In Proceedings of the 34th International Symposium on Microarchitecture, December, 2001

Dynamic Speculative Precomputation

Jamison D. Collins[†], Dean M. Tullsen[†], Hong Wang[‡], John P. Shen[‡]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

[‡]Microprocessor Research Lab
Intel Corporation
Santa Clara, CA 95052-8119

Abstract

A large number of memory accesses in memory-bound applications are irregular, such as pointer dereferences, and can be effectively targeted by thread-based prefetching techniques like Speculative Precomputation. These techniques execute instructions, for example on an available SMT thread context, that have been extracted directly from the program they are trying to accelerate. Proposed techniques typically require manual user intervention to extract and optimize instruction sequences. This paper proposes Dynamic Speculative Precomputation, which performs all necessary instruction analysis, extraction, and optimization through the use of back-end instruction analysis hardware, located off the processor's critical path. For a set of memory limited benchmarks an average speedup of 14% is achieved when constructing simple p-slices, and this gain grows to 33% when making use of aggressive optimizations.

1. Introduction

The CPU-memory gap, the difference between the speed of computation and the speed of memory access, continues to grow. Meanwhile, the working set of the typical application is also growing rapidly. Thus, despite the growing size of on-chip caches, performance of many applications is increasingly dependent on the observed latency of the memory subsystem.

Cache prefetching is one technique that reduces the observed latency of memory accesses by bringing data into cache before it is accessed by the CPU. Cache prefetching comes in three varieties. Hardware cache prefetchers [4, 10, 9] observe the data stream and use past access patterns and/or miss patterns to predict future misses. Software prefetchers [13] insert prefetch directives into the code with enough lead time to allow the cache to acquire the data before the actual access is executed. Recently, the expected emergence of multithreading processors [21, 20] has led to thread-based prefetchers [6, 23, 11, 2, 16], which execute code in another thread context, attempting to bring data into the shared cache before the primary thread accesses it.

All three techniques have advantages and disadvantages. Hardware prefetchers only work when the data stream exhibits predictable patterns. Software prefetching is constrained to the control flow being executed by the main thread prior to the access. Software and thread-based prefetchers are generated by the compiler (or programmer) and must operate strictly based on information known statically at compile time. Software and thread-based prefetchers do not work on code compiled for machines that do not have the same prefetch support, or in some cases do not work on code compiled for a different cache organization.

This paper presents a new form of cache prefetching — hardware-constructed thread-based prefetching. Like thread-based prefetching, it can use the actual code of the program to prefetch irregular, data-dependent access patterns that cannot be predicted. Like thread-based prefetching, the prefetch code is decoupled from the main code, allowing much more flexibility than software prefetching. Like hardware prefetching, this technique works on legacy code and does not sacrifice software compatibility with future architectures, and can operate on dynamic information rather than static to initiate prefetching and to evaluate the effectiveness of a prefetch stream.

We call our hardware-constructed thread-based prefetch mechanism Dynamic Speculative Precomputation. It is based on the (software and manually-constructed) thread-based technique Speculative Precomputation [6], which has similarities to [11, 23]. Like those techniques, it executes threads decoupled from the main thread to generate prefetches, but unlike those techniques, the threads are constructed, spawned, improved upon, evaluated by, and possibly removed by hardware.

This paper is organized as follows. Section 2 describes related prior research. Section 3 describes the simulation methodology and benchmarks studied. Section 4 presents details on the hardware used to capture p-slices at runtime. Section 5 explores advanced p-slice optimizations. Section 6 explores Dynamic SP when relaxing ideal hardware assumptions. Section 7 concludes.

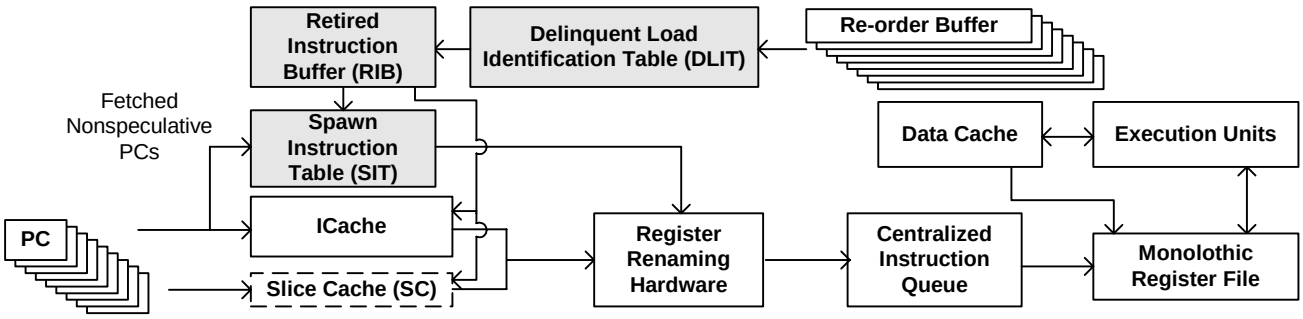

Figure 1. Pipeline organization of modeled simultaneous multithreading processor.

2. Related Work

Several thread-based prefetching paradigms have been proposed previously, including Collins et al.'s Speculative Precomputation (SP) [6], Zilles and Sohi's Speculative Slices [23], Roth and Sohi's Data Driven Multithreading [16], Luk's Software Controlled Pre-Execution [11], Annavaram et al.'s Data Graph Precomputation [2], and Moshovos et al.'s Slice-Processors [12].

SP works by identifying the small number of static loads, known as delinquent loads, that are responsible for the vast majority of memory stall cycles. Precomputation Slices (p-slices), minimal sequences of instructions which produce the address of a future delinquent load, are extracted from the program being accelerated. When an instruction in the non-speculative thread that has been identified as a trigger instruction reaches some point in the pipeline, the corresponding p-slice is spawned into an available thread context on a simultaneous multithreading (SMT) processor [21, 20]. Also introduced is the concept of a chaining trigger, which permits a speculative thread to directly spawn further speculative threads. SP focuses on the benefits from precomputation on an SMT-capable in-order processor implementing the Itanium^{TM} ISA [8]. Significant focus is on the use of existing Itanium features to simplify implementation. P-slices using chaining triggers are constructed manually.

Speculative Slices [23] focuses largely on the use of precomputation to predict future branch outcomes and to correlate predictions to future branch instances in the non-speculative thread. A thread repeating technique similar to the chaining p-slices of this paper is explored, but control of such threads is limited to placing a maximum number of repeats on thread execution. P-slices are constructed manually.

Software Controlled Pre-Execution [11] focuses on the use of specialized, compiler inserted code that is executed in available hardware thread contexts to provide prefetches for a non- speculative thread.

Data Graph Precomputation [2] explores the runtime construction of instruction dependence graphs (similar to the p-slices of SP) through analysis of instructions currently within the instruction queue. The constructed graphs are speculatively executed on a specialized secondary execution pipeline.

Slice-Procssors [12] dynamically construct instruction slices to prefetch delinquent loads. Slices corresponding to the basic p-slices of this work are constructed using back-end hardware and executed on a specialized pipeline.

Dependence Based Prefetching [15] targets pointer-based data structures by identifying relationships between loads that produce data and loads which dereference these data. A prefetch engine traverses data structures ahead of the executing program by loading and dereferencing data structure fields.

Other research has also explored the use of back-end instruction processing on behalf of the main thread. The trace cache fill unit [14] groups committed instructions together into program traces, possibly applying transformations on those instructions [7]. Instruction Path Coprocessors [5] have been proposed as a software controlled back-end processor.

Previously proposed hardware prefetchers [4, 10, 9], including those targeted at irregular memory access patterns [9, 15], all rely on pattern-based and history-based predictability to enable accurate prefetching.

Sundaramoorthy et al. proposed Slipstream Processors [17] in which a non-speculative version of a program runs alongside a shortened, speculative version. Outcomes of certain instructions in the speculative version are passed to the non-speculative version, providing a speedup if the speculative outcome is correct. Their work focuses on implementation on a chip-multiprocessor (CMP).

In contrast to most previous research, this work focuses exclusively on automatic, hardware-based p-slice construction, management, and optimization, performed entirely at program runtime through the use of a back-end instruction analyzer. Techniques proposed for automatic extraction of p-slices are significantly more aggressive than previously proposed automatic techniques, and are comparable to the most aggressive manually applied optimizations. Additionally, SP in the context of multiple non-speculative threads is explored.

3. Simulation Methodology

Benchmarks are simulated using SMTSIM, a cycle accurate, execution driven simulator [18] that simulates an out-of-order, simultaneous multithreading processor. SMTSIM executes unmodified alpha binaries. Benchmarks from

Pipeline Structure	8 stage pipeline, 1 cycle misfetch penalty, 6 cycle mispredict penalty
Fetch	8 instructions total from up to two threads
Branch Predictor	16k entry GSHARE, 256 entry 4-way associative BTB
Execution Resources	8 total int units, 4 can perform mem ops, 3 fp. All units pipelined, 200 int and 200 fp renaming regs 128 entry int and fp instruction queues. Each thread has a 256 entry ROB
Memory Hierarchy	32KB, 2-way ICache, 64KB, 4-way data cache, 1024KB, 4-way shared L2 cache (10 cycle latency) Memory has 110 cycle latency, 128 entry instruction and data TLB TLB misses handled by pipelined, on chip TLB miss handler, 60 cycle latency
Multithreading	8 total hardware thread contexts
Dynamic SP	64 entry DLIT, 32 entry SIT, 8 entry SAT, 512 entry RIB

Table 1. Details of the modeled processor

the SPEC suite are compiled for a base SPEC build, and Olden [3] benchmarks are compiled with gcc -O4. All simulations with a single non-speculative thread are executed for 300 million total committed instructions.

Because it is essential that they execute as quickly as possible, fetch preference is given to speculative threads over non-speculative ones, and preference is given to older speculative threads over younger ones.

Table 1 shows the configuration of the processor modeled in this research. Unless noted, simulation results assume speculative threads are spawned instantaneously when a trigger instruction reaches the rename stage, and that p-slices are fetched from an on-chip slice cache [16]. More realistic assumptions in both areas are evaluated in Section 6.3.

This paper evaluates SP with 10 memory-limited benchmarks (between 48% and 315% speedup when assuming a perfect L2 cache) and five ILP limited benchmarks (between 1% and 18% speedup from a perfect L2 cache). Details are shown in Table 2.

The memory limited benchmarks fall into four categories — pointer based applications with complex control flow (*mcf* and *vpr* from SPECINT), array based applications with simple control flow (*art*, *equake*, *mgrid* and *swim* from SPECFP), loop based pointer traversal applications (*mst* and *em3d* from Olden) and recursive applications with a small recursive function body (*perimeter* and *treeadd* from Olden). Both *swim* and *mgrid* contain significant software prefetching. ILP limited benchmarks are drawn from the SPEC suite and consist of three integer and two floating point applications.

4. Hardware Support for Dynamic Speculative Precomputation

A processor implementing dynamic speculative precomputation must support three basic operations (1) identify a set of delinquent loads to target, (2) construct p-slices that target those loads, and (3) spawn and manage the execution of speculative threads. Figure 1 shows the processor modeled in this research; structures shown in gray are specialized structures necessary to fulfill these precomputation requirements. The hardware slice cache (shown dashed) is not necessary but may improve performance. The following sections describe these structures.

4.1. Identifying and Tracking Delinquent Loads

Delinquent loads are identified at runtime by a hardware structure known as the Delinquent Load Identification Table (DLIT). DLIT entries are allocated first-come first-serve to static loads which missed in L2 cache on their most recent execution. After a load occupies an entry for 128K total committed instructions, it is evaluated for delinquency. A load is considered delinquent if, over the previous 128k instructions, it had been executed at least 100 times, and, on average, each instance, spent four or more cycles as the oldest instruction in its thread, waiting for its memory request to be satisfied. This is a form of critical path prediction, based on the *ALOld* metric described by Tune et al. [22], and henceforth referred to as the number of cycles spent blocking commit. Other metrics could also be used (such as counting cache miss rate at different levels of the memory hierarchy), but we found this one to be effective. When a load is found to be delinquent, it is passed on to the next phase and its entry locked into the DLIT.

In programs which contain software prefetch instructions that do not entirly eliminate memory latency, the prefetches themselves should be treated as delinquent. They will not be identified as such by the above metric, however, because prefetch instructions commit even if their memory request is outstanding. A prefetch is considered delinquent if, on average, the prefetched line is accessed by a non-prefetch load at least four cycles before it arrives in cache. Only *swim* and *mgrid* were impacted by targeting delinquent prefetch instructions.

Each load in the DLIT is specified by a tuple – the PC of the load itself and the PC of the call instruction which invoked the function that contains the load. Thus, the same load PC may be present in multiple DLIT entries if it is within a function which has multiple callers.

4.2. Hardware to Construct P-Slices

Once a load is identified as delinquent, a p-slice is constructed to prefetch the load, using a hardware structure called the Retired Instruction Buffer (RIB). The RIB stores a trace of committed instructions, similar to a large trace cache

Bench	Instr Skipped	Input	Speedup Perf L2$
Mem Limited Benchmarks			
mcf	5B	Ref	132%
vpr	5B	Ref	48%
art	5B	Ref	60%
equake	5B	Ref	103%
mgrid	5B	Ref	54%
swim	5B	Ref	236%
em3d	120M	25000 nodes	315%
mst	2B	3407 nodes	113%
perimeter	380M	12 levels	55%
treeadd	940M	22 levels	170%
ILP Limited Benchmarks			
ammp	5B	Ref	9%
crafty	5B	Ref	2%
galgel	5B	Ref	18%
gzip	5B	Ref	1%
vortex	5B	Ref	4%

Table 2. Details of studied benchmarks.

fill unit [14]. For each instruction, it holds the PC, logical source and destination register numbers, and a "marked" bit, which is used during analysis to mark instructions for inclusion in the p-slice. Because it analyzes instructions which have already committed, the RIB is entirely off the processor critical path. Instructions are stored as a FIFO, overwriting the oldest instruction when a new one is inserted. We assume a 512 entry RIB.

The RIB operates in three states — *idle*, *instruction-gathering mode* and *slice-building mode*. For most applications, the RIB spends the vast majority of its time (more than 99%) in idle mode, and thus could be easily designed for low power. The RIB remains in idle mode until a delinquent load lacking a p-slice is committed, causing a transition into instruction-gathering mode. In this mode the RIB accepts instructions (but performs no analysis on them) as they are committed by that thread.

When a second instance of the delinquent load is inserted into the RIB, it transitions into slice-building mode, where a p-slice will be constructed by identifying those instructions which produce register values the delinquent load is data dependent on. Following slice construction, the RIB transitions back into idle mode.

If too many instructions are committed between the two instances of the load, the first instance will be evicted from the RIB. Even if this happens we can still create a p-slice, but, because delinquent load instances are typically clustered, such a control flow likely represents a rarely executed control path. Thus we will re-capture traces up to five times, attempting to capture both instances of the load. After that, we will use a trace with just the second instance, constructing a p-slice starting with whatever instruction happens to be the oldest in the RIB.

```
struct DATATYPE {                    loop:
  int val[10];                  I1  load    r1=[r2]
};                              I2  add     r3=r3+1
                                I3  add     r6=r3-100
DATATYPE * data [100];          I4  add     r2=r2+8
                                I5  add     r1=r4+r1
for(j = 0; j < 10; j++) {       I6  load    r5=[r1]
  for(i = 0; i < 100; i++) {    I7  add     r5=r5+1
    data[i]->val[j]++;          I8  store   [r1]=r5
  }                             I9  blt     r6, loop
}
```

Figure 2. Example C and assembly loop targeted with SP.

4.3. Basic P-slice Construction

During RIB analysis, instructions are analyzed serially, from the most recent to the oldest. Analysis builds up the p-slice (initially consisting only of the delinquent load) by identifying instructions which produce registers consumed by the current partial p-slice. This set of registers is known as the p-slice live-in set. When such an instruction is encountered, it is added to the p-slice (by setting its marked bit) and the live-in set is updated by clearing the dependence on that instruction's destination register and adding dependences for its source registers.

The live-in set is efficiently implemented as a bit vector with length equal to the total number of logical registers. Slice construction is similar to compiler live range analysis [1], but is not iterative, for basic p-slice construction, because we walk over a single linear path — since all branches in the RIB are executed branches, there is no control-flow uncertainty.

When RIB analysis completes, marked instructions indicate the p-slice, and the live-in set specifies the registers that must be copied from the parent thread when the p-slice is spawned.

Secondarily, the RIB is used to identify a trigger instruction, which will cause a speculative thread to be spawned and execute the p-slice when the instruction is renamed. For initial analysis, the last analyzed instruction (typically the delinquent load itself) is conservatively chosen as the trigger instruction.

4.4. Example

This section provides an example of p-slice construction via the RIB. Figure 2 shows the code from a sample loop in both C and assembly, with the delinquent load highlighted. Figure 3 shows the state of the RIB when the instructions between two instances of the delinquent load have been analyzed. To the right of the RIB four columns of information are shown — if the instruction is included in the p-slice, its source registers, its destination registers and the new live-in set after analyzing the instruction.

Initially the p-slice consists of only the delinquent load, with the live-in set as the source register for this instruction, r1. Analysis proceeds from the delinquent load through older instructions until the second instance of the delinquent load is encountered. On the far right is the final p-slice, as well as its set of live-in values.

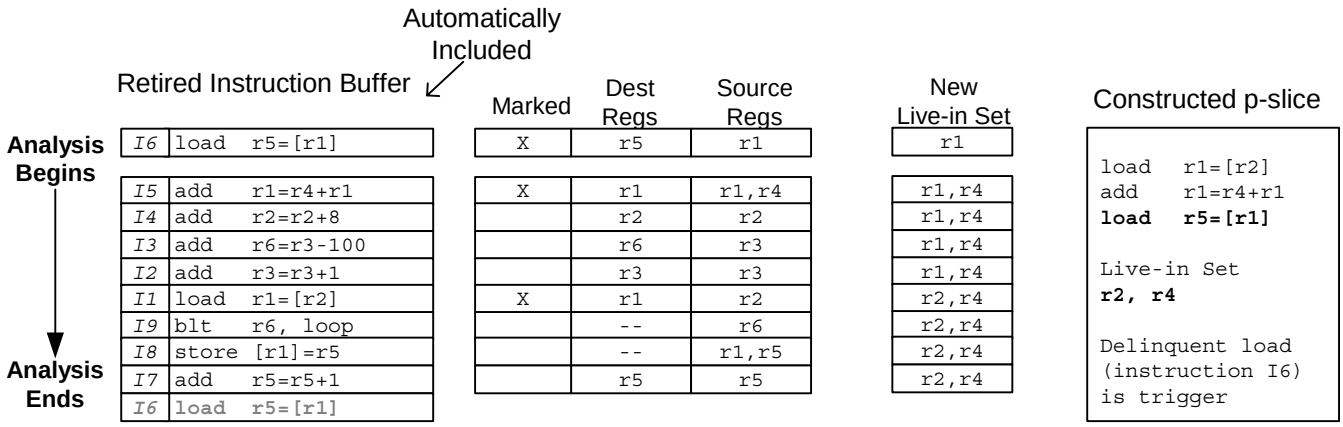

Figure 3. State of RIB as example code loop from Figure 2 is analyzed.

4.5. Basic P-slice Optimizations

This section describes some optimizations which are applied to the basic construction algorithm. More aggressive optimizations will be examined in Section 5, but those described here require only minor changes to the basic p-slice construction algorithm.

Store-Load Dependence Detection Because RIB analysis occurs over instructions within a small instruction window, heap-based store-load dependencies involving p-slice instructions (which create undetected data dependencies) are unlikely. However, store-load dependencies in the form of stack-based saving and restoring of values can occur. Such dependencies are identified by adding a small stack address table (SAT) to the RIB.

When a load with the stack pointer as its address base is included in a p-slice, the address accessed by this load is also stored in the SAT. When a store instruction (again with the stack register as its base) is analyzed, the SAT is queried with the target address. If an address match is found, then a store-load dependence has been identified. When the load and store instructions involve the same non-stack register (ie the register was saved to the stack then later restored), the store-load dependence only has an effect if this register is overwritten by an instruction included in the p-slice between the load and the store. Otherwise, the load will simply overwrite the register with the same value, and neither the load nor store needs to be included in the p-slice. To make use of the SAT, the stack memory addresses accessed by load and store instructions must be available during RIB analysis. The full width of the accessed address need not be saved, however, as only the stack pointer offset is needed.

This table does not significantly complicate the p-slice construction algorithm because the read of the memory location will be encountered before the write, just as with register dependencies. The load and store instructions involved in the dependence are converted into move instructions, and pass their value through an unused logical register. This work assumes an eight-entry fully-associative SAT, though typically fewer entries are needed.

Targeting Recursive Programs During RIB analysis it is possible to encounter multiple instances of the same delinquent load PC (but with distinct tuples because the function containing it was called from different locations). This situation is taken as a hint that the delinquent load is within a recursive function, and such loads are automatically added to the p-slice (even though it does not produce a p-slice live-in value). This allows a single p-slice to target multiple loads.

Prefetch Conversion Because the final instruction in a p-slice loads a value which is never consumed, its destination register can be changed to the zero register (r31 on the modeled ISA), converting it into a software prefetch. Because these instructions commit without waiting for their memory access to complete, thread contexts executing such p-slices can be freed much earlier. We call this optimization prefetch conversion.

For most p-slices, only the final load in the p-slice can be converted. However, loads which are only included in the p-slice because of the recursive optimization previously described can also be converted, as they also produce a value which is not consumed. If prefetch conversion is applied to the example above, the final p-slice instruction becomes `load r31=[r1]`.

Instruction Removal The size of a p-slice can be reduced by removing unecessary updates to the stack and global pointers. For example, the corresponding stack pointer updates at the beginning and end of a function call are unecessary to include in a p-slice when no intervening read to the register is included.

Detecting such instruction pairs is achieved by recording (in a stack based structure) each p-slice instruction that updates the global or stack pointer. If no instruction which reads the updated register is included before one which undoes the previous update (considering only adds followed by

subtracts or vice versa), neither instruction is included in the p-slice.

4.6. Hardware to Spawn and Manage P-slices

The Slice Information Table (SIT) initiates speculative threads and manages p-slices. Every cycle it is queried with the addresses of the instructions decoded on that cycle on behalf of a non-speculative thread. If a trigger instruction has been decoded, hardware in the register rename stage is informed, and a speculative thread is spawned the following cycle (when the trigger instruction reaches the rename stage). If no thread context is available, the spawn request is ignored. For initial analysis, the child thread's live-in values are assumed to be instantaneously copied from the parent; Section 6.3 evaluates more realistic assumptions. SIT entries are allocated by the RIB after a p-slice is constructed.

The SIT also evaluates p-slice effectiveness. A p-slice is considered effective when its prefetches cause a greater reduction in the memory stall cycles incurred by loads in the non-speculative thread than instructions were fetched on behalf of the p-slice when it had been spawned from the correct path. The precise impact on memory stall cycles is impractical to directly measure in a real processor, so we approximate this value by assuming every prefetched cache line is both useful and timely, saving the full latency to the accessed memory structure. Thus, a prefetch that hits in L2 cache saves 10 cycles, and one that accesses memory saves 110, even if the prefetched data is never eventually accessed. Ten cycles are subtracted for each prefetch evicted from L1 cache before being accessed. This computes number of saved memory cycles optimistically, but classifications are still accurate because most p-slices are highly biased toward being useful or useless.

For every 128K total committed instructions, each p-slice which had been spawned at least once is evaluated to determine its effectiveness. When a basic p-slice is found to be ineffective (meaning more instructions were fetched on its behalf than cycles were saved), it is eliminated and its DLIT entry cleared; a new p-slice will be captured for the load if the DLIT again classifies the load as delinquent at a future time. P-slices employing optimizations are treated differently, as described in Section 5.4.

The RIB employs two simple filters to reduce the likelihood of generating ineffective p-slices. If fewer than 32 instructions occur between the two instances of the delinquent load or the p-slice consists of more instructions than the average number of cycles the load had spent blocking commit over the previous 128K instructions, the p-slice construction is aborted. The first filter insures that the p-slice will actually be able to trigger cache misses before the data is accessed by the non-speculative thread, and the second filter implements a slightly modified version of the test for effectiveness described above. If p-slice construction is aborted more than five times, the DLIT entry is cleared.

If a line is prefetched by multiple p-slices, only the first p-slice to prefetch the line gets the "credit" for the prefetch.

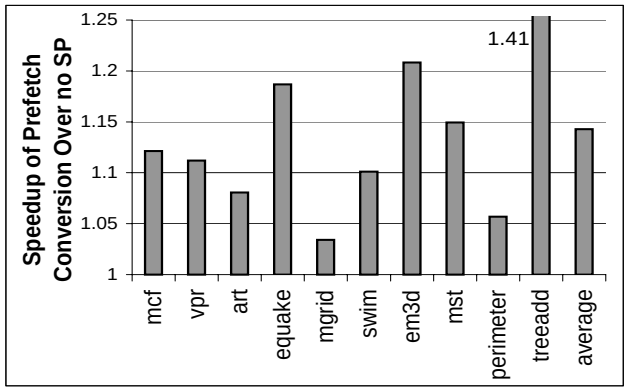

Figure 4. Memory limited benchmark speedup provided by basic Dynamic SP over baseline processor without SP.

Thus, p-slices must compete with each other to save memory cycles, enabling a more recently constructed p-slice which initiates prefetches earlier to cause an existing p-slice to be eliminated.

4.7. Speculative Thread Instruction Fetch

Speculative threads fetch their instructions from a logical structure called the slice cache (SC). This structure can be implemented in hardware [16, 12], eliminating fetch conflicts between speculative and non-speculative threads, or in software as a specially allocated memory buffer. In the software implementation, the OS allocates a special region of memory into which p-slice instructions are stored. Speculative threads fetch from this buffer, automatically bringing their instructions into the ICache. The performance of these approaches is compared in Section 6.3.

Once a p-slice is constructed, its instructions must be stored in the SC. For each instruction in the p-slice, the ICache is accessed and the instruction read out, possibly manipulated (by applying, for example, prefetch conversion), then written into the (hardware or software) slice cache. Instructions are read out either by stealing ICache read ports (if the cache is multiported) or by stalling fetch entirely. Because p-slice construction is rare, the performance impact is negligible in either case.

4.8. Basic P-slice Performance

This section shows the performance of applying the baseline form of Dynamic Speculative Precomputation described above, applying the basic optimizations of Subsection 4.5. Figure 4 shows the speedup achieved over a processor with no SP. All programs show speedups, ranging from 3.4% (*mgrid*) to 40.1% (*treeadd*), with an average of 14.3%. This includes significant speedup for *mcf* and *vpr*, both of which have complex control flow, making them a difficult target for traditional prefetching approaches.

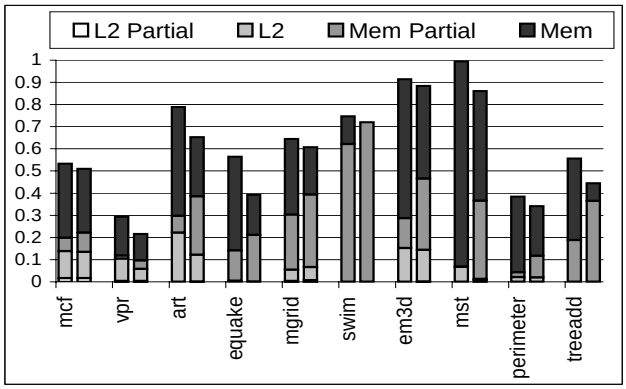

Figure 5. Breakdown of delinquent load memory accesses without Dynamic SP (left) and with basic Dynamic SP (right).

Figure 5 shows where in the memory hierarchy delinquent loads are satisfied for the baseline configuration (without SP, left) and the basic SP implementation for each benchmark (right). Delinquent load instances are divided into five categories, according to whether they were satisfied in L1 cache, L2 cache or main memory, and whether the latency was partially eliminated by a previous access (not necessarily a prefetch). Loads that hit in L1 cache are not shown; thus, a lower total bar height represents a higher L1 cache hit rate. Even in the baseline configuration, *vpr* hits in cache frequently, yet still achieves a significant speedup from SP. A large number of delinquent loads are covered by SP, but many prefetches are not timely. For example, in both *em3d* and *mst* most of the useful prefetched cache lines had not arrived before non-speculative access.

5. Aggressive P-slice Optimizations

The basic p-slice construction scheme outlined in Section 4 covers a good number of delinquent loads, but many prefetches have not arrived in cache before non-speculative access. This section presents advanced p-slice optimizations which achieve improved timeliness: (1) more aggressive trigger instruction placement, (2) targeting delinquent loads multiple loop iterations ahead via induction unrolling [16] and (3) chaining p-slices (which are similar in goal to chaining triggers [6]). These optimizations are only applied to p-slices when two instances of the delinquent load are present in the RIB. However, p-slices for which this is not true already provide good timeliness because their trigger comes at least 512 instructions (the size of the RIB) before the targeted delinquent load.

5.1. Improved Trigger Instruction Selection

The baseline construction scheme chooses the targeted delinquent load as the p-slice trigger instruction. This insures that all live-in producing instructions have been renamed (a requirement for correct p-slice execution), but spawns the p-slice significantly later than necessary. Alternatively, choosing the final live-in producing instruction results in an earlier trigger, and still retains correct execution.

This optimization improves delinquent load timeliness by spawning p-slices earlier, but also improves delinquent load coverage. With the baseline trigger choice, a delinquent load instance is only covered if the delinquent load was also executed on the previous loop iteration. This is because it is the delinquent load instance from the prior loop iteration which spawns the p-slice targeting the load instance in the following iteration. If the delinquent load is not executed in a particular loop iteration, no p-slice is spawned and the next instance of the delinquent load will not be covered. When employing this optimization, we find that the instruction selected as the trigger is typically executed much more often; one reason for this is that this algorithm often ends up choosing a loop induction variable update as a trigger.

The improved trigger instruction is identified by adding a second RIB analysis pass to the basic scheme. The live-in set from the first pass is retained, and instructions are analyzed in the same order until a live-in producing instruction is found, which is then selected as the trigger instruction.

Figure 6a shows the p-slice generated when this optimization is applied to the loop of Figure 2. Instruction I4 is chosen as the trigger instruction because neither I5 nor I6 produce a p-slice live-in value.

5.2. Induction Unrolling

For delinquent loads which have high average access latencies, the memory latency may not be entirely eliminated, even when spawning p-slices as early as possible. A technique known as Induction Unrolling (IU) [16] enables prefetching for delinquent loads multiple loop iterations ahead of the non-speculative thread by performing multiple loop induction variable updates in the p-slice.

Induction unrolling is applied by performing multiple RIB passes, where the nth pass identifies the instructions which must be executed n loop iterations before the targeted delinquent load. The first pass is the same as basic p-slice construction. Each further pass uses the live-in set generated on the previous pass as its initial live-in set. Following each pass, marked instructions are prepended at the head of the partial p-slice, and all marked bits are cleared.

Figure 6b shows a p-slice for the loop in Figure 2, targeting the delinquent loop four iterations ahead. For each additional loop iteration, a copy of the instruction `I4:add r2=r2+8` is included because it both consumes and produces r2 — r2 is a live-in value after the first pass.

5.3. Chaining P-slices

Chaining p-slices are able to spawn future instances of themselves, decoupling thread spawning from non-speculative thread execution. This allows delinquent loads far ahead of the non-speculativce thread to be targeted, but without significantly increasing executed speculative instructions, unlike induction unrolling. A chaining p-slice is

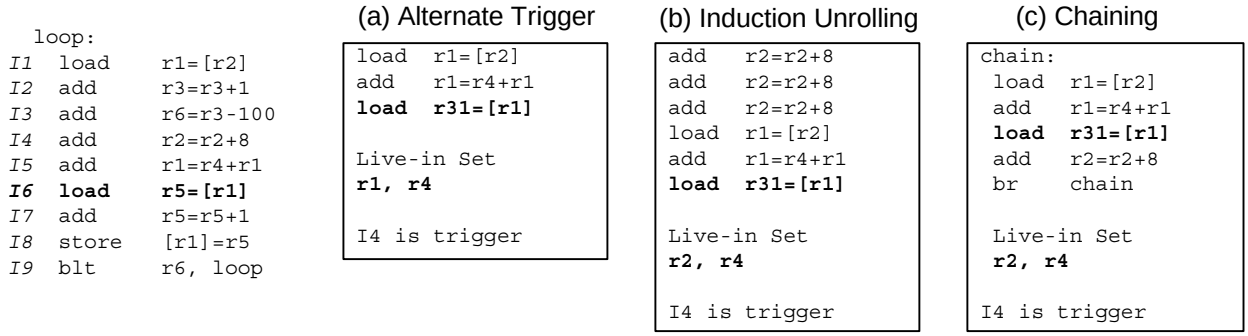

Figure 6. P-slices constructed when applying aggressive p-slice optimizations to code from loop in Figure 2.

implemented by repeating the p-slice in the same thread context (unlike the chaining triggers in [6], which spawn each iteration in a separate thread, an unnecessary feature in an out-of-order processor). Thus, contention for hardware thread contexts is significantly lower than with the baseline p-slices or induction unrolling, which spawn a p-slice for each iteration.

A chaining p-slice is simply a basic p-slice which is terminated by an unconditional branch to the beginning. When repeatedly executed (namely, "chained" together), accurate prefetches are generated for future instances of the load. When one p-slice chains another, the new p-slice is considered the youngest p-slice in the machine with regards to fetch policy, despite occupying the same thread context. This prevents chaining p-slices from monopolizing fetch bandwidth.

However, two dangers exist from the use of chaining p-slices. First, speculative threads must be prevented from executing so far ahead that useful data is evicted from the cache before being accessed. Second, speculative threads must not be allowed to chain indefinitely, but must be terminated when the non-speculative thread leaves the targeted code section (a loop body, for example). For other p-slices this occurs implicitly, as they are only spawned in response to execution of a trigger instruction. We introduce a modified version of the Outstanding Slice Counter (OSC) [6] to address these issues.

Outstanding Slice Counter The OSC is a structure which controls the execution of chaining p-slices in two ways — it limits execution distance ahead of the non-speculative thread (counted in loop iterations) and kills chaining p-slices when the non-speculative thread leaves the targeted section of code.

Because it encompasses an entire loop iteration, the range of non-speculative PCs analyzed in the RIB during p-slice construction is taken as the targeted program section. The effect of function calls within this code section, however, will distort this instruction range and must be filtered out. Thus, only the instructions within the RIB at the shallowest (least nested) call depth (at which no return instructions are present in the RIB) are counted. Also, the call depth between the delinquent load and the shallowest level is captured.

When a chaining p-slice is spawned, an OSC entry is initialized with a counter indicating the number of levels between the delinquent load and the shallowest level. The OSC monitors the non-speculative thread, incrementing this counter for each call executed and decrementing for each return. If the non-speculative thread commits an instruction when this counter is negative (indicating it is "below" the targeted instruction range), or outside this range when the counter is zero, the corresponding speculative thread is killed.

Additionally, when a load which produces a live-in value to a future p-slice instance dereferences an invalid address, the p-slice is killed. This does not include, for example, the final load in the p-slice if prefetch conversion has been applied.

Each OSC entry also tracks the number of loop iterations ahead of the non-speculative thread that its p-slice executes. When a chaining p-slice is spawned, this counter is initialized to the upper limit for that p-slice. This counter is decremented each time the p-slice chains and incremented each time the non-speculative thread completes a loop iteration. Speculative threads are not permitted to fetch when the counter is zero, giving the non-speculative thread a chance to catch up.

Identifying the completion of a loop iteration, however, is not a trivial matter. Execution of the targeted delinquent load itself cannot be used because the load may not be executed every loop iteration. Instead, we assume that the instruction with the smallest PC value (at the shallowest call level) which was analyzed during p-slice construction marks the top of the loop. Each time this instruction is executed in the non-speculative thread, it indicates the completion of a loop iteration. Section 5.4 presents an algorithm for determining the permitted runahead distance.

Capturing Chaining P-slices Chaining p-slices are created in multiple passes, but, unlike for induction unrolling, the number of passes is not known a priori. This is because the register live-in set is more difficult to compute, since a chaining p-slice must not only produce values needed by its prefetch, but also those needed for all future instantiations of the chaining p-slice; thus, our analysis mechanism must iteratively identify loop-carried dependences which affect future

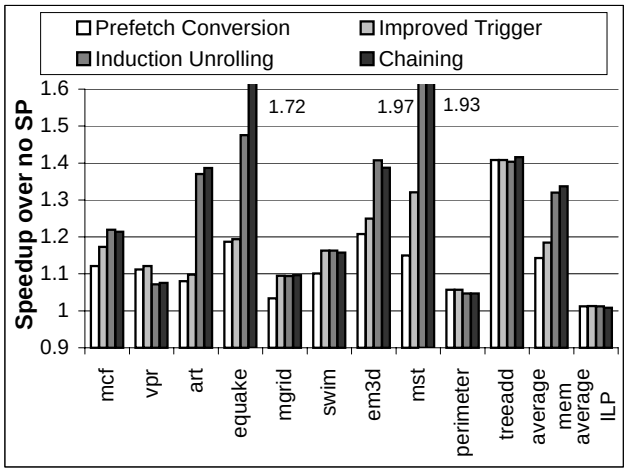

Figure 7. Speedup provided by Dynamic SP when using advanced optimizations over baseline processor without SP. Only the average performance for the five ILP limited benchmarks is shown.

delinquent loads. This is achieved by executing additional RIB passes until a pass in which no further instructions are included in the p-slice (during this final pass the set of instructions which kill the p-slice when dereferencing an invalid address is determined). Normally, this requires very few passes; however, heavily software pipelined loops may require several. Section 6.3 shows that dynamic SP performance is extremely tolerant of p-slice construction latency and that the cost of these passes is not high.

When using chaining p-slices, a slight complication arises in applying prefetch conversion — if the data loaded by the delinquent load in one p-slice instance is consumed by a future instance (such as in the traversal of a linked data structure), the delinquent load must not be converted. This situation is identified if the destination register of the delinquent load is in the p-slice's live-in set.

Figure 6c shows a chaining p-slice targeting the loop of Figure 2. All instructions producing values needed in future instances are added to the p-slice, including `I4:add r2=r2+8`, even though its result value is not necessary when targeting only a single loop iteration ahead. Instructions are ordered within a chaining p-slice in the same order as they occurred in the RIB. Thus, `I4:add r2=r2+8` is located after the delinquent load prefetch, despite coming before it in program order. The p-slice ends with an unconditional branch back to the beginning.

5.4. Induction Unrolling and Chaining Distance

Induction unrolling and chaining are applied progressively to p-slices which are effective, but generate untimely prefetches. When a delinquent load is first targeted, the p-slice is constructed without employing chaining or induction unrolling. When the p-slice is evaluated, if it satisfies the following conditions, a new p-slice which makes use of the more aggressive technique (induction unrolling or chaining) is created. The p-slice must be found to be effective (using the metric of Section 4.6), and more than 10% of its prefetches not having arrived before non-speculative access. Additionally, optimizations are only applied to p-slices in which fewer than 10% of memory accesses are to invalid addresses (which are a sign that the main program followed a different control path from what was expected), and which are executed 250 times in the previous 128K instructions, indicating that delinquent load instances typically occur close together.

The basic p-slice construction algorithm presented in Section 4.3 is slightly modified to now save the pattern of branch outcomes it encounters during p-slice construction into the corresponding SIT entry. When applying induction unrolling or chaining it is possible to use this information to target the same control flow as before by remaining in instruction-gathering mode until this control path is repeated.

When a p-slice employing either of these optimizations is evaluated as effective but with more than 10% of its prefetches untimely, it is modified to target an additional loop iteration ahead. When such a p-slice is found to be ineffective, it is modified to target one less loop iteration ahead of the non-speculative thread. If the p-slice employs induction unrolling, a new p-slice must be captured. Chaining p-slices require only modifying the maximum runahead distance. If the runahead distance reaches one loop iteration ahead, the optimization is abandoned and a basic p-slice is recaptured.

5.5. Performance of Aggressive Optimizations

Figure 7 shows performance gains from applying the three described optimizations. All configurations make use of prefetch conversion. Additionally, the chaining and unrolling configuration makes use of improved triggers.

Initially, when applying the improved triggers optimization to the recursive benchmarks, performance was severely degraded. This is because logical control flow in recursive programs differs significantly from loop based programs; moving the trigger instruction "earlier" in the RIB may actually place it at the end of the targeted recursive function, greatly increasing the number of useless prefetches generated. Thus, in the results above, the improved trigger optimization is not applied to p-slices identified as recursive (using the hint from Section 4.5).

For other benchmarks, improved triggers often provide a moderate performance gain, with an average speedup of 3.7% over only applying prefetch conversion. Induction unrolling and chaining p-slices achieve very large speedups for programs they can be applied to, especially those with predictable control flow. For instance, *mst* achieves a speedup of more than 90% for either technique. Overall, induction unrolling and chaining give average speedups of 31.9% and 33.7% respectively, compared to a processor which does not implement SP.

Performance improvement from applying SP to the five ILP limited benchmarks is fairly modest (average speedup 1%). However, this does show that the described mecha-

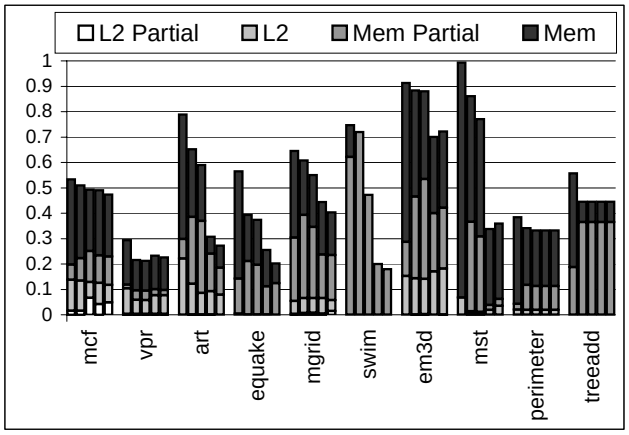

Figure 8. Breakdown of delinquent load memory accesses for different SP optimizations. Configurations shown are, left to right, no SP, prefetch conversion, improved triggers, induction unrolling and chaining.

nisms do prevent SP from being applied in cases for which it is not beneficial.

Figure 8 shows where in the memory hierarchy delinquent loads are satisfied when assuming the p-slice optimizations described above. Generally, these optimizations provide significant improvements in prefetch timeliness compared to those with only basic optimizations. For example, applying chaining or unrolling to *mst* results in more than 60% of delinquent loads hitting in L1 cache, compared to only 14% when using basic p-slices.

6. Further SP Evaluations

Results to this point have reflected a processor (1) having a large number of total hardware thread contexts, (2) executing only a single non-speculative thread, and (3) assuming ideal RIB analysis and thread spawning hardware. Next we evaluate the impact of varying these assumptions. Unless stated otherwise, results reflect only memory limited benchmarks.

6.1. Reduced Thread Contexts

Figure 9 shows the speedup from applying SP on a processor with two, four and eight total hardware thread contexts. Predictably, speedup drops as the number of contexts is reduced, but even a processor with only two thread contexts and no advanced optimizations achieves a 5.5% average speedup. Due to their more efficient use of thread contexts, chaining p-slices provide the largest speedup for any number of thread contexts, and perform nearly as well with only two total contexts as a processor with eight total thread contexts which implements only basic p-slices.

6.2. Multiple Non-speculative Threads

SP techniques are most useful when the system is executing a single non-speculative thread, because the processor is least able to hide memory latencies and idle contexts are in abundance. However, SP also provides benefit when a processor executes multiple non-speculative threads. This is true even if only one of the threads is memory limited, as accelerating this thread will reduce contention for execution resources, such as renaming registers, which would otherwise be occupied while the thread waits for a memory request to be resolved [19].

When multiple non-speculative threads execute, SP is applied to the worst behaving loads in the machine, regardless of which thread they belong to. Thus, if one thread is significantly memory limited, all SP resources may be allocated exclusively to target delinquent loads within that thread.

However, to insure some degree of sharing among available hardware thread contexts, all currently executing chaining p-slices are killed every 128K total committed instructions. This gives other threads an opportunity to spawn their own chaining p-slices if they have constructed any. Otherwise, the thread contexts are quickly reoccupied, with little overall performance loss. Because non-chaining p-slices are very short-lived, no special actions are required to prevent monopolization of resources by one thread. DLIT and SIT entries are allocated on a first-come, first-serve basis.

The fetch mechanism is changed slightly over a previously proposed SMT processor [20]. First, the *fetch family* with lowest total ICOUNT is selected for fetch, where a fetch family is the aggregate of a non-speculative thread and its speculative children. If speculative threads exist in the fetch family and are ready to fetch, the oldest is allowed to fetch. Otherwise, the non-speculative thread fetches.

Figure 10 shows performance when applying SP with more than one non-speculative thread executing. For each configuration, pairs of benchmarks are simulated for 300 million total instructions without employing SP, and the number of instructions executed by each of the threads is recorded. For simulations which enable SP, each benchmark executes as many instructions as it had in the baseline case, then goes idle. Thus, the program region simulated from each thread is the same between the baseline and the SP configurations. Speedup is given as the ratio of throughput (combined IPC) between the two cases.

Results are shown for eight total hardware contexts both for combining two memory limited benchmarks and combining one memory limited and one ILP limited benchmark. Chaining p-slices provide the largest speedup in both cases. This is especially true when combining two memory limited threads, where the more efficient utilization of thread contexts is critical. On average, chaining p-slices provide a speedup of 8.9% when executing a memory and ILP limited thread and a speedup of 17.6% when executing two memory limited threads. Thus we see, as indicated in previous research [23], SP has the effect of actually reducing the overall execution resources needed by the thread it targets, despite

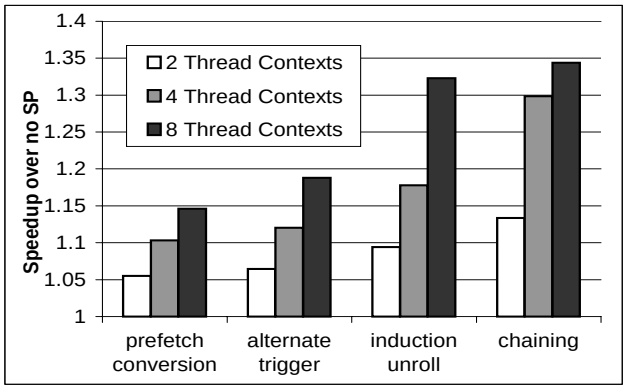

Figure 9. Speedup from Dynamic SP when executing a single non-speculative thread and reduced thread contexts.

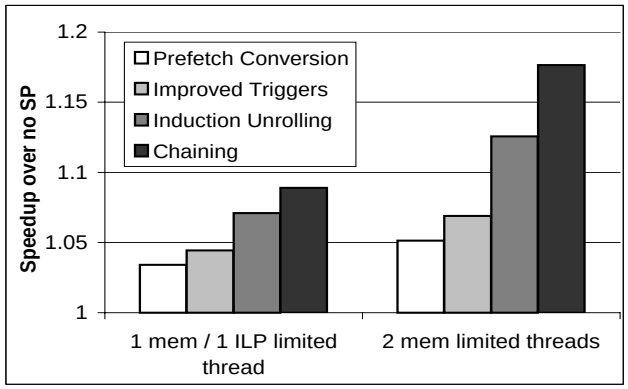

Figure 10. Speedup from Dynamic SP when executing multiple non- speculative threads.

the execution of speculative threads. This results in overall improvements in instruction throughput, even if not all executing threads are directly targeted by SP.

6.3. Relaxing Idealized Hardware Assumptions

To this point, results reflect three idealized SP hardware assumptions, (1) RIB analysis occurs instantaneously, (2) speculative instructions are fetched from a hardware slice cache, and (3) threads are spawned from the non-speculative thread without overhead to copy live-in values. Next, we evaluate the performance impact of more realistic implementations.

A realistic RIB is modeled by adding an analysis delay for each instruction. While busy, the RIB cannot accept new instructions from any thread. We model this delay as a very conservative 10 cycles per instruction in order to demonstrate the latency tolerant nature of p-slice construction. This delay is imposed for each time an instruction is analyzed during p-slice construction and improved trigger placement, and accounts for multiple pass analysis.

As described in Section 4.7, when using a software slice cache, p-slice instructions are stored in an OS controlled memory region, rather than a hardware structure. We model a 32KB software slice cache, which is divided into 256 byte (64 instruction) blocks. When constructed, a p-slice is written into a slice cache block so that, when brought into the ICache, it will be located as far away from the delinquent load it targets as possible. This eliminates ICache conflicts between the p-slice and the program code it is attempting to accelerate.

Two alternate schemes for p-slice spawning are modeled. *Move insert* moves values between threads by inserting explicit inter-thread move instructions into the non-speculative thread instruction stream immediately following the trigger instruction. Because these instructions are renamed before any p-slice instructions, the p-slice executes correctly. *Move stall* operates similarly, but doesn't inject the move instructions until the following cycle, and renames only move instructions on that cycle. Thus, move instructions are renamed one cycle after the trigger instruction is renamed, and the instruction originally following the trigger is renamed two cycles after the trigger.

Figure 11 shows the performance of each of these processor configurations. RIB construction latency plays a small performance role for all configurations, including those making use of chaining and induction unrolling. Even though these p-slices require multiple RIB analysis passes, p-slice construction is sufficiently rare that analysis delays do not impact performance. The alternate thread spawning mechanisms have a greater impact, but even when implementing move stall an 11.0% average speedup is still achieved using only basic p-slices. Chaining p-slices perform only slightly worse than when assuming ideal spawning hardware because the vast majority of p-slices are spawned from chaining p-slices, rather that from the non-speculative thread. In contrast, induction unrolling provides a 6.0% smaller speedup than it did with ideal hardware. The impact of the software slice cache is negligible, due to the small instruction footprint of the p-slices.

7. Conclusion

This paper presents Dynamic Speculative Precomputation, a runtime technique that identifies a program's delinquent loads, and generates precomputation slices to prefetch them. P-slices are generated using a back-end structure, the Retired Instruction Buffer, which captures a sequence of instructions as they are committed by the non-speculative thread. Even when only minimal p-slice optimization is performed, a speedup of 14% is achieved on a varied set of memory-limited benchmarks. More aggressive p-slice optimizations yield an average speedup of 33%. SP also provides improved throughput when applied to multiple non-speculative threads, even if only one of the threads benefits directly from SP. In sum, the use of Dynamic Speculative Precomputation enables available hardware thread contexts to target the worst behaving loads in a processor, improving

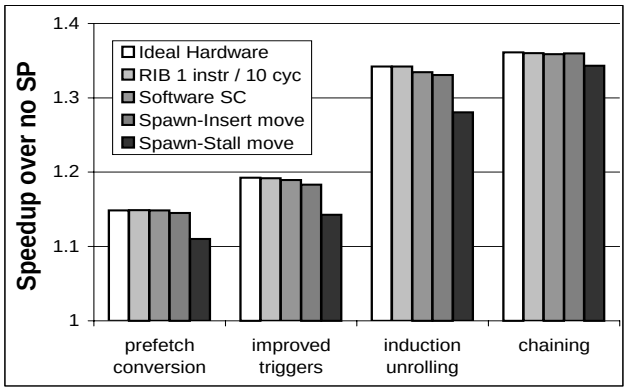

Figure 11. Speedup from Dynamic SP with realistic hardware assumptions.

both the targeted thread's latency as well as overall instruction throughput.

Acknowledgements

We would like to thank the referees of this paper for their extremely useful suggestions. Additionally, we would like to thank Robert Jenkins for data structure source code that was integrated into SMTSIM. This work was supported in part by a Focht-Powell fellowship, NSF grant CCR-9808697, and equipment donations from Compaq Computer Corporation.

References

[1] A. Aho, R. Sethi, and J. Ullman. Compilers: Principles, techniques and tools.

[2] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *28th Annual International Symposium on Computer Architecture*, pages 52–61, July 2001.

[3] M. Carlisle. Olden: Parallelizing programs with dynamic data structures on distributed-memory machines. In *PhD Thesis, Princeton University Department of Computer Science*, June 1996.

[4] T. Chen. An effective programmable prefetch engine for on-chip caches. In *28th International Symposium on Microarchitecture*, pages 237–242, Dec. 1995.

[5] Y. Chou and J. Shen. Instruction path coprocessors. In *27th Annual International Symposium on Computer Architecture*, pages 270–281, June 2000.

[6] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[7] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *31st International Symposium on Microarchitecture*, pages 173–181, Dec. 1998.

[8] Intel Corporation. Intel IA-64 architecture software developer's manual.

[9] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[10] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[11] C. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, pages 40–51, July 2001.

[12] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *15th International Conference on Supercomputing*, pages 321–334, June 2001.

[13] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. In *Journal of Parallel and Distributed Computing*, pages 87–106, June 1991.

[14] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *28th International Symposium on Microarchitecture*, pages 24–35, Dec. 1996.

[15] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[16] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Seventh International Symposium on High Performance Computer Architecture*, pages 37–48, Jan. 2001.

[17] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[18] D. Tullsen. Simulation and modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[19] D. Tullsen and J. Brown. Handling long-latency loads in a simultaneous multithreaded processor. In *34th International Symposium on Microarchitecture*, Dec. 2001.

[20] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[22] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 185–195, Jan. 2001.

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