Hardware Support for Prescient Instruction Prefetch

Tor M. Aamodt† Paul Chow†
†*Dept. of Elec. & Comp. Eng.*
University of Toronto
Toronto, Canada
{*aamodt,pc*}*@eecg.toronto.edu*

Per Hammarlund‡
‡*Desktop Platforms Group*
Intel Corporation
Hillsboro, OR 97124

Hong Wang § John P. Shen §
§*Microarchitecture Research Lab*
Intel Corporation
Santa Clara, CA 95054
{ *per.hammarlund, hong.wang, john.shen* }*@intel.com*

Abstract

This paper proposes and evaluates hardware mechanisms for supporting prescient instruction prefetch—an approach to improving single-threaded application performance by using helper threads to perform instruction prefetch. We demonstrate the need for enabling store-to-load communication and selective instruction execution when directly pre-executing future regions of an application that suffer I-cache misses. Two novel hardware mechanisms, safe-store *and* YAT-bits*, are introduced that help satisfy these requirements. This paper also proposes and evaluates* finite state machine recall*, a technique for limiting pre-execution to branches that are hard to predict by leveraging a counted I-prefetch mechanism. On a research Itanium® SMT processor with next line and streaming I-prefetch mechanisms that incurs latencies representative of next generation processors, prescient instruction prefetch can improve performance by an average of 10.0% to 22% on a set of SPEC 2000 benchmarks that suffer significant I-cache misses. Prescient instruction prefetch is found to be competitive against even the most aggressive research hardware instruction prefetch technique:* fetch directed instruction prefetch.

1. Introduction

Instruction supply may become a substantial bottleneck in future generation processors that have very long memory latencies and run application workloads with large instruction footprints such as database servers [20]. Prefetching is a well-known technique for improving the effectiveness of the cache hierarchy. This paper investigates the use of spare simultaneous multithreading (SMT) [27, 10, 13] thread resources for prefetching instructions and focuses on single-threaded applications that incur significant I-cache misses.

Even though SMT has been shown to be an effective way to boost throughput performance with limited impact on processor die area [11], the performance of single-threaded applications does not directly benefit from SMT and running such workloads may result in idle thread resources. Recently, a number of proposals have been put forth to use idle multithreading resources to improve single-threaded application performance by running small *helper threads* that reduce the latency impact of D-cache misses, and branch mispredictions that foil existing hardware mechanisms [9, 5, 6, 30, 28, 22, 18, 3, 17, 8, 7, 16, 15].

However, there has been little published work focused specifically on improving I-cache performance using such helper threads. A key challenge for instruction prefetch is to accurately predict control flow sufficiently in advance of the fetch unit to tolerate the latency of the memory hierarchy. The notion of prescient instruction prefetch [1] was first introduced as a technique that uses helper threads to improve single-threaded application performance by performing judicious and timely instruction prefetch.

The key contribution of this paper is the introduction of simple hardware mechanisms that provide support necessary to feasibly implement prescient instruction prefetch. When supplied with prescient instruction prefetch enabled application software, these hardware mechanisms can yield significant performance improvement while remaining complexity-efficient. While the potential benefit of prescient instruction prefetch was demonstrated previously via a limit study [1], the techniques presented in this paper show how to implement prescient instruction prefetch under realistic constraints. In addition, we evaluate the performance scalability of prescient instruction prefetch as memory latency scales up, and in comparison to aggressive hardware-only I-prefetch mechanisms. We demonstrate that prescient instruction prefetch may yield similar or better performance improvements than the most aggressive hardware-based instruction prefetch technique, by leveraging existing SMT resources without large amounts of additional specialized hardware.

The rest of this paper is organized as follows: Section 2 reviews the prescient instruction prefetch paradigm; Section

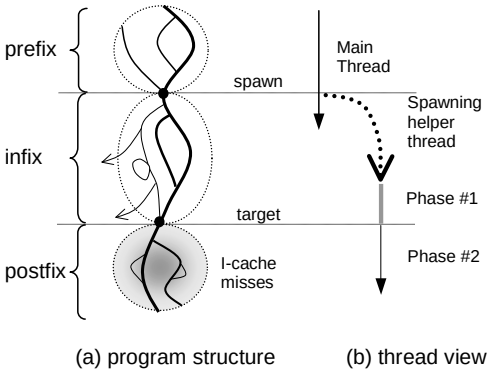


Figure 1. Prescient Instruction Prefetch

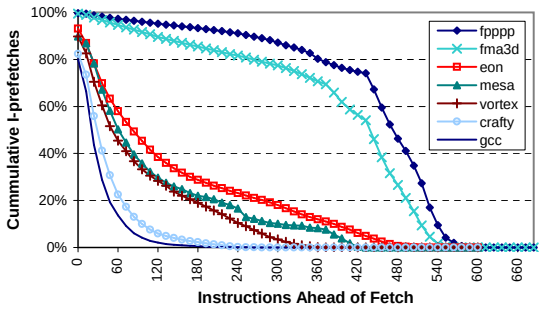


Figure 2. Prefetch distribution for a branch predictor guided I-prefetch mechanism

3 presents a straightforward technique for implementing prescient instruction prefetch called *direct pre-execution*, along with hardware mechanisms supporting it; Section 4 introduces an enhanced technique for implementing prescient instruction prefetch called *finite state machine recall*, which also leverages hardware support for a counted instruction prefetch operation and precomputes only hard to predict branches; Section 5 presents a performance evaluation of both techniques; Section 6 reviews related work; and Section 7 concludes.

2. Prescient instruction prefetch

Prescient instruction prefetch uses helper threads to perform instruction prefetch on behalf of the main thread. Figure 1 illustrates prescient instruction prefetch by highlighting a program fragment divided into three distinct control-flow regions by two points labeled the *spawn* and *target*. Of particular interest is the region following the target, called the postfix region that is known from profiling to suffer significant I-cache misses. Once a spawn-target pair is identified (at compile time, using profile feedback) a helper thread is generated and attached to the original program binary (i.e., the main thread). At runtime, when a spawn-point is encountered in the main thread, a helper thread can be spawned to begin execution in an idle thread context. The execution of the helper thread prefetches for anticipated I-cache misses along a control flow path after the target.

In the context of this work the term “prescient” carries two connotations: One, that helper threads are initiated in a timely and judicious manner, and two, that the instructions prefetched are actually useful as the helper thread closely follows the same path through the program that the main thread will follow when it reaches the target.

An efficient methodology for optimizing the selection of spawn-target pairs based on a rigorous statistical model of dynamic program execution has been described previously [1].

Prescient instruction prefetch is similar to *speculative multithreading* [24] insofar as it exploits both control-independence (also referred to as control-equivalence) to skip ahead in program execution, and bandwidth readily available for multiple fetch streams on multithreading processors. Furthermore, the use of precomputation for producing live-ins to a future task has been explored in *master/slave speculative parallelization* [29]. However, whereas these techniques primarily achieve speedup by exploiting thread-level parallelism available by reusing the architectural results of speculative computation, prescient instruction prefetch improves performance by reducing memory latency, in particular by reusing the microarchitectural side-effects (i.e., cache warm-up) of speculative precomputation threads.

Whereas *trace preconstruction* [14] employs a hardware-based breadth-first search of future control-flow to cope with weakly-biased future branches, prescient instruction prefetch uses precomputation to resolve which control-flow path to follow. Furthermore, as the precomputation frequently contains load instructions, prescient instruction prefetch often improves performance by prefetching data.

In contrast to branch predictor guided instruction prefetch mechanisms such as *fetch directed instruction prefetch* [21], prescient instruction prefetch uses a macroscopic view of control-flow gained by a profile-driven statistical analysis of program behavior. Such global analysis allows helper threads to accurately anticipate potential performance degrading code regions from a long distance ahead. For instance, the distance between spawn and target could be thousands of dynamic instructions.

To illustrate how prescient instruction prefetch may potentially improve performance beyond existing techniques, Figure 2 shows a plot of the cumulative distribution of I-prefetch distances for prefetches successful in reducing or eliminating the latency impact of an I-cache miss generated by a branch predictor guided instruction prefetch technique called *fetch directed instruction prefetch* [21] (abbreviated

to FDIP herein, see Table 1 for implementation details). A point (x, y) in Figure 2 indicates that y percent of instruction prefetches required to avoid an I-cache miss were issued x or more dynamic instructions ahead of the fetch unit. The effectiveness of this prefetch mechanism is a consequence of the branch predictor's ability to anticipate the path of the program through code regions not in the I-cache. As memory latency grows, the minimum prefetch distance required to tolerate that latency increases, and hence the fraction of timely prefetches tends to decrease. The same trend occurs for a fixed memory latency as more parallelism is exploited by the processor core.

By exploiting the global macroscopic control-flow correlation of spawn and target, prescient instruction prefetch threads can provide ample, scalable slack and achieve both timeliness and accuracy.

3. Direct pre-execution

We begin by investigating a straightforward implementation of prescient instruction prefetch we call *direct pre-execution*. During direct pre-execution, instructions from the main thread's postfix region are prefetched into the first-level I-cache by executing those same instructions on a spare SMT thread context. In this section requirements for effective direct pre-execution are examined and hardware mechanisms supporting these requirements are described.

3.1. Requirements for direct pre-execution

We begin by highlighting several significant challenges to obtaining effective direct pre-execution.

3.1.1. Constructing precomputation slices. For direct pre-execution to correctly resolve postfix branches, the outcome of the backward slice of each postfix branch must be accurately reproduced. This slice may contain computations from both the infix and postfix regions. Thus, as shown in Figure 1(b), direct pre-execution consists of two phases: The first phase, live-in precomputation, reproduces the effect of the code skipped over in the infix region that relates to the resolution of branches in the postfix region. We refer to these precomputation instructions as the infix slice. Similar to speculative precomputation [8, 16], infix slices for direct pre-execution helper threads could be encoded as additional instructions embedded in a program's binary image. In the second phase, the helper thread executes the remaining slice operations while executing the postfix region. As instructions in the infix region not related to any branch instruction in the postfix region are not duplicated in the infix slice they are neither fetched, nor executed by the helper thread which enables the helper thread to start executing through the postfix region before the main thread.

To construct infix slices, we consider dynamic slices similar to those studied in previous work [2, 30]. A dynamic slice can be decomposed into value, address, control and existence sub-slices [30]. In the present context, the value slice refers to those operations that compute values directly used to determine the outcomes of branches in the postfix region. The address slice includes those operations that compute load and store addresses in the value slice to resolve store-to-load dependencies between memory operations in the value slice. Likewise, the control slice resolves the control dependencies of the value and address slices. Lastly, the existence slice is the portion of the control slice that determines whether the target is reached.

Similar to Zilles and Sohi [30], in this study we examine variable infix-slices, extracted per dynamic spawn-target instance from the control-flow through the infix and postfix region, assuming perfect memory disambiguation, and containing the value, and address sub-slice, as well as the subset of the control sub-slices that generate instruction predicates for computation in the value and address slice.

3.1.2. Handling stores in precomputation threads. Helper threads run ahead of the main thread. In the postfix region helper threads execute the same code as the main thread and will inevitably encounter store instructions. These stores should not be executed the same way as in the main thread, but they cannot simply be ignored: On the one hand, allowing stores from a helper thread to commit architecturally could violate program correctness (e.g. a load from the main thread may end up erroneously depending on a later store committed by a helper thread). On the other hand, as shown in Section 5, store-to-load communication can indeed be part of the slice required to resolve control flow through the postfix region. To satisfy these conflicting requirements a novel mechanism called the *safe-store* is introduced in Section 3.2.2.

3.1.3. Handling ineffectual instructions. To ensure the effectiveness of prescient instruction prefetch, helper threads should avoid executing postfix instructions that are not related to the resolution of postfix branches. These instructions are called *ineffectual instructions*[1]. Although computing these instructions may potentially bring beneficial side effects such as data prefetching (if data addresses are computed correctly), they could waste processor resources. Worse yet, these ineffectual instructions may even corrupt the computation responsible for resolving postfix branches if they are not filtered out.

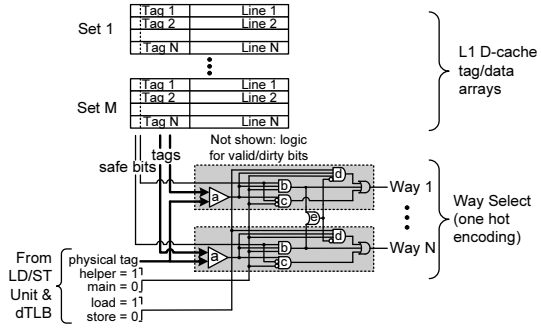
For example, Figure 3 shows four instructions from a postfix region where asterisks are used to indicate opera-

1 Sudaramoorthy et al. [25] use this term to describe those dynamic instructions that do not impact the results of program execution.

```
<spawn>
...
<target>
S1       cmp p3 = r1,r2 // r2 arbitrary
S2  (p3) add r1 = ...
S3*      cmp p5 = r3,r1
S4* (p5) br X
```

The code after <target> is the postfix region for a helper thread started when the main thread reaches <spawn>. p3 and p5 are predicate registers controlling whether S2 and S4 are executed. Mnemonics are "cmp" for compare, "add" for addition, "br" for branch (if predicate is true).

Figure 3. Postfix pre-execution and filtering



Way selection enabled if: (a) conventional tag match found; and: (b) line has safe-bit set and access is from helper thread; or (c) line does not have safe-bit set and access is from main thread; or (d) single matching line does not have its safe bit set, and access is from helper thread.

Figure 4. Safe-store way-selection logic

tions in the slice leading to the resolution of a branch. Statement S2 is found to be predicated "false" most of the time (so register `r1` is usually not updated at S2). Hence, the dynamic slice algorithm considered statements S1 and S2 ineffectual and excluded them from the slice, however, executing S1 and S2 may cause the helper thread to diverge from the main thread's future path: As `r2` contains an arbitrary value at S1, S2 may (incorrectly) be predicated "on" by the helper thread, resulting in an update to `r1`. This update to `r1` may result in S3 computing the wrong value for `p5` leading to control flow divergence after S4[2].

To filter out ineffectual computation a novel mechanism called the *YAT-bit* is introduced in Section 3.2.3 that would allow only S3 and S4 to execute in the preceding example.

3.2. Hardware support

In this section, we describe hardware mechanisms to support direct pre-execution.

3.2.1. Helper thread spawning. When the main thread commits a spawn-point a helper thread is spawned if there is an idle thread context. The helper thread first reads the set of live-in register and memory values required for infix-slice precomputation from the main thread. We consider a register or memory value to be *live-in* to a region if the value is used before being defined in that region. The infix slice is responsible for computing the set of live-in register and memory values relevant to branch resolution in the postfix region. Once infix slice precomputation completes, the helper thread jumps to the target and starts executing the main thread's code in the postfix region.

3.2.2. Safe-stores. To honor store-to-load dependencies during helper-thread execution without affecting program correctness we propose *safe-stores*—a simple microarchitectural mechanism for supporting a type of speculative store. Store instructions encountered during helper thread execution are executed as safe-stores. A safe-store is written into the first-level D-cache, but cannot be read by any other non-speculative thread. This selectivity is achieved by extending the cache tags with a safe-bit to indicate whether or not the line was modified by a helper thread.

To implement safe-stores the tag match logic is extended to include comparison of the safe-bit as shown in Figure 4. The safe-bit is initially cleared for cache lines brought in by the main thread but is set when a line is modified by a helper thread. The augmented tag matching logic guarantees that a load from the main thread will never consume data produced by a store in a helper thread, thus ensuring program correctness. However, upon a D-cache hit, a load from the helper thread is allowed to speculatively consume data stored by the main thread or another helper thread.

For example, a safe-store modified line may be evicted from the D-cache by a store from the main thread with the same tag, which is then followed by a dependent load from the helper thread that hits in the cache and receives the data written by the main thread (we call this condition a *safe-store eviction*). Hence it cannot be guaranteed that the data consumed was produced by the safe-store upon which it logically depends. Similarly, when multiple helper threads operate concurrently, they may (erroneously) read values stored by each other (we call this *safe-store aliasing*). If multiple helper threads often write to the same lines concurrently, performance may improve if safe-store modified lines are further distinguished by adding a couple of helper thread instance specific tag bits in addition to the safe-bit.

After a helper thread exits, a safe-store modified line may remain in the cache long enough that a subsequent helper

2 The problem illustrated is not unique to predication: Executing ineffectual stores can also lead to control flow divergence.

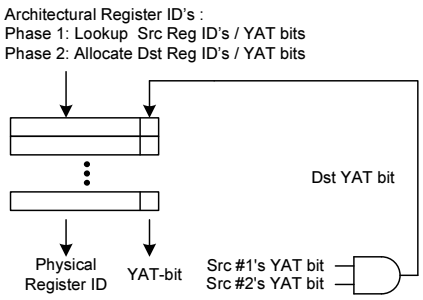


Figure 5. YAT-bit rename logic

thread reads a stale value from it. One way of reducing the chance of this happening is to mark loads that read helper thread live-in values so they invalidate matching safe-store modified lines. As shown in Section 5, we found the safe-store mechanism can frequently enforce the correct store-to-load dependency within a helper thread.

If the D-cache uses a write-through policy, the line modified by a safe-store is prevented from being written into the next level cache hierarchy. If the D-cache instead uses a write-back policy, a line with safe-bit set behaves as an invalid line when it participates in a coherence transaction. Furthermore, when using a write-back policy, a write-back of non-speculative state must be generated when a helper thread writes to a line modified by the main thread. When a safe-store modified line is evicted it is not written back to the next level of the memory hierarchy. Safe-store modified lines may reduce performance by increasing the number of D-cache misses incurred by the main thread, and by triggering additional write-backs.

3.2.3. YAT-bits. To avoid executing ineffectual instructions, we introduce the *YAT-bit*[3], a simple extension to existing register renaming hardware. Register renaming is used in out-of-order processors to eliminate false dependencies resulting from reuse of architectural registers. Similarly, a form of register renaming is used to reduce function call overhead, via register windowing, and software pipelining overhead, via register rotation, in the Itanium® architecture [12]. In the proposed extension each architectural register is associated with an extra bit, called the YAT-bit that indicates whether the corresponding physical register likely contains the same value as the main thread will when it reaches the point where the helper thread is currently executing. Upon spawning a helper thread all YAT-bits for the helper thread are cleared. After copying live-in registers from the main thread, the YAT-bit for each live-in register is set to indicate the content is meaningful, then precomputation of the infix slice begins. YAT-bits propagate as follows: When an instruction enters the rename stage, the YAT-bits of each source operand are looked up. If all source YAT-bits are valid, the destination register YAT-bit(s) will also be marked valid, and the instruction will execute normally. However, if any source register YAT-bits are invalid, the instruction's destination register YAT-bits will be invalidated, and the instruction will be treated as a "no-op" consuming no execution resources down the rest of the pipeline (see Figure 5)[4].

The hardware support used to implement the YAT-bit mechanism is reminiscent of that used to support runahead execution [19]. In runahead execution the register-renamer tracks so-called invalid registers—those registers dependent upon a value that would be produced by a load that misses in the L2-cache which is instead ignored by the runahead thread. YAT-bit operation fundamentally differs in that an invalid YAT-bit indicates a register is dependent upon a live-in that was not copied from the main thread. An invalid YAT-bit therefore indicates that the value is both inaccurate and likely irrelevant when resolving postfix branches.

It is interesting to note a few concerns specific to architectures supporting predication and/or register windows. If an instruction is predicated "false", the destination registers are not updated and hence the associated YAT-bits should remain unchanged. If the predicate register YAT-bit is invalid the correct value of the predicate is unknown and hence it may be unclear whether the destination YAT-bits should change. In practice we found it was effective to simply let the destination register YAT-bits remain unchanged.

On architectures using register windowing a register stack engine typically spills and fills the values in registers to/from memory automatically. If this mechanism is triggered during helper thread execution, saving and restoring YAT-bits can improve prefetch accuracy and therefore may improve performance.

3.2.4. Helper thread kill. To ensure helper threads do not run behind the main thread or run astray, a helper thread is terminated when the main thread catches it (in practice–when both threads having the same next fetch address), or when a maximum number of postfix operations (predetermined during spawn-target pair selection) have executed.

4. Exploiting predictability

In this section we examine a technique for reducing the amount of precomputation required for achieving prescient instruction prefetch. Typically, many postfix branches are either strongly biased, thus predictable statically, or dynamically predictable with high confidence using a mod-

3 YAT-bit stands for "yes-a-thing bit" in analogy to the Itanium® NAT-bit ("not-a-thing bit") used for exception deferral.

4 The infix slice may leave stale temporary register values, clearing their YAT-bits before postfix precomputation improves prefetch accuracy.

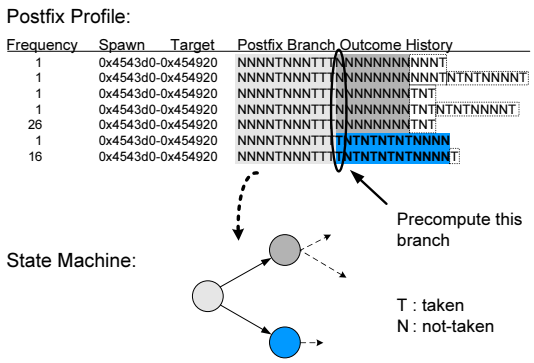


Figure 6. Postfix profile & FSM construction

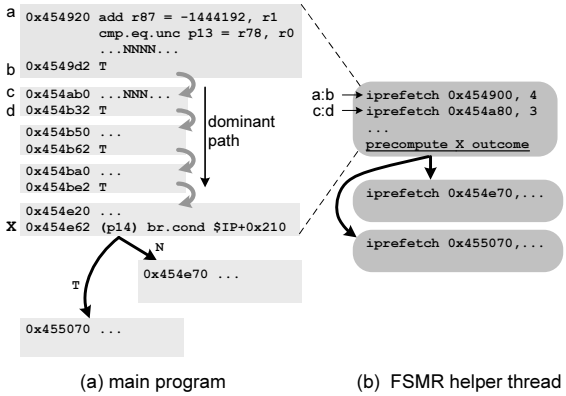


Figure 7. FSMR example showing use of counted prefetch "iprefetch x,n" that triggers a hardware counted prefetch engine to prefetch n cache lines starting from address x. Underlined statement in part (b) is a t-slice.

ern hardware branch predictor. Hence, using branch prediction can reduce the amount of precomputation required to ensure accurate instruction prefetch.

An interesting alternative to the direct pre-execution technique uses a counted I-prefetch operation encoded in the processor's instruction set to decouple I-prefetch from precomputation. The precomputation slices used by this technique, called transition slices (t-slices), merely resolve weakly-biased branches. A counted I-prefetch operation provides the processor with a starting address, and number of lines to prefetch as a hint (i.e., the action may be ignored without affecting program correctness). A hardware prefetch engine queues these instruction prefetch requests, and issues them to the memory system at a later time. The counted I-prefetch operations we envision resemble the I-prefetch hints encoded in the `br.many` and the `br.few` instructions of the Itanium® architecture, except the hint would also express the number of lines to be prefetched.

The following section describes how to implement prescient instruction prefetch helper threads that use counted instruction prefetches.

4.1. Finite state machine recall

The control flow paths through a postfix region alternate between path segments containing strongly-biased branches (which may be summarized with a sequence of counted prefetches) separated by weakly-biased branches. For example, the top half of Figure 6 shows profile information (from 186.crafty) for a single spawn-target pair's postfix region illustrating that the target is followed by 10 strongly-biased branches ending at a weakly-biased branch. These dominant path segments can be summarized as states in a finite state machine in which transitions represent the resolution of weakly-biased branches (bottom half of Figure 6).

In *finite state machine recall* (FSMR) a t-slice is constructed for each of these weakly-biased branches. If some of these weakly-biased branch could be predicted accurately via some hardware mechanism then an instruction that queries the predictor could be used instead of a t-slice.

Figure 7 shows an FSMR helper thread example. In Figure 7(a) the block starting at 'a' and ending at 'b' is merged together with the block starting at 'c' and ending at 'd' into the first state in the FSMR helper thread because the branch at 'b' is strongly biased to be taken. Figure 7(b) shows a fragment of the finite state machine encoding the region shown in part (a). Note that only non-contiguous blocks of code require separate counted prefetch operations.

In contrast to the infix slice used in direct pre-execution, the code in a t-slice may include copies of computation from both the infix and the postfix region. If the outcome generated by a t-slice does not match one of the transitions in the state machine, the helper thread exits. Note FSMR helper threads may benefit from using safe-stores to enable store-to-load communication.

Since the instructions prefetched by any state are independent of the t-slice outcome, we scheduled counted prefetches first to initiate prefetches for code in the main thread before precomputing a t-slice to determine the next state transition. Counted prefetch memory accesses continue in parallel with the execution of the subsequent t-slices. A more aggressive implementation might spawn t-slices speculatively based upon less reliable control flow prediction techniques (cf. chaining speculative precomputation [8] where essentially the prediction is that a backward branch is taken), or reuse common sub-computation from one t-slice to the next. In the extreme case, provided there are a sufficient number of idle thread contexts and fetch bandwidth, from one single state multiple t-slices can be ea-

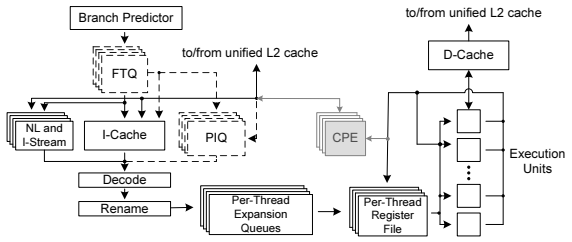


Figure 8. Processor model (FTQ and PIQ for FDIP only, CPE for FSMR only)

gerly spawned for multiple successor states.

Our simulator models a modest per-thread context decoupled counted prefetch engine. When a counted prefetch reaches the execute stage of the pipeline the counted prefetch is inserted to the back of the queue for the associated thread context. Prefetch requests are issued from this queue upon available bandwidth to the L2 cache.

4.2. FSM and t-slice construction

To construct the finite state machine and t-slices, the profile guided spawn-target pair selection algorithm [1] is used to supply spawn-target pairs. Spawn-target pairs for FSMR are selected assuming the number of instructions prefetched per cycle by a helper thread allows it to keep the same pace as the main thread. For each pair, postfix path profiling is applied to the postfix region and the results of this postfix path-profile are then used to construct a separate FSM for each static spawn-target pair as described in Section 4.1. The t-slices are extracted similarly to dynamic slices. The data in Table 2 shows that these finite state machines are typically small. For example, the number of t-slice transitions per dynamic helper thread instance ranges from 0.39 to 6.8 (see Row 15 in Table 2), even though the number of branches encountered varies from 1.53 to 47 (Row 8).

5. Performance evaluation

In this section we evaluate the performance of the proposed techniques.

5.1. Processor model

We model a research Itanium® SMT processor with four hardware thread contexts based upon SMTSIM [27] (see Figure 8). The processor configurations listed in Table 1 are modeled. To gauge performance scalability with respect to future memory latencies, we evaluate two memory hierarchies, labeled 2x and 4x, representative of 2GHz and 4GHz Itanium® processors, respectively.

The baseline configuration includes the following I-prefetch support: Upon a demand miss, a request is generated for the next sequential cache line. In addition, Itanium® instruction prefetch hints (e.g., `br.many`) are used to trigger a streaming prefetch address generator that can send additional requests up to four cache lines ahead of the fetch address. If an instruction prefetch request encounters a second-level iTLB miss that requires accessing the page table in main memory, the request is ignored without generating an iTLB update. Instruction cache miss requests are prioritized in the memory hierarchy. Demand fetches have highest priority, followed by next line and streaming prefetches, followed by counted prefetch requests. For FDIP configurations the Itanium® stream I-prefetch mechanism is not used.

In our processor model, helper threads contend for fetch and issue bandwidth with the main thread, but have lower priority. For I-cache access, each cycle the fetch unit can access two cache lines. If the main thread is currently waiting for an outstanding I-cache miss, up to two active helper threads are selected (in round-robin order) to access the I-cache. Otherwise, one access from the main thread, and one from an active helper thread are allowed.

During instruction issue, ready instructions are selected from the main thread provided there is availability of function units. Any left over issue bandwidth is available for helper thread instructions.

5.2. Prescient instruction prefetch models

We model both direct pre-execution and FSMR mechanisms for prescient instruction prefetch.

5.2.1. Direct pre-execution. For direct pre-execution we model both an idealized version that assumes *perfect live-in prediction* (PLP) and a more realistic version that uses dynamic infix slices to perform precomputation, called *variable slice* (VS). PLP models the ideal case where helper threads begin executing at the target as soon as the spawn is encountered by the main thread and initially see the same architectural state that the main thread will be in when it also reaches the target. As in the earlier limit study [1], for PLP an infinite store buffer model is assumed (rather than the safe-store mechanism), and a helper thread is triggered when the main thread commits a spawn-point (if no contexts are available the spawn-point is ignored). The helper thread begins fetching from the target the following cycle and runs the maximum number of instructions, or until it is caught by the main thread, at which point it exits. For the VS model, after a helper thread is spawned it first pre-executes the infix slice and only afterwards jumps to the target-point to start executing (and thereby effectively prefetching) instructions in the postfix region. For VS safe-stores are modeled with a 2-bit tag that reduces interference between con-

Threading	SMT processor with 4 thread contexts	Issue		priority main thread, helper threads round-robin
Pipelining	In-order: 8-stage (2x), or 14-stage (4x) pipeline			
Fetch	1 line per thread, 2 bundles from 1 thread, or 1 bundle from 2 threads priority main thread, helper threads ICOUNT [26]	Function Units		4 int., 2 FP, 3 br., 2 mem. units
		Register Files (per thread)		128 general purpose, 128 FP, 64 predicate, 8 branch, 128 control regs.
Instruction Prefetch	maximum 60 outstanding I-cache requests of any type (a) next line prefetch on demand miss (b) I-stream prefetch, max. 4 lines ahead of fetch (c) counted i-prefetch engine (CPE) - allocates into L1 I-cache 64-entry FIFO per thread, max. 2 issued per cycle (d) fetch-directed instruction prefetch [21]: enqueue cache probe filter (enqueue-CPF) dedicated port for tag lookup 16- entry fetch target queue (FTQ), FIFO prefetch instruction queue (PIQ)	Cache	2x	L1 (separate I&D): 16 KB, 4-way (each) L2 256KB, 4-way 14-cycle L3 3MB, 12-way 30-cycle 64-byte lines, D-cache uses a write-through, no write allocate policy
			4x	latencies changed to: L2 28-cycle L3 60-cycle
Branch Predictor	2k-entry GSHARE, 64-entry RAS with mispec. repair [23] 256-entry 4-way associative fetch target buffer (FTB)	Memory		2x: 230-cyc. TLB miss 30 cyc. 4x: 450-cyc. TLB miss 60 cyc.

Table 1. Processor Configurations

row	description	benchmark abbrev.	145.fpppp fpppp	176.gcc gcc	177.mesa mesa	186.crafty crafty	191.fma3d fma3d	252.eon eon	255.vortex vortex
1	Static spawn-target pairs		62	3545	34	166	34	152	1348
2	Dynamic spawn-target pairs		18483	16105	10515	32092	16973	20786	22035
3	Avg. spawn-target distance (insts.)		638	645	1161	553	835	690	1028
4	Avg. PLP postfix region size (insts.) (2x/4x)		150 / 146	85 / 92	239 / 181	98 / 104	103 / 64	136 / 130	124 / 122
	Direct pre-execution (4x)								
5	Avg. # live-ins (register / memory)		0.88 / 1.07	4.5 / 13.7	8.9 / 14	8.0 / 12.9	3.4 / 5.3	3.9 / 7.4	5.1 / 12.1
6	Avg. # infix slice instructions		4.9	9.1	26	35	20	27	26.1
7	Avg. # infix store-to-load pairs		0.158	0.94	1.19	1.52	0.74	2.2	1.67
8	Avg. # postfix branches (static / dynamic)		2.6 / 1.53	25 / 29	22 / 18.8	31 / 13	1.09 / 2.4	17.0 / 8.4	26 / 47
9	Fraction of YAT-ed postfix inst.		8%	68%	54%	74%	24%	55%	76%
10	Avg. # YAT-ed loads / stores		3.1 / 0.61	16.1 / 2.1	19 / 2.2	15.0 / 3.0	7.1 / 0.73	11.2 / 2.4	18.1 / 4.3
11	Avg. # threads with a safe-store	safe-bit	14 / 9	19.7 / 113	100 / 88	37 / 96	0 / 0	13.4 / 11.7	78 / 173
12	eviction / alias (per 1000)	+ 2-bit tag	0.18 / 0.31	2.9 / 2.1	0.54 / 0.33	18.3 / 12.2	0 / 0	0 / 0.09	59 / 10
	FSMR (4x)								
13	Avg. # total t-slice instructions		7.4	94	25	140	38	79	14
14	Avg. # sliced postfix branches (static)		0.72	13.3	12.1	30	0.35	9.9	1.20
15	Avg. # precomputed postfix branches (dyn.)		0.50	6.8	1.91	6.4	0.39	2.1	0.53

Table 2. Helper Thread Statistics

current helper threads (increasing D-cache misses incurred by the main thread). If a helper thread load, known to read a live-in value, hits a safe-store modified line it invalidates the line and triggers a D-cache fill of non-speculative state, or reads the non-speculative value if it is already in the D-cache. When the main thread catches up with a helper thread (has matching next instruction fetch address), that helper thread stops fetching instructions. Once the helper thread's instructions drain from the pipeline the thread context is available to run other helper threads.

The PLP model serves to gauge the upper bound for the direct pre-execution approach, while the VS model serves to evaluate the impact of live-in precomputation overhead, the performance penalty of safe-stores due to increased misses in the D-cache; the ability of YAT-bit filtering to reduce resource contention, and prevent ineffectual instructions from corrupting postfix precomputation; and the relatively less D-cache data prefetching side effects compared to PLP as fewer loads are executed by helper threads.

5.2.2. FSMR. For FSMR, counted prefetch requests are queued into a FIFO buffer, and eventually issued to the memory system (if no buffers are available, new requests are discarded). Similar to direct pre-execution, counted prefetches issued via FSMR are allocated right into the I-cache rather than a prefetch buffer. Safe-stores are modeled as with VS, but without additional tag bits.

5.3. Simulation methodology and workloads

We selected six benchmarks from SPEC2000 and one from SPEC95 that incur significant instruction cache misses on the baseline processor model. For spawn-target pair selection we profiled branch frequencies and I-cache miss behavior by running the programs to completion. To evaluate performance we collect data for 5 million instructions starting after warming up the cache hierarchy while fast-forwarding past the first billion instructions assuming no prescient instruction prefetch. We found that longer simula-

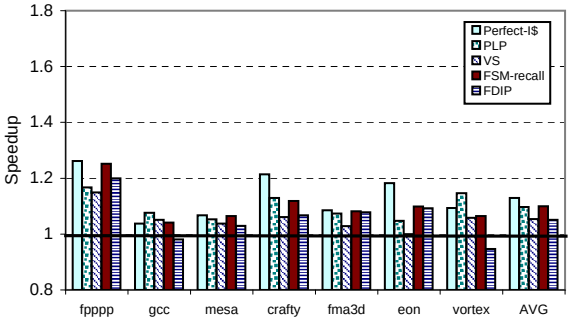


Figure 9. Performance on 2x configuration

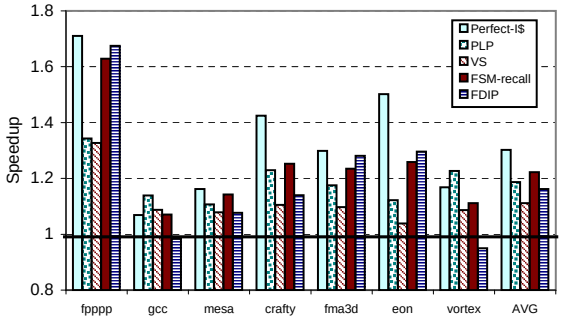


Figure 10. Performance on 4x configuration

tions did not exhibit significant IPC variation for the workloads we employ. For FSMR, postfix path profiling was also performed over this simulation window.

Our spawn-target generation algorithm selected between 34 and 3545 spawn-target pairs per benchmark as shown in Row 1 of Table 2, which also quantifies several characteristics of the slices executed for the VS and FSMR configurations. The small number of static spawn-target pairs implies a small instruction storage requirement for prescient instruction prefetch helper threads. Hence, we model single-cycle access to infix slice and t-slice instructions, and assume they can be stored in a small on-chip buffer.

Comparing the average spawn-target distances (i.e., the average number of instructions executed by the main thread between spawn, and target) in Row 3 of Table 2 with the data in Figure 2 for FDIP highlights the fact that prescient instruction prefetch has the potential to provide significantly larger slack. For the benchmarks we study, most I-cache misses are satisfied from the low latency L2 cache so they do not show substantial benefit from this larger slack (i.e., applications with larger instruction footprints than those we study may obtain larger benefits from prescient instruction prefetch).

The number of infix slice instructions averaged between 4.9 and 35 (Row 6), which is comparable in size to those required for speculative precomputation of load addresses for data prefetching [8, 18], and implies prescient instruction prefetch can start executing through the postfix region long before the main thread will reach the target. On average the number of store-to-load dependencies within an infix slice was less than three (Row 7). Furthermore, very few safe-store evictions, or safe-store alias errors occur for the 4-way set associative D-cache we modeled. For FSMR there were almost no safe-store evictions, and less than 1.54% of helper threads encountered a safe-store alias error. For VS less than 5.9% of all helper threads encounter a safe-store eviction, and fewer than 1.22% of helper thread instances encounter a safe-store alias error when using a 2-bit tag in addition to the safe-bit mechanism (Row 12). (Row 11 shows the increase in evictions and aliasing that occur for VS using safe-bits without additional tag bits.)

The average postfix region sizes were fairly large, ranging from 85 to 239 instructions (Row 4). The number of branches in a postfix region can be quantified both in terms of the number of static branches it contains, and by the number of dynamic branch instances executed each time a helper thread executes through it. Averages for both are given in Row 8, and indicate most benchmarks are control flow intensive in addition to having poor I-cache locality.

For FSMR the total number of slice instructions per helper thread instance tends to be larger than for VS, however, FSMR t-slices may also include operations from the postfix region. Furthermore, slice operations from one branch precomputation may intersect the set of slice operations from a subsequent branch. For FSMR the average number of branch outcomes precomputed per helper thread instance is between 0.5 and 6.8 (Row 15), which is significantly lower than the number of branches encountered in the postfix region per helper thread instance.

5.4. Performance evaluation

Figures 9 and 10 display the performance for the 2x and 4x memory hierarchy configurations described more fully in Table 1. Each figure shows five bars per benchmark. The first bar from the left, “Perfect I$”, represents the speedup obtained if all instruction accesses hit in the I-cache. The next few bars, labeled “PLP”, “VS”, and “FSMR” represent the speedup of the various models of prescient instruction prefetch described earlier. The bar labeled “FDIP” shows the performance of our implementation of Reinman *et al.*’s fetch directed instruction prefetch mechanism using the configuration described in Table 1.

On the 2x memory configuration, PLP obtains a harmonic mean speedup of 9.7%, VS obtains a 5.3% speedup, FSMR obtains a 10.0% speedup, and FDIP obtains a 5.1% speedup. Similarly, on the 4x memory configuration PLP obtains a harmonic mean speedup of 18.7%, VS obtains a

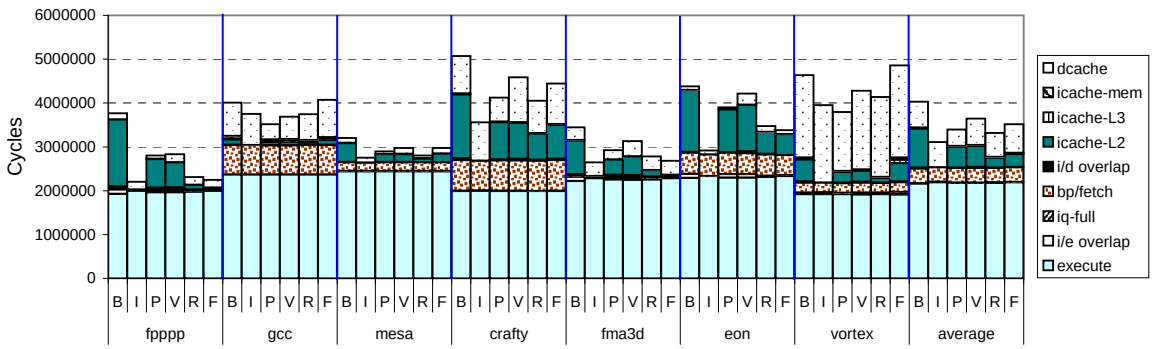


Figure 11. Classification of execution cycles (4x memory configuration). B=baseline, I=Perfect-I$, P=PLP, V=VS, R=FSMR, and F=FDIP.

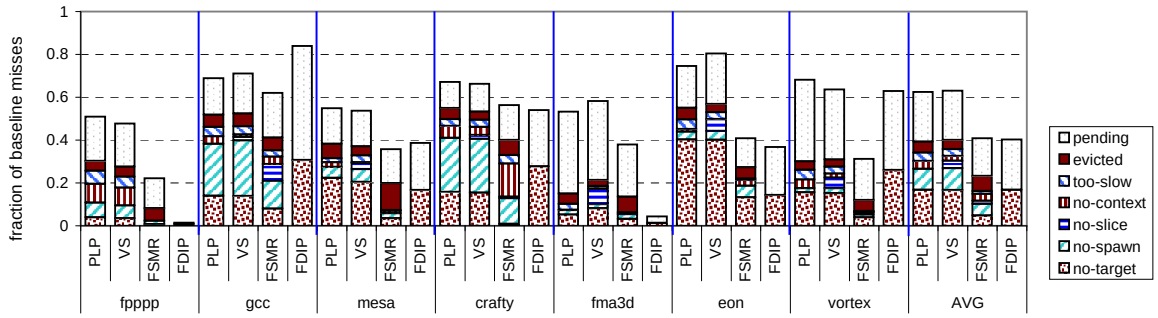


Figure 12. Classification of remaining I-cache misses (4x memory configuration)

11.1% speedup, FSMR obtains a 22% speedup, and FDIP obtains a 16.2% speedup. The data shows that prescient instruction prefetch can obtain significant speedups over our baseline model and in particular, FSMR is competitive with, and often outperforms fetch directed instruction prefetch.

To provide insight into this result Figure 11 provides a breakdown of how execution cycles are spent, and Figure 12 analyzes the cause of remaining I-cache misses.

Figure 11 shows the total execution time broken down into nine categories: *d-cache* represents cycles lost to D-cache misses; *icache-L2*, *icache-L3* and *icache-mem* represent cycles spent waiting for instructions to return from the associated location in the cache hierarchy; *i/d overlapped* and *i/e overlap* represents I-cache stall cycles overlapped with D-cache and execution cycles; *bp/fetch* represents cycles lost due to branch mispredictions, misfetched branch targets, taken branches, and crossing cache line boundaries; *iq-full* represents cycles lost due to the need to refill the front-end of the pipeline after the expansion queue overflows; finally, *execute* represents the number of cycles required given available function units and the schedule produced by the compiler.

On the baseline configuration each benchmark spends an average of about 22% of total execution time waiting for instruction fetch requests to return from L2, which drops to 14% for PLP, 13% for VS, 7% for FSMR, and 8% for FDIP. Hence, all of the mechanisms yield performance gains by reducing stall cycles due to I-cache misses. However, the prescient instruction prefetch techniques also reduce execution time by prefetching some data. For example FSMR reduced D-cache stall cycles on gcc, mesa, crafty and vortex. Most notably, 177.mesa has a 42% reduction in D-cache stall cycles for FSMR, translating into a 1.6% speedup, and 176.gcc has a 33% reduction in D-cache stall cycles for VS, translating into a 6.3% speedup. Even larger gains are seen for PLP because all loads in the postfix region are executed and helper thread stores do not write to the D-cache. On the other hand, some benchmarks see an increase in data cache stall cycles for the VS and FSMR configurations. This increase is due to safe-stores increasing the number of D-cache misses seen by the main thread as described in Section 3.2.2. For FDIP, the D-cache stall cycle component increases by an average of 10.9% and 11.8% on the 2x and 4x memory configurations due to increased contention with I-prefetches for bandwidth in the cache hierarchy.

The cause of the remaining I-cache misses are analyzed

in Figure 12. For FDIP, the bottom portion of each bar represents complete I-cache misses, and the top portion represents partially prefetched misses. For prescient instruction prefetch, I-cache misses are classified in more detail by examining the history of events leading up to them. If no target was encountered recently enough that the corresponding postfix region would have included the instruction that missed, the classification is *no target*. If a target was found, but no preceding spawn could have triggered a helper thread to start at this target, the classification is *no spawn*. For a given implementation technique (e.g., FSMR), "no-spawn" and "no-target" I-cache misses can only be reduced by selecting spawn-target pairs that yield higher coverage. On average about 43% and 25% of remaining I-cache misses fall in these categories for VS, and FSMR respectively.

The FSMR mechanism has significantly fewer "no target" and "no spawn" misses than VS because the spawn-target selection for FSMR was performed with the expectation that FSMR helper threads can make faster progress through the postfix region than VS because fewer branches need to be precomputed. The shorter spawn-target distance provides more flexibility for spawn-target selection and hence higher coverage. Thus, faster helper threads may lead to better performance because they enable more flexible spawn-target selection.

If both a spawn and target are found, but an idle thread context did not exist at the time the spawn-point was committed by the main thread, the classification is *no context* (4% and 8% of I-cache misses on average for VS and FSMR, respectively). If a helper thread was successfully spawned there are three remaining possibilities: First, that the helper thread ran *too slow* so the main thread caught it before it could prefetch the instruction (5% and 3%); second, that the helper thread ran so far ahead that the instruction was brought into the cache but then *evicted* (7% and 17%). The number of "evicted" misses increases for FSMR relative to VS on 177.mesa. So while faster helper threads may help spawn-target selection, it may be helpful to eliminate such I-cache misses by regulating the subsequent issuing of counted prefetches. Finally, the access may already be *pending* (37% and 43%)[5].

FSMR improves performance more than FDIP for all benchmarks on the 2x configuration but only for gcc, mesa, crafty, and vortex on the 4x memory configuration. Comparing this data with that in Figure 2 we see that the benchmarks on which FDIP does best were the same three which obtained the largest prefetch distances. FSMR improved performance more on those benchmarks that showed smaller prefetch distances with FDIP in Figure 2.

5 Slicing was limited to dynamic spawn-target pairs under 4000 instructions apart leaving a small number of I-cache misses due to *no slice*.

6. Related work

Song and Dubois proposed *assisted execution* as a generic way to use multithreading resources to improve single-threaded application performance [9]. Chappell *et al.* proposed *simultaneous subordinate multi-threading* (SSMT), a general framework for leveraging otherwise spare execution resources to benefit a single-threaded application [5], and later proposed hardware mechanisms for dynamically constructing and spawning subordinate microthreads to predict difficult-path branches [6]. Zilles and Sohi analyzed the dynamic backward slices of performance degrading instructions [30]. They subsequently implemented hand crafted speculative slices to precompute branch outcomes and data prefetch addresses [28]. Roth and Sohi [22] proposed using *data-driven multi-threading* (DDMT) to dynamically prioritize sequences of operations leading to branches that mispredict or loads that miss. Moshovos *et al.* proposed *slice processors*, a hardware mechanism for dynamically constructing and executing slice computations for generating data prefetches [18]. Balasubramonian proposed a mechanism for allowing a future thread to advance ahead of the main thread when a long latency D-cache miss stalls the processor [4]. Annavaram *et al.* proposed dependence graph precomputation [3]. Luk proposed software controlled pre-execution [17] as a mechanism to prefetch data by executing a future portion of the program. Collins *et al.* proposed speculative precomputation [8], and later dynamic speculative precomputation [7] as techniques to leverage spare SMT resources for generating long range data prefetches and showed the importance of chaining helper threads to achieve effective data prefetching. Liao *et al.* extended this work by implementing a post-pass compilation tool to augment a program with automatically generated precomputation threads for data prefetching [16]. Finally, Kim and Yeung developed a source-to-source translator for generating data prefetch threads [15], and Mutlu *et al.* proposed runahead execution to prefetch when a data-cache miss would otherwise stall the processor [19].

7. Conclusion

This paper demonstrates the effectiveness of prescient instruction prefetch. In particular, we examine two categories of helper thread implementation techniques: direct pre-execution and finite state machine recall. The former achieves its efficiency via a combination of judicious construction of helper threads and two simple and novel microarchitectural support mechanisms: safe-store and YAT-bits. Safe-stores allow store-to-load dependencies to be observed in the helper thread without affecting the execution

of the main thread, while YAT-bits enable filtering out ineffectual computation when directly pre-executing a region of the main thread's code. Helper threads employing FSMR use a finite state machine like sequence of code and counted prefetch operations to prefetch along control flow path segments with biased branches and apply precomputation only to resolve otherwise unpredictable branch outcomes thus improving prefetch efficacy relative to direct pre-execution.

8. Acknowledgements

We would like to thank Andreas Moshovos, David Lie, Guy Lemieux, Steve Liao, Ravi Rajwar, Ronny Ronen, Greg Steffan, Michael Voss, Perry Wang and the anonymous referees for their valuable comments on this work. Tor Aamodt and Paul Chow were partly supported by funding from the Natural Sciences and Engineering Research Council of Canada. This work was performed while Tor Aamodt was an intern at Intel Corporation in Santa Clara.

References

[1] T. M. Aamodt, P. Marcuello, P. Chow, A. González, P. Hammarlund, H. Wang, and J. P. Shen. A Framework for Modeling and Optimization of Prescient Instruction Prefetch. In *SIGMETRICS*, pages 13–24, 2003.

[2] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *PLDI*, pages 246–256, 1990.

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *ISCA-28*, pages 52–61, 2001.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. In *ISCA-28*, pages 26–37, 2001.

[5] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *ISCA-26*, pages 186–195, 1999.

[6] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *ISCA-29*, pages 307–317, 2002.

[7] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic Speculative Precomputation. In *MICRO-34*, pages 306–317, 2001.

[8] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *ISCA-28*, pages 14–25, 2001.

[9] M. Dubois and Y. Song. Assisted execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California, October 1998.

[10] J. Emer. Simultaneous Multithreading: Multiplying Alpha's Performance. Microprocessor Forum, October 1999.

[11] G. Hinton and J. Shen. Intel's Multi-Threading Technology. Microprocessor Forum, October 2001.

[12] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5):12–23, 2000.

[13] Intel Corporation. Special Issue on Intel Hyper-Threading Technology in Pentium 4 Processors. Intel Technology Journal. Q1 2002.

[14] Q. Jacobson and J. E. Smith. Trace preconstruction. In *ISCA-27*, pages 37–46, 2000.

[15] D. Kim and D. Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *ASPLOS-X*, pages 159–170, 2002.

[16] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *PLDI*, pages 117–128, 2002.

[17] C.-K. Luk. Tolerating Memory Latency Through Software-Controlled Pre-execution in Simultaneous Multithreading Processors. In *ISCA-28*, pages 40–51, 2001.

[18] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-Processors: An Implementation of Operation-Based Prediction. In *15th International Conference on Supercomputing*, pages 321–334, 2001.

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *HPCA-9*, 2003.

[20] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero. Code Layout Optimizations for Transaction Processing Workloads. In *ISCA-28*, pages 155–164, 2001.

[21] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *MICRO-32*, pages 16–27, 1999.

[22] A. Roth and G. S. Sohi. Speculative Data-Driven Multithreading. In *HPCA-7*, pages 37–48, 2001.

[23] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. In *MICRO-31*, pages 259–71, 1998.

[24] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA-22*, pages 414–425, 1995.

[25] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *ASPLOS-IX*, pages 257–268, 2000.

[26] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ISCA-23*, pages 191–202, 1996.

[27] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA-22*, pages 392–403, 1995.

[28] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA-28*, pages 2–13, 2001.

[29] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *MICRO-35*, pages 85–96, 2002.

[30] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *ISCA-27*, pages 172–181, 2000.