Speculative Parallel Threading Architecture and Compilation

Xiao-Feng Li, Zhao-Hui Du, Chen Yang, Chu-Cheow Lim†, Tin-Fook Ngai
Intel China Research Center, Beijing, China *†Santa Clara, California, USA*
Intel Corporation
{xiao.feng.li, zhao.hui.du, chen.yang, chu-cheow.lim, tin-fook.ngai}@intel.com

Abstract

Thread-level speculation is a technique that brings thread-level parallelism beyond the data-flow limit by executing a piece of code ahead of time speculatively before all its input data are ready. This technique appears particularly appealing for speeding up hard-to-parallelize applications. Although various thread-level speculation architectures and compilation techniques have been proposed by the research community, scalar applications remain difficult to be parallelized. It has not yet shown how well these applications can actually be benefited from thread-level speculation and if the performance gain is significant enough to justify the required hardware support. In an attempt to understand and realize the potential gain with thread-level speculation especially for scalar applications, we proposed an SPT (Speculative Parallel Threading) architecture and developed an SPT compiler to generate optimal speculatively parallelized code. Our evaluation showed that with our SPT approach 10 SPECint2000 programs can achieve an average of 15.6% speedup on a two-core SPT processor by exploiting only loop parallelism. This paper describes the SPT architecture and the SPT compiler which performs aggressive cost-driven loop selection and transformation, and presents our performance evaluation results.

1. Introduction

Thread-level speculation is an emerging technique that can bring thread-level parallelism beyond the program data-flow limit by speculatively executing a piece of code before all its dependences are resolved. The speculated execution is checked at runtime after all its dependence are resolved to determine if there are any dependence violations. When there is no dependence violation, the speculation results can be safely committed. Otherwise, the speculation results are invalid and need to be recovered.

With thread-level speculation, a sequential program can be executed in multiple parallel threads but still observing the original sequential semantics. When the speculation is correct most of the time at runtime, the threads exhibit substantial dynamic parallelism and can speed up the application execution if the overhead for the speculation support is relatively small.

Thread-level speculation appears particularly appealing for speeding up hard-to-parallelize applications. Figure 1 illustrates the speculative parallelization of a sequential loop with thread-level speculation. It shows an example loop from parser in SPECint2000 before and after speculative parallelization. The original loop in Figure 1(a) traverses a linked list pointed by variable c and frees the list node one by one. Traditional parallelization cannot parallelize this loop because of the sequential dependence of the link list chasing. However, by speculating that the next pointer is likely not a null pointer, it becomes possible to execute consecutive iterations in parallel threads. Figure 1(b) shows the corresponding code after speculative parallelization. After determining the next node of the link list, the main program thread will execute the

```
while( c!= NULL ){
   c1 = c->next;
   free_Tconnector(c->c);
   xfree(c,
sizeof(Clause));
   c = c1;
}
```

(a) Original loop

```
while( c!= NULL ){
SPT_001;
   c = temp_c;
   c1 = c->next;
   temp_c = c1;
   SPT_FORK(SPT_001);
   free_Tconnector(c->c);
   xfree(c, sizeof(Clause));
   c = c1;
}
```

(b) after parallelization

Figure 1. An example loop before and after speculative parallelization

SPT_FORK(SPT_001) statement to fork a thread to execute the next iteration speculatively at the label SPT_001. After forking the speculative thread, the main program thread and the speculative thread are executing in parallel, one freeing the current list node and the other freeing the next list node speculatively. Our evaluation shows that this speculative parallel loop can speed up the original loop by more than 40%. Only 5% of the speculatively executed instructions were invalid and 20% of the speculative threads ran perfectly parallel with the main program thread without misspeculation.

Many architecture models have been proposed to support thread-level speculation [1, 5, 8, 9, 11, 12, 13]. They differ in processor organization (such as chip multiprocessor, multi-core processors or simultaneous multithreaded processors), and in the thread-level speculation support (such as thread communication and synchronization, speculation state transition, data dependence checking, value prediction support and speculation commit/recovery mechanism.) Simulation evaluation of these architecture models showed significant speculative parallelism in existing applications, especially in scientific/multimedia applications. For large scalar applications like SPECint2000, the reported speedups either are small or depend on aggressive hardware assumptions.

Various compilation techniques have been proposed to parallelize applications with thread-level speculation [16, 14, 17]. However, scalar applications remain difficult to be parallelized [17].

Despite the appealing concept of thread-level speculation and sporadic evidence of speculative parallelism, it has not yet shown how well scalar applications can actually be benefited from thread-level speculation and if the performance gain is significant enough to justify the required hardware support.

1.1 Speculative Parallel Threading

In order to understand and realize the potential gain with thread-level speculation especially for scalar applications, we proposed an SPT (Speculative Parallel Threading) architecture and developed an SPT compiler to generate optimal speculatively parallelized code.

With the knowledge of limited parallelism in scalar applications, we primarily focus ourselves on small-scale but tightly coupled multiprocessors. The proposed SPT architecture consists of two tightly-coupled in-order pipeline cores. The two pipeline cores share the same instruction and data caches, and are functionally asymmetric, i.e., one runs the main or the architectural thread while the other runs speculative threads. One key feature of the SPT architecture is its selective re-execution recovery mechanism. Most other speculative multithreaded architectures trash all speculation results and re-execute the entire speculative thread upon misspeculation. On the contrary, upon misspeculation the SPT architecture commits correct speculation results and selectively re-executes only those misspeculated instructions.

We have developed a speculative auto-parallelization compiler to generate optimal speculative parallelized code for the SPT architecture. This allows us to effectively evaluate and demonstrate the actual benefit of thread-level speculation in scalar applications. The SPT compiler uses a comprehensive cost-driven compilation framework which aggressively transforms loops into optimal speculative parallel loops and selects only those loops whose speculative parallel execution is likely to improve program performance. The compiler also supports and uses enabling techniques such as loop unrolling, software value prediction and dependence profiling to expose more speculative parallelism.

Our evaluation shows that the SPT compilation and architecture is effective in generating good speculative multithreaded code and delivering good performance with scalar applications. Ten SPECint2000 benchmarks achieve an average 15.6% speedup on a 2-core SPT processor.

This paper is different from the work in [4] in that, this paper is focused on architecture instead on the compilation techniques only and it evaluated the full potential of the architecture instead of the benefits of the compiler framework.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 discusses related work in thread-level speculation architecture and compilation, focusing on the performance of scalar applications. Section 3 and Section 4 describe SPT architecture and compilation respectively. In Section 5, we evaluate our SPT solution. We report the amount of speculative parallelism being found and exploited in loops in SPECint2000 benchmarks and present the overall performance results. We conclude the paper in Section 6.

2. Related Work

The Wisconsin's Multiscalar work [5, 15] is the first and well-known work that studied both hardware and software supports in thread-level speculation. The Multiscalar compiler applied various task size, control flow and data flow heuristics to break a program into fine-grained tasks [16]. Task selection in Multiscalar processor is crucial to the delivered performance, and the generated code for integer program contains only 10-20 instructions. Hence the performance result is very sensitive to a particular hardware design and its associated overhead. Recently, Park evaluated the SPEC benchmarks on an implicitly-multithreaded processor using the Multiscalar compiler [11]. He showed that the SPECint2000 benchmark could achieve 20% speedup with a fine-tuned 8-context implicitly-multithreaded SMP processor.

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 [14]. For their experiments, the benchmark programs were manually transformed at the source level.

Garzaran et al. [6] proposed a software-only undo log system on hardware support designed in [18], which demonstrates the performance is only 10% less than a full hardware implementation.

Zhai studied code scheduling and optimization techniques to improve thread-level speculation performance [17]. She showed that the compiler could reduce value communication delay between the threads significantly by forwarding scalar values to the speculative threads and inserting synchronization. Unfortunately, the overall average performance gain for SPECint2000 is less than 5% on a multiprocessor machine whose processors are 4-issue 128-entry-ROB out-of-order superscalar processors.

Chen proposed and evaluated a Java runtime parallelizing machine [3]. The machine is a chip multiprocessor with thread-level speculation supports. He showed 1.5-2.5 speedups with integer on a 4-processor machine. While the compilation techniques being used are generally applicable to other programs, the benchmarks evaluated are small Java programs. It is not clear if the performance results apply to other speculative multithreaded architectures or to other scalar applications.

3. SPT Architecture

A SPT architecture has been proposed to evaluate the performance potential with thread-level speculation in scalar applications. Here are a few major design considerations: First, even with thread-level speculation the amount of parallelism in scalar application is expected to be limited. We focus ourselves on small scale tightly coupled multiprocessors in order to maintain reasonable utilization of hardware resource and performance-cost ratio. Second, we prefer well-established implementation techniques and micro-architecture structures. Similar to superscalar processors that executes in-order instructions out of order, the SPT processor operates like executing in-order threads out of order. Third, even if a speculative thread fails due to dependence violation, those speculation results that remain correct should be kept and reused. Recall the example loop in Figure 1, only 20% of the speculative threads could run perfectly parallel with the master thread without any dependence violation. In other words, 80% of the speculative threads failed. However, results of 95% of the speculatively executed

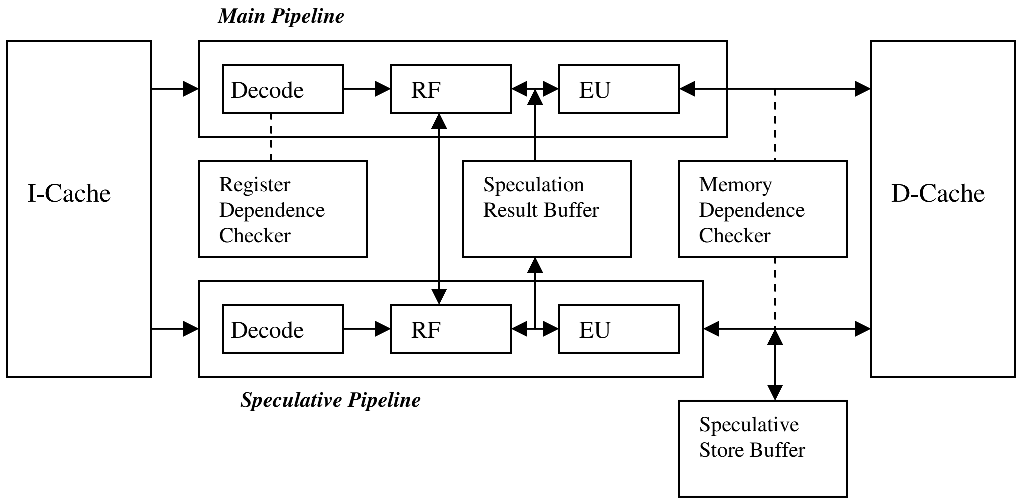

Figure 2. The SPT Architecture

instructions were correct. Their reuse can lead to considerable performance gain.

Figure 2 shows the proposed SPT architecture. It is a tightly-coupled asymmetric multi-processor which consists of two in-order pipeline cores. One processor is the main processor which always executes the main program thread. The other processor is the speculative processor which executes only speculative threads. Each processor has its own register file and program state. They share the same memory hierarchy and subsystem. They may have separate L1 caches, but the caches are always coherent.

Since the speculative thread execution is speculative, the speculative thread cannot modify the architectural state of the program before commit. All speculative stores by the speculative thread are stored in the speculative store buffer. Correspondingly, all speculative loads by the speculative thread first look up dependent stores in the speculative store buffer and access the shared cache/memory only when they cannot locate their data in the speculative store buffer.

3.1 SPT Execution Model

There are two special SPT instructions, spt_fork and spt_kill, for explicit hardware threading support. When the main processor executes the spt_fork instruction, it forks a speculative thread and the speculative processor executes the speculative thread starting at the IP address specified by the spt_fork instruction. When the main processor executes the spt_kill instruction, any running speculative thread on the speculative processor is killed. The program point where the main thread forks a speculative thread is called the fork-point. The program point where a speculative thread starts execution is called the start-point. Both spt_fork and spt_kill instructions are no-ops to the speculative processor.

When the main processor executes the spt_fork instruction, its register context is copied to the speculative processor before the speculative thread starts execution.

During the parallel execution, there is no register communication or any explicit thread synchronization between the threads. All speculatively executed instructions and their execution results are stored in program order in the speculation result buffer. The speculation result buffer is a FIFO queue similar to ROB in out-of-order processors. When the speculation result buffer is full, the speculative thread stalls until there are free buffer spaces.

When the main thread eventually arrives at the start-point of the speculative thread (assuming correct control speculation), it checks for dependence violation. If there is no dependence violation, the entire speculative state of the speculative processor can be committed at once. This includes copying the register context of the speculative processor back to the main processor and writing back all outstanding stores in the speculative store buffer. We call this a fast-commit. When there are dependence violations, the main processor starts replaying the speculative execution from the speculation result buffer. The speculated instructions are examined one by one in its program order. For those instructions that are speculatively executed correctly, the main thread reads their speculative results from the speculation result buffer and commits them directly. For those instructions whose speculative execution is incorrect, the main thread re-executes them. If an instruction is misspeculated and re-executed, all instructions that use its execution result are also misspeculated and need to be re-executed. The identification of misspeculated instructions and their dependent instructions is performed by the dependence checkers which are described in the following subsection.

The main thread stops replaying the speculative execution when either the speculation result buffer is empty or it re-executes a branch instruction and the next instruction does not match the one in the speculation result buffer. The former case occurs when the main thread catches up with the speculative thread. The latter case occurs when the speculative thread begins executing down a wrong path. In both case, the speculative thread is killed and the main thread resumes normal execution at the program point it stops replaying.

The replay and selective re-execution recovery mechanism allows the SPT execution to recover all correct speculative execution results even when the speculative thread is misspeculated.

3.2 Data Dependence Check

The SPT architecture checks both register dependences and the memory dependences.

For register dependence check, the main thread keeps track of its register modifications since the fork-point. All registers that has modified after the fork-point are marked updated. When it arrives at the start-point of the speculative thread, if any register ever read by the speculative processor is an updated register, that means the speculative thread has read an obsolete register value, a dependence violation occurs. During replay, the main processor continues to use scoreboarding to keep track of the register modifications. Any speculated instruction that has referenced an updated register is misspeculated and after re-execution, its destination register is also marked updated. A more sophisticated dependence check is to compare the new register values at start-point with the old register value of the register at fork-point when the main thread arrives at the speculation start-point. Only those registers whose values have changed are updated registers and cause dependence violation.

For memory dependence check, the SPT architecture maintains a speculative load address buffer in addition to the speculative store buffer. A speculative load accesses the cache/memory only when there is no matched data in the speculative store buffer. The memory dependence checker records all such cache/memory loads in the load address buffer. Whenever the main thread stores to the cache/memory before it arrives at the start-point, the store address is checked against all load addresses in the speculative load address buffer. Any match indicates a memory dependence violation. The misspeculated load is marked and will be re-executed during replay.

4. SPT Compilation

We have developed an SPT compiler based on Open Research Compiler (ORC [10]) for loop-level parallelism by executing successive loop iterations speculatively in parallel threads. For SPECint2000 benchmarks, 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.

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). Most analyses and transformations were done in ORC's SSA form [2]. For readers who are interested in the details of our compiler framework, please refer to paper [4]

SPT compiler has a comprehensive cost-driven compilation framework that selects only the loops that are likely to improve delivered performance, and transforms them into speculative parallel loops, called SPT loops. An SPT loop execution scenario is illustrated in Figure 3.

In Figure 3, the main thread executing iteration i forks the speculative thread for iteration i+1. The insertion of instruction spt_fork effectively partitions the loop body into pre-fork and post-fork regions. The source of any cross-iteration dependence is called violation candidate. Since the speculative thread starts after the pre-fork region of the main thread is finished,

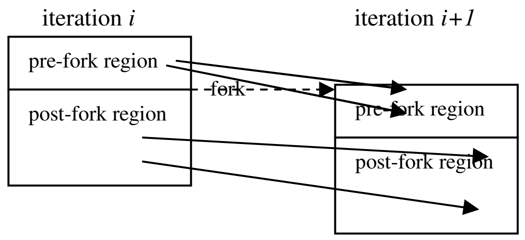

Figure 3. SPT loop execution scenario.
(The loop body is shown as square and the true dependences are shown as arrows. The cross-line in the loop body refers to the fork-point., and the start-point is always at the loop body start.)

all the dependences originated from the main thread's pre-fork region are guaranteed to be satisfied; so we care only those dependences from post-fork region. On the other hand, we can not put all violation candidates into pre-fork region, because the code in pre-fork region is sequentially executed by the threads. Amdahl's law requires the pre-fork region size small enough compared to the post-fork region in order to bring any parallelism benefits, especially when the threading overhead is considered.

The misspeculation penalty caused by a dependence is determined by the probability of the dependence really happening at runtime, and the misspeculation computation amount caused by this dependence that needs to be re-executed. The overall effects of all the dependences in a loop is computed as misspeculation cost, which is the expected misspeculated computation amount within a speculative executed loop iteration that needs to be re-executed if the loop is transformed with SPT compilation and run on SPT architecture.

The goal of SPT compiler is to find out an optimal loop partition for each loop candidate that has minimal misspeculation cost. SPT compiler accomplishes the goal with its cost-driven compilation framework in a two-pass compilation process, as we describe in following sections.

4.1 Cost-driven Compilation

The key element in SPT compiler is its cost-driven compilation. The misspeculation cost computation is the central component of the framework, which is based on a misspeculation cost model built with annotated control-flow graphs and data-dependence graphs (DD graph) as shown in Figure 4. The control-flow graph is annotated with reach probability and the data-dependence graph with dependence probability. By querying the cost computation core, SPT compiler searches optimal loop partition for each loop and applies SPT loop transformation upon selected loops.

Misspeculation cost is actually computed with cost graph, which is built based on the control-flow graph and the data-dependence graph. Basically, the inner nodes of the cost-graph represent operations of the speculative thread annotated with the operation's computation amount, and each edge X→Y in the cost-graph is annotated with a probability p, which is the conditional probability that a re-execution at X causes Y to be misspeculated and re-executed, given that X is misspeculated.

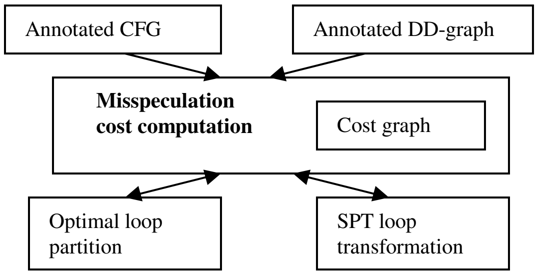

Figure 4. SPT compilation framework

For a given loop partition, we need first estimate of the re-execution probability of each operation in the speculative thread by walking through the cost graph in topological order. Then the misspeculation cost of the given partition is computed as:

$$\sum_{c} P(c) * Cost(c) \qquad (1)$$

The summation is for all nodes c in the cost graph, P(c) is the re-execution probability of node c and Cost(c) is the amount of computation in node c.

In order to select and transform only good SPT loops without missing any good ones, SPT compiler goes through two compilation passes. The first pass selects loop candidates according to simple selection criteria like loop body size and trip count, and apply loop preprocessing such as loop unrolling and privatization for more opportunities of thread-level parallelism. Then, the SPT compiler finds out the optimal loop partition for each loop candidate, and determines its speculative parallelism amount. The results of all loop candidates' optimal partitions are output and the first pass finishes without any real permanent transformation. Next, the second pass reads back the partition results and evaluates all loops together, then selects all good and only good SPT loops. These loops are again preprocessed, partitioned, and transformed so as to generate final SPT code.

4.2 Optimal Loop Partition Searching

With the misspeculation cost of a given partition, searching for the optimal loop partition of a loop is not a trivial work. A direct algorithm might exhaustively enumerate all the combinations of the loop body statements but our experiments showed the direct approach is not valid because of both correctness and performance issues.

We observe that, although a partition is decided by a set of statements in pre-fork region (or post-fork region because they are complementary), the misspeculation cost of a given partition is decided uniquely by the violation candidates in the post-fork region. That means a combination of violation candidates can uniquely decide a loop partition. Since the number of violation candidates of a loop is usually much smaller than that of its statements, so we can substantially reduce the search space by searching only partitions that have different combinations of violation candidates in the pre-fork region.

We use two constraint functions to reduce more the search space and computations effectively. One is cost-bounding function that computes the misspeculation cost of a given partition; the other is size-bounding function that computes the pre-fork region size of a partition. 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. We utilize this monotone property of the two functions to avoid lots of redundant search and computations.

4.3 Loop Transformation

With the resulted optimal partition of a good SPT loop candidate, SPT compiler uses code re-ordering to transform the original loop: Assuming the SPT_FORK statement (i.e., the partition boundary, compiled into spt_fork instruction) is initially inserted at the loop body start, the compiler re-orders the loop body so that the violation candidates in the pre-fork region are all before the SPT_FORK statement.

There are two complications in the SPT loop transformation. One is that the life-range of different definitions of the same symbol may be overlapped, which is solved by introducing temporary variables to break the live range. The other one is to deal with code motion of partial conditional statements. When there is a branch statement inside loop body and some code control dependent on the branch statement is to be moved into pre-fork region, the control dependence relation must be maintained. We solve the problem by copying the branch statement into pre-fork region as well.

4.4 Software Value Prediction

In order to achieve best transformed loop that can deliver optimal performance with SPT architecture, we have invented some novel techniques, which are either essential or helpful to thread-level speculation. In this section, we only introduce software value prediction technique which we believe is essential for thread-level speculation.

Not all the dependences originated from post-fork region can be moved into pre-fork region because its size cannot be too large. If we cannot eliminate a critical cross-iteration dependence from post-fork region, we can apply software value prediction [7] to reduce its dependence probability, which reduces the misspeculation cost as well. With software value prediction, the compiler identifies the critical variables that incur this kind of dependences, then instruments the program to profile the value patterns of the corresponding variables. If the values are found predictable and both the corresponding value-prediction overhead and the misprediction cost are acceptably low, the compiler inserts the appropriate software value prediction code to generate the predicted values. It also generates software check and recovery code to detect and correct potential value misprediction. Figure 5 illustrates how a loop looks like before and after the application of software value prediction.

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

(a) Original loop

```
pred_x = x;
while(x){
SPT_001:
    x = pred_x;
    pred_x = x+2;
    SPT_FORK(SPT_001);
    foo(x);
    x = bar(x);
    if(pred_x != x)
       pred_x = x;
}
```

(b) SVP-transformed loop

Figure 5. Software value prediction example

In this example, the across iteration dependence from the definition statement x = bar(x) to the use statement foo(x) causes high misspeculation cost. On the other hand, the potential side effects within the functions foo() and bar() prevents the definition statement x = bar(x) to be moved before foo(x). The compiler therefore profiles the value x. Assume it finds that x is often incremented by 2 by bar(x). It then decides to value-predict x. It implements the software value predictor x+2 to compute the predicted value pred_x before the SPT_FORK statement. The predicted value is used to set the value x in the next iteration. If the actual value of x is different to the predicted value, the predicted value is corrected with the right value.

5. SPT Evaluation

We proposed the SPT architecture and developed the SPT compiler to generate optimal speculative parallelized code. Our purpose is to understand and realize the performance potential with thread-level speculation in scalar applications. We performed simulation experiments to gather data to assess the potential amount of speculative parallelism in scalar applications and to determine how well our SPT solution can realize the performance potential.

In this section, we first describe our evaluation methodology. Then we report the loop characteristics of the benchmarks, the amount of speculative parallelism in loops in the benchmark and the SPT performance results.

5.1 Evaluation Methodology

The SPECint2000 benchmarks were evaluated in our experiments as representative scalar applications. We compiled SPECint2000 benchmarks with our SPT compiler and run the executables on an SPT simulator.

The SPECint2000 benchmarks were compiled to generate two different versions of executable code. Both versions are generated using ORC –O3 optimization level with profile-guided optimizations. One version is the ordinary optimized Itanium code and serves as our baseline reference; the other version of executable code is our optimal speculatively parallelized code. All SPECint2000 benchmarks except eon and perlbmk were evaluated in this study. Eon and perlbmk failed to run on our simulator because eon requires C++ library supports and perlbmk requires an additional system call support.

The generated code was simulated using an in-house trimmed down input set that is derived from the SPEC reference input set. All simulation runs ran to program completion. The derived input sets were created to exhibit representative program behavior similar to those generated by the reference input sets but were substantially smaller than the reference input set to allow realistic full program simulation. Each program run normally executes about 20 billion instructions, which is about 5% of that with the reference input set. Like the reference input sets, the derived input sets contain similar multiple runs for each benchmarks.

Table 1. SPT simulator configuration

Processor cores	2 Itanium2 in-order cores
Cache hierarchy	L1: separate I/D cache, 16KB, 4-way, 64B-block, 1-cycle latency L2: 256KB, 8-way, 64B-block, 5-cycle latency L3: 3MB, 12-way, 128B-block, 12-cycle latency
Memory latency	150 cycles
Normal / re-execution fetch width	6
Normal / re-execution issue width	6
Replay fetch width	12
Replay issue width	12
RF read/write ports	12
Branch predictor	GAg with 1K entries
Mispredicted branch penalty	5 cycles
RF copy overhead	1 cycle minimum
Fast commit overhead	5 cycles minimum
Speculation result buffer size (default)	1024 entries
Misspeculation recovery mechanism (default)	Selective re-execution with fast-commit (SRX+FC)
Register dependence checking (default)	Value-based

The simulator we use is a trace-driven simulator. It reads in an execution trace of the sequential execution of the program, and simulates the trace on two separate pipelines. It maintains two separate cycle counters for the main pipeline and the speculative pipeline to keep track the speculative parallel execution. The cache system is also simulated. Each cache and memory access is tagged with the corresponding time stamp to maintain the proper temporal ordering. Table 1 describes the default machine configuration. The processor core and memory subsystem has the same configuration as an Intel's Itanium2 machine.

5.2 Benchmark Characteristics

In order to understand the capability of our SPT solution, we have to know the potential and characteristics of SPECint2000 benchmarks.

Figure 6 shows the loop coverage of the applications. The curves show the accumulative coverage in term of dynamic execution cycles of all loops whose body size (in term of number of instruction) is within certain limit.

As shown in the figure, the accumulative coverage of loops whose average body size is less than a few hundred instructions is considerable high even though the applications are traditionally considered to be not loop-intensive. Besides gap and vortex, the total loop coverage is more than 60%. For gap, the total coverage increases sharply from 35% to 95% once loops with average loop body size of 2500 instructions are included. This is contributed by one highly skewed but very hot loop whose loop body size is usually not large but can occasionally becomes huge when certain function calls are made in the loop body. Vortex has very little loop coverage even with loops whose body size has a few tens of thousand of instructions. This implies that the performance opportunity of speculative parallel loops in vortex is insignificantly small. We do not expect any speedup for vortex. The figure indicates that most of the applications execution time is dominated by loops whose loop body size are less than 10K instructions.

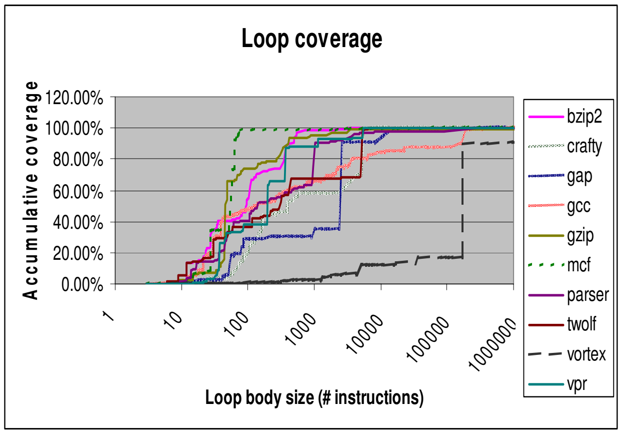

Figure 6. Loop coverage of SPECInt2000

5.3 SPT loops coverage

With the loop coverage shown in Figure 6, we collect data on how many speculative parallel loops are generated by SPT compiler for each benchmark. Figure 7 shows the SPT loops number and their total coverage as compared with the corresponding maximum loop coverage for all loops with same size limit.

Except the special case in gap, we only considered loops whose estimated loop body size is less than 1000 instructions in this data because of practical architecture considerations. For gap, because of one hot loop mentioned above, we considered loops with average loop body size less than 2500 instructions. The figure shows SPT compiler does a pretty good job in parallelizing interesting loops. At the same time, the data suggests the SPT loop parallelization is also effective to derive thread-level parallelisms from scalar applications: On the average, only 32 SPT loops are generated but they cover as high as 53% of the total execution cycles of the program.

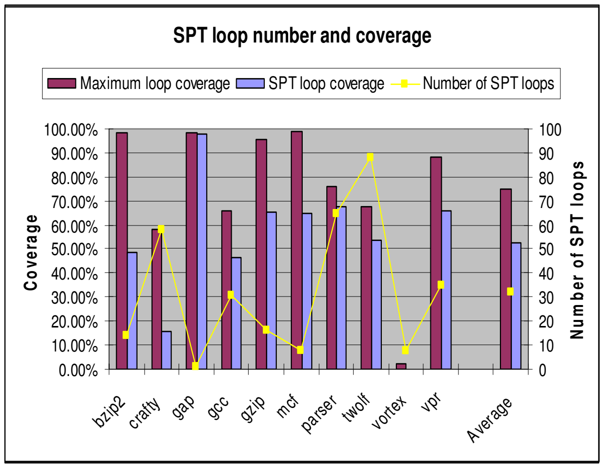

Figure 7. Number and coverage of SPT loops

5.4 SPT Loops Performance

Now we understand the SPT loops coverage is really encouraging, we also want to know the performance achieved at loop-level with our SPT solution. Figure 8 shows the loop-level performance of the SPT loops. On the average, an SPT loop that we generated is able to speed up by 35%. The success ratio of speculative parallelization is pretty high. Out of all speculative threads being spawned during run-time, 64% of them could be fast-committed successfully (i.e., without any dependence violation, shown as the bars of Fast commit ratio). Only 1.2% of all speculatively executed instructions was misspeculated and required re-execution (see the curves for misspeculation ratio).

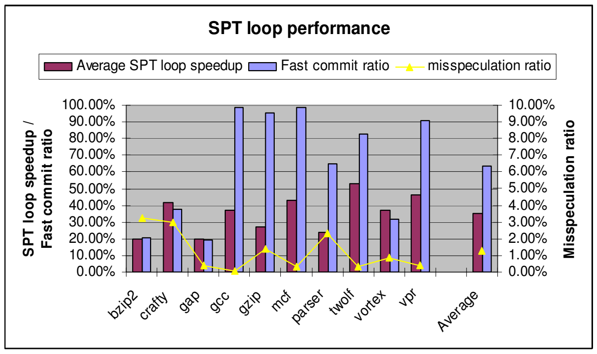

Figure 8. SPT loop performance

This data essentially exhibits the degree of inter-iteration true dependences of scalar application loops. The high fast-commit ratio means good iteration-level parallelism. Interesting enough is that, although the fast-commit ratio is high, there is still misspeculation existing. This observation verifies that the substantial thread-level parallelisms of the loops can only be extracted dynamically at runtime.

5.5 SPT Overall Performance

Finally, Figure 9 shows the overall program performance of speculative parallel loops in Spec2000Int benchmarks when running on the default machine configuration (that is, with selective replay and fast commit and 1024-entry speculation result buffer). It shows that the speculative parallel loops are able to achieve an average of 15.6% program speedup on the two-core SPT architecture as against the optimized non-SPT code running on one core. Notably, the known hard-to-parallelized gcc is able to achieve 14.3% speedup. The gain in bzip2 is hurt by indirect global memory updates via function calls. Crafty has many loops of short iteration counts that is inefficient to parallelize at iteration level. Vortex does not show any performance gain as expected.

The figure also shows the breakdown of the source of program speedup. On the average, out of the 15.6% program speedup, 8.4% comes from reduction in pipeline execution cycles, 1.7% comes from reduction in pipeline stall cycles and 5.5% comes from reduction of D-cache stall cycles. This indicates that speculative parallelization is effective in exploiting both computation parallelism and memory parallelism.

6. Conclusion

Speculative multithreaded computation is an emerging technique that can parallel sequential programs by speculatively breaking the data

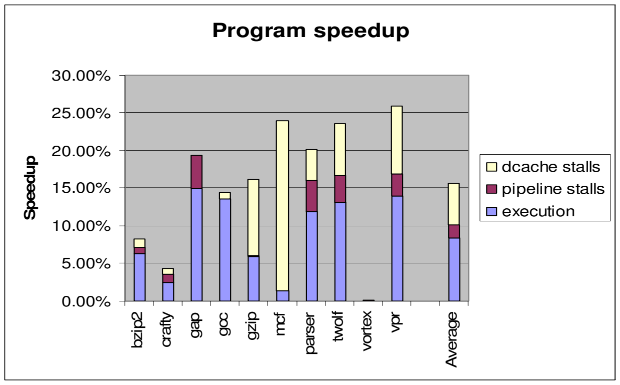

Figure 9. SPT performance with SPECint2000

dependences. Although the idea behind the technique is appealing, the practical solution and actual benefits of the thread-level speculation for scalar applications are not fully understood. In this paper, we describe our SPT (Speculative Parallel Threading) solution, including SPT architecture proposal and a cost-driven compiler. Data shows SPECint2000 benchmarks have considerably high loop coverage and the SPT solution can extract substantial loop parallelism from the applications. Up to an average of 15.6% speedup on a two-core SPT architecture is achieved in our work.

While the demonstrated performance is encouraging, we note that there are still loops that are not speculatively parallelized because of too big or small loop body size, too many violation candidates, too large misspeculation cost or too small iteration count. Except the case of too large misspeculation cost, which means intensive dependences between consecutive iterations hence little inherent parallelism, all of the rest cases can be improved in further studies. Region-based speculation is believed to be a potential approach, which tries to parallelize a sequential piece of code by executing its first half and second half in parallel. The other important future work is to fully understand the implications of various architectural parameters upon the delivered performance.

References

[1] H. Akkary and M. Driscoll, *A Dynamic* Multithreading *Processor*, International Symposium on Microarchitecture, MICRO-31, December, 1998.

[2] 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.

[3] M. Chen and K. Olukotun, *The Jrpm System for* Dynamically *Parallelizing Java Programs*, Proceedings of the 30th Annual Symposium on Computer Architecture, June, 2003.

[4] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao and Tin-Fook Ngai, *A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs*, in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI), Washington, D.C., June 9-11, 2004.

[5] M. Franklin, *The Multiscalar Architecture*, PhD thesis, University of Winsonsin-Madison, 1993.

[6] M. J. Garzaran, M. Prvulovic, J. M. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas, *Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation*, Intl Conf. on Parallel Architectures and Compilation Techniques (PACT), September 2003.

[7] Xiao-Feng Li, Zhao-Hui Du; Qingyu Zhao, Tin-Fook Nagi; Software *Value Prediction for Speculative Parallel Threaded Computations*, First Value-Prediction Workshop, held in conjunction with the 30th ISCA, San Diego, California, June 7, 2003.

[8] P. Marcuello and A. Gonzlez, *A Clustered Speculative Multihreaded Processors*, Proceeding of the ACM International Conference on Supercomputing, June, 1999.

[9] K. Olukotun, L. Hammond and M. Wi ley, *Improving the performance of speculatively parallel applications on the Hydra CMP*, Proceedings of the 1999 International Conference on Supercomputing, 1999.

[10] Open Research Compiler, Intel Co. Ltd., http://ipf-orc.sourceforge.net/.

[11] I. Park, B. Falsafi and T. Vijaykumar, *Implicitly-Multithreaded Processors*, Proceedings of the 30th Annual Symposium on Computer Architecture, June, 2003.

[12] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai and Todd C. Mowry, A scalable approach to thread-le*vel speculation*, The 27th Annual International Symposium on Computer architecture, 2000.

[13] J-Y Tsai, J. Huang, C. Amlo, D.J. Lilja, P-C. Yew, *The superthreaded processor architecture*, IEEE Transactions on Computers, Volume 48, No. 9 , Sept. 1999, pp 881-902.

[14] 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.

[15] T. N. Vijaykumar, *Compiling for the Multiscalar Architecture*, PhD thesis, Computer Science Department, U. Wisconsin-Madison, Jan. 1998.

[16] T.N. Vijaykumar and Gurindar S. Sohi, *Task Selection for a Multiscalar processor*, Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31), 1998, Pages 81-92.

[17] A. Zhai, C. B. Colohan, J. G. Steffan and T. C. Mowry, *Compiler Optimization of Scalar Value Communication Between Speculative Threads*, The Tenth Intl Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, Oct 7-9, 2002.l

[18] Y. Zhang, L. Rauchwerger, and J. Torrellas, *Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors*, In Proceedings of the 5th Intl. Symp. on High-Performance Computer Architecture (HPCA), pages 135–139, January 1999.