In Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO-35), November 2002.

Pointer Cache Assisted Prefetching

Jamison Collins Suleyman Sair Brad Calder Dean M. Tullsen

Department of Computer Science and Engineering
University of California, San Diego
{jdcollin,ssair,calder,tullsen}@cs.ucsd.edu

Abstract

Data prefetching effectively reduces the negative effects of long load latencies on the performance of modern processors. Hardware prefetchers employ hardware structures to predict future memory addresses based on previous patterns. Thread-based prefetchers use portions of the actual program code to determine future load addresses for prefetching.

This paper proposes the use of a pointer cache, which tracks pointer transitions, to aid prefetching. The pointer cache provides, for a given pointer's effective address, the base address of the object pointed to by the pointer. We examine using the pointer cache in a wide issue superscalar processor as a value predictor and to aid prefetching when a chain of pointers is being traversed. When a load misses in the L1 cache, but hits in the pointer cache, the first two cache blocks of the pointed to object are prefetched. In addition, the load's dependencies are broken by using the pointer cache hit as a value prediction.

We also examine using the pointer cache to allow speculative precomputation to run farther ahead of the main thread of execution than in prior studies. Previously proposed thread-based prefetchers are limited in how far they can run ahead of the main thread when traversing a chain of recurrent dependent loads. When combined with the pointer cache, a speculative thread can make better progress ahead of the main thread, rapidly traversing data structures in the face of cache misses caused by pointer transitions.

1 Introduction

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

Data prefetching is one technique that reduces the observed latency of memory accesses by bringing data into the cache or dedicated prefetch buffers before it is accessed by the CPU. One can classify data prefetchers into three general categories. Hardware data prefetchers [3, 9, 10, 23] observe the data stream and use past access patterns and/or miss patterns to predict future misses. Software prefetchers [15] 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 multithreaded processors [27] has led to thread-based prefetchers [1, 5, 6, 13, 14, 18, 19, 24, 28], which execute code in another thread context, attempting to bring data into the shared cache before the primary thread accesses it.

However, traditional prefetching techniques have difficulty with sequences of irregular accesses. A common example of this type of access is pointer chains, where the code follows a serial chain of loads (each dependent on the previous one) to find the data it is looking for.

Some hardware approaches [23] can run ahead of the main program when traversing a pointer-chain, since they can predict the next address. This assumes that a history of the pointer traversal over the miss stream can be captured in the predictor. The accuracy of these techniques drops when running too far ahead of the main thread, since the techniques are based entirely on prediction. Accuracy can degrade because of (1) the address predictor being used, and (2) having pointer traversals in a program guarded by branches (e.g., tree traversal) and these techniques do not incorporate control flow into their prediction.

Thread based techniques, such as speculative precomputation [5, 6] (SP), have the advantage of using code from the actual instruction stream, allowing them to accurately precompute load addresses. A potential shortcoming of speculative precomputation is that cache misses can prevent the speculative thread from making progress faster than the main thread when traversing pointer chains with little other intervening computation,

In this paper we introduce a specialized cache used to aid prefetching that focuses just on pointer loads. The *Pointer Cache* is specifically targeted at speeding up recurrent pointer accesses. The pointer cache only stores pointer transitions (from one heap object to another), effectively pro-

viding a compressed representation of the important pointer transitions for the program. We examine using the pointer cache as a prediction table of object pointer values when a load executes. We also use a hit in the pointer cache to initiate a prefetch of the object to provide pointer-based prefetching for the main thread of execution. To keep the pointer cache up to date in the face of changing data, our design uses *Store Teaching*, to update the pointer transitions when they change.

We find that using a pointer cache to provide pointer base addresses overcomes serial accesses to recurrent loads for an SMT processor with support for prefetching via speculative precomputation. The pointer cache is particularly effective in combination with speculative precomputation for two reasons. It prevents the speculative thread from being hampered by long, serial data accesses. Data speculation occurs in the speculative thread rather than the main thread, allowing greater distance between the speculative computation and the committed computation. Furthermore, the control instructions in the speculative threads allow them to follow the correct object traversal path resulting in very accurate preloading of cache lines to be required by the main thread.

The pointer cache organizations we examine in this paper are not necessarily small (or fast) structures, but are an alternative to overly large on-chip cache structures that provide little marginal performance on many applications. We show that a pointer cache in combination with a smaller traditional cache can be a more effective use of processor transistors.

The rest of the paper is organized as follows. Section 2 discusses prior prefetching models. Section 3 describes the Pointer Cache, Section 4 its use in accelerating single-thread performance, and Section 5 its use in accelerating speculative precomputation. Simulation methodology and benchmark descriptions can be found in Section 6. Section 7 presents performance results, and Section 8 concludes.

2 Related Work

We build on research from both hardware-based pointer-chain prefetchers and thread-based prefetching. In this section we discuss these prior prefetching architectures and other related work.

2.1 Hardware Pointer Prefetchers

In Shadow Directory Prefetching by Charney and Puzak [2], each L2 cache block has a shadow address associated with it. The shadow address points to the cache block accessed right after the corresponding cache block, providing a simple Markov transition. A hit in the L2 cache with a useful shadow entry triggers a prefetch of the shadow address.

Joseph and Grunwald introduce using Markov predictors for prefetching [9]. When a cache miss occurs, the miss address is used to index into the Markov prediction table to provide the next set of possible cache addresses that previously followed this miss address. After these addresses are prefetched, the prefetcher stays idle until the next cache miss. They do not use the predicted addresses to re-index into the table to generate more predictions for prefetching.

Farkas et al. [8] propose a stream buffer architecture that uses a *PC-based* stride predictor to provide the stride on stream buffer allocation. Their PC-stride predictor determines the stride for a load instruction by indexing into a stride address prediction table with the instruction PC. The PC-stride predictor records the last miss address for *N* load instructions, along with their program counter values in an associative buffer. Thus, the stride prediction for a stream buffer is based only on the past memory behavior of the load for which the stream buffer was allocated.

A stream buffer is allocated on an L1 cache miss subject to allocation filters [8]. The stride predictor is accessed with the missing load's PC to store the stride into the stream buffer. The stream buffer starts prefetching cache blocks separated by the constant stride, starting with the one that missed in the L1 cache. On subsequent misses, the stream buffers are probed and if the reference hits, that block is transferred to the L1 cache. A stream buffer can issue prefetches only if it has empty entries available for prefetching. Once all entries are full, the stream buffer waits until it incurs a hit and an entry becomes available or until the stream buffer is reallocated to another load.

Sherwood, et al. [23] propose a decoupled architecture for prefetching pointer-based miss streams. They extend the stream buffer architecture proposed by Farkas et al. [8] to follow prediction streams instead of a fixed stride. Predictor-directed stream buffers achieve timely prefetches since the stream buffers can run independently ahead of the execution stream, filling up the stream buffer with useful prefetches. Different predictors can be used to direct this architecture allowing it to find both complex array access and pointer chasing behavior over a variety of applications. In addition, they present a new stream buffer allocation and priority scheduling technique based on confidence.

Recently, Cooksey et al. [7] presented a content-aware prefetching technique that examines each address-sized word in a cache line for a potential prefetch address on a demand L2 miss fill. After translation, addresses are issued as prefetches into the L2 cache. Upon returning from the memory hierarchy, prefetched cache lines are further examined for potential prefetch addresses provided they are within a certain prefetch distance threshold.

Our baseline architecture for the simulations in this paper includes the per PC-stride predictor proposed in [8] along with the confidence based stream buffer allocation and priority scheduling techniques detailed in [23].

2.2 Thread-Based Prefetchers

Several thread-based prefetching paradigms have been proposed, including Collins et al.'s Speculative Precomputation (SP) [6], Zilles and Sohi's Speculative Slices [28], Roth and Sohi's Data Driven Multithreading [18], Luk's Software Controlled Pre- Execution [13], and Annavaram's Data Graph Precomputation [1].

Speculative precomputation [6] 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), sequences of dependent instructions which, when executed, 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 (typically commit or rename), the corresponding p-slice is spawned into an available SMT thread context.

Speculative slices [28] focus largely on the use of precomputation to predict future branch outcomes and to correlate predictions to future branch instances in the non-speculative thread, but they also support load prefetching.

Software controlled pre-execution [13] 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 [1] 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.

2.3 Additional Pointer-based Prefetchers

Jump pointers are a software technique for prefetching linked data structures. Artificial jump pointers are extra pointers stored into an object that point to other objects some distance ahead in the traversal order. On future traversals of the data structure, the targets of these extra pointers are prefetched. Natural jump pointers are existing pointers in the data structure used for prefetching. It is assumed that when an object is visited one of its neighbors will also be accessed in the near future and prefetches are issued for all the pointer fields in the object. These techniques were introduced by Luk and Mowry [12] and refined in [11] and [17]. Roth et al. [16] also looked at dependence based prefetching which identifies producer-consumer pairs of accesses. A prefetch engine then speculatively traverses these and prefetches them as it goes.

Recently Chilimbi and Hirzel [4] proposed an automated software approach based on correlation. Their scheme first gathers a data reference profile via sampling. Next, they process the trace to extract data reference sequences that frequently repeat in the same order. At this point, the system inserts prefetch instructions to detect and prefetch these frequent data references. The sampling and optimization are done dynamically at runtime with very low overhead.

3 Pointer Cache

The Pointer Cache holds mappings between heap pointers and the address of the heap object they point to. This structure is organized like a cache, but with only word-length lines since we want to store only the important pointer transitions to maximize the utility of the structure. Since we are only interested in capturing pointer transitions, the pointer cache stores load values *only if the address of the pointer and the address of the object it points to fall within the range of the heap*.

The primary function of the pointer cache is to break the serial dependence chains in pointer chasing code. When one load depends on the data loaded by another, a cache miss by the first load forces the second load to stall until the first load completes. When executing a long sequence of such dependent pointer-chasing loads, instructions can only be executed at the speed of the serial accesses to memory.

3.1 Identifying Pointer Loads

Only pointer loads are candidates to be inserted into the pointer cache. We assume simple hardware support for identifying pointer loads, which are loads that access a heap address, loading a value which is also in the range of the heap. A load is identified as a pointer load if the upper N bits of its *effective address* match the upper N bits of the *value being loaded*. In this study we assume N to be 6 bits. We found that not inserting loads into the pointer cache whose effective addresses point to the stack provided higher performance than including them. Therefore, all load instructions with the stack pointer as a source register are classified as not being pointer loads in our study. This approach for dynamically identifying heap loads was proposed by Cooksey et al. [7]. They found that comparing the upper 8 bits was sufficient for their benchmark suite.

3.2 Pointer Cache Architecture

Figure 1 shows a processor organization for including a pointer cache. The pointer cache is queried in parallel with the L1 cache and returns the address of the object pointed to. This value is consumed by instructions dependent on the load, breaking the serial nature of the memory access. Pointer cache entries are optionally tagged with the source memory address they correspond to in order to avoid false hits. In this study, we assume partial tags.

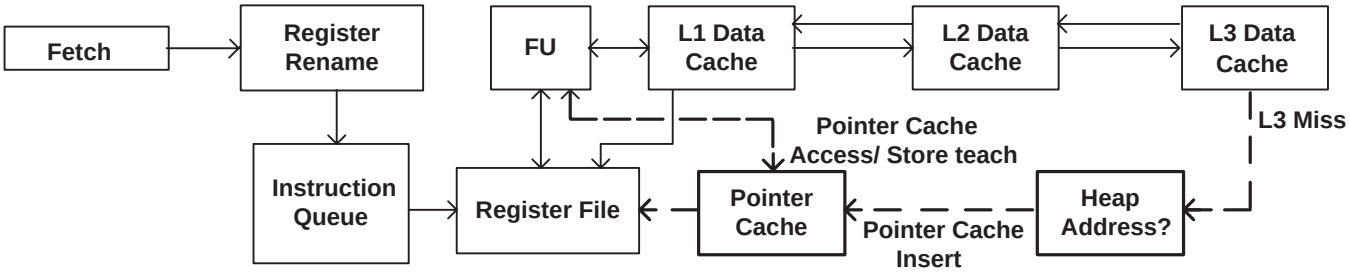

Figure 1. Pipeline organization of a processor with a pointer cache.

Because the pointer cache predicts a full memory address, the returned value can be directly used by instructions dependent on the load. Subsequent loads may result in further accesses to the cache and/or pointer cache, allowing parallel access to these serial data structures.

The pointer cache is updated by pointer based accesses (loads or stores) when they commit. When a pointer based load (a heap load which loads a pointer to another heap location) commits, it queries the pointer cache with the address it had accessed. If no entry is found for this address, the load is classified as a pointer load as described above, and the load had missed in L3 cache, a new pointer cache entry is allocated for the load. We only install a new pointer entry on L3 misses. This reduces contention for pointer cache entries and ensures maximum benefit is derived from each pointer cache entry. More aggressive or targeted filtering techniques are also possible. If a pointer load hits in any level of data cache, but the loaded value does not match the value stored in the pointer cache entry, then that pointer cache entry is updated.

Existing pointer cache entries must be updated when the program modifies pointer values. Otherwise, the pointer cache will lose effectiveness by returning incorrect information when queried. Such pointer cache misspeculations (in which a pointer cache hit occurs, but an incorrect memory value is returned), are avoided through the use of *Store Teaching* to compensate for the dynamic nature of data structures. This technique queries the pointer cache on all stores, and, on a hit, updates the relevant pointer cache entry with the store value. Store teaching updates occur in the commit stage of the pipeline. If a pointer cache with full tags employs store teaching, and is exposed to the system's cache coherence mechanisms, then values loaded from the pointer cache can safely be used by the processor without special care for misspeculation. However, in this research we assume only partial tags, so the value returned from the pointer cache could be incorrect. Even though we found this situation to be rare, the address returned by the pointer cache must be treated as speculative. Therefore, the load cannot commit until it verifies the predicted value against the actual loaded value. Section 4 describes this situation in more detail.

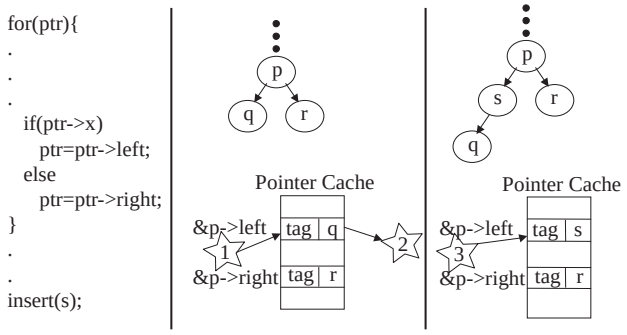

Figure 2. This figure illustrates the different operations performed on the pointer cache: training (1), prediction (2), and store teaching (3).

3.3 An Example Pointer Cache Use

Figure 2 illustrates the possible operations on a pointer cache. In this example `q` and `r` are the left and right children of `p` respectively. On the first traversal of this data structure, the pointer cache is updated with `q` at `&p->left` when we execute the `ptr=ptr->left` transition (1). Similarly, when we follow the other link we store `r` at `&p->right`. On a subsequent traversal of the data structure, a dereference of `ptr->left` which misses in cache results in a pointer cache hit, returning the address of `q` (2). Finally, when a later modification of the data structure takes place, (in this case, causing `p->left` to point at `s`), the pointer cache detects the update via store teaching, and updates the corresponding pointer cache entry (3).

4 Using Pointer Cache for Value Prediction and Main Thread Prefetching

We first examine using the pointer cache for a single-threaded wide-issue superscalar processor. This section describes using the pointer cache for two distinct purposes. The first purpose is to use the values out of the pointer cache to break true data dependencies exposing additional ILP. The

second purpose is to initiate prefetching when a load hits in the pointer cache.

4.1 Using the Pointer Cache to Predict Values

We access the pointer cache with a load's effective address in parallel with the data cache. A pointer cache hit provides a predicted value for the load. This provides the base address for an object to potentially be used by subsequent loads to access that object's fields. For programs where the critical loop path consists of instructions performing serialized pointer chasing, value predicting the base addresses effectively allows multiple loop iterations to execute in parallel.

If the predicted address differs from the value which was loaded, the misspeculation must be repaired, either by reexecuting instructions dependent on the load, or by flushing the pipeline. Note that very few misspeculations (less than 0.2% of all pointer cache accesses) occurred in our results because of the use of store teaching and 10-bit partial tags.

4.2 Using the Pointer Cache to Aid Prefetching

When using a pointer cache to provide predicted values, the consumers of this value are typically loads to various fields of the object. The predicted object's cache blocks will not be loaded into the data cache (assuming they are not already there) until the first consumers of each block of the object are executed. It can take several cycles from the time the pointer cache prediction was made until these first consumers are executed. This can occur because of resource contention, or because the first consumer of a block may be statically compiled a reasonable distance from the base pointer load. Consequently, programs can benefit from initiating the prefetch of the object when we get a hit in the pointer cache.

To use the pointer cache for main thread prefetching, when a load hits in the pointer cache, we initiate a prefetch of the pointed-to address and the sequential next line after that into the L2 cache. We blindly prefetch two blocks for the object, since many objects do not fit into one cache block. More accurately predicting the object size for prefetching is a goal of planned future work.

5 Using Pointer Cache with Speculative Precomputation

This section describes the use of the pointer cache to aid speculative precomputation, enabling it to get farther ahead of the main thread of execution. Unlike the main thread, speculative threads are not bound by the correctness requirement, and can freely integrate pointer cache address predictions into their thread context without verification.

5.1 Control Oriented Speculative Precomputation

This work assumes a Speculative Precomputation architecture similar to that described in [6], on a simultaneous multithreading [26, 27] (SMT) processor. For this work, speculative threads are generated off-line and trigger instructions are identified manually. Speculative threads fetch instructions out of a dedicated hardware Slice Cache, avoiding fetch related conflicts with the main thread.

Unlike the prior speculative precomputation work in [6], speculative threads constructed for this research can contain important conditional branches. One of the primary benefits of SP in this architecture is the ability to correctly follow control flow. Allowing SP to resolve branch instructions and recover from branch mispredictions provides the ability to accurately traverse control flow to find out what pointer transition to follow next. This is particularly important when there are several possible next pointer transitions for a given loop construct.

Control speculation in the speculative threads is handled just like a non-speculative thread — fetch is guided by branch prediction, and on a branch misprediction all instructions following the mispredicted branch are squashed and the speculative thread is redirected down the correct control path.

Because slices contain branches in this research, poor branch prediction resulting from a cold branch predictor could hamper the speculative threads' ability to get ahead of the main thread. Therefore, we assume an architecture that allows the speculative thread to benefit from the branch predictor training of the main thread. For conditional branch prediction, we would ideally like to have the same hash bits for the speculative thread branch PCs as the original main thread code. Thus, we assume that the speculative thread code is laid out so that the PC bits used for the branch prediction hash are the same between the branch in the main thread and its duplicate in the speculative thread. Therefore, the speculative thread will use the same 2-bit counters for prediction as the main thread, but in our implementation the speculative threads do not update the conditional branch 2-bit counters when the branch completes. For branch target address prediction, the speculative branch stream needs to be executed once and inserted into the branch target buffer before taken branches on the SP thread can enjoy no fetch stalls.

5.2 Using the Pointer Cache with Speculative Precomputation

The pointer cache is used by speculative threads only as a value prediction. When a speculative thread executes a load, the pointer cache is accessed in parallel. On a pointer cache hit, the load is marked as ready to commit (in addition to waking up its dependent instructions) *even though its mem-*

ory request is still outstanding. Since the speculative thread doesn't have to be correct, we do not wait until the load comes back from memory before committing a load that hits in the pointer cache. In comparison, the main thread, when using the pointer cache speculatively, must validate the value prediction before the load and its dependencies can complete execution, which may cause the main thread's fetch to stall due to a full instruction window. Thus, SP makes it possible for speculative computation to advance well ahead of the main thread when the speculative thread incurs a sequence of pointer cache hits.

5.3 Making Speculative Threads

Speculative threads are constructed manually, using an approach similar to the automatic construction described in [5]. We start with loads that account for the majority of misses for the program, which we call *delinquent loads*. The program's static instructions are analyzed in the reverse order of execution from a delinquent load, building up a slice of instructions the load is directly and indirectly dependent upon. As instructions are traversed, the set of registers (the register *live-in* set) necessary to compute the address accessed by the delinquent load is maintained — when an instruction is encountered which produces a register in this set, the instruction is included in the slice, and the register set is updated by clearing the dependence on the destination register, and adding dependences for the instruction's source registers. If, during analysis, another load which has been identified as delinquent is analyzed, it is automatically included in the slice.

If only a single dominant control path leads to the delinquent load, only this control path is traversed (in reverse order) when constructing the slice, and the conditional branches necessary to compute this control path are not included in the slice. Slice construction terminates when analyzing an instruction far enough from the delinquent load that a spawned thread can provide a timely prefetch, or when further analysis will add additional instructions to the slice without providing further performance benefits. In this form, a slice consists of a sequence of instructions in the order they were analyzed.

The single path slices constructed in this work are triggered typically between 40 and 100 instructions prior to the delinquent load, and contain between 10 and 15 instructions. In some cases, moving the trigger instruction further back would make the slice less profitable because doing so requires a significant increase in the number of executed speculative instructions. For example, when the targeted delinquent loads are preceded by computation of a hash function, including this hash function in the slice may degrade performance.

If multiple control paths lead to the delinquent load, the constructed slice must contain the dependence chains from each path, and must also include the relevant branches to determine which path to take when executing the slice. In addition, if there are multiple delinquent loads on different paths that define the same register, then those loads and their corresponding branches are included into the same slice. For example, the slice formed for Figure 2 includes the loop branch, the `if (ptr->x)` branch and its register operand definitions, and the two loads `ptr=ptr->left` and `ptr=ptr->right`.

Slices which contain multiple paths are typically targeting delinquent loads within a loop. The slice formation is terminated when analysis reaches an instruction sufficiently far preceding the loop (in which case the last instruction added is marked as a trigger instruction), or when the live-in set converges for all of the control flow paths and the instructions in the slice. Branches that are not critical to the slice are marked as *predict-only*. This means that those branches will only generate a prediction, and the speculative thread will not verify the correctness of the prediction. In addition, these branches cannot cause a speculative thread to be terminated, which we describe in more detail below. These predict-only branches are inserted into speculative threads in order to keep the global branch history information accurate for the main thread, and to keep the speculative branch outcome FIFO synchronized with the branches seen by the main thread. Keeping global branch history accurate in the speculative thread is important since the main thread and speculative thread share the 2-bit predictor entries.

The multiple-path slices with control flow consist of between 8 and 40 static instructions for the programs we examined, with some instructions unique to a particular control path. In comparison, the original targeted loops contain between 27 and 171 instructions. Some slices consist of a significant number of the instructions making up a loop ($>$50% of the loop instructions) when address computation is complex, or when a significant number of branch outcomes must be computed.

5.4 Spawning Speculative Threads

When an instruction in the main thread which has been identified as a trigger instruction reaches the register rename stage, a speculative thread is spawned into a hardware thread context if one is available and another instance of the same speculative thread is not already executing. The speculative thread's context is initialized by copying the necessary register live in values from its parent thread, and by copying the global history register value (at the time the trigger instruction was fetched) from its parent thread, ensuring the accuracy of the branch predictions performed by the speculative thread.

Speculative threads spawn further child threads via explicit child spawn instructions (the *chaining triggers* of [6]) in the speculative thread. Unlike when spawning threads off

of the main thread, a child thread is spawned even if there is already another instance of that particular thread executing. A speculative thread that attempts to fetch a spawn instruction when all thread contexts are occupied is prevented from doing so, and stops fetching until a thread context becomes available.

5.5 Maintaining Control of the Speculative Threads

The branch predictor for our SMT architecture is shared among all of the threads, but each thread maintains its own global history of executed branches. We call this the branch outcome FIFO queue. When a speculative thread is forked, the parent thread's speculative global history register, when the trigger point was fetched, is copied to the speculative thread. From that point on, the speculative thread performs its own branch predictions and keeps track of its predicted global branch history in the FIFO global history queue.

The speculative thread keeps track of the outcomes of all executed branches that have not committed in the main thread, starting from the trigger point. This enables two important mechanisms — killing of speculative threads which have deviated from the main thread's control flow, and a feedback mechanism to control how far ahead a speculative thread is allowed to go.

The first mechanism is achieved by comparing the head of this FIFO with each main thread branch that commits, and killing the speculative thread when they differ. If the branch outcomes agree, the head entry is removed. A speculative thread can be on the wrong path because the formation of the speculative thread has left out an instruction (e.g., a store) that would affect at a later point an outcome of a branch on the speculative thread. Note, that if the FIFO entry was produced by a predict-only branch, this check is not performed. Predict-only branches do not cause a speculative thread to be terminated.

The second mechanism is achieved by preventing a speculative thread from fetching when this FIFO exceeds a limit. Each slice can have its own limit. The multi-path slices we created had this limit set between 8 and 256 branches, to target traversing several loop iterations ahead.

The branch outcome FIFOs can also provide a simple mechanism for guiding main thread branch prediction with outcomes computed in a speculative thread. However, because this study focused only on the use of speculative threads for prefetching, computed branch outcomes are not used for this purpose. This is a topic of future work.

6 Methodology

Benchmarks are simulated using SMTSIM [25], a cycle accurate, execution driven simulator that simulates an out-of-order, simultaneous multithreading processor. SMTSIM executes unmodified, statically linked Alpha binaries. Table 1 shows the configuration of the processor modeled in this research. Programs are simulated for 300 million committed instructions or until completion, starting with a cold cache.

This paper studies eight memory limited benchmarks. `mcf`, `parser`, and `vpr` are from the SPECINT2k suite and `em3d` is from the Olden suite. `Dot` is taken from the AT&T's GraphViz suite. It is a tool for automatically making hierarchical layouts of directed graphs. Automatic generation of graph drawings has important applications in key technologies such as database design, software engineering, VLSI and network design and visual interfaces in other domains. We also used `gawk`, which is the GNU Project's implementation of the AWK programming language. Finally our benchmark suite includes `sis` which is an interactive program for the synthesis of both synchronous and asynchronous sequential circuits, and `vis` which integrates the verification, simulation, and synthesis of finite-state hardware systems.

Benchmarks from the SPEC suite are compiled for a base SPEC build, and other benchmarks are compiled with gcc -O4. All simulations except `em3d` and `gawk` run for 300 million instructions after first skipping program initialization code as determined by the SimPoint tool [21, 22]. `Em3d` and `gawk` are run from the beginning until completion. Table 2 presents additional details on the simulated benchmarks, including the fraction of simulated instructions which are loads, and data cache miss rates for each level of cache when executing each program with no prefetching.

We examine two architectures that use stream buffers. The first is the baseline program counter based stride prefetcher proposed by Farkas et al. [8] (stride prefetching). The second is the *Stride-filtered Markov* (SFM) predictor proposed by Sherwood et. al. [23]. For all stream buffer configurations, we use the confidence-based allocation and priority-based scheduling described in [23]. The stride configuration uses a 256-entry, 4-way associative stride address prediction table. For the stride-filtered Markov scheme, we utilize a similar 256-entry 4-way stride address prediction table to filter stride predictions out of a 256K-entry Markov table – this makes the Markov table roughly comparable in size to our most common pointer cache configuration. For both the stride and the stride filtered Markov architectures we use 8 stream buffers, each with 4 entries. All stream buffers are checked in parallel on a lookup. In addition, when a stream buffer generates a prediction, all stream buffers are checked to guarantee that the stream buffers do not follow overlapping streams. Unless otherwise noted, all the techniques utilize the PC-based stride predictor along with 8 stream buffers with 4 entries each.

The speculative precomputation technique makes use of an SMT processor with 8 thread contexts. As described in the previous section, the precomputation slices are created

Pipeline Structure	8 stage pipeline, 1 cycle misfetch penalty, 6 cycle minimum mispredict penalty
Fetch	8 instructions total from up to two threads
Branch Predictor	88kbit 2Bc-gskew branch predictor modeled after the EV8 [20] branch predictor but with instantaneous global history update (which also accounts for its smaller size)
	256 entry 4-way associative BTB
Execution Resources	6 total int units, 4 can perform mem ops, 3 fp. All units pipelined, 256 int and 256 fp renaming regs 128 entry int and fp instruction queues. Each thread has a 384 entry commit buffer
Memory Hierarchy	64KB, 2-way instruction cache, 64KB, 2-way data cache 256KB, 4-way shared L2 cache (10 cycles round trip time) 2048KB, 8-way shared L3 cache (30 cycle round trip time) Memory has a 230 cycle round trip time, 128 entry instruction and data TLB TLB misses handled by pipelined, on chip TLB miss handler, 60 cycle latency
Prefetching	Stride prefetcher with a 256-entry, 4-way stride table, 8 stream buffers of 4 entries with 1 cycle access time
Multithreading	8 total hardware thread contexts

Table 1. Assumed baseline architecture simulation parameters.

Bench	FFwd	% lds	L1 MR	L2 MR	L3 MR
`dot`	5.1B	40.0%	26.7%	88.7%	97.1%
`em3d`	0	30.9%	7.6%	89.0%	74.5%
`GNU awk`	0	28.3%	1.2%	25.5%	67.8%
`mcf`	50.8B	30.8%	10.6%	82.2%	83.2%
`parser`	228B	23.4%	2.3%	59.9%	37.8%
`SIS`	5B	27.6%	2.3%	58.4%	23.0%
`VIS`	5B	24.5%	1.9%	84.4%	86.4%
`vpr`	31.8B	32.6%	2.8%	67.9%	50.1%

Table 2. Details of simulated benchmarks. Data cache miss rates are shown for a processor which performs no hardware prefetching.

by hand.

Each pointer cache entry is comprised of a 10-bit partial tag and a 26-bit differential address field. The address field stores the difference between the indexing address and the value stored in the entry to conserve area. Using 26 bits of difference was enough to cover the whole memory space for the programs we examined. The overall size of the 256K entry pointer cache is 1.1MB, which is about the size of a 1MB cache along with its tags. As for the partial tags, tag-width sensitivity analysis showed that tags wider than 8 bits result in very few tag mismatches — less than 1%.

7 Prefetching Performance

This section compares the pointer-cache assisted uniprocessor and speculative precomputation architectures to prior prefetching techniques which have been shown to perform well on pointer intensive programs. The prior techniques include speculative precomputation [6] (SP) alone and Predictor-directed Stream Buffers [23] (PSB) as described in Section 2.

7.1 Previous Prefetching Mechanisms

We first explore the performance impact of prior prefetching schemes. Figure 3 shows performance results with no prefetching, prefetching using a program counter stride predicted stride stream buffer (stride prefetching), Stride-Filtered Markov (SFM), Speculative Precomputation (SP), and a new combination not examined in prior literature, which combines stride stream buffers with speculative precomputation.

The results show that stride-based prefetching provides large speedups for `dot`, `em3d`, and `parser`. Speculative precomputation by itself performs well on `sis` and `vpr`, but misses a lot of potential benefit the simple stride prefetcher achieves. Therefore, we combined the stride prefetcher with speculative precomputation, and found that this provided the best overall results on average.

When combining speculative precomputation with stride prefetching, we allow the speculative threads to access the stream buffers and the stride predictor along with the main thread. To make the stride predictor thread-aware, we extended the load PCs used to update the predictor with thread IDs. This increases the accuracy of the predictor on a per-thread basis, maximizing the stream buffer efficiency. Speculative threads are treated as equals with the main thread with respective to all stream buffer functions (e.g., spawning stream buffers on a data cache miss). The results show that combining stride with speculative precomputation provides on average 7% speedup over only using stride, and 21% speedup over only using speculative precomputation.

Unlike the other benchmarks, `em3d` shows a very dramatic speedup (almost 300%) from using only stride prefetching over the next best approach. The traversal of `em3d`'s data structures occur in the same order in which

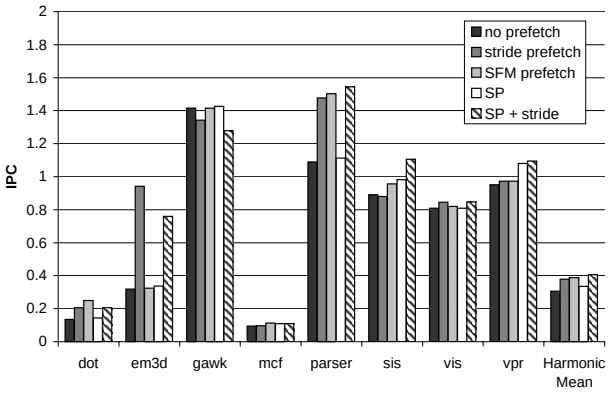

Figure 3. Performance improvement from previous prefetching schemes.

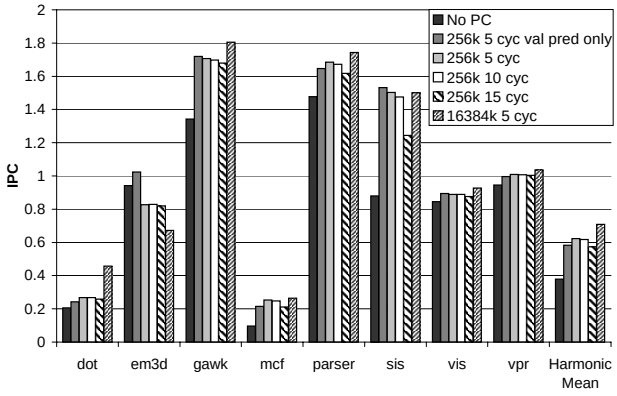

Figure 4. Performance impact when the main thread is permitted to access a pointer cache of varying sizes. All configurations also use the stride prefetcher. No PC refers to the baseline with only stride prefetching. Val pred only shows results using the pointer cache for only value prediction. All of the rest of the PC results use the pointer cache for both value prediction and prefetching.

they memory allocated, and the data structure pointers do not change during execution. Because of this and the fact that the memory allocator allocates the objects to sequential addresses in a linear order, `em3d`'s object traversal is highly stride prefetchable.

Two benchmarks, `dot` and `mcf`, involve significant numbers of dependent pointer dereferences. The stride-filtered Markov scheme provides the best performance for these two programs because the stream buffers are able to use address prediction to run ahead of the main thread. Speculative precomputation, in contrast, is hampered by the serial nature of these memory accesses, and has difficulty prefetching ahead of the main thread. Subsection 7.3 illustrates how the pointer cache enables SP to overcome this limitation.

7.2 Main Thread Pointer Cache Prefetching

In this section we evaluate the benefits of allowing the main thread to use the pointer cache (PC) to hasten issue of instructions dependent on a pointer load. We compare several pointer cache configurations with varying capacity and access latency to a baseline architecture with stride prefetching (labeled No PC in the graphs).

Figure 4 displays the effects of permitting the main thread access to a pointer cache with varying access times (between 5 and 15 cycles). All the pointer cache configurations are 4-way set-associative and have 256K entries, except the last one with 16 Meg entries. The first pointer cache bar shows the effect of only using the pointer cache for value prediction. The remaining bars use the pointer cache for value prediction and prefetching into the L2 on a pointer cache hit. In addition, all results have a baseline stride prefetching stream buffer.

The results show that access latency is not a significant factor in performance. This can be attributed to the pointer cache allowing the main thread to hide long latencies asso-

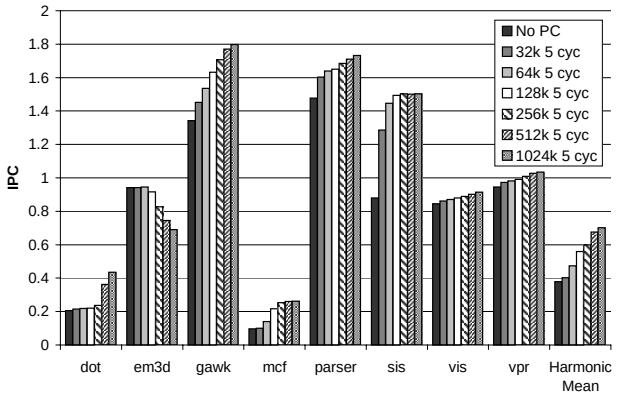

Figure 5. Performance impact varying the number of entries in the pointer cache. No PC refers to stride only prefetching. All the other configurations also make use of stride prefetching.

ciated with off-chip cache misses. Increasing the access latency beyond the 10 cycle L2 access to 15 cycles adversely affects `Sis`, because most of the L1 misses in `sis` are serviced by the L2 cache. Prefetching the pointer cache target into L2 cache works particularly well for `mcf` and `parser` over only using the pointer cache for value prediction. Since an increase in pointer cache hit latency results in a delay in initiating prefetches for the target object, these benchmarks suffer some performance degradation with a 15 cycle pointer cache. In comparison, `dot` also enjoys a significant benefit from additional prefetching, but it is not affected by the increase in access time since it is more capacity-limited than latency-limited as described below.

For two benchmarks, `em3d` and `sis`, value prediction only performs better than prefetching. This is because data structures in both these programs are small, and prefetching two cache lines pollutes the cache by bringing in irrelevant data. This is especially true for `em3d`, where stride prefetching is already effective at prefetching the same loads which are targeted by the pointer cache.

Figure 5 shows results for using the pointer cache for value prediction and prefetching varying the number of pointer cache entries. All configurations utilize stride prefetching. These results show that performance is more sensitive to the capacity of the pointer cache than to the latency to access it. With the exception of `dot` and `gawk`, the pointer data set for each program fits into a 256K entry, 4-way pointer cache. `Dot` has a large working set — it has a 3% hit rate on a 2M, 8-way L3 cache — and incurs pointer cache capacity misses for pointer caches with less than one million entries. `Gawk` also traverses long recursive data structures, creating the need for a large pointer cache. It is interesting to note that `gawk`, `parser` and `sis` attain noticeable speedups even with a relatively small 32K entry pointer cache.

7.3 Pointer Cache Assisted Speculative Precomputation

Figure 6 shows the performance impact when combining SP and stride prefetching while permitting speculative threads to access a pointer cache for value prediction only. For these results the pointer cache is *not* used by the main thread of execution. The performance benefits from permitting speculative threads access to the pointer cache are often significant, varying between 2% and 236% over a processor with no pointer cache.

In comparing Figures 4 and 6, some benchmarks achieve a smaller speedup when combining SP with the pointer cache than when only the main thread using the pointer cache. It is not always possible (or profitable) to construct slices targeting some loads in a program, but these loads may achieve hits in the pointer cache when accessing it directly by the main thread. Maximal benefit is achieved by permitting both

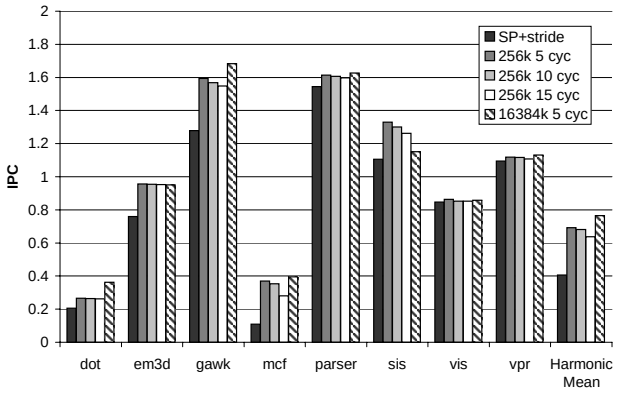

Figure 6. Performance impact from combining SP and stride prefetching when only using the pointer cache for speculative precompuation threads. The main thread does not use the pointer cache.

the main thread and speculative threads access to the pointer cache, which we present next.

7.4 Summary of Results

We now compare the performance of the pointer cache architecture to a baseline architecture that uses the transistors we devote to the pointer cache to instead increase the size of the L3 cache. Figure 7 compares the performance of speculative precomputation with stride prefetching using a 3 MB L3 cache to the performance a 1 MB pointer cache and a 2 MB L3 cache. The 1 MB pointer cache is 4-way associative and has 256K entries, each requiring 36 bits (as described in Section 6). The 1 MB pointer cache is accessed in 20 cycles to present a fair comparison to the 3MB L3 cache with 30 cycles access time (which is the L2 access time plus 20 cycles). We present results for four pointer cache configurations: (1) allowing only the main thread to access the pointer cache to perform value prediction, (2) allowing only the main thread to access the pointer cache for value prediction and prefetching target objects, (3) allowing only the speculative threads to access the pointer cache for value prediction, and finally (4) allowing both the main thread and the speculative threads to access the pointer cache for value prediction while using it also for prefetching for the main thread.

For programs which experience a large number of serialized pointer chasing related misses, the pointer cache provides very significant performance benefits over increasing the L3 cache size; `mcf` achieves a speedup of more than 268% from using a pointer cache compared to increasing the size of the L3 cache. Only `em3d` and `vpr`, which do not significantly benefit from the pointer cache, perform better with the larger L3 cache.

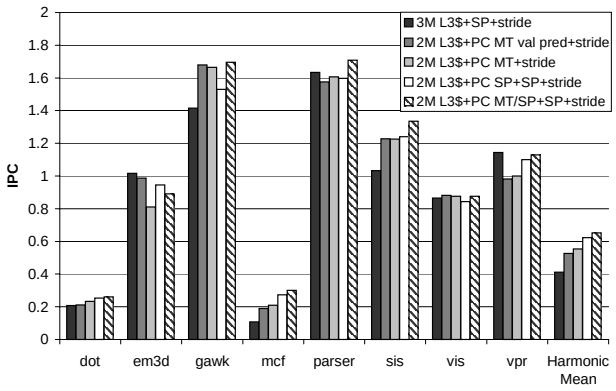

Figure 7. Performance results comparing a processor with a 2M 8-way L3 cache and 256k entry 4-way, 20 cycle pointer cache against a baseline stride prefetching architecture with a 3M, 12-way L3 cache. The pointer cache is used for value prediction and prefetching for all main thread (MT) results, except the second bar, where the pointer cache is only used by the main thread for value prediction. All configurations access their L3 cache in 30 cycles.

8 Conclusions

A wide range of applications — from games to database management systems — are based on dynamic data structures linked together via pointers. However, such accesses are often not governed by the localities exploited by traditional cache organizations. Furthermore, misses to such pointer-based loads, especially recurrent load accesses, significantly restrict parallelism and expose the full latency to memory.

In this paper we propose using a Pointer Cache to accelerate processing of pointer-based loads. The pointer cache provides a prediction of the object address pointed to by a particular pointer. If the load misses in cache, consumers of this load can issue using this predicted address from the pointer cache, even though the load itself is still outstanding. Using the pointer cache for just value prediction provided a 50% speedup over stride prefetching for a single-threaded processor.

We also examine using the pointer cache to initiate prefetches for the main thread of execution. On a pointer cache hit, prefetches for the first two cache blocks of the object are initiated. This provides an additional 5% speedup on average over using the pointer cache for just value prediction because the object's cache blocks are accessed at the time of prediction instead of waiting until their first use.

Another contribution of this paper is the examination of adding stride prefetching and the pointer cache to speculative precomputation. We found that a speculative precomputation architecture that incorporated a stride prefetching architecture provided 7% speedup over using only a stride prefetcher. Even with this improvement, we found that applying speculative precomputation with stride to a suite of pointer intensive applications was ineffective for half of the programs we examined, often because of recurrent loads.

To address this, we found that using the pointer cache increases the effectiveness of speculative precomputation by supplying speculative threads with pointer load values when they otherwise would have been forced to stall due to cache misses. This enables the speculative thread to prefetch far ahead of the main thread. When executing recurrent loads, the pointer cache is invaluable, as otherwise a cache miss by any load prevents the speculative thread from making any further progress ahead of the main thread until the data returns. A challenge for speculatively prefetching object transitions, is dealing with objects that have several possible next transition fields. To address this, we present a new form of speculative thread that includes additional control flow in order to accurately guide which next object transition to take. This allows the speculative threads to preload the cache lines that will soon be demanded by the main thread of execution. Our results show that stride prefetching with speculative precomputation using a 2 MB L3 cache and a 1 MB pointer cache is able to achieve 54% speedup on average over stride-based prefetching with a 3 MB L3 cache. When both the main thread and speculative precomputation threads using the pointer cache for prefetching, the speedup increases to 62% on average. This shows that a pointer cache can be an attractive alternative to increasing the L3 on-chip cache to help reduce the memory bottleneck.

Acknowledgments

We would like to thank Tim Sherwood for helping develop early Pointer Cache concepts. We would also like to thank Antonio González and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045, NSF grant CCR-0105743, a Focht-Powell fellowship, and a grant from Compaq Computer Corporation.

References

[1] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[2] M.J. Charney and T.R. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3), May 1997.

[3] T.F. Chen and J.L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[4] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the SIGPLAN'02 Conference on Programming Language Design and Implementation*, June 2002.

[5] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precomputation. In *34th International Symposium on Microarchitecture*, December 2001.

[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*, June 2001.

[7] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

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

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

[11] M. Karlsson, F. Dahgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[12] C.-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[13] C.K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[14] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *International Conference on Supercomputing*, June 2001.

[15] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992.

[16] 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*, October 1998.

[17] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[18] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[19] A. Roth, C. B. Zilles, and G. S. Sohi. Micro-architectural miss/execute decoupling. In *International Workshop on Memory access Decoupled Architectures and Related Issues*, October 2000.

[20] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[21] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[23] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[24] Y. Song and M. Dubois. Assisted execution. Technical Report CENG 98-25, University of Southern California, October 1988.

[25] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[26] D.M. Tullsen, S.J. Eggers, J. Emer, H.M. 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.

[27] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[28] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, June 2001.