A Minimal Dual-Core Speculative Multi-Threading Architecture

Srikanth T. Srinivasan, Haitham Akkary, Tom Holman, Konrad Lai
Microarchitecture Research Lab, Intel Corporation
{srikanth.t.srinivasan, haitham.h.akkary, tom.holman, konrad.lai}@intel.com

Abstract

Speculative Multi-Threading (SpMT) can improve single-threaded application performance using the multiple thread contexts available in current processors. We propose a minimal SpMT model that uses only two thread contexts. The model achieves significant speedups for single-threaded applications using a low-overhead scheme for detecting and selectively recovering from data dependence violations, and a novel Wrong Path Predictor to reduce the number of speculative threads executing along the wrong path. We also study the interactions between three previously proposed SpMT thread spawning policies that can be implemented dynamically in hardware – Fork on Call, Loop Continuation and Run Ahead policies – and show it is beneficial to implement all three policies together in a processor. While the individual thread spawning policies show performance benefits of 14%, 5% and 4% respectively on our SpMT model over a base processor that does not exploit SpMT, combining all three policies shows an average performance gain of 20%. Finally, we identify the sources of SpMT benefits – on average, 58% of the performance benefits due to SpMT comes from cache prefetching, 33% from instruction reuse, and 9% from branch precomputation and show all three sources of SpMT benefits must be utilized to realize the full potential of SpMT.

1. Introduction

Multithreaded processors are becoming increasingly common. Major hardware vendors are providing multiple cores on chip (Power5 [6]) or multiple threads on the same core (Pentium 4 [8]). While such designs benefit throughput oriented applications, mechanisms that exploit this trend towards multithreaded architectures to help single-thread performance are highly desirable.

Speculative Multi-Threading (SpMT) [1, 3, 5, 10, 15, 17] is an approach to speed up the performance of single-threaded applications by breaking the application into many threads of control, each running on a different context. At any time, the oldest thread is always non-speculative and the remaining threads are speculative. The non-speculative thread benefits from cache prefetching, instruction reuse, and branch precomputation effects produced by the speculative threads resulting in improved single-threaded application performance. Hence, SpMT is an attractive option for applications running on multiple context processors that do not exhibit thread-level parallelism. While SpMT has been extensively studied, realistic implementations proposed typically use many thread contexts but gain little performance on many non-numeric applications. Implementation overhead and performance potential of a minimal SPMT model that uses the least number of threads (only 2 contexts) are not clear.

This paper presents a study of a low-overhead dual-core SpMT model that exploits various sources of speculative multithreading benefits: cache prefetching, branch precomputation, and instruction reuse. In this model, one core executes the speculative threads, while the other executes non-speculatively. To detect data dependence violations in the speculative threads, the model features a simple register dependence violation detection scheme that uses a single bit vector, and a simple, novel memory ordering mechanism that allows selective recovery from memory ordering violations in the speculative threads. When a data dependence violation is detected, only instructions affected by data dependence violations are re-executed, while the results of instructions that are executed by the speculative threads but not affected by data dependence violations are buffered and later committed by the non-speculative thread. The SpMT model uses a cache hierarchy where the first level caches are physically separate but logically shared for inter-thread memory data value communication, and to allow the speculative threads to prefetch into the first level caches of the non-speculative processor.

Using this SpMT model, we identify one key performance challenge in speculative multithreading, and that is ensuring the speculative threads remain on the correct path with respect to the non-speculative thread. Wrong path execution is an important source of performance loss for SpMT processors. Unlike conventional processors where a wrong path execution is detected and corrected when the branch is resolved, in SpMT processors the speculative threads are executing based on incomplete information–the speculative thread lacks register values computed by the non-speculative thread after the speculative thread is spawned. We show significant performance loss due to wrong path execution, yet prior work on SpMT has failed to address this important aspect. To ensure speculative threads exe-

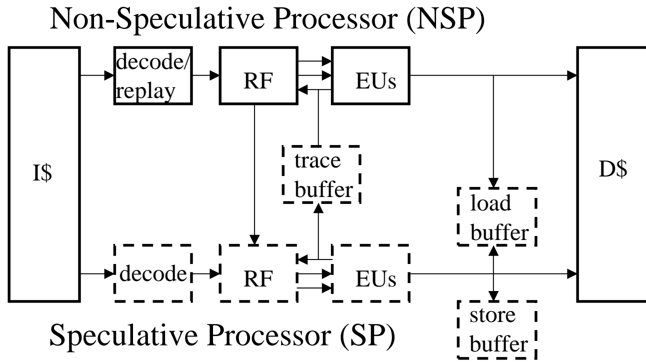

Figure 1. SpMT Machine Model

cute along the correct path, we propose a novel Wrong Path Predictor. The Wrong Path Predictor is based on the fact that sometimes a speculative thread branch prediction could convey better information about a program's correct path than the branch execution in the speculative thread context. Section 4 discusses Wrong Path Predictor results in detail and shows the mechanism significantly reduces the number of speculative threads executing along the wrong path.

Paper Contributions: The paper makes the following contributions:

1. A minimal dual-core SpMT model that achieves significant performance benefits. The model uses a simple memory ordering scheme that allows selective recovery of results computed by speculative threads.

2. A novel Wrong Path Predictor to overcome a key SpMT performance limitation.

3. A detailed study of the interaction of SpMT thread spawning policies and their performance benefits.

4. Analysis of the source of performance benefits in SpMT. We find 58% of the benefits comes from Instruction and Data Cache prefetching, 33% from Instruction reuse and 9% due to Branch precomputation.

This paper is organized as follows. Section 2 describes our SpMT machine model and Section 3 discusses our simulation methodology. Section 4 presents the Wrong Path Predictor results. Section 5 compares different thread spawning policies and their interactions. Section 6 analyzes the source of SpMT benefits and Section 7 discusses related work. Finally Section 8 presents concluding remarks.

2. SpMT Machine Model

Our SpMT machine model uses 2 processor cores/pipelines as shown in Figure 1. One of the cores is the Non-Speculative Processor (NSP) and the other core is the Speculative Processor (SP). Each core supports a single thread context. The results in this paper were simulated for Intel Itanium 2 [13] processor core.

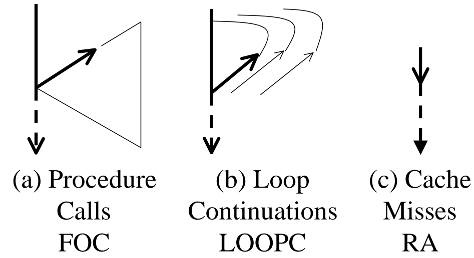

Figure 2. Thread Spawning Policies

Only the NSP spawns threads and the SP speculatively executes these threads. The NSP spawns threads (Figure 2) when it encounters a procedure call (FOC) or a loop (LOOPC), and spawns a Run Ahead (RA) [4, 12] thread once a load misses the cache and accesses DRAM. We call such points spawn points. When the NSP spawns a thread, it remembers the starting program counter of the thread, which is the address of the first instruction after a call, end of loop, or a load that misses the cache. We call the start of a speculative thread the speculation point. While the NSP executes the code between the spawn point and the speculation point (Figure 2, solid line), the SP executes code after the speculation point (Figure 2, dashed line).

The SP executes using the live-out register values from before the spawn point as inputs. Since the SP does not have the results computed between the spawn point and the speculation point, SP's execution of the threads is data speculative and hence we call these threads speculative threads. SP's execution cannot begin until it receives all of NSP's live-out registers from before the spawn point. We model a minimum register transfer time of 2 cycles. For design simplicity, we do not use value prediction to initialize live-in values to a thread.

The SP cannot modify the architectural state of the processor and hence does not commit its results to the architectural register file or to the memory system. Instead, the SP accumulates its execution results in a Trace Buffer (TB). Once the NSP reaches a thread's speculation point, it enters replay mode and starts reusing/committing valid results from the thread's TB, while re-executing instructions from the SP thread that are invalid due to data mis-speculation. Our SpMT model uses eight 512-entry TBs. There are more TBs (8) than thread contexts (1) in the SP to handle nested procedure calls and loops. In the event of such nested program structures, a thread corresponding to an outer procedure/loop gets killed to give way to a thread corresponding to an inner procedure/loop. The results from the killed outer procedure/loop thread are kept in a trace buffer for later reuse by the NSP. Thus, in our model, even though only one speculative thread can execute at any time in the SP, the results of up to eight speculative threads can be waiting to be committed by the NSP.

During replay mode, when the NSP replays all instructions in the trace buffer computed by the SP, or if the NSP finds the SP has mis-executed a branch and has gone down the wrong path, the NSP exits replay mode, resets the speculative thread, and resumes regular execution.

2.1. SpMT Memory Communication

To handle communication of memory data values, we have a single 128 entry (32 sets, 4 way associative) store buffer in the SP. The store buffer is indexed by load or store address. Stores from the SP write to the store buffer that forwards data to later speculative loads to the same memory address from the same thread. When a new thread starts executing in the SP, the store buffer is cleared.

"NSP stores" to "SP loads" data communication happens using shared data caches at all levels in the hierarchy. The level one data caches are physicaly separate but are logically shared in SpMT mode since committed stores from the NSP and line fills from the L2 cache write to the L1 caches of both the NSP and the SP. The other cache levels are physically shared. This organization lets loads in the SP prefetch (and in some cases, pollute) data values needed by the NSP. If the SP processor issues a load that depends on a non-committed NSP store a memory dependence violation occurs that will later be detected and corrected by the NSP processor, as described in the next subsection.

2.2. SpMT Data Dependence Violation Detection

To identify which instructions can be reused from the TB without re-execution, the NSP needs to resolve register and memory dependences between the non-speculative thread and a speculative thread.

2.2.1. Register Dependences. To correctly resolve register dependences, the NSP maintains a list of all registers that it modifies between the spawn point and the speculation point[1]. During replay mode, the NSP re-executes only instructions dependent on one of the modified registers. This information is propagated down dependence chains using a simple bit vector, with a single bit per register. The re-execution information can be determined very early in the pipeline (in the register rename stage) and hence instructions not requiring re-execution simply pass through the pipeline similar to a no-op and get committed.

2.2.2. Memory Dependences. To detect memory dependence violations due to SP load execution, the NSP uses a load buffer.

1 The Pentium architecture does not use a register stack for procedures like Itanium. Hence, register value matching is needed to identify SP FOC mis-speculated inputs after NSP reloads registers from memory stack.

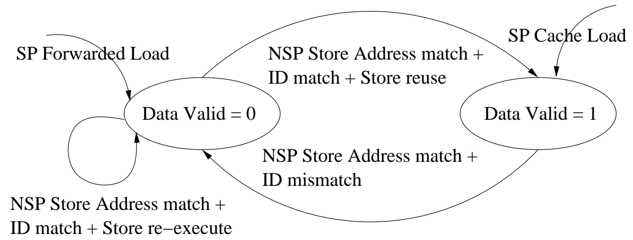

Figure 3. Load Buffer Entry State Diagram

Load Buffer Organization: We use a 512 entry, 4 way associative load buffer in the NSP. The load buffer is accessed using load and store addresses. Each entry in the load buffer has an address tag to identify the load, a thread ID to identify the speculative thread to which the load belongs, and a store ID to identify a forwarding store if the SP store buffer has forwarded data to the load. Each load buffer entry also has a Data Valid bit that indicates if such forwarding was valid or not.

Load Operation in the SP: When a load executes in the SP, it is allocated a load buffer entry. In case the store buffer forwards to the load, the forwarding store ID and thread ID are written in the entry, and the data valid bit is set to 0 indicating that the forwarding ID needs to be validated later by the NSP processor. Otherwise, the load gets its data from the cache and the data valid bit in the load buffer entry is initialized to 1, as shown in Figure 3. If there is no free load buffer entry in the set corresponding to the load address, the TB entry of the load is marked as having to re-execute during replay mode.

NSP Stores Snoop Load Buffers: The NSP re-validates all load-store dependences observed by the SP thread. All stores are committed in-order by the NSP and snoop the load buffer. Multiple stores can hit the same load buffer entry, but the last store that hits should be the data forwarding store since stores are committed to memory in order.

Figure 3 shows the state transitions of the load buffer entry Data Valid bit. The final value of the Data Valid bit depends on the last store that hits. If the ID of the last store matches the forwarding store ID in the entry, the valid bit is set to 1. Otherwise, the valid bit is cleared. Moreover, in order to support selective recovery of instructions affected by memory dependence violations, when a store ID match occurs on a snoop hit and the store is re-executing due to an earlier data dependence violation, the data valid bit is set to 0. This way, load instructions dependent on a re-executed store can be identified and re-executed.

NSP Loads Snoop Load Buffer: The NSP issues all loads to check the Data Valid bits in the load buffer entries of their speculative counterparts. If the Data Valid bit for a load is 0, a memory dependence violation has occured and the pipeline is flushed to re-execute the load and all instruc-

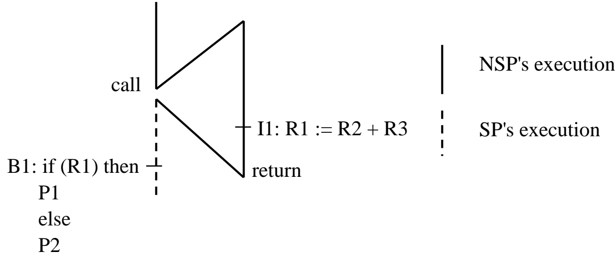

Figure 4. Branch Example

tions after it in the pipeline.

Minimizing Costly Pipeline Flushes: A valid bit in the trace buffer load entry mirrors the Data Valid bit in the load buffer entry. The TB entry valid bit is checked early in the pipeline when a NSP load is renamed. This early detection of memory dependence violations often allows the NSP to re-execute and correct a mis-speculated load and its dependents, without having to flush the pipeline.

2.3. Wrong Path Predictor

Branch mis-predictions and the associated wasted speculative execution are a known problem in conventional micro-architectures. They are more of a problem in a machine exploiting SpMT for the following reason. Since the SP executes using speculative data (the SP does not have the results of computations that occur between the spawn point and the speculation point), often the SP is unaware of the program's correct path. Hence it is possible that even after a branch has finished execution in the SP, the SP may continue execution along the wrong program path.

Consider the scenario shown in Figure 4, where a branch (B1) in the speculative thread is dependent on a result (R1) that is computed by instruction I1 in the NSP. Assume B1 is 'Taken'. However, since the value of R1 is not available to the SP, SP's execution may incorrectly resolve B1, i.e. SP's execution using the wrong value for R1 may indicate B1 should be 'Not Taken'. Once B1 has been resolved by the SP, execution will resume/continue along the 'Not Taken' path in the SP. However, since the correct direction for B1 is 'Taken', the SP must be prevented from executing along the 'Not Taken' path and forced to continue down the 'Taken' path. We use a Wrong Path Predictor (WPP) for this purpose. The WPP is trained to detect cases where the path found by SP's execution does not match the actual program path. In such cases, it will be beneficial to follow WPP's predicted path rather than SP's execution path.

The WPP chooses between SP's branch prediction and the branch outcome computed by SP's execution. Since the purpose of the WPP is to detect cases where SP's execution does not match NSP's execution, the WPP is trained using NSP's execution since the NSP alone knows the program's correct path. The WPP's choice matters only during the event that SP's prediction and the branch outcome computed by SP's execution differ. If the WPP chooses SP's prediction and the SP determines that the branch execution differs from the branch prediction, the SP ignores the branch execution and continues executing along the predicted path without invoking a branch recovery event.

3. Simulation Methodology

3.1. Experimental Setup

Our simulator is cycle-accurate and execution-driven, running binaries compiled with Intel Electron compiler [2, 9] for the Itanium architecture with maximum compiler optimizations, including those based on profile driven feedback, such as aggressive software prefetching, software pipelining, control speculation and data speculation.

We use 9 benchmarks (crafty, gap, gcc, gzip, mcf, parser, twolf, vortex and vpr) from the SPECint2000 suite for which user binaries are available to us. We omit the SPECfp2000 benchmarks because they have a lot of loops and hence benefit the most from the loop iteration thread spawning scheme (LOOPI). The LOOPI scheme works best with compiler support to setup loop index variables and minimize loop carried dependences [16]. Since this paper focuses on hardware only schemes, we present all results using just the integer benchmarks.

We use the train data sets of the benchmarks. For each benchmark, we execute 5 different samples of 2.5 million instructions each. We fast-forward the simulator for 1 billion (1B), 1.5B, 2B, 2.5B, and 3B instructions for the different samples. Since the complete runs of crafty, gzip, and parser run longer than the other benchmarks, these three benchmarks are sampled at 11B, 11.5B, 12B, 12.5B, and 13B instructions. The results presented for each benchmark are averages of the 5 samples. We warm up the caches before the start of each simulation.

3.2. Base Machine Parameters

Our base machine smulator models an EPIC Itanium 2 processor [13]. It is a 2-bundle wide machine where each bundle comprises up to 3 instructions. It also consists of a 16KB L1 cache with a 2 cyle latency, a 256KB L2 cache with a 6 cycle latency and a 3MB L3 cache with a 14 cycle latency, on die. Our base machine uses a 2K entry gshare branch predictor with a 11-bit global history. Since our base processor uses an in-order execution model, we did not experiment with complex branch predictors.

3.3. Thread Spawning Policy Implementation

This section presents implementation details specific to individual thread spawning policies.

Fork On Call (FOC) Policy: For the FOC policy, a procedure call instruction is the spawn point and the first instruction after the corresponding return instruction is the speculation point. The spawn point and speculation point can be trivially identified by examining the instruction opcode and the current program counter (PC).

Loop Continuation (LOOPC) Policy: For the LOOPC policy, we consider any backward branch as a potential loop. Thus, any backward branch is a potential spawn point. If the same backward branch is encountered multiple times consecutively corresponding to successive iterations of a loop, only one LOOPC thread is spawned. We also need the PC of the first instruction after the loop, to be used as the thread speculation point. For most loops, it is the fall-through PC of the loop's backward branch. However, for some loops the loop exit may be a taken branch in the middle of the loop body and identifying the speculation point in such cases may not be trivial.

For this purpose, we monitor the NSP's commit stream and associate the first PC greater than the loop's backward branch as the speculation point for the loop. This works well in most cases, but in some cases the first PC following a loop could vary and hence the speculation point predicted by the above scheme may never be reached after loop exit. We have a throttling mechanism to weed out such threads from being spawned after several consecutive failures to reach the speculation point.

Run Ahead (RA) Policy: For the RA policy, an L3 cache load miss is the spawn point and the first instruction after the load is the speculation point. The RA thread executes in the SP and can continue execution even after the cache miss is satisfied. In order not to be constrained by a finite window size, the speculative thread pseudo-commits [12] completed instructions as well as instructions dependent on incomplete loads and frees up window space. We extened the RA model to include instruction reuse in addition to prefetching benefits.

4. Wrong Path Predictor Results

This section presents the performance benefits of the Wrong Path Predictor using FOC spawn policy. The WPP benefits for the other policies are similar.

Figure 5 shows the performance benefits due to SpMT with and without the WPP. The performance improvement of SpMT with the WPP over without the WPP, is 4% on average. We find WPP reduces the percentage of threads getting killed because they execute along the wrong path by 50% on average. Correspondingly, the amount of wasted speculative execution due to threads executing along the wrong path also decreases from 28% to 15%.

The benchmarks twolf and vpr benefit the least from WPP. For these benchmarks, both SP's prediction and the speculative execution are wrong and hence the WPP does not help. gzip benchmark benefits the most from WPP. For gzip, 80% of the threads are killed because they execute along the wrong path without the WPP. With WPP this number drops to almost 0%. The performance benefit due to SpMT increases from -1% without WPP to 16% with WPP.

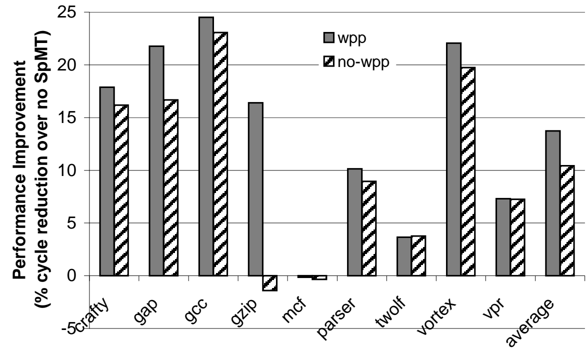

Figure 5. Wrong Path Predictor Benefits

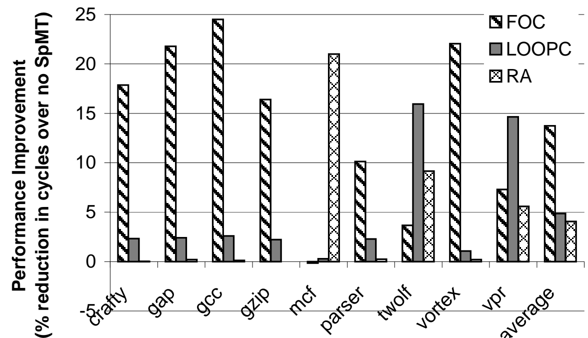

Figure 6. SpMT Performance Benefits

5. Thread Spawning Policy Interactions

In this section we use our SpMT model with the Wrong Path Predictor turned on, and present results on individual thread spawning policies as well as their interactions and benefits when combined all together.

5.1. Individual Thread Spawning Policies

Figure 6 shows the performance benefits due to the three policies. The average performance gain due to the FOC, LOOPC and RA policies are 14%, 5% and 4% respectively.

The benchmarks gap, gcc and vortex benefit the most from the FOC policy. These benchmarks have a procedure call every 200 instructions and their performance gain from FOC is greater than 20%. gcc shows the highest performance gain of 25%. parser too has a call every 200 instructions but most of these called procedures are small (less than 60 instructions) and so the NSP catches up quickly with the SP. crafty and gzip have a call nearly every 400 instructions

Table 1. Thread Policy Characteristics

Characteristic	FOC	LOOPC	RA
Useful Speculative Execution	76	57	84
Re-execution percentage	28	31	21
Speculative Processor Busy	42	18	6

and their performance gain due to FOC is around 15%. vpr, twolf and mcf have fewer than one call every 1000 instructions and their performance benefit from FOC is also lower.

The benchmarks twolf and vpr benefit the most from the LOOPC policy, 16% and 15% respectively. For these two benchmarks, a LOOPC thread is spawned every 500 instructions and each thread executes 200 instructions on average. The other benchmarks either have very few loops (less than one loop every 1000 instructions) or short loops (average LOOPC thread length less than 100 instructions) and hence do not benefit much from the LOOPC policy.

The benchmark mcf benefits the most from the RA policy (21%). mcf misses the L3 cache often and stalls waiting for data from main memory. An RA thread is created nearly every 600 instructions for mcf and each thread executes on average 200 instructions. The benchmarks twolf and vpr also have a significant number of L3 cache misses (and hence RA threads) and show 9% and 6% performance improvements respectively.

Table 5.1 shows different characteristics of each thread spawning policy. The Useful Speculative Execution metric shows the percentage of instructions executed by the SP, that are replayed by the NSP. The remaining instructions executed by the SP are wasted because they are on the wrong path due to data mis-speculation. The Re-execution percentage metric shows the percentage of replayed instructions that need to be re-executed. The LOOPC policy has more wasted speculative execution and more re-executions than the FOC and RA policies. This suggests that for the LOOPC threads, there are more data dependences between instructions before and after the speculation point. Note that less than a third of the replayed instructions need to be re-executed for all the policies.

The Speculative Processor Busy metric shows the percentage of cycles the SP is busy executing speculative threads. Since the FOC policy shows the most benefits, it also keeps the SP busiest. The individual benchmarks that benefit the most from the LOOPC and RA policies do occupy the SP more than the average. From the Speculative Processor Busy numbers, we can see that the SP is not busy all the time with just one thread spawning policy and there is scope for increasing the SP busy cycles by implementing more than one policy at a time.

The best individual spawning policy in our model is FOC, with 14% performance gain. Next section shows the performance gain from combining all spawning policies.

Table 2. Speculative Coverage Intersection

	F	L	R	FL	LR	RF	FLR
crafty	34	4	0	0	0	0	0
gap	48	4	1	2	0	0	0
gcc	32	4	0	1	0	0	0
gzip	37	7	0	5	0	0	0
mcf	2	1	32	0	0	0	0
parser	21	3	0	1	0	0	0
twolf	6	27	15	0	5	2	0
vortex	53	4	0	1	0	1	0
vpr	22	31	8	4	1	1	0
average	28	9	6	2	1	0	0

5.2. Combined Thread Spawning Policies

Before combining the different policies, we study the intersection in speculative coverage areas between the policies. Speculative coverage area is the region of a program benefitting from SpMT. Since two different policies could be helping the same portion of an application, such a study would give us an idea of the maximum speculative coverage area we can hope to achieve by combining these policies. To do this, for each thread spawning policy, we generate traces of all instructions that were speculatively executed and then retired using that policy. We then post-process the traces, to determine the intersecting and non-intersecting portions for different combinations of policies.

Table 5.2 provides the speculative coverage intersection results. In Table 5.2, we refer to the FOC, LOOPC and RA policies as F, L and R respectively. The first three columns show the areas exclusively covered by individual policies and indicate no policy is a proper subset of another policy. Each policy has some exclusive speculative coverage area and hence implementing all three policies together in a machine could be beneficial.The next four columns show the common speculative coverage areas for groups of two and three policies. These numbers indicate there is not much intersection in the speculative coverage areas of the three policies. The benchmarks gzip, twolf and vpr have some intersections between the FOC, LOOPC and RA policies, but the intersections are not dominant.

Figure 7 presents performance results for all policies together (F+L+R). For each benchmark, the MAX(F,L,R) scheme statically selects a single policy that performs the best for that benchmark. We find for crafty, parser and vpr benchmarks, the benefits due to F+L+R is significantly better than any individual policy. For vpr, the maximum performance gain from any individual policy is 15% achieved by the LOOPC policy while F+L+R achieves a 22% performance improvement. On average, F+L+R achieves an overall speed up of 20% over a base configuration without SpMT. This compares well with the 18% for MAX(F,L,R) (which would also require all three policies to be imple-

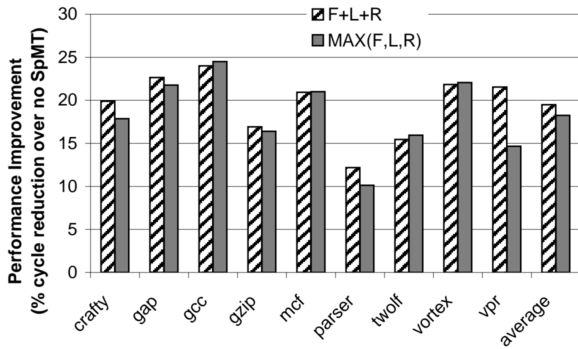

Figure 7. Thread Policy Interactions

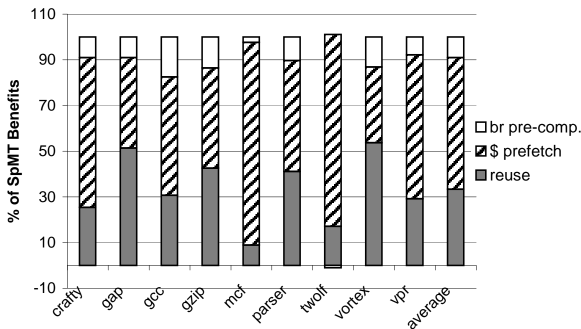

Figure 8. Source of SpMT Benefits

mented in a machine) and 14% if only one thread spawning policy were to be implemented.

6. Source of SpMT Benefits

SpMT benefits come from three main sources: cache prefetch, reusing speculative thread instructions that do not violate data dependences, and branch precomputation where the NSP avoids mispredictions from branches precomputed by the speculative threads. To isolate the benefits of SpMT from cache prefetching, instruction reuse and branch precomputation, we first turn off branch precomputation by not reusing branch execution information from the TB during replay mode. Next, we disable instruction reuse altogether and use SP's execution only for prefetching.

Figure 8 plots the portion of the SpMT benefits due to the three different sources listed above. These results are for the combined F+L+R configuration. On average, 58% of the SpMT benefits are due to Instruction and Data Cache prefetching, 33% due to Instruction reuse, and 9% due to Branch precomputation. However, these results vary for the different benchmarks. The benchmarks gap and vortex benefit the most from Instruction reuse, while gzip and parser benefit equally from both cache prefetching and instruction reuse. Hence, to realize the full potential of SpMT, all three sources of SpMT benefits must be utilized.

7. Related work

SpMT Models: The Multiscalar [5, 15] model relies on the compiler to identify parallel tasks and to convey register and memory dependences between tasks. Our model is completely dynamic and maintains binary comaptibility. DMT [1] is a hardware-based scheme but requires a large and complex data-flow engine outside the execution pipeline including a large ROB that spans all threads to recover from data and control mispredictions. The SpMT architecture in this paper has a continuous non-speculative thread that consumes speculative thread results, detects data dependence violations, and initiates recovery.

Marcuello and Gonzalez [10] propose a Data Speculative Multi-threaded (DaSM) Architecture to parallelize loop iteration threads. They use data value prediction, and predict number of reads and writes in each thread. Their recovery mechanism is coarse and they squash a thread violating a data dependence and all subsequent threads. In contrast, our model does not require value prediction or number of reads and writes prediction, and allows fine grain recovery.

Zilles and Sohi [17] use speculative helper threads to prefetch data required by later delinquent loads and to pre-execute hard to predict branches. Their model is simple but achieves only two of the three benefits of SpMT - cache prefetching and branch precomputation. However, our results show that instruction reuse constitutes a significant portion (33%) of the benefits of SpMT on average. Instruction reuse is the main contributor to SpMT benefits for several benchmarks as we see in Section 6.

The Slipstream processor [14] uses two streams, an advanced stream (A-stream) and a redundant stream (R-stream). The A-stream prefetches data and control information to the R-stream to use. The R-stream verifies A-stream's exeuction and identifies ineffectual instructions and passes the information to the A-stream, so it can skip executing those instructions. In SpMT, the SP core is similar to the A-stream that runs ahead and the NSP that consumes the results of SP's computation is similar to the R-stream. In SpMT during replay mode, the NSP needs to execute only a subset of instructions exeucted by the SP that need re-execution. However, the R-stream in Slipstream needs to execute all program instructions (albeit at a faster rate).

SpMT Thread Spawning Policies: Marcuello et. al. [11] present a control quasi-independent points (CQIP) scheme where the compiler determines points in the program that have very few control and data dependences. They show the CQIP scheme performs better than a combination of FOC, LOOPI and LOOPC policies, using an aggressive 16 processor elements SpMT model. Codrescu and Willis [3] evaluate the Fork On Call (FOC) and Loop Iteration (LOOPI) thread spawning policies individually and propose improv-

ing speculative thread coverage using the static mem-slicing algorithm to create additional threads at memory operation boundaries. In contrast, our SpMT model uses simple extensions to a dual-core processor and a combination of the dynamic thread spawning policies, FOC, LOOPC and RA to exploit the benefits of SpMT.

SP Branches: Gummaraju and Franklin [7] deals with fragmented branch history and incorrect history to improve branch prediction for SpMT processors. They do not address incorrect SP branch execution.

8. Concluding Remarks

This paper presents a dual-core speculative multithreading design that achieves significant performance benefit with low-overhead hardware. One key design feature of the proposed model is a novel and effective inter-thread memory dependence handling scheme. We identify a key problem of SpMT processors that previous research in the field has not addressed – correctly resolving branch directions in the back-end of a speculative processor – and propose the Wrong Path Predictor optimization to handle it. The Wrong Path Predictor significantly reduces the number of speculative threads executed along the wrong path and improves the overall performance gain due to SpMT. Measurements using our SpMT model reveals all three policies, Fork on Call, Loop Continuation and Run Ahead, are beneficial for different applications or different portions of the same application, and that all three sources of SpMT benefits, cache prefetching, branch precomputation and instruction reuse, must be harvested to realize the full potential of SpMT.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–236, 1998.

[2] J. Bharadwaj, W. Y. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The Intel IA-64 compiler code generator. *IEEE Micro*, 20(5):44–53, September 2000.

[3] L. Codrescu and D. S. Willis. On dynamic speculative thread partitioning and the mem-slicing algorithm. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 40–46, October 1999.

[4] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, 1997.

[5] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 58–67, 1992.

[6] P. N. Glaskowsky. IBM raises curtain on Power5. *Microprocessor Report*, Oct. 2003. http://www.mdronline.com/mpr/h/2003/1014/174101.html.

[7] J. Gummaraju and M. Franklin. Branch prediction in multi-threaded processors. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT '00)*, pages 179–188, Philadelphia, October 15–19, 2000. IEEE Computer Society Press.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technical Journal*, Feb. 2001. Q1 2001 Issue.

[9] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C.-C. Lim, J. Ng, and D. Sehr. An advanced optimizer for the IA-64 architecture. *IEEE Micro*, 20(6):60–68, November 2000.

[10] P. Marcuello and A. Gonzalez. Exploiting speculative thread-level parallelism on a smt processor. In *Proc. Int'l Conf. on High Performance Computing and Networking*, 1999.

[11] P. Marcuello and A. Gonzalez. Thread spawning schemes for speculative multi-threading. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA '02)*, pages 55–64, Boston, Massachusetts, February 2002.

[12] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the Ninth IEEE International Symposium on High Performance Computer Architecture*, 2003.

[13] S. D. Naffziger and G. Hammond. The implementation of the next-generation 64b Itanium microprocessor. In *Digest of Technical Papers, 2002 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 45, February 2002.

[14] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 269–280, December 2000.

[15] T. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Jan. 1998.

[16] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th Architectural Support for Programming Languages and Operating Systems, ASPLOS-X*, October 2002.

[17] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.