problem and how we search for an optimal partition.

We formulate the optimal loop partitioning problem as an optimization problem: To find a legal loop partition such that its misspeculation cost is minimum under the constraint that the pre-fork region size is no more than a given threshold.

Legal partitions are those which maintain program correctness. Any partition that maintains all forward intra-iteration dependence edges is a legal partition of the loop. A partition that causes a forward intra-iteration edge to become backward is illegal[3].

The fork region size threshold is imposed to limit the amount of sequential execution, thus allowing some minimum degree of parallelism after the speculative parallelization.

A branch and bound search algorithm is used to find the optimal partition. Search efficiency is one of our major concerns. We reduce the search space by focusing on violation candidates only. We construct a violation candidate dependence graph (VC-dep graph) to facilitate the search. Furthermore, the search tree is pruned more effectively by observing that both the misspeculation

[3] This restriction can be relaxed when intra-thread speculation is allowed.

cost and size of pre-fork region are monotone functions of the set of statements in the pre-fork region. When additional statements are moved into the pre-fork region, the misspeculation cost will be reduced (compared to the partition prior to the move) and the size of the pre-fork region becomes larger.

5.1 Construction of VC-dep graph

A violation-candidate dependence graph (VC-dep graph) is built from the data dependence graph. The node set of the VC-dep graph is the set of violation candidates. A node N in the VC-dep graph is a successor of another node S iff the corresponding violation candidate for N is directly or indirectly dependent on the corresponding violation candidate for S in the data dependence graph. Only intra-iteration data dependences are considered.

All nodes in the VC-dep graph are topologically sorted before the search starts. That is for any two nodes u and v in the graph, if u is dependent on v, u must have a larger topological order number than v.

5.2 The optimal search algorithm

The search algorithm starts with an empty pre-fork region. In each step, we will add one node of the VC-dep graph into the pre-fork region if all its predecessors have been added into the pre-fork region. Since there could be multiple candidate nodes to be added, each candidate is added separately to explore all possible cases whenever there is a branch in the search space.

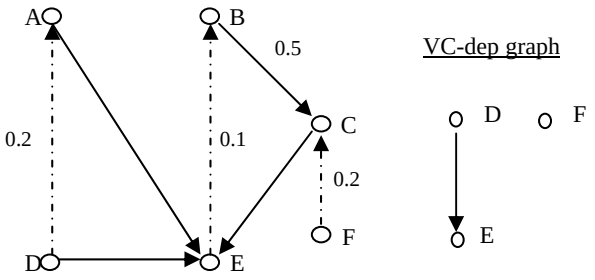

Figure 7: VC-dep graph for the dependence graph in Figure 5

Let us consider the dependence graph example in Figure 5 again. Figure 7 shows the corresponding VC-dep graph. There are three violation candidates D, E and F. E is a successor of D in the data dependence graph.

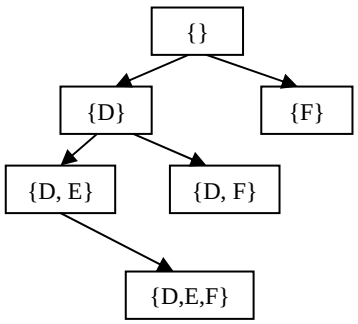

Figure 8: Search space. Each node represents a pre-fork region which completely defines a partition.

Figure 8 gives the search space formed by searching all the different pre-fork regions.

To avoid searching the pre-fork region {D, E, F} multiple times, a constraint is added. At each step, only the node with larger topological order number could be added into the pre-fork region.

5.2.1 Pruning heuristics

To speedup the search, we adopt two pruning heuristics:

1. When a pre-fork region's size exceeds the given threshold, there is no need to search down the search tree.
2. Suppose the current search node has a pre-fork region R. Because we never move the nodes in current post-fork region that has smaller topological order number than those in R into the pre-fork region when searching the offspring nodes of the current search node, we could estimate the lower bound cost for all offspring nodes in the search space. If this lower bound is already larger than the minimum misspeculation cost found so far, there is no need to search successors of the current node.

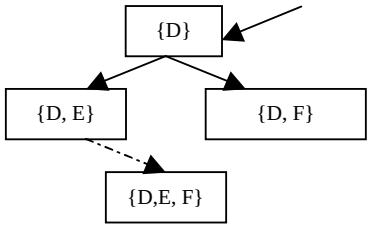

Figure 9: Search space pruning. The dash edge is pruned.

Figure 9 illustrates the application of heuristic 1 applied to the search of the search space in Figure 8. The search is pruned at node {D, E, F} because the size of its pre-fork region already exceeds the threshold.

The search algorithm still has exponential time complexity. To avoid exceedingly long compilation time, loops with too many violation candidates are skipped. Our experiments show that only a few loops in Spec2000Int have more than 30 violation candidates and got skipped due to this reason.

6. SPT LOOP SELECTION & TRANSFORMATION

After the first compilation pass, we obtain all optimal partitions and their associated optimal misspeculation costs for all loop candidates. In the second compilation pass, our speculative parallelization examines all loop candidates together (such as all nesting levels of a loop nest) and select all those good SPT loops that are likely to deliver performance gain and does the final SPT transformation to generate SPT code. This section describes the final SPT loop selection and transformation.

6.1 SPT loop selection criteria

We select SPT loops based on the following criteria:

1. Misspeculation cost. The optimal misspeculation cost of the loop must be smaller than a predefined threshold which is a

fraction of the loop body size. This attempts to limit the potential performance loss due to misspeculation.

2. Pre-fork region size. The size of the pre-fork region is less than a predefined threshold which is a fraction of the loop body size. This limits the sequential execution portion in the speculative parallel execution. The same threshold is used in the first pruning heuristic (Section 5.2.1).
3. Loop body size. Loop body size is a fundamental characteristic of an SPT loop. It shows how far ahead of the current execution a speculative thread starts the speculative execution. It also indicates roughly the size of the speculative thread.
    - It should be larger than a predefined threshold. If the loop body size is too small, the performance gain of speculative parallelism will not be enough to compensate for the overhead of forking a thread.
    - It should be smaller than a machine-dependent threshold. Hardware resources can only support speculative execution of limited size.
4. Iteration count. A small iteration count (especially a number smaller than 2) means the next iteration is not likely to be executed and any speculative thread is likely to be killed. We avoid speculatively parallelizing such loops.

6.2 SPT loop transformation

After the SPT loop selection, the original loop bodies of the selected loops are transformed into SPT loops using the optimal partition results obtained earlier.

To simplify the transformation procedure, the CFG of original loop is duplicated with empty basic blocks as the initial CFG of the pre-fork region. Then the statements are moved from the original loop body (which becomes the post-fork region) into the pre-fork region according to the optimal partition result. After that, the pre-fork and post-fork regions are connected together with the SPT_FORK statement inserted between them.

There are two complications in the SPT loop transformation. One is how to deal with overlapped live ranges after code reordering. The other one is to deal with code motion of partial conditional statements.

```
i_3=phi(i_1,i_2)   (1)
foo(i_3);          (2)
i_2=i_3+1;         (3)
```

(a) Original loop

```
i_3=phi(i_1,i_2)   (1)
i_2=i_3+1;         (3)
SPT_FORK(loopid)
foo(i_3);          (2)
```

(b) SPT loop partition

Figure 10: Life-range overlap after partition

After code reordering, the life ranges of different definitions of the same variable may be overlapped. For example, in Figure 10, suppose the partition result requires statements (1) and (3) to be moved into the pre-fork region. After the code is moved, we find that the life range of *i_2* overlaps with the life range of *i_3*. This is not permitted in the SSA form in ORC.

```
   temp_var=i_1;
start_of_loop:
   i_3=temp_var;
   temp_i_3=i_3;
   i_2=i_3+1;          (3)
   temp_i_2=i_2;
   temp_var=i_2;
   SPT_FORK(loopid)
   i_3'=temp_i_3;
   foo(i_3');          (2)
   i_2'=temp_i_2;
```

Figure 11: Temporary variable insertion to avoid the life-range overlap in Figure 10.

(Code is in SSA form with phi-nodes not shown.)

The above problem is solved by introducing temporary variables to break the life ranges. Figure 11 shows the immediate results after the temporary variables are inserted. The temporary variable *temp_i_2* is used to avoid the life-range overlap within the loop body while the temporary variable *temp_var* carries the cross-iteration definition (3).

After the above code motion and temporary variable insertions, the code is immediately cleaned and optimized by applying SSA renaming, copy propagation and dead code elimination in ORC.

When there is a branch statement inside the loop body and some code control-dependent on the branch statement is to be moved into the pre-fork region, the branch statement needs to be copied into the pre-fork region as well. In Figure 12, the partial conditional statement '*if (x<y) s++;*' is moved into the pre-fork region. The branch statement '*if (x<y)*' is replicated so that the transformed code maintains the correct control flow in both the pre-fork and post-fork regions.

```
…
if (x<y) {
    s++;
    foo();
}
```

```
…
temp_cond=(x<y);
if (temp_cond) {
    s++;
}
SPT_FORK(loopid)
…
if (temp_cond) {
    foo();
}
```

Figure 12: Moving a partial conditional statement into the pre-fork region

7. SPT-ENABLING TECHNIQUES

While the core SPT speculative parallelization attempts to identify and transform the best SPT loops, other SPT-enabling techniques are needed to expose more speculative parallelism to the compiler. Our compilation framework can be easily extended to support and use some of these techniques. This section

describes three techniques which our framework has incorporated – loop unrolling, software value prediction and data-dependence profiling.

7.1 Loop unrolling

Our current SPT compilation is focused on generating speculative threads for next iterations of a loop. It is therefore important that the loop body size be sufficiently large to compensate for the overheads of the speculative thread execution. Loop unrolling is a useful technique to increase the body size of a loop.

The SPT compilation does loop unrolling as follows. Before the Loop Nested Optimization (LNO) phase, based on the loop body size and the minimum SPT loop body size requirement, the SPT compilation module decides if a loop should be unrolled and what the unroll factor will be. A loop-unrolling pragma is inserted into the loop's header. We modified LNO to recognize this pragma and perform the required loop unrolling (including outer loop unrolling).

Loop unrolling is always enabled in all our experiments.

Since ORC can only unroll DO loops in LNO, many loops are still not unrolled automatically in the current compiler.

7.2 Software value prediction

We have developed a software value prediction (SVP) technique to predict critical values purely in software without any hardware support. Based on our misspeculation cost model, the compiler identifies critical dependences that cause unacceptably high misspeculation cost. The compiler then instruments the program to profile the value patterns of the corresponding variables. If the values are found to be predictable, and both the corresponding value-prediction overhead and the mis-prediction cost are acceptably low, the compiler inserts the appropriate software value prediction code to generate the predicted values. It also generates the software check and recovery code to detect and correct potential value mis-prediction [6].

```
while (x) {
   foo(x);
   x=bar(x);
}
```

(a) Original loop

```
pred_x = x;
while (x) {
   x = pred_x;
   pred_x = x + 2;
   SPT_FORK(loopid);
   foo(x)
   x = bar(x);
   if (x != pred_x) {
      pred_x = x;
   }
}
```

(b) Speculative parallel loop

Figure 13: Speculative parallelization with software value prediction

Figure 13 gives a software value prediction example. The compiler profiles the values for *x*. We assume that a pattern is identified whereby *x* is often incremented by 2 by *bar(x)*. A check-and-recovery code is also inserted at the end of the loop iteration. If the actual value of *x* is different from the predicted value during execution time, *pred_x* is corrected with the right value.

In our cost-model, *x=bar(x)* is a violation candidate which cannot be moved to the pre-fork region because of the legality constraints imposed by code reordering. SVP provides a way to overcome this limitation so the loop can become an SPT loop.

7.3 Data-dependence profiling

As described in Section 4, data dependence probability is an essential basis for speculation. Data dependence profiling is one important means to obtain reliable dependence probability information.

Our current SPT compilation includes a data-dependence profiling tool to complement the static type-based memory disambiguation analysis in ORC. The profiling is done offline. The results are used during pass 1 compilation (together with the reaching probability information of the control-flow graph) to annotate both intra-iteration and cross-iteration true data dependence edges. These edge probabilities are in turn used to annotate the edges in the cost graph (Section 4.2.2). The goal is to better estimate the re-execution probabilities that reflect the runtime behavior, and in the process, be able to parallelize more loops with more accurate and hopefully lower misspeculation costs.

In order to use the data-dependence instrumentation and profile feedback results, we only need to modify the annotation of data-dependence probabilities of the dependence graph to take the profiling input. There was no change to the underlying cost computation module.

8. RESULTS AND LESSONS

We evaluated our framework by generating SPT code for an SPT architecture [15] and running the generated code on a simulator. The simulated machine is a tightly-coupled multiprocessor with one main core and one speculative core. Each processor core is an in-order Itanium2-like core. The cores have their own register files but share the memory/cache hierarchy. The main core always executes the main thread. Speculative threads must run on the speculative core. Both processor pipelines, branch predictors and the cache hierarchy are simulated. The memory/cache hierarchy has the same configuration and latencies as the Intel's Itanium2 systems. Branch misprediction penalty is 5 cycles. The minimum overheads to fork and commit a speculative thread are 6 and 5 cycles respectively.

To evaluate our speculative parallelization framework, we compiled 10 Spec2000Int benchmarks[4] using our SPT compiler, ran the generated code on the simulator and collected performance data. All compilation, both for the non-SPT base reference code and for the different versions of SPT code, used O3 level optimization, profiled guided optimization and type-based alias analysis. The generated code was simulated using an in-house trimmed down input set that is derived from the SPEC

[4] The remaining two Spec2000Int benchmarks (eon and perlbmk) were not evaluated because they failed to run on our simulator. Eon requires C++ library supports and perlbmk requires additional system call supports.

reference input set but runs about 5% as long while exhibiting similar program behavior. All simulation runs ran until program completion. Table 1 shows the performance of the non-SPT base reference code on a single core. The IPC (instruction per cycle) numbers shown exclude nop instructions. All speedup numbers reported in this section are based on comparing the execution time of the generated SPT code against the execution time of the non-SPT base reference code.

Table 1: IPC (excluding nops) of the non-SPT base reference

Program	IPC	Program	IPC
bzip2	1.69	mcf	0.44
crafty	1.49	parser	1.30
gap	1.30	twolf	1.05
gcc	1.33	vortex	0.56
gzip	1.77	vpr	1.22

Three sets of SPT code were evaluated. The first set of SPT code was generated by the basic compilation which used our cost model and all basic SPT optimizations (such as code reordering and loop unrolling). This basic compilation used only control flow edge profiling. The second set of SPT code was generated by the current best compilation which in addition to the basic compilation applied software value prediction (SVP) and data dependence profiling feedback. The third set of SPT code was generated by the current best compilation plus manual application of a few additional enabling techniques which were not yet implemented. These enabling techniques include while-loop unrolling, privatization and the export of global variables beyond their visible scopes. We anticipate that the last set of SPT code is what our compiler can achieve when those enabling techniques are in place. We refer to this last set of code as the result of the anticipated best compilation. Their performance results are reliable indicators of what can be practically achieved.

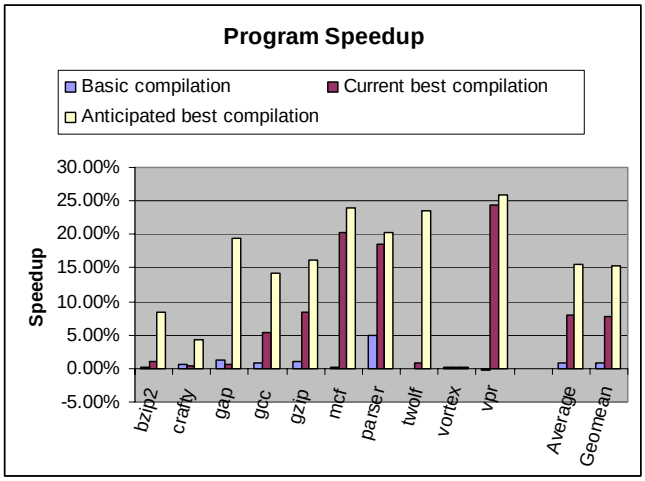

Figure 14: Performance gains from speculative parallelization

Figure 14 summarizes our evaluation results. It shows that simple techniques such as loop unrolling and code reordering are not enough to achieve good performance. The basic compilation achieves only 1% average speedup as compared to 8% in the best compilation and 15.6% in the anticipated best compilation. This also shows that only control profiling and type-based alias analysis are not enough to identify speculative parallelism. Software value prediction is an important SPT-enabler because it both helps to reduce misspeculation cost and enables more code reordering. The anticipated compilation results are encouraging. It shows that our framework is able to perform aggressive speculative parallelization when more speculative parallelism gets exposed. Note that in this work, the current compiler does not introduce intra-thread speculation to exploit more inter-thread speculative parallelism.

We now discuss the results of the current best compilation in detail.

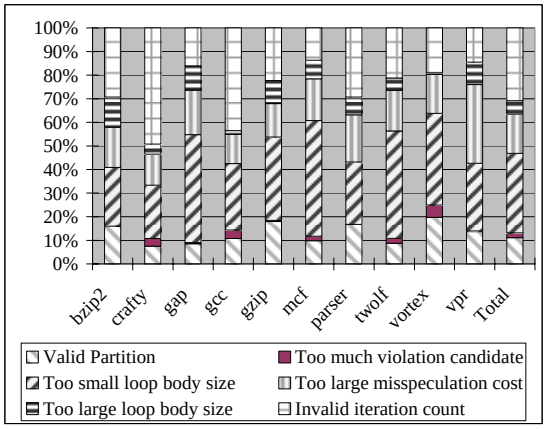

Figure 15: Breakdown of loops with respect to whether they can be SPT-transformed and why if not

Figure 15 shows the breakdown of loops with respect to whether the loops can be SPT transformed and the reasons why they cannot be transformed under our framework. The fraction labelled as 'Valid Partition' are the percentage of loops that satisfies the SPT loop selection criteria. For those that cannot be transformed, only a few loops failed because of too many violation candidates. 35% of the loops cannot be transformed because they have either too small iteration count or too large body size. We note that 34% of the loops are not transformed because their loop bodies are too small. These loops are while loops. Our current compiler can only unroll DO loops to increase their body sizes. As indicated in our anticipated best compilation result, while-loop unrolling is one important enabling technique.

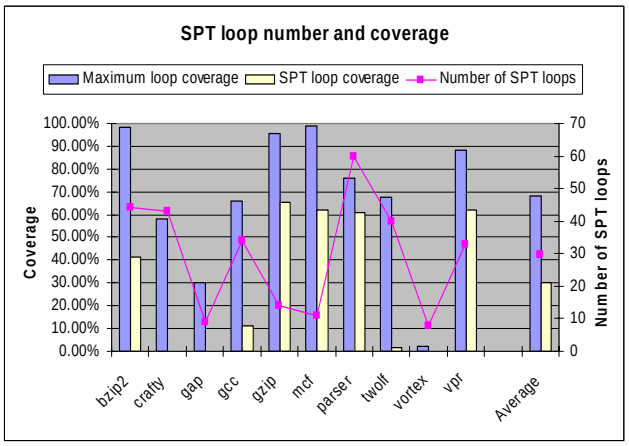

Figure 16: Runtime coverage of the generated SPT loops

Figure 16 shows the runtime coverage of the loops and the number of speculative parallel loops. Runtime coverage refers to the percentage of total program execution cycles being spent in the SPT loops. Our current best compilation is able to generate SPT loops to cover 30% of the total program execution cycles. Compared to the 68% maximum coverage of all loops with the same maximum loop size limit[5], our compiler does a decent job, successfully realizing 40% of the potential opportunity. On average, only 30 SPT loops were generated per benchmark. This indicates that a few hot loops were selected and transformed. There are still ample opportunities – this is not surprising with the anticipated best compilation results.

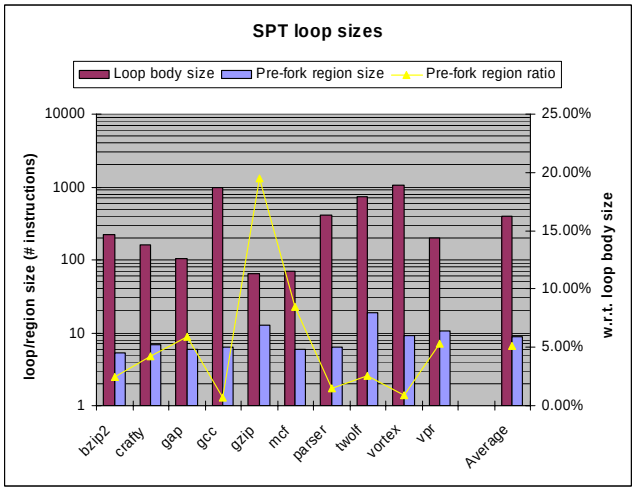

Figure 17: General characteristics of the SPT loop partitions

Figure 17 shows the average loop body size and the general characteristic of the partitions of the SPT loops. On average, a speculative parallel loop executes about 400 instructions per iteration (i.e., the loop body size). The pre-fork region has about 9 instructions (i.e., the pre-fork region size). About 5% of the loop body computations occur in the pre-fork region (i.e., the pre-fork region ratio). In other words, there was a significant amount of overlap in the execution of the main and speculative threads.

[5] The maximum loop size limit used in the experiments is 1000 instructions.

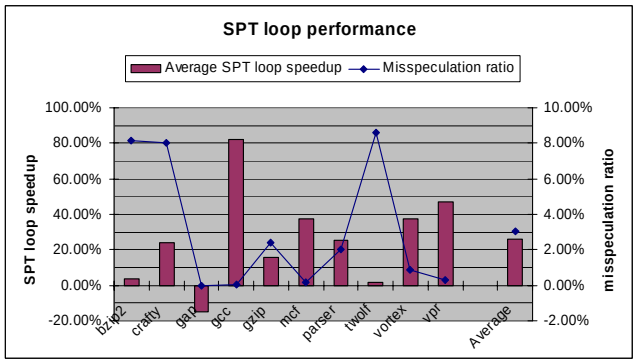

Figure 18: Actual performance of the generated SPT loops

One main goal of our compiler framework is to be able to speculatively parallelize only loops with low misspeculation costs. Figure 18 shows the actual performance characteristic of the SPT loops generated by the current best compilation. On average, the misspeculation ratio is only 3% while the speedup over the original loop is about 26%.

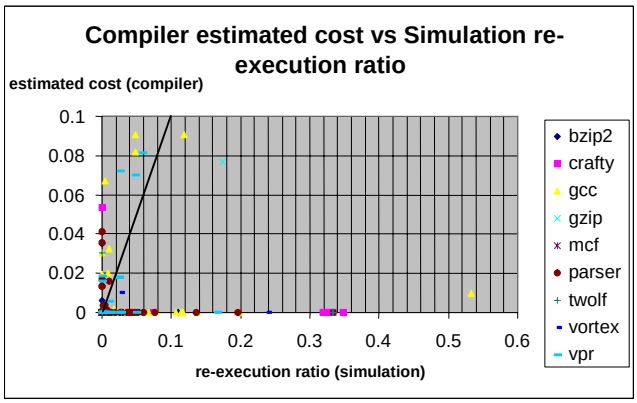

Figure 19: Correlation of compiler estimated misspeculation costs vs. actual simulation re-execution ratio

Our framework is built on the premise that the cost-driven approach can help to guide the selection of loops for the best speculative parallelization. Figure 19 shows a scatter plot that plots, for each loop, the compiler-estimated misspeculation cost against its actual re-execution ratio. The re-execution ratio of a loop is the fraction of computation of a loop iteration that is re-executed due to misspeculation. Figure 19 shows that the costs and re-execution ratios are generally well-correlated, except that the estimated costs tend to be conservative and over-estimate the re-execution ratio (as the data is more clustered near the y-axis.) Figure 19 also shows some loops near to the x-axis. Our analysis shows that the discrepancies come from function-calls inside these loops, which will modify and use some global variables

unknown to the caller loops. This points an area for improvement in the cost estimation.

9. FUTURE WORK AND CONCLUSIONS

The emerging hardware support for thread-level speculation opens new opportunities to parallelize sequential programs beyond the traditional limits. To take full advantage of this new execution model, a program needs to be programmed or compiled in such a way that it exhibits a high degree of speculative thread-level parallelism. We propose a comprehensive cost-driven compilation framework to perform speculative parallelization. Based on a misspeculation cost model, the compiler aggressively transforms loops into optimal speculative parallel loops and selects only those loops whose speculative parallel execution is likely to improve program performance. The framework also supports and uses enabling techniques such as loop unrolling, software value prediction and dependence profiling to expose more speculative parallelism. The proposed framework was implemented on ORC compiler. Our evaluation showed that the cost-driven speculative parallelization was effective. Our compiler was able to generate good speculative parallel loops in ten Spec2000Int benchmarks, which currently achieve an average 8% speedup. We anticipate an average 15.6% speedup when all enabling techniques are in place.

We noted in Figure 15 that there are loops which are not transformed because either their body size is too large, or their iteration count is too small. Such cases can be handled if we generalize our work to perform speculative parallelization for general code regions. For example, a speculative thread may be forked for a section of the loop body within the same iteration, or for a section of code after the loop body.

10. ACKNOWLEDGMENTS

Our thanks to Jesse Fang for his encouragement and support of this work.

11. REFERENCES

[1] Michael K. Chen and Kunle Olukotun. TEST: A tracer for extracting speculative threads. Proceedings of 2003 Intl Symposium on Code Generation and Optimization, San Francisco, CA, Mar 2003.

[2] Michael Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing Java programs. Proceedings of the 30th Annual Symposium on Computer Architecture, Jun 2003.

[3] M. Franklin. The Multiscalar Architecture. PhD thesis. University of Wisconsin at Madison, 1993.

[4] L. Hammond, B. Hubbert, et al. The Stanford Hydra CMP. IEEE Micro, Volume.20 No.2, Mar. 2000, pp. 71--84.

[5] Yuan-Shin Hwang, Peng-Sheng Chen, Jenq Kuen Lee, Roy Dz-Ching Ju. Probabilistic points-to analysis. Lanaguages and Compilers for Parallel Computing, 14th Intl Workshop, LCPC 2001, Cumberland Falls, KY, Aug 2001, pp. 290-305.

[6] Xiao-Feng Li, Zhao-Hui Du, Qingyu Zhao, and Tin-Fook Ngai. Software value prediction for speculative parallel threaded computations. The First Value-Prediction Workshop, San Diego, CA, June 7, 2003, pp. 18-25.

[7] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai and Sun Chan. A compiler framework for speculative analysis and optimizations. Proceedings of the ACM Sigplan 2003 Conference on Programming Language Design and Implementation, San Diego, CA, Jun 2003, pp. 289-299.

[8] Open Research Compiler for Itanium™ Processor Family. http://ipf-orc.sourceforge.net/

[9] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. Proceedings of Intl Symp. On Computer Architecture 2000, pp. 1-24.

[10] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew. The Superthreaded processor architecture. IEEE Transactions on Computers, Volume 48, Number 9, September 1999, pp. 881-902.

[11] Jenn-Yuan Tsai, Zhenzhen Jiang and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. Intl Journal of Parallel Programming, 27(1), 1999, pp 1-19.

[12] T. N. Vijaykumar. Compiling for the Multiscalar Architecture. PhD thesis. Computer Science Department, University of Wisconsin at Madison, Jan 1998.

[13] A. Zhai, C.B. Colohan, J.G. Steffan and T. C. Mowry, Compiler optimization of scalar value communication between speculative threads. The Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, CA, USA, Oct 7-9, 2002.

[14] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In Proc. of the Sixth Int'l Conf. on Compiler Construction, pages 253--267, April 1996.

[15] Xiao-Feng Li, Zhao-Hui Du, Chen Yang, Chu-Cheow Lim, Qingyu Zhao, William Chen and Tin-Fook Ngai. Speculative parallel threading architecture and compilation. Submitted to the 9th Asia-Pacific Computer Systems Architecture Conference, 2004.