Compiler Orchestrated Prefetching via Speculation and Predication

Rodric M. Rabbah[1], Hariharan Sandanagobalane[2], Mongkol Ekpanyapong[3], Weng-Fai Wong[2]
[1] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, rabbah@mit.edu
[2] Department of Computer Science, National University of Singapore, {harihar1, wongwf}@comp.nus.edu.sg
[3] School of Electrical and Computer Engineering, Georgia Institute of Technology, pop@ece.gatech.edu

ABSTRACT

This paper introduces a compiler-orchestrated prefetching system as a unified framework geared toward ameliorating the gap between processing speeds and memory access latencies. We focus the scope of the optimization on specific subsets of the program dependence graph that succinctly characterize the memory access pattern of both regular array-based applications and irregular pointer-intensive programs. We illustrate how *program embedded precomputation via speculative execution* can accurately predict and effectively prefetch future memory references with negligible overhead. The proposed techniques reduce the total running time of seven SPEC benchmarks and two OLDEN benchmarks by 27% on an Itanium 2 processor. The improvements are in addition to several state-of-the-art optimizations including software pipelining and data prefetching. In addition, we use cycle-accurate simulations to identify important and lightweight architectural innovations that further mitigate the memory system bottleneck. In particular, we focus on the notoriously challenging class of pointer-chasing applications, and demonstrate how they may benefit from a novel scheme of *sentineled prefetching*. Our results for twelve SPEC benchmarks demonstrate that 45% of the processor stalls that are caused by the memory system are avoidable. The techniques in this paper can effectively mask long memory latencies with little instruction overhead, and can readily contribute to the performance of processors today.

Categories and Subject Descriptors

B.1.4 [**CONTROL STRUCTURES AND MICROPROGRAMMING**]: Microprogram Design Aids—*Languages and compilers, Optimization*; B.3 [**MEMORY STRUCTURES**]: Design Styles; C.0 [**COMPUTER SYSTEMS ORGANIZATION**]: General—*Instruction set design*

General Terms

Performance, Design, Experimentation

Keywords

precomputation, speculation, predicated execution, prefetching

ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

Modern trends in the design of high-performance microprocessors continue to widen the gap between the rate of data consumption in the processor and the rate of data delivery from the memory system. This rate disparity implies a system must effectively tolerate (or mask) potentially long memory latencies, lest it suffer severe performance degradation as the processor stalls and awaits the delivery of data from memory. The performance implications are exacerbated when applications make extensive use of dynamically allocated objects which complicate the analysis of the application reference pattern and often inhibit the development of static techniques for masking long memory access latencies.

In this paper, we propose and evaluate a compiler technique to orchestrate the prefetching of data ahead of eventual processor references. The prefetching is driven by the precomputation of future memory addresses via speculative execution of load dependence chains. Each *load dependence chain* (LDC) represents a subset of the program dependence graph that is pertinent to the address calculation of a specific *delinquent* (i.e., long latency) memory operation. The compiler identifies the LDCs and statically embeds the precomputation within the program instruction stream, along with prefetch operations to mask potentially long memory access latencies.

This strategy of *program embedded precomputation via speculative execution* (PEPSE) relies on the compiler's ability to inject the precomputation into the host program without lengthening the critical path of the application. The key enabler for PEPSE is the limited ability of compilers to extract sufficient ILP (instruction level parallelism) to fully utilize the parallelism afforded by some architectures such as the Itanium Processor Family (IPF). Thus, once an application is optimized and scheduled, our compiler-orchestrated prefetching system is allowed to "steal" any remaining resources to embed speculative address precomputation within program regions that are likely to benefit from prefetching.

We have incorporated our ideas into the Open Research Compiler (ORC) for the IPF, and using our prototype, we have measured speedups up to 65% for nine benchmarks (seven of which are from the SPEC CFP suite) running on a 900 MHz Itanium 2 workstation. The performance gains are in addition to prefetching technology [28] available in ORC. Furthermore, we measured a 45% reduction in data-induced processor stall cycles for twelve SPEC benchmarks, seven of which are SPEC CINT applications. In contrast to the array-based SPEC CFP applications which are amenable to prefetching, the SPEC CINT benchmarks are pointer-heavy and often defeat prefetching strategies.

In this paper, we also introduce *sentineled prefetching for EPIC architectures* (SPEAR) as a technique to address a major challenge when prefetching in pointer-based programs. When point-

ers are involved, the instruction overhead for generating prefetch addresses is non-trivial because the load dependence chains will contain memory instructions that may miss in the data cache. In SPEAR, the speculative precomputation instructions are predicated such that if any load in the chain suffers a cache miss, a special predicate is set so that subsequent precomputation instructions are ignored. Thus, SPEAR allows for greater synergy between static prefetch orchestration and run-time adaptation in response to information propagated from the memory system. The ideas in this context draw from the early work on *informing loads* [17] and may lead to other interesting optimizations in the future. Our results show that sentineled prefetching affords a clear advantage in several scenarios and warrants further research.

We begin in Section 2 with a description of program embedded precomputation via speculative execution (PEPSE), and in Section 3 we introduce SPEAR as a mechanism for sentineled prefetching. In Section 4 we evaluate PEPSE using the Open Research Compiler for the Itanium Processor Family, and we demonstrate that our methodology complements traditional prefetching strategies that are considered quite effective in the context of Fortran and scientific applications. In Section 5 we look at the role of compiler-orchestrated prefetching in the context of pointer-intensive applications. Specifically, we look at how program embedded precomputation and prefetching impact performance. Section 6 discusses related work, and Section 7 concludes the paper with directions for future work.

2. SPECULATIVE PRECOMPUTATION

The strategy for program embedded precomputation via speculative execution (PEPSE) consists of four steps. First, certain program memory operations are identified as delinquent loads. Next, for each delinquent load, a specific subset of the program dependence graph (PDG)—consisting exclusively of the instructions necessary to compute the target address of that load—is isolated. Each PDG subset represents a load dependence chain (LDC) that is independently transformed to provide an adequate level of foresight. The transformations are especially important in cyclic program regions where prefetch instructions must issue several iterations ahead of the eventual load. In the final step, each LDC is embedded within the original application such that as the program executes, the operations in the LDC execute speculatively and trigger prefetch instructions ahead of their corresponding delinquent loads. The process of embedding the LDCs occurs just before code generation and subsequent to other compiler optimizations, and thus, the prefetch-orchestration system can exploit any available machine resources (e.g., registers and functional units) to schedule the precomputation viz. the load dependence chains. It is worthy to note that the PEPSE strategy outlined in this paper may be applied in continuous optimization scenarios such as Dynamo [6] and DELI [11], or even in JAVA Just-In-Time compilers.

2.1 Delinquent Load Selection

We narrow the scope of our optimization to a tractable subset of the *delinquent* load instructions in a program, and by doing so, we can devise simple and effective algorithms for extracting the precomputation chains and their injection into the program. Our prefetch-orchestration system identifies delinquent load instructions via *profiling*. It collects the requisite information by running an instrumented version of the original application alongside a cycle-accurate extension of the Dinero IV cache simulator [13]. The output of the profiling stage is a detailed record of the cache hits and misses, as well as the total number of cycles spent servicing each static load in the application. Empirically, we have observed that a small number of load instructions account for more than 90% of the total data stalls that a program suffers (see Table 1). Our results are in accord with prior work [2, 31] that also observed the number of delinquent loads in a program is small when compared to the total number of loads in the same program. This characteristic allows the prefetching system to focus the memory optimizations to a manageable set of instructions, and indeed, we exploit this characteristic in our work.

Table 1: Number of static load instructions accounting for more than 90% of the data stalls (assuming an Itanium 2 memory hierarchy configuration.

Benchmark	Total Number of Static Loads	Number of Delinquent Loads
132.ijpeg	5079	43
164.gzip	1226	9
175.vpr	5289	30
181.mcf	515	14
183.equake	945	30
188.ammp	776	3
197.parser	4368	6
255.vortex	21298	361
256.bzip2	1064	28
300.twolf	10695	99

Because our methodology is profile driven, we recognize the importance of addressing the issue of profile sensitivity to different input workloads. That is, does the set of delinquent loads for an application remain relatively constant across different inputs? In our work, we focus largely on the SPEC benchmarks and we use one set of input workloads for profiling, and another distinct set of workloads for performance measurements. Furthermore, we have performed some empirical analysis to show that the set delinquent loads remains substantially constant for the majority of the SPEC benchmarks and input workloads.

2.2 Extraction and Scheduling

In this section we outline a simple algorithm for identifying a *program slice* [5] of the set of operations that contribute to the address calculation of a particular load instruction. Each slice is called a load dependence chain (LDC), or equivalently, a precomputation chain. Our algorithm assumes a model of computation (e.g., VLIW) where the compiler explicitly schedules operations and manages the architecture resources. The input to the algorithm is an already optimized and scheduled program expressed in a low-level IR (i.e., assembly or similar intermediate representation). The algorithm also inputs the memory profiling information to identify delinquent loads. The output is a program with embedded precomputation and prefetching instructions. The algorithm avoids some of the complexities attributed to slicing by leveraging the following program properties:

- Each function consists of a set of blocks or regions. Each block has single *entry* instruction, and it is the first operation in the block.
- Each operation o in a block is a member of a unique instruction word or bundle w_o. The bundles are issued in order to the processor, and the operations within a bundle are processed in parallel when the program executes. Without loss of generality, we assume that instructions within the same bundle do not have any dataflow dependencies.

```
    R1 = &list
    R5 = 0
loop:
    w1: R2 = R1 + 4
    w2: R3 = *[R2]
    w3: R4 = R1 + 8
    w4: R1 = *[R4]          # delinquent load
    w5: R5 = R5 + R3
    w6: br loop (R1 != NULL)
```

The load in w_4 is delinquent. Its LDC is:

```
    p2: R1 = *[R4]          # second LDC operation
    p1: R4 = R1 + 8         # first LDC operation
```

(a) Original code

```
    R1 = &list
    R5 = 0
    R6 = R1 + 8
loop:
    w1: R2 = R1 + 4;            p2: R7 = *[R6]
    w2: R3 = *[R2]
    w3: R4 = R1 + 8
    w4: R1 = *[R4]
    w5: R5 = R5 + R3;           p1: R6 = R1 + 8
    w6: br loop (R1 != NULL)
```

The LDC operations are scheduled and the destination registers renamed. Note in this example we do not propagate the original cyclic dependence in the LDC (i.e., R7 is not used in p_1). Hence we can convert p_2 to a normal prefetch instruction.

(b) Orignial code with an embedded LDC

```
    R1 = &list
    R5 = 0
    R6 = R1 + 8
loop:
    w1: R2 = R1 + 4;      p2: R7 = *[R6]
    w2: R3 = *[R2];       p3: R8 = R7 + 8
    w3: R4 = R1 + 8;      p4: R9 = *[R8]
    w4: R1 = *[R4]
    w5: R5 = R5 + R3;     p1: R6 = R9 + 8
    w6: br loop (R1 != NULL)
```

The LDC is unrolled twice (i.e., a new LDC is formed by chaining together two copies of the original LDC) and scheduled with register renaming. In this example we propagate the original cyclic dependence in the LDC (i.e., R9 is used in p_1).

(c) Unrolled precomputation with cyclic dependence

```
    R1 = &list
    R5 = 0
    R6 = R1 + 8
loop:
    w1: R2 = R1 + 4;      p2: R7 = *[R6]
    w2: R3 = *[R2];       p3: R8 = R7 + 8
    w3: R4 = R1 + 8;      p4: prefetch *[R8]
    w4: R1 = *[R4]
    w5: R5 = R5 + R3;     p1: R6 = R1 + 8
    w6: br loop (R1 != NULL)
```

In this example we do not propagate the original cyclic dependence and instead we convert the last LDC operation to a normal prefetch instruction.

(d) Unrolled precomputation without a cyclic dependence

Figure 1: Example pointer-chasing code and various PEPSE scenarios.

- The schedule time of a bundle w is $t(w)$, and hence the schedule time of an operation o is $t(w_o)$. Within a block, each bundle has a unique schedule time.

To identify delinquent loads, the algorithm identifies the set of memory operations that are responsible for 90% or more of the total program data stall cycles. For every load l in the set, the algorithm considers each operation p occurring in the bundles preceding w_l, and if there exists a dataflow edge from p to l, a speculative version of p is added to the LDC of l. The LDC is maintained as a queue with operations inserted at the head to preserve data dependencies. When the algorithm encounters a block's entry instruction, it may cross the region boundary to continue building the LDC. If a block has multiple (control) predecessors, the LDC is replicated, and for each predecessor, a path-specific LDC is built. For each LDC, the algorithm uses branch profiling to restrict the extraction to the two most frequently traversed paths. This threshold was selected empirically after numerous experiments, and it serves to limit the number of LDCs that may otherwise arise. The path that is traversed during the LDC construction defines the set of *hosts* within which the precomputation chain is later scheduled.

The algorithm terminates a precomputation chain according to several heuristics. One of the stopping conditions is triggered when there are no remaining dataflow edges to process. Another terminating condition is triggered when the length of the LDC reaches a predefined limit. The length constraint throttles the amount of precomputation. In this paper, we use an experimentally determined threshold of seven instructions; in later stages, we may expand the LDC subject to the desired prefetching distance and the availability of resources in the machine. A third stopping condition is triggered when a dataflow cycle is detected within the LDC. Cycles occur when the result of a delinquent load impacts its future address computation. This is a common occurrence in pointer-chasing applications and Figure 1(a) presents a common code fragment. In the figure, a loop iterates over a linked data structure, and in every iteration, it accesses some field in the current object (bundle w_2) and then dereferences a pointer (bundle w_4) to retrieve the next object in the list. In the figure, the delinquent load is w_4 and the LDC consists of the operations shown in bold.

When the algorithm stops building a precomputation chain, a scheduler begins the process of assigning the LDC instructions to available resource slots within the appropriate host regions. Scheduling begins in the block that was visited last, following the last processed bundle in the block. When all of the scheduling slots in that block are exhausted, the scheduler searches for additional resources in neighboring hosts—this is to avoid lengthening the program schedule. Note that in loop regions, a visit to neighboring blocks may result in traversing the loop *backedge*. It is therefore possible that scheduling begins in the tail end of a block, and continues at the head of the loop (or even the same block as shown in Figure 1(b)).

The scheduler begins with the first instruction in the LDC. When the LDC is cyclic, the chain is reversed and then scheduled. As each LDC operation is assigned to a resource slot, its destination operand is renamed and register allocated. The scheduler maintains a mapping of the old operand names to the new names, and propagates the new operands throughout the precomputation chain. In Figure 1(b), the scheduler assigns the first LDC instruction to bundle w_5 and the second LDC instruction is assigned to bundle w_1 at the top of the loop; note that if the backedge is not traversed, the loop length would have increased by one bundle. In the example, we assume that the processor can issue two instructions per bundle, and we use a semicolon to separate instructions within the same bundle. Note that when the first LDC instruction is scheduled, it is assigned a new destination register (R6), and the source operand of the second LDC instruction is updated to reflect the change. The renaming step assures that the registers that are updated by the precomputation do not affect the host program.

Finally, in the case of acyclic LDCs, when the last LDC instruction is scheduled, an appropriate prefetch operation is added. In cyclic LDCs, the last scheduled operation is a load instruction whose result is normally required by the first LDC instruction. And herein lies the challenge with prefetching pointer-codes: if the cyclic LDC dependence is preserved, then the result of the last LDC operation (a load) is committed to a register that is subsequently read by the first LDC operation. If the load suffers a cache miss that is not serviced by the time the dependent precomputation instruction is issued, the precomputation forces the processor to stall—an adverse outcome. In effect, the precomputation simply shifts the pattern of processor stalls to occur earlier in time. Hence what is needed is a mechanism for dynamically changing the behavior of the loads in the precomputation chains, and this is the topic of the next section where we describe an architectural mechanism for mitigating the challenges imposed by pointer-programs.

For the example shown in Figure 1(b), however, there is a simple solution. We can convert the load in w_1 to a normal prefetch operation. In general, we can always break cyclic LDC dependences in this manner. Unfortunately if the precomputation chain has two or more loads, we can not avoid the problem of stalling the processor since the result of some LDC-issued loads are necessary for the precomputation to proceed. In our experiments, we found that LDCs rarely have only one load instruction. And in many cases, LDCs will have several load operations as a result of LDC "unrolling", a transformation that our compiler will apply to better tailor the prefetching distance to the host program region. Specifically, in loop regions, the compiler must schedule a prefetch to issue in iteration k of a loop such that the data arrives in time for processing in a future iteration m. LDC unrolling is a lightweight transformation, and it simply entails creating γ copies of the LDC and chaining them all together as shown in Figure 1(c) for $\gamma = 2$. The unroll factor γ is ideally equal to $\lceil L_l/K \rceil$ where L_l represents the average miss latency of a delinquent load l, and K represents the length of the longest path in the loop region.

Referring to the example in Figure 1(c), observe that if the cyclic LDC dependence is preserved, and hence the result of the last LDC instruction (R9) is consumed by the first operation in the chain (in bundle w_5), then iteration k of the loop will access the k^{th} object in the linked list while the precomputation accesses objects $(2 \times k)+1$ and $(2 \times k)+2$. By contrast, if the last LDC operation is converted to a prefetch operation—thereby removing the cyclic dependence—the precomputation will prefetch object k and $k+1$ with every iteration of the loop. Thus the prefetching patterns are different. In the former case where the cyclic dependences are preserved, the precomputation can "run ahead" quite significantly. In the latter case where cyclic dependences are broken, the precomputation maintains a fixed "lookahead" distance (at the cost of some prefetching redundancy). Furthermore, in the latter scenario, the compiler does not need to properly initialize the live-in values to the LDC because the precomputation *synchronizes* with the appropriate values calculated in the original loop at every iteration (Figure 1(d)).

Note that the constant synchronizing of values between the main program and the precomputation serves to maintain a constant lookahead. In contrast, if a cyclic LDC dependence is preserved (as in Figure 1(c)), the precomputation risks running too far ahead of the program. Furthermore, in the context of the the sentineled prefetching scheme (next section), the precomputation may end up lagging behind its host.

Lastly, we note that it is possible to reuse values generated by the precomputation to avoid redundant calculation of the same values by the host region. We do not consider such an optimization at this time, although some recent work [12] has shown the idea is beneficial in the context of speculative helper threads.

3. SENTINELED PREFETCHING

In PEPSE, the precomputation chains consist of operations that are "binding", and therefore the operations are semantically equivalent to the normal instructions in the program. In pointer-heavy applications, instructions within the precomputation chain may stall the processor as they await the delivery of outstanding data. In this section we introduce the concept of sentineled prefetching for EPIC architectures (SPEAR) as an innovative approach to mitigating the challenges that arise in pointer-codes.

SPEAR instructions allow the program to adapt in response to dynamic memory events. Specifically, the execution of the precomputation chain is predicated on special-purpose *bits* of information that signal the program when certain memory requests are outstanding. The special-purpose bits are akin to event registers that might indicate a cache miss, or the "operand not ready bit" in conventional processors. In sentineled prefetching, the precomputation is *canceled* when any of the operations in the chain suffer a cache miss. In this scheme, the "guarding predicate" is controlled by the memory system which can distinguish normal program loads from speculative prefetch instructions. The latter are viewed as *informing* memory operations and require an extension to the instruction set architecture (ISA).

3.1 Informing Memory Operations

The assembly language syntax for an informing load is the same as a standard predicated Itanium load instruction:

```
(p) iLD rd = [rs]
```

Semantically, an informing memory operation clears a designated predicate register—called the *spearing bit* (sb)—when it suffers a cache miss. More specifically, the operation is described as follows:

- The execution of the operation is guarded by a predicate (p). When the predicate is cleared (p = 0) the operation is nullified (i.e., the state of the machine does not change). Otherwise, the informing load executes as follows when the predicate is affirmed (p = 1).
- If the address in rs hits in the TLB *and* in the primary data cache, then iLD behaves as a standard load instruction; a value is read from the address specified by register rs and placed in register rd.

- Otherwise, the iLD behaves as a non-binding prefetch instruction; the line containing the address specified by the value in register rs is moved to the highest level of the data memory hierarchy. In addition the spearing bit is cleared (i.e., set to FALSE as in sb = 0).

The following example illustrates the use of the iLD instruction. Suppose the LDC chain consists of the following instruction pattern:

```
LD  r1 = [r0]        # cache miss
...
ADD r1 = r1, 4       # processor stalls
LD  r2 = [r1]
...
```

In this first scenario, when the first load suffers a cache miss, an in-order EPIC processor will stall if the memory request is outstanding when the ADD operation is issued. However, with informing memory operations, the processor can ignore the operations that are no longer profitable (i.e., they may unduly stall the pipeline):

```
iLD r1 = [r0]         # cache miss, sb = 0
...
(sb)  ADD r1 = r1, 4  # operation ignored
(sb)  iLD r2 = [r1]   # operation ignored
...
```

In the above sequence, the processor ignores the embedded precomputation when the spearing bit is not affirmed. Thus when the first informing load results in a miss, the spearing bit is cleared and subsequent operations in the LDC are canceled. This scheme allows the precomputation to greedily proceed along, and affords a mechanism for prefetching data for any one of multiple loads in the LDC.

3.2 Architecture Implementation

The addition of a new predicate register in the form of the spearing bit is straightforward. The spearing bit may be implemented as global predicate registers, or as an implicit argument to SPEAR instructions. Either approach conserves the address space of predicate registers, although in each case, an explicit clearing instruction is necessary to reaffirm the spearing bit. Naturally, aliasing effects can occur if there are overlapping LDCs in the same program region. That is, the precomputation of one LDC may cancel the precomputation of another even though the two LDCs are distinct and do not share data. Our empirical analysis indicates that two precomputation chains often overlap, whereas three or more LDCs in the same program region rarely occur. Hence using two distinct spearing bits (that are exposed to the compiler) can reduce aliasing effects.

An alternate approach for sentineled prefetching employs a spearing bit per register, and in order to reduce the associated architectural complexity, an ISA may dedicate a subset of its register file for sentineled prefetching. In this approach, the mechanism to set and clear the spearing bits is similar to the propagation of exceptions flags with NaT bits in EPIC architectures. When an informing load is issued, the spearing bit associated with its destination operand is cleared. The bit is later set when the memory request is serviced, and in general, the bit is reaffirmed whenever a value is written to a register. A SPEAR instruction is canceled if any of the its source operands have a cleared spearing bits. When a SPEAR instruction is suppressed, the spearing bit of the destination register is cleared to propagate the cancellation of the precomputation chain.

Table 2: Benchmarks and input workloads used for profiling and evaluation.

Benchmark	Profile	Evaluate	Delinquents
101.tomcatv	train	ref	70
168.wupwise	train	ref	10
171.swim	train	ref	120
172.mgrid	train	ref	70
179.art	train	ref	100
183.equake	train	ref	60
189.lucas	train	ref	70
em3d	2000 2 50	30000 2 200	8
tsp	8000 0	8000000 0	10

Table 3: Measured `user` CPU times.

Benchmark	Time (in secs)		Speedup
	ORC	PEPSE	
101.tomcatv	45.6	29.3	1.56
168.wupwise	622	591	1.05
171.swim	317	230	1.38
172.mgrid	369	257	1.44
179.art	489	297	1.65
183.equake	471	220	2.14
189.lucas	366	307	1.19
em3d	6.39	5.29	1.21
tsp	80.7	75.9	1.06
total	2766.7	2012.5	1.37

4. PEPSE EVALUATION

This section evaluates our compiler-orchestrated prefetching system using the Open Research Compiler (ORC) [29]. ORC is an open source infrastructure for the Itanium Processor Family, and it includes a comprehensive set of components including a large suite of loop transformations, inter-procedural analysis and optimizations, region-based compilation, global instruction scheduling, software pipelining, data prefetching, and several other state-of-the-art technologies.

For our experiments, we used a 900 MHz Itanium 2 processor, with 16 Kb primary data and instruction caches, a 256 Kb secondary cache, and a 1.5 Mb tertiary cache. The first level caches have access latencies of of 1 cycle, and the second and third level caches have minimum access latencies of 5 cycles and 12 cycles respectively. The processor can issue up to six instructions at a time, and has an 8-stage pipeline with a minimum branch misprediction penalty of 6 cycles.

Our evaluation benchmark set (Table 2) consists of all of the SPEC benchmarks that we managed to successfully compile using our prototype compilation infrastructure. In all there were nine benchmarks: seven are from SPEC CFP and two from OLDEN. Two of the SPEC benchmarks—179.art and 183.equake—are C programs while the others are implemented in Fortran. In Section 5 we include results for SPEC CINT benchmarks.

4.1 Results

Table 3 reports the running time (in seconds) for each benchmark from the evaluation suite. For our baseline in column two, we used ORC to compile the benchmarks. The compiler applied several optimizations, including its most aggressive form of data prefetching. Column three reports the running time of the benchmarks when PEPSE was also enabled; note that because the software pipeliner interferes with our memory instrumentation routines, we disable software pipelining when enabling PEPSE. Our

Table 4: Number of static prefetch instructions for different prefetching strategies.

Benchmark	pft-1	pft-2	pepse
101.tomcatv	36	68	70
168.wupwise	101	101	10
171.swim	91	153	120
172.mgrid	71	79	70
179.art	36	40	100
183.equake	8	35	60
189.lucas	112	122	70
em3d	1	1	8
tsp	0	0	10

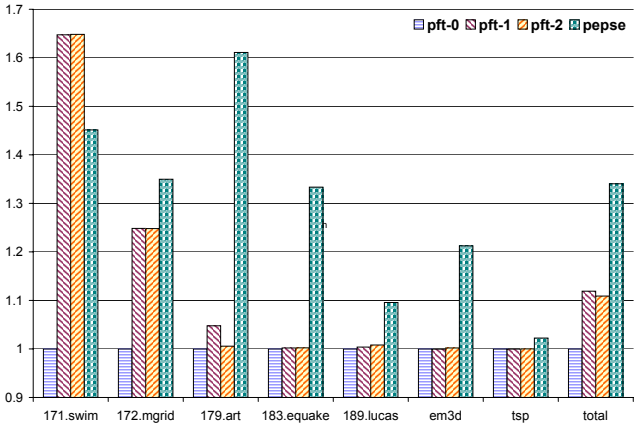

Figure 2: Measured speedup for different prefetching strategies.

results show a 27% reduction to the cummulative running time of the benchmarks (from 46 minutes to just under 34 minutes) when using our prefetching approach.

The prefetching technology in ORC is based on Mowry's doctoral thesis [28] and it is well suited for prefetching array references. The technique analyzes loop nests to determine which array references are likely to suffer a cache miss, and based on the discovered miss pattern, the algorithm decides what to prefetch and in which iteration of a loop to trigger the appropriate prefetch. We generated four compiled versions of each application: `pft-0`, `pft-1`, `pft-2` and `pepse`. The first applies all of the ORC optimizations (including software pipelining) with the exception of data prefetching. The second and third versions enable increasingly aggressive ORC prefetching optimizations (via `-LNO:prefetch=1` and `-LNO:prefetch=2` respectively, in addition to software pipelining). The last version is compiled with PEPSE enabled (and no software pipelining). For the SPEC benchmarks, ORC statically scheduled an average of 65 prefetch operations in the `pft-1` versions, and in the case of `pft-2`, there were 86 prefetch instructions on average. In the `pepse` version of the benchmarks, there were 74 static prefetch instructions on average, with a total of 18 prefetch instructions in the OLDEN kernels. The kernels use recursive (pointer) data structures and are not amenable to ORC's prefetching. Table 4 details the extent of the prefetching that is achieved by each prefetching strategy.

Next we ran all benchmarks and used PAPI [32] to monitor the performance counters in the Itanium. For each benchmark, we record the total number of cycles elapsed from the start of the application until it exits, along with the number of cache accesses and misses at every level of the hierarchy. Figure 2 reports the speedups achieved by each strategy (compared to `pft-0`). The data used to generate the figure was derived from the cycle counts reported by PAPI[1]. The last bar in the figure corresponds to the cumulative running time of the benchmarks. Note that the figure does not include data for `101.tomcatv` and `168.wupwise` because of infrastructure difficulties beyond our control.

Interestingly, both `179.art` and `183.equake` are array-heavy but the ORC prefetching strategy is not as effective as with the other SPEC CFP benchmarks. As we noted earlier, these two benchmarks are implemented in C and dynamically allocate their arrays. As a result, ORC may have made pessimistic assumptions during its reference-analysis stage. For example, of the top twenty five delinquent loads in `183.equake`, only eighty are targeted for prefetching in `pft-2`, whereas PEPSE targets all twenty five. Consequently, the PEPSE versions of the two benchmarks run 30 - 60% faster than the baseline (`pft-0`). In `179.art`, PEPSE reduces the tertiary cache misses nearly 10%.

It is also worth noting that the number of PEPSE-issued prefetch instructions greatly exceeds the number of ORC-issued prefetch requests: for `183.equake` and `189.lucas` there are 43× more prefetch requests, and for the other SPEC benchmarks, the number of requests vary by a factor of five on average. Surprisingly, less than one percent of the total prefetch instructions request data that are never used (or are evicted from the cache before they are referenced by the program). Two exceptions are `179.art` (6.3%) and `183.equake` (2.6%). In contrast, the fraction of the prefetch requests that hit in the primary cache can be much higher. In `183.equake` for example, 61% of the prefetch requests are redundant (i.e., they request data already in the cache) compared to only 13% for `pft-2`—note however that `pepse` issues 47× more dynamic prefetch requests in this case. In `172.mgrid`, PEPSE results in 2.3× more prefetch requests than `pft-2` and only 25% of the requests are redundant, compared to 37% for `pft-2`.

With the exception of `171.swim`, PEPSE delivers better performance than ORC's prefetching strategies. In `171.swim`, the impact of the software pipeliner is quite significant. Turning off software pipelining in `pft-0` slows `171.swim` down by 48%. A similar experiment with `pft-2` slows the application down 21%, and at that point `pepse` is 6% better in terms of running time. Hence we expect that as our compiler matures to simultaneously enable PEPSE and software pipelining, the overall speedups will increase.

Our ORC-based prefetching system remains under development, and we plan on more evaluations with a greater emphasis on SPEC CINT benchmarks. The current Itanium results are encouraging and provide concrete evidence that PEPSE is a viable optimization strategy. It does not rely on complex analysis of reference patterns and it is more robust in complex loops. In addition, the technique benefits from memory profiling and can issue a large number of prefetch requests with very low instruction-overheads. Furthermore, PEPSE is complementary to any existing prefetching scheme. For example the combination of PEPSE and ORC's prefetching results in a 10% gain in performance for `172.mgrid` compared to using either strategy exclusively.

5. PEPSE VERSUS SPEAR

Pointer-chasing programs pose a major challenge to prefetching systems. In the context of our work, precomputation chains with multiple load instructions can stall the processor if the result of a load is necessary to further propagate the precomputation. For example, if a load within the LDC suffers a cache miss that is not ser-

[1]We ran each benchmark version a total of three times and recorded the minimum cycle count that was reported.

Table 5: Benchmarks and input workloads used for profiling and evaluation.

Benchmark	Profile	Evaluate
101.tomcatv	train	lgred
168.wupwise	train	lgred
171.swim	test	train
172.mgrid	train	lgred
188.ammp	train	lgred
130.li	train	au, boyer, puzzle0
132.ijpeg	vigo.ppm	penguin.ppm
164.gzip	input.random	lgred.graphic
181.mcf	train	lgred
197.parser	train	lgred
256.bzip2	input.random	lgred.graphic
300.twolf	train	lgred

Table 6: HPL-PD processor model.

Functional Units	4 INT units, 2 FP units, 3 BRANCH units, 2 MEMORY units
Register Files	128 INT registers, 128 FP registers, 64 PREDICATE registers, 8 BRANCH registers
Caches	TLB (split) : 8 Kb I and D fully-associative, 32 bytes per line 8 cycles miss latency
	L1 (split) : 32 Kb I and D caches 4-way, 32 bytes per line 2 cycles hit latency (D cache)
	L2 (unified) : 1 Mb cache 4-way, 64 bytes per line 10 cycles hit latency, 200 cycles miss latency

viced in time for a subsequent LDC operation, the processor stalls and awaits data delivery. With sentineled prefetching, the precomputation chain abandons the prefetching process when it might otherwise stall the processor. In this section we compare the PEPSE and SPEAR strategies.

Our evaluation benchmark set consists of array-based and pointer-based benchmarks. In Table 5, we group the benchmarks into two categories. The first represents the array-heavy benchmarks drawn from SPEC CFP, and the second represents pointer-chasing benchmarks drawn from SPEC CINT. We used as many benchmarks as possible, limited only by what the Trimaran [38] infrastructure can successfully compile.

Trimaran is a publicly available compilation and simulation framework for EPIC research based on the HPL-PD [20] architecture (a precursor to the IPF). The compiler only provides a C front-end[2], and includes a number of classic and high level optimizations such as loop unrolling, copy propagation, common subexpression elimination, dead code elimination, aggressive register allocation, and software pipelining [33]. For our results, we do not use superblock [18] and hyperblock [27] optimizations; the former applies control speculation, the latter also adds predication to removes branches. During our experiments, we found that PEPSE provides better latency masking compared to these ILP optimizations which have a negligible impact on the memory system performance.

Our simulation environment consists of a cycle accurate HPL-PD simulator coupled with the Dinero IV [13] cache simulator. The in-order HPL-PD processor is configured as shown in Table 6. In addition, there is a BTB, and a two-level gshare branch predictor.

[2]We convert the Fortran SPEC CFP benchmark to C using `f2c`.

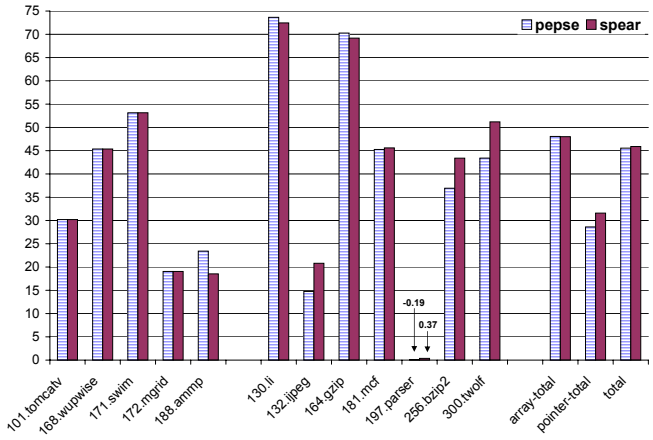

Figure 3: Percent reduction in processor data-stall cycles.

We have substantially modified the HPL-PD and Dinero simulators to model contention in the memory system and to perform cycle-accurate accounting of processor stalls. We used a *stall-on-use* model, meaning the processor pipeline only stalls when an instruction is ready to issue but its source operands are not yet available (due to an outstanding memory request). Additional instructions are not issued when the processor is stalled. Processing eventually resumes when the outstanding data items reach their destination. The caches in our simulated model are non-blocking, and we assume the memory units are pipelined such that they can each process a new memory request every cycle.

For some of the benchmarks in our evaluation suite, we use the MinneSPEC [23] reduced input workloads to reduce simulation time. The MinneSPEC workloads are recognized by SPEC and are distributed with Version 1.2 and higher of the SPEC CPU 2000 benchmark suite. The results that follow were obtained by *fully* simulating each of the benchmarks.

5.1 Precomputation Overhead

We quantify the runtime overhead associated with our methodology by measuring the total number of dynamic instruction bundles that are issued before and after prefetch orchestration. Each instruction bundle consists of a set of operations that issue simultaneously and execute in parallel. The extent to which the prefetch orchestration lengthens the program is reflected in the number of bundles that are processed. When the compiler embeds the LDCs in regions with an adequate number of available resources, the dynamic bundle count will not change relative to the baseline (where no prefetch orchestration is performed). Please note that SPEAR differs from PEPSE only in the semantics of the operations within the LDC. Therefore, both PEPSE and SPEAR employ the same LDC extraction, transformation, and scheduling algorithms, and thus they incur the same instruction overhead.

In our experiments, the average increase in the number of dynamic bundles was 3.66%. If we do not consider the static schedule length of host regions during LDC scheduling, we observe as much as a 32% increase in dynamic bundles. Our current scheduling algorithm uses a simple heuristic to avoid increasing the static length of a region. The scheduler compares the length of the precomputation chain to the number of available resource slots in the host regions, and it discards a load dependence chain if its computational requirements exceed the number of scheduling slots that are available.

5.2 Memory System Performance

Figure 3 reports the impact of PEPSE and SPEAR on the number of cycles that the processor stalled awaiting data delivery. The striped bars represent the percent reduction in stall cycles when PEPSE is used (compared to the number of stall cycles incurred by the processor in the absence of precomputation and prefetching; greater percentages are better). The solid bars represent the percent reduction in stall cycles when sentineled prefetching is used (compared to the same baseline). We aggregate the data into three categories. The first group (i.e., five sets of bars on the left side of the graph) consists of the array-heavy SPEC CFP benchmarks. The second group (i.e., the middle seven bars) consists of the pointer-heavy SPEC CINT benchmarks. The last three sets of bars represent the percent reduction in total stall cycles for the array-based (`array-total`) and the pointer-based (`pointer-total`) categories, as well as the overall reduction in stall cycles for all of the benchmarks (`total`).

As might be expected, SPEAR does not have an edge in array-based codes, but does have a slight advantage (4.17% on average and 13.75% in the best case) in the pointer-heavy benchmarks. In the case of `197.parser`, PEPSE results in a small performance degradation (0.19% more data stall cycles), and SPEAR shows virtually no improvement (0.37% fewer stall cycles). The degradation is caused by processor stalls induced by the precomputation chain, and worse, they are due to spilling and restoring registers used by the precomputation—particularly along function boundaries. We suspect that better inter-procedural cooperation of the precomputation chains can help in this case. In SPEAR, the precomputation induced spill and restore operations are nullified.

The performance benefit of PEPSE is quite significant, even in pointer-codes: nearly 30% fewer data stall cycles. Thus, when the precomputation stalls the processor—and as long as the correct addresses are calculated—it shifts the pattern of stalls to occur earlier. The performance benefit of SPEAR is interesting in a few cases (`132.ijpeg`, `256.bzip2`, and `300.twolf`) and negligible in others. As we investigated, we found two interesting issues.

First, when SPEAR does not outperform PEPSE, it is due to the behavior of load operation within the computation chain. If the LDC contains a load operation q whose average miss latency L_q is less than the miss latency L_l of the target delinquent load, PEPSE wins almost exclusively. This is because if the precomputation waits for the results of load q, the potential reward is L_l, for a total positive gain of $L_l - L_q$. If on the other hand, the average miss latency of the delinquent load is less than some other load within the LDC, SPEAR holds the advantage since it will not wait for the result and cancels the remaining precomputation; concomitantly, the outstanding memory transaction becomes a prefetch request. This phenomenon arises when the load dependence chain has more than one load operation, and suggests there are opportunities for selectively using informing loads and normal binding loads within the same precomputation chain. We refer to LDCs with both PEPSE and SPEAR operations as *hybrid* LDCs. Our initial results which experiment with various heuristics show some promise. The simplest approach calculates an estimated wait-time for an LDC-initiated memory request, and when the wait-time exceeds the potential for latency masking, the remaining LDC operations are converted to SPEAR instructions. This strategy results in better performance in some benchmarks. For example, in `256.bzip2`, hybrid LDCs deliver 15% fewer stalls compared to PEPSE, and that is 5.5% better then SPEAR.

The second noteworthy observation involves cooperative prefetching when LDCs that are not mutually exclusive end up in the same program region. When LDCs overlap (in terms of the precomputation they carry out), they are often nested in pointer-chasing loops with complex control flow. We found that SPEAR performs better in regions where overlapping and cooperative LDCs occur. We have found that the interactions can be quite complex, and further investigation is needed.

6. RELATED WORK

Due to the volume of research in the field, we briefly mention some of the most recent and relevant work. However, many of the related studies fall short of our proposed methodology which (i) combines an existing framework of speculative execution with prefetching to (ii) ensure accurate and timely precomputation of load addresses with (iii) little resource overhead.

6.1 Predictive Prefetching Schema

Data prefetching is the early delivery of data to the processor or nearby storage. The strategy is effective as long as accurate information is available about future memory references (e.g., location in memory and time of access), and many techniques are designed to either statically [7, 22, 28, 30, 40] or dynamically [8, 14, 19, 35] predict future memory references and drive the optimization. Static prediction generally requires little architecture investments beyond an instruction set architecture (ISA) that supports data prefetching. By contrast, dynamic prediction mechanisms often require non-trivial architecture modifications [8, 14, 19, 35]. While the strategies are sometimes effective, they are vulnerable to the irregular memory access patterns that are common to pointer-heavy applications where the lack of interaction between the application, the operating system and the architecture, leads to complex memory reference patterns that are not amenable to prediction. Consequently, the wrong data is prefetched and resources are wasted.

6.2 Prefetching via Precomputation

Some attractive alternatives to prediction-based prefetching include precomputation-based strategies [4, 9, 10, 21, 24, 25, 39, 41], where subsets of an application are redundantly executed by specialized hardware components to provide the information necessary to perform data prefetching. Such precomputation is often effective in the timely discovery of future memory references, but at a substantial cost in architectural investments which can include dedicated micro-engines, threads, or even co-processors. Furthermore, an actual implementation of helper-threads for prefetching (using a hyperthreaded Pentium 4), has shown that synchronization costs between the helper-threads and the main program are non-deterministic, ranging from ten thousand to thirty thousand CPU cycles, and that it is only with special hardware support that the overhead can be reduced to a few thousand cycles [21]. This casts serious challenges on achieving the claims of various helper-thread techniques. Hence, it is desirable to implement a prefetching strategy capable of acquiring substantial foresight about future memory reference patterns, but with the architectural complexity of static prediction techniques. Toward this end, program embedded precomputation and sentineled prefetching are two important contributions. The ideas proposed in this paper are applicable in high-end EPIC architectures as well as in embedded VLIW processors.

6.3 Speculative Execution and Scheduling

Speculative execution was present in some of the earliest dynamically scheduled processors [37], and subsequent research [36] demonstrated that exposing speculative execution to a compiler can boost the performance of superscalar architectures. Others later proposed a different scheme for speculative execution that requires the scheduling of sentinels to guard against exceptions [26]. A

variant of this scheme was incorporated into the HPL-PD architecture [20] from which the IPF borrowed heavily. The primary aim of speculative execution as envisioned by previous research was to improve instruction level parallelism.

Some other related techniques include speculative load instructions [34], advanced loads that perform dynamic memory disambiguation [15], modulo scheduling [33], and load sensitive scheduling [1, 3]. Modulo scheduling is the overlapping of loop iterations to boost ILP and mask memory latencies. Load sensitive scheduling attempts to maximize the distance between load instructions and their dependents. The scheduler focuses on memory operations that are not on the critical path, and it may inject prefetch instructions when possible.

Our methodology differs from the aforementioned optimizations in that PEPSE is the redundant execution of program operations for the purpose of early address generation and prefetching. In PEPSE we can also target operations that are on the critical path. In cyclic program regions, the embedded precomputation may prefetch data several iterations ahead of their actual use. Our methodology also exploits the available scheduling slots in an already optimized program to minimize instruction overhead. In addition, we provide concrete validation of our optimization using an Itanium 2 machine, and earlier, we compared our results to software pipelining.

As an experiment, we also compared PEPSE to an in-house implementation of load sensitive scheduling within ORC. The scheduling algorithm leverages memory profiling information to annotate the dataflow edges between a load and its dependents with the load's average miss latency. The annotations enable the scheduler to make better scheduling decisions as it tries to maximally separate load operations from their successors. Our implementation of a load sensitive scheduler did not yield very significant performance improvements compared to the baseline compiler (i.e., ORC with `-O3` optimizations). In the best case, the running time of `101.tomcatv` (a vectorized mesh generator with lots of ILP) was reduced by 5%. In the worst case, `183.equake` ran nearly 10% slower. For the other benchmarks from the evaluation suite (Table 2), there was negligible performance impact (i.e., the running times differed by less than one percent).

Our proposed sentineled prefetching scheme for EPIC architectures is inspired by previous work on informing memory operations [16, 17]. However, instead of using the informing load instructions to invoke special handling routines upon a cache miss, we use SPEAR instructions as a way to predicate the execution of the embedded precomputation chains.

7. CLOSING REMARKS

In this paper we advocate the role of a compiler as an orchestrator of prefetching strategies via speculative and predicated execution. We propose the concept of program embedded precomputation via speculative execution (PEPSE) as a new methodology for data prefetching. PEPSE uses simple and lightweight algorithms to identify the load dependence chains (LDC) of delinquent loads. LDCs serve as the building blocks for orchestrating address precomputation and prefetching, and represent precomputation chains that are embedded within host regions of the application—such that as the program executes, the LDC operations also execute and carry out address generation and prefetching. Using the Open Research Compiler (ORC) for the Itanium Processor Family, we demonstrate that PEPSE is a viable optimization strategy, and delivers as much as a 38% reduction in execution time when compared to the ORC optimizations that include software pipelining and prefetching technology.

We also propose an innovative sentineled prefetching strategy for EPIC architectures to mitigate some of the PEPSE limitations in pointer-chasing applications. We refer to the strategy as SPEAR, and using a cycle accurate simulator, we show that it can also serve as a viable prefetching strategy. Our extensive simulation and experimental analysis has shown that not only are both strategies effective, but there are several important research directions worthy of pursuit. One idea is combine the predicated and informed nature of SPEAR operations with the PEPSE concepts to better tailor the prefetch orchestration to the program. Other research can focus on the orchestration of cooperative LDCs for the sake of better latency masking.

We believe that compiler-orchestrated prefetching, via speculation and predication, can effectively mask long memory latencies with little instruction overhead. Furthermore, we believe the techniques presented in this paper can readily contribute to the performance of processors today.

8. ACKNOWLEDGMENTS

The authors thank Jessica Baugh, Mark Stephenson, Bill Thies, and the anonymous reviewers for their comments on previous drafts of this paper. We also thank George Dan Ciobanu who helped with the Itanium experiments. Hariharan Sandanagobalane was partially funded by A*STAR Grant 012/106/0046. This work was supported in part by DARPA contract F30602-00-2-0564.

9. REFERENCES

[1] S. Abraham and T. Johnson. Load sensitive scheduling. Personal Communication, HP Labs.

[2] S. Abraham and B. R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, HP Labs, Dec. 1994.

[3] S. Abraham, R. Sugumar, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, Dec. 1993.

[4] M. Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of 28th International Symposium on Computer Architecture*, July 2001.

[5] H. Argawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[8] M. Charney and A. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb. 1995.

[9] J. Collins, D. Tullsen, D. Wang, and J. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[10] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Long-range prefetching of delinquent loads. In *Proceedings of 28th International Symposium on Computer Architecture*, July 2001.

[11] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher. Deli: A new run-time control point. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002.

[12] Z. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and T. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 71–81, 2004.

[13] J. Edler and M. Hill. Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/˜markhill/DineroIV/.

[14] J. Fu and J. Patel. Stride directed prefecthing in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, Dec. 1992.

[15] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 183–195, 1994.

[16] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. Informing loads: Enabling software to observe and react to memory behavior. Technical Report Technical Report No. CSL-TR-95-673, Stanford University, 1995.

[17] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. Informing memory operations: providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[18] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, Jan. 1993.

[19] D. Joseph and D. Grunwald. Prefetching using Markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, Feb. 1999.

[20] V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-9380 (R.1), HP Labs, Feb. 2000.

[21] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Gikar, and J. Shen. Physical experimentation with prefetching helper threads on Intel's Hyper-Threaded processors. In *Proceedings of the Second Annual IEEE/ACM International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, Mar. 2004.

[22] A. Klaiber and H. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, 1991.

[23] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, 2002.

[24] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128, June 2002.

[25] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[26] S. Mahlke, W. Chen, R. Bringmann, R.Hank, W. Hwu, B. R. Rau, and M. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4), Nov. 1993.

[27] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.

[28] T. Mowry, M. 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*, pages 62–73, Oct. 1992.

[29] Open Research Compiler for the Intel Itanium. http://ipf-orc.sourceforge.net.

[30] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Nov. 1995.

[31] V. Panait, A. Sasturkar, and W. Wong. Static identification of delinquent loads. In *Proceedings of the Second Annual IEEE/ACM International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, Mar. 2004.

[32] PAPI: Performance application programming interface. http://icl.cs.utk.edu/papi/.

[33] B. R. Rau. Iterative modulo scheduling. Technical Report Technical Report HPL-94-115, HP Labs, Nov. 1995.

[34] A. Rogers and K. Li. Software support for speculative loads. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1992.

[35] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 42–53, 2000.

[36] M. Smith, M. Lam, and M. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, 1990.

[37] R. Tomasulo. An efficient hardware algorithm for exploiting multiple arithmetic units. *IBM Journal*, 44-5:25–33, Jan. 1967.

[38] TRIMARAN: An infrastructure for research in instruction level parallelism. http://www.trimaran.org.

[39] S. Vanderwiel and D. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the International Conference on Computer Design*, pages 372–377, 1999.

[40] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 210–221, 2002.

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