Microarchitectural Miss/Execute Decoupling

Amir Roth, Craig B. Zilles and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin - Madison
{amir, zilles, sohi}@cs.wisc.edu

Abstract

The decoupled access/execute architecture described a machine that enables the access of memory values to be decoupled from the consumption of those values. Although never widely adopted in its original form, the decoupled design is a compelling way to tolerate memory latency. In this paper, we propose and demonstrate a novel implementation of decoupling, one based on the following two refinements of the original idea. First, because the latency of cache hits can generally be tolerated, we only decouple from the main program accesses that are likely to miss in the cache. Second, our decoupling takes place at the microarchitectural level, not the architectural level. By treating the access stream as a speculative thread and not allowing it to modify the architectural state of the machine, we relax the correctness constraints that were placed on it in the original design. For many programs, this added flexibility enables a level of decoupling and, consequently, latency tolerance that could not be achieved under the more constrained architectural model.

1 Introduction

The Decoupled Access/Execute Architecture was proposed in 1982 [S82] and described a processor with a two instruction stream interface. Architectural decoupling was attractive at the time because it provided a means to sidestep the Flynn bottleneck of fetching/decoding a single instruction per cycle and enabled some load/use slip without the complexity of a full out-of-order implementation. The appeal of these particular features of the architecture has largely disappeared over the past twenty years as nearly all microprocessor vendors now ship superscalar, out-of-order processors. However, access decoupling itself is still appealing, perhaps more so than ever. Today's processors can tolerate latencies of about 10 cycles — enough to cover the bulk of first level cache misses. However, the rapid increase in processor frequency has caused relative main memory access latencies to exceed 100 cycles. Access decoupling is a compelling method for tolerating these latencies, since scaling the out-of-order mechanism for this purpose has proven difficult.

The access decoupling scheme we propose for today's processors differs from the original design in two major ways. First, we restrict decoupling to only those accesses that are likely to result in cache *misses*. Second, we perform the decoupling at the *microarchitectural*, rather than architectural, level. Instead of extracting an architectural access stream from the program, we extract multiple speculative microarchitectural *miss streams*. Rather than running these miss streams on a second architected pipeline, we execute them on the additional hardware contexts of a multithreaded processor. Instead of requiring that miss streams bind correct values to registers on behalf of the execute stream, we simply ask them to perform performance-enhancing data movement, freeing them to use speculation in order to perform this task more efficiently. By removing correctness responsibilities from miss streams, decoupling and latency tolerance are more easily achieved for a wider class of programs.

Our implementation of microarchitectural miss decoupling is called *speculative data-driven multithreading (DDMT)*. DDMT is an new processing model that supports the execution of speculative non-contiguous code sequences called *data-driven threads (DDTs)*. In this implementation, the miss stream is split into multiple sub-streams each of which is microarchitecturally forked at a certain point by the original program and speculatively executed as a DDT. The main stream and several DDTs execute in parallel by using an underlying microarchitecture that supports multiple threads, like a simultaneous multithreading (SMT) processor [HK+92, YM95, TE+96]. The main thread then attempts to *reuse* the results produced by the DDTs.

The rest of the paper is organized as follows. In the next section, we describe the logical (not necessarily historical) chain of observations that lead from the original decoupled architecture to the speculatively decoupled microarchitecture we propose. We then present a short characterization of the estimated usefulness of speculative miss streams. We conclude with a brief description of DDMT and a short performance evaluation.

2 From Decoupled Architecture to Speculative Decoupled Microarchitecture

The original decoupled architecture called for a two stream interface. The *access stream* contained all memory accesses and their *backward slices* (all instructions that transitively contribute values to the address calculation). The *execute stream* contained everything else. The streams communicated via a set of architected queues. When the access stream did not need values from the execute stream, it could run ahead as far as result queueing would allow and, in doing so, effectively "absorb" access latency on behalf of the execute stream. The original decoupled architecture was realized, but did not gain widespread acceptance. Over the past twenty years, several embodiments of decoupling were proposed (and some were actually built) that successively solved some of the problems that hampered predecessor designs.

One obstacle that slowed the acceptance of the original decoupled architecture was a marketplace increasingly dominated by single stream interface machines and systems with relatively high latency support for inter-stream communication. The desire for a single stream solution was met early on with the observation that, when adequate decoupling is possible, the access stream can simply be "shifted up" with respect to the execute stream and the two streams can subsequently be *re-merged*. Software pipelining [RG+82] is the premier instantiation of this concept.

Traditional re-merging is effective but requires that decoupling be possible in the first place. Decoupling works well for programs that contain many operations that obviously don't contribute to address calculation. It is not surprising that workloads circa 1982 were dominated by programs that fit this description — floating-point programs. It may be that multimedia programs, which have similar structure, will dominate future workloads. However, non-numeric programs, which have been and will likely continue to be an important part of workloads, have features — complex control flow, many procedure calls and numerous statically unanalyzable accesses — that make statically distinguishing "purely execute" computations difficult [SP+98]. An observation that helped expand the scope of decoupling to difficult-to-split, non-numeric applications was that splitting the program *per se* is not strictly necessary. A load consists of two sub-operations. The first, bringing a data value to the processor, is not architecturally visible and can take a long time while the second, binding the value to a register, is architecturally visible and takes a fixed, small amount of time. The availability of an architectural mechanism that can perform the data movement function independently of the result binding removes the true decoupling requirement in the following way. Instead of decoupling and shifting an access stream within the program, we construct an *additional* stream that performs only data movement — the latency critical yet non-architected portion of the task. We then shift and merge this additional stream into the original program. An access stream that provides only data movement has the advantage of not requiring binding communication through the precious architectural register namespace; the data movement effect is implicitly communicated through the much larger namespace of the cache. Reduced register pressure can greatly unconstrain an optimizing compiler and improve overall code quality. Adding a second access stream to a program does incur some execution overhead. However, this overhead is much smaller than that of a adding a second conventional (full) access stream. Specifically, since it provides only data movement services, the added stream is constructed to deal only with accesses that are likely to have long data movement components, in a sense becoming a *miss stream* rather than an access stream. Indeed, the introduction of non-binding prefetch instructions coupled with the demonstration that the set of static loads that miss in the cache is both small and predictable [AS+93] combined to spawn a host of effective instantiations of this basic technique [ML+92, CB94, LS+95, LM96].

The single re-merged stream approach has been widely adopted, but does forfeit the true decoupling of the original architecture. The fetch schedule, and consequently the execute schedule, of the miss stream relative to the rest of the program is rigid and pre-determined. The miss stream cannot asynchronously run ahead when the main program stream is "pinned" by the processor's reorder buffer. This problem was solved by re-splitting the added miss stream from the original program and executing it as a separate thread on a multithreaded processor [SD98]. Since communication is uni-directional (main program to miss) and occurs only at a miss stream fork, a conventional multithreaded system could be used. Executing the miss stream in microcode [CS+99] has also been proposed. Finer control over miss stream scheduling is achieved by splitting the miss stream into several sub-streams and forking each one separately.

While multithreading restores decoupling, executing miss streams in architecture-level threads obligates the machine to execute them in full, fixing the amount of imposed overhead. A better approach is to specify miss streams as *hints* and to allow the *microarchitecture* to execute them non-architecturally at its discretion, using either dedicated [RM+98] or generic [RS01] hardware. By giving the microarchitecture flexibility in allocating resources to miss streams, miss stream resource consumption and overhead can be controlled dynamically and adapted to anticipated need and available bandwidth. The use of speculative, non-architectural multithreading has another advantage: it can be used to return miss streams to being parts of the original program. In fact, depending on the underlying thread implementation, the execution of a miss stream may contribute directly to the execution of the original program thread, and as such would no longer constitute overhead. Consider the following arrangement. Rather than split miss streams from the original program, the program is kept intact and instructions belonging to miss streams are instead *annotated*. Since they are composed of original program instructions, miss streams are "binding" again. The processor executes the annotated portions of the original program as speculative threads. Since these portions execute ahead of their architected place in the original program, their accesses implicitly perform the data movement function. The bindings themselves, however, are *buffered* rather than exposed architecturally. Now, when the main program thread catches up to the speculative thread, it “recognizes” it as actually being the same piece of code it is about to execute. Rather than repeating the work, the main program thread simply “picks up” the buffered bindings of the appropriate instructions and skips over their execution. Overhead is minimized because miss streams actually contribute to the execution of the original program. This is exactly the arrangement proposed by the original decoupled architecture!

```
for (l = list; l; l = l->next)       A1:  addq r1, zero, list
   process(l->data);                  A2:  beq r1, A7
                                      A3:  ldq r2, 4(r1)
                                      A4:  jsr process
                                      A5:  ldq r1, 0(r1)
                                      A6:  jmp A2
```

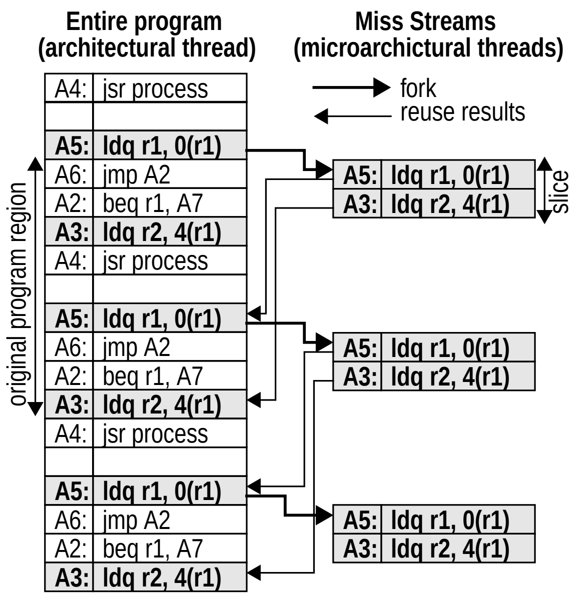

FIGURE 1. *An example of microarchitectural miss/execute decoupling.*

A simple example of microarchitectural decoupling is shown in Figure 1. At the top, we show source and corresponding Alpha machine code for a simple list traversal loop. The instructions in bold are *A3*, the first data access to each node — which results in a cache miss — and *A5*, the pointer-chasing loop induction — which when unwound comprises the full backward slice of *A3*. To tolerate *A3* latency, we choose the speculative miss streams to contain one instance of *A5* and one of *A3*. A miss stream is forked by the previous instance of the induction instruction *A5*. This choice of miss stream and fork point allows us to leverage one full loop iteration, including the entire invocation of *process*, for latency tolerance purposes. The bottom of the figure shows an abstract execution of these speculative miss streams on a multithreaded processor. Each instance of the induction instruction *A5* forks a new speculative miss stream to absorb the latency of the next iteration’s accesses. When the architectural thread (executing the entire program) reaches the next iteration, it reuses the values computed by the miss stream rather than re-executing the corresponding instructions. As shown in this example, miss streams can be small and have low overhead, while allowing likely-to-miss accesses to be efficiently decoupled from the main program. In this example, decoupling allows *A5* and *A3* of a given iteration to proceed while the processor is executing the *process* invocation code from the previous iteration.

3 Characterizing Speculative Miss Streams

Miss streams are *dynamic backward slices* (or just *slices*) of loads that are likely to miss in the cache. A slice is constructed by walking the dynamic instruction stream backward from the offending load and adding any instruction that satisfies one of three kinds of active dependences: (1) a *register dependence* in which the instruction writes a register that is read by an instruction already in the slice, (2) a *memory dependence* if the instruction is a store that writes to the same address of a load in the slice or (3) a *control dependence* if the instruction is a branch on which an instruction in the slice is control-dependent.

In this section, we present a short qualitative study of the slices we intend to extract and pre-execute as miss streams. A more complete study can be found here [ZS00]. We characterize slices in terms of their size relative to the original program regions with which they overlap — these definitions are illustrated in Figure 1. The size metric tells us how far ahead of its original place in the program a miss stream can be shifted and, consequently, how much latency it can hide. It also tells us roughly how many resources a miss stream will consume when executed in parallel with the original program. A “good” miss stream is small and highly concentrated towards end of the original program region, near the offending load. Such a distribution implies that the miss stream can be forked at the beginning of the region and finish execution before the targeted load is processed by the original program thread.

The graph in Figure 2 shows several size accumulations for the slice of a single static load from the program *compress*. These results are illustrative of slices in all benchmarks. We show a single slice because averaging the slices of multiple static loads blurs the important qualitative details. Three accumulations are shown. The top accumulation (the one with the largest area underneath it), is obtained by including all register, memory and control dependences in the slice. For the next slice, we exclude control dependences. Doing so makes the resulting miss stream *greedy* — unable to determine at runtime which computations really need execution and

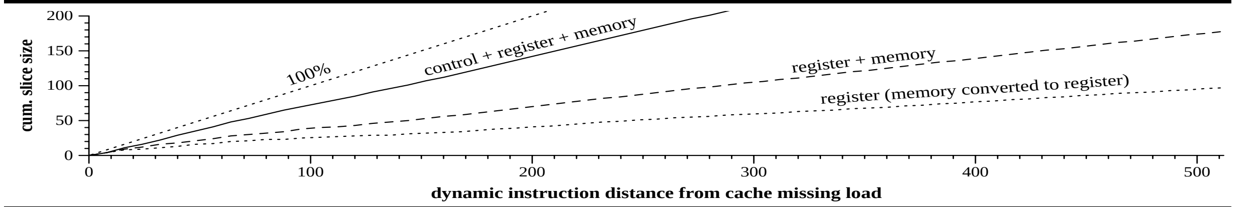

FIGURE 2. Average cumulative slice sizes as functions of dynamic instruction distance from the load for a single static load from compress.

which can be ignored, forcing it to execute all of them. However, it also drastically reduces the size of the miss stream, enhancing its run-ahead ability.

In the final accumulation, we convert stable memory dependences into register dependences. This is accomplished by identifying store/load pairs that consistently communicate (incidentally, this is the only way a miss stream will ever contain a store), annotating the store/load communication, and excluding the address calculation of both instructions from the slice since it is no longer needed to propagate the data value. This micro-architectural "register allocation" requires a hardware mechanism for passing values from stores to loads without calculating the addresses. Cloaking [MS97] is one such mechanism.

Our results indicate that conservative generation of miss streams yields large streams that likely will not provide sufficient lookahead and will exact too much overhead. However, by utilizing the speculative nature of miss streams to optimize away first control dependences then memory dependences, we can construct streams that are short enough to support sufficient decoupling to tolerate long memory latencies.

4 An Implementation of Miss Decoupling: Speculative Data-Driven Multithreading

The fact that miss streams are small relative to the original program region they overlap and that the bulk of the work is concentrated towards the latter end of the region tells us that the potential to hide latency by pre-executing miss streams exist. To realize this potential, we need an engine that can execute miss streams.

The engine we propose is a *simultaneous multithreading (SMT)* processor modified to support the speculative sequencing and execution of non-sequential pieces of code. We call this new execution model speculative *data-driven multithreading (DDMT)* [RS01]; the non-sequential code segments are called *data-driven threads (DDTs)*. DDMT's unique support for non-contiguous speculative threads is important because miss streams are not composed of sequential instructions. It should be noted that other, more conventional speculative thread models that support *control-driven (sequential) threads* [F93, SB+95, DO+95, SM98] may be used to execute miss streams. However, these require that miss streams be expanded to include *all* instructions that are sequentially interleaved with the computation (slice) of interest. Including irrelevant instructions in miss streams detracts from their ability to run ahead and hide latency and increases the load on the system.

The ability to execute non-contiguous code sequences does come at an implementation cost. Since a DDT cannot be sequenced with a program counter, an additional mechanism is required to precisely describe the instructions that comprise the DDT and how these are ordered. In DDMT, the mechanism that provides this functionality is the *data-driven thread cache (DDTC)*. The DDTC contains static representations of DDTs "straightened out" and packed to look like linear code sequences. The DDTC isolates the fetch engine from the sequencing details of DDTs in much the same way that a trace cache [RB+96] abstracts the sequencing details of control-driven code. A DDMT processor fetches DDTs from the DDTC in chunks and places their component instructions into the instruction queue. The rest of the processor is oblivious to the non-contiguous nature of the DDT instructions. DDT instructions are renamed, scheduled and executed like any other instructions, although they are not retired nor are they allowed to modify architected state.

One component of DDMT we mention here but do not describe in detail is the *integration* facility. DDTs' non-contiguous nature prevents them from describing a complete picture of program state. As a result, the original program thread must re-sequence and re-execute all work performed in DDTs. Integration is a mechanism that removes the re-execution portion of that requirement. Re-sequencing is still mandatory, but integration uses the re-sequencing process to match up DDT instructions with their corresponding original program thread counterparts. The main thread integrates (incorporates) results computed by DDT instructions by making the buffered result bindings architecturally visible (a process we alluded to earlier). The decision about whether or not a given instruction can be integrated is implemented as an extension to register renaming. By choice, we allow only instructions that have completed execution in the DDT before being renamed in the main thread to be integrated.

5 Performance Evaluation

We evaluate a DDMT implementation of miss/execute decoupling for six programs — selected for their relatively high data cache miss rates — two each from the SPEC95, SPEC2000 and Olden benchmark suites. We compile the programs for the Alpha EV6 architecture using the Digital UNIX V4 `cc` compiler with flags `-O3 -fast` and simulate them in their entirety.

Our cycle-level simulator is built using the SimpleScalar 3.0 [BA97] Alpha toolkit. It models an 8-wide SMT processor with out-of-order speculative execution and a maximum of 128 instructions, 64 loads or 32 stores in-flight. The pipeline has 3 fetch, 2 decode/rename and 2 schedule/register-read stages. Up to 2 loads and 2 stores may issue per cycle. Address generation takes 1 cycle, with an additional cycle for either a first level cache hit or a store queue bypass. Loads issue in the presence of older stores with unknown addresses — on a mis-speculation, the load and younger instructions are squashed. The memory system consists of a 32KB instruction cache and a 64KB data cache, both 2-way set-associative with 32 byte lines, a shared 1MB, 4-way set-associative, 64-byte line second level cache and 32-entry TLB's. Up to 16 load misses can be simultaneously outstanding. The second level cache takes 12 cycles to access, main memory access takes 70 cycles. The second level cache and memory buses are 32 and 16 bytes wide and operate at full and one-fourth processor frequency, respectively. The simulated processor has 4 hardware contexts which share all resources, and is capable of running the original program thread and up to three concurrent speculative streams. Thread priority is explicit at fetch only with bandwidth allocated in round-robin fashion among active threads on a cycle basis.

We model an offline, profile-driven implementation of DDT-annotation generation. The selection algorithm processes a program trace that is generated from a run using a different, shorter input data set. We permit a maximum DDT length of 32 instructions and require a run-ahead head-start of at least 64 dynamic instructions. We assume that the resulting DDT annotations are encoded into the executable and are loaded from the executable into the DDTC on demand. None of our program use more than 11 distinct DDT miss streams.

Performance results are summarized in Table 1. Performance improvements range from a negligible 0.1% for *li* to 17.3% for *mst*. The higher speedups tell us that, while a significant portion of second level cache hit latency can be hidden by a machine with 128 instruction re-ordering capability, the additional decoupling provided by DDMT, specifically its ability to generate cache misses while the original program thread is "pinned", further increases memory-level parallelism (MLP) — the degree of memory access overlapping.

We provide several metrics to support these results. Load latency is the average difference between the issue and completion times of every *committed* load. MSHR occupancy is the average number of simultaneously outstanding misses — an MLP measure. In most cases, average load latency decreases while MLP increases suggesting that the DDT's are overlapping misses that are further away than a single instruction window's worth. In *mst*, the dominant DDT encapsulates a hash table search including hash function calculation. Execution of this DDT overlaps a second hash bucket traversal with the one taking place in the main program thread, doubling the MLP but increasing bus contention to a level that *slightly increases* the average load latency.

The observed speedups are somewhat smaller than the latency and MLP diagnostics suggest. The reason for this is that miss streams contend for resources with the main program thread, slowing it down. We approximate the contention effect by measuring the number of instructions fetched by DDTs. These numbers range

		compress	**li**	**gzip**	**vpr**	**em3d**	**mst**
Insns committed (millions)		331.39	1188.37	3367.27	692.50	248.88	230.77
Loads (millions)		42.68	302.63	677.78	198.37	71.59	32.58
Base	L1 misses (millions)	3.08	3.46	23.12	8.46	24.50	4.16
	Avg. load latency (cycles)	3.63	2.70	2.76	3.41	42.41	19.85
	Avg. MSHR occupancy (/cycle)	1.89	1.51	0.83	1.67	10.76	0.94
Base + DDMT	DDT's forked (millions)	3.57	4.32	26.28	7.09	0.80	0.52
	DDT insns fetched (millions)	104.08	29.20	671.02	145.24	23.20	16.24
	DDT insns integrated (millions)	15.82	13.38	248.52	37.14	12.82	10.30
	DDT loads integrated (millions)	2.12	4.00	13.80	5.04	5.61	2.52
	Avg. load latency (cycles)	3.34	2.44	2.11	3.37	29.58	21.19
	Avg. MSHR occupancy (/cycle)	3.05	2.50	1.05	1.81	12.58	1.81
	Speedup over base	**1.6%**	**0.1%**	**15.4%**	**12.0%**	**7.2%**	**17.3%**

TABLE 1. Using speculative data-driven multithreading to pre-execute miss streams.

from reasonable 5-10% to a quite high 30% for *compress*. In the latter case, when overhead and contention increase to the point of offsetting all benefit, DDMT should be suppressed. A mechanism for doing so dynamically is straightforward but beyond the scope of this paper.

One interesting metric is the percentage of fetched DDT instructions that are eventually integrated — lower than 60% for all benchmarks and as low as 15% for *compress*. Low integration rates indicate that DDTs fetch and execute many instructions unnecessarily. This is due in part to working set differences in data sets used for DDT selection and execution, but more so to the inherently greedy nature of DDTs. As we mentioned earlier, DDTs represent control greedily, not explicitly. A DDT may contain computations that exist along dynamically exclusive paths. However, rather than take the time to synchronize with the original program thread or to execute a piece of code to decide which computations to execute and which to discard, it simply executes all of them. The result is some amount of wasted work. Note, DDT efficiency is *not* tied to main thread branch prediction accuracy. In fact, efficiency may increase with decreased prediction accuracy. We are investigating DDT extensions for reducing this waste.

6 Summary

As initially proposed, the decoupled access/execute architecture has not been widely adopted. However, its motivating observations and key features have formed the basis for many load latency tolerance techniques. The latest incarnation is data-driven multithreading (DDMT). Unlike other recent variants, DDMT implements decoupling at the microarchitectural, rather than architectural, level, freeing the access stream from correctness obligations. This added flexibility minimizes synchronization with the main thread and promotes higher degrees of decoupling and, consequently, latency tolerance.

Acknowledgements

This work was supported in part by National Science Foundation grants MIP-9505853 and CCR-9900584, donations from Intel Corp. and Sun Microsystems, the University of Wisconsin Graduate School and Intel Ph.D Fellowships. The authors thank the anonymous referees for their reviews.

References

[AS+93] S. Abraham, R. Sugumar, D. Windheiser, B. Rau and R. Gupta. Predictability of Load/Store Instruction Latencies. *Proc. MICRO-26,* Dec. 1993.

[BA97] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[CB94] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Prefetching Techniques. *Proc. ISCA-21,* Apr. 1994.

[CS+99] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). *Proc. ISCA-26,* May 1999.

[DO+95] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading. *Proc. PACT-1995*, Jun. 1995.

[F93] M. Franklin. The Multiscalar Architecture. Ph.D. Thesis. Technical Report #CS-TR-1993-1196, Computer Sciences Dept., University of Wisconsin-Madison, Nov. 1993.

[HK+92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, T. Nishizwa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In Proc. of the 19th Annual International Symposium on Computer Architecture, May 1992.

[LS+95] M. Lipasti, W. Schmidt, S. Kunkel and R. Roediger. SPAID: Software Prefetching in Call and Pointer Intensive Environments. *Proc. MICRO-28,* Nov. 1995.

[LM96] C-K. Luk and T. Mowry. Compiler Based Prefetching for Recursive Data-Structures. *Proc. ASPLOS-7,* Oct. 1996.

[MS97] A. Moshovos and G. Sohi. Streamlining Inter-Operation Memory Communication via Data-Dependence Prediction. *Proc. MICRO-30,* Dec. 1997.

[ML+92] T. Mowry, M. Lam and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *Proc. ASPLOS-5,* Oct. 1992.

[PG99] J.-M. Parcerisa and A. Gonzalez. The Synergy of Multithreading and Access/Execute Decoupling. *Proc. HPCA-5,* Jan. 1999.

[RG+82] B. Rau, C. Glaeser, and R. Picard. Efficient Code Generation For Horizontal Architectures: Compiler Techniques and Architectural Support. Proc. ISCA-9, Apr. 1982.

[RB+96] E. Rotenberg, S. Bennett and J. Smith. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. *Proc. MICRO-29,* Dec. 1996.

[RM+98] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. *Proc. ASPLOS-8,* Oct. 1998.

[RS01] A. Roth and G.S. Sohi. Speculative Data-Driven Multithreading. *Proc. HPCA-7 (to appear)*, Jan. 2001.

[RS00] A. Roth and G.S. Sohi. Register Integration: A Simple and Efficient Implementation of Squash Reuse. *Proc. MICRO-33 (to appear)*, Dec. 2000.

[SP+98] S. Sastry, S. Palacharla and J. Smith. Exploiting Idle Floating Point Resources for Integer Execution. *Proc. PLDI '98,* Jun. 1998.

[S82] J. Smith. Decoupled Access/Execute Computer Architecture. *Proc. ISCA-9*, Jul. 1982.

[SB+95] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. *Proc. ISCA-22*, Jun. 1995.

[SD98] Y. Song and M. Dubois. Assisted Execution. Technical Report #CENG 98-25, Dept. of EE-Systems, University of Southern California, Oct. 1998.

[SM98] J. Steffan and T. Mowry. The Potential for Using Thread Level Data-Speculation to Facilitate Automatic Parallelization. *Proc. HPCA-4*, Feb. 1998.

[TE+96] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *Proc. ISCA-23*, May 1996.

[YM95] W. Yamamoto and M. Nemirovsky, Increasing Superscalar Performance Through Multistreaming. *Proc. PACT-95*, Jun. 1995.

[ZS00] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. *Proc. ISCA-27*, Jun. 2000.