Ameliorating the Memory Bottleneck by Speculative Execution: The Load Dependence Graph

RODRIC M. RABBAH, MONGKOL EKPANYAPONG, AND WENG-FAI WONG
Georgia Institute of Technology

The effective use of the memory hierarchy is crucial for achieving good performance in all modern processors. In many applications, however, a very small number of delinquent memory operations are responsible for the bulk of the cache misses incurred by an application. In this paper, we propose an innovative, lightweight, and effective compiler framework that deals with delinquent loads by leveraging architectural features that exist in the class of Explicitly Parallel Instruction Computing (EPIC) processors – of which the Intel Itanium is an important representative. We will introduce the concept of a Load Dependence Graph and show how we can effectively make use of it to insert prefetching code in an application such that (*i*) *the prefetch is highly precise,* (*ii*) *the technique is applicable to both numerical and pointer-intensive applications,* (*iii*) *we require no new hardware support, and* (*iv*) *the overhead is negligible. Our results show that when implemented in the* TRIMARAN *EPIC research infrastructure, we achieve a speedup of 26% on average. Experiments conducted on an Itanium processor demonstrate a 11.67% reduction in total execution time. Similar experiments performed on an Itanium* II *processor showed an average performance improvement of 7.14%. Our application test bed is drawn from the well-known* OLDEN *and* SPEC2000 *suites of integer and floating point benchmarks.*

Categories and Subject Descriptors: B.3 [**Hardware**]: Memory Structures; D.2 [**Software**]: Software Engineering

General Terms: algorithms, performance, design

Additional Key Words and Phrases: speculation, prefetching, precomputation

1. INTRODUCTION

In a 1946 preliminary discussion on the logical design of an electronic computing instrument, Burks, Goldstine, and von Neumann [3] first remarked on the importance of a storage or memory as a critical component in the design of a "satisfactory general-purpose computing architecture". Furthermore, they "recognized the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible". Today, roughly fifty five years later, the memory hierarchy is a ubiquitous component available in almost all computing platforms; contemporary memory hierarchies are mainly comprised of several storage elements called *caches*. For example, the Intel Itanium processor consists of a three-level cache hierarchy: a 32 Kb primary cache, a 96 Kb secondary cache, and a tertiary cache as large as 4 Mb [13], with access latencies ranging from 14 to 30 cycles. Such long access latencies dramatically decrease processor throughput and hence magnify the need for latency masking techniques. This is especially true in the context of the explicitly parallel instruction computing (EPIC) platforms which afford significant and often massive instruction level parallelism (ILP).

Explicitly parallel processors are largely derived from the very long instruction word (VLIW) architecture paradigm[1]. They continue to gain wider acceptance and play a significant role in various aspects of today's computer industry, ranging from high end server platforms such as the Itanium Processor Family (IPF) [13], to digital signal processing engines such as the TI-C6x processors [31], to custom computing systems such as the Trimedia VLIW products [33]

[1]EPIC will be used to implicitly to include VLIW.

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

This work is supported in part by DARPA contract F30602-00-2-0564, Hewlett Packard Laboratories and Yamacraw.

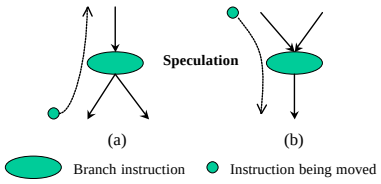


Figure 1. Examples of control speculation.

and the HP-STMicroelectronics Lx processors [10]. Unlike high performance superscalar processors which hide many architectural details from the compiler, EPIC processors advocate a new paradigm whereby the architecture is exposed to the compiler via extensions to the instruction set architecture (ISA). Thus, the compiler is capable of communicating memory management hints to the processor through special operations, and hence direct the movement of data across the memory hierarchy [13].

This paper proposes a novel *compiler* algorithm to effectively manage the memory hierarchy of an EPIC processor such as the Intel Itanium (it is also useful in the context of the wide range of simpler VLIW processors mentioned eariler). In particular, we shall describe a methodology which utilizes the ISA to *speculatively* precompute the addresses of future memory references and initiate their delivery to lower levels of the hierarchy, ahead of actual processor requests. To the best of the authors' knowledge, this is the first such successful technique for EPIC architectures. In Section 5, we shall provide detailed results and analysis in the context of the experimental HPL-PD [16] microarchitecture, a precursor to the IPF. Such results offer significant insight and allow for fine grained investigation of the proposed methodology. However, we recognize the importance of concrete evidence in support of the proposed schema, and hence we shall also present some results gathered using actual Itanium processors (Section 6). Briefly, the performance speedup for an Itanium-like HPL-PD processor is 26% on average for seven well-known OLDEN and SPEC2000 benchmarks. Preliminary experiments conducted on an Itanium machine produced an average speedup of 11.67%. Similar experiments carried out on an Itanium II showed an average reduction of 7.14% in execution time.

While the technique proposed in this paper may be generally classified as a *prefetching* schema, it differs from traditional prefetching methodologies in four significant ways:

— **Precomputation based prefetching.** Unlike traditional pre-fetching techniques which attempt to predict future memory references [11, 6, 30], the proposed precomputes future memory references using the program itself. Hence, the number of useless prefetch instructions due to mispredictions is drastically reduced.

— **Simultaneously applicable to array and pointer based applications.** Because conventional prefetching techniques are predictive in nature, they are generally vulnerable and do not perform well in the context of irregular memory access patterns (MAPs) intrinsic to real-world, pointer-centric dynamic applications. In other words, the strategies that work so well with regular, array based applications waste bandwidth and pollute caches when data is unnecessarily prefetched. By contrast, the proposed optimization is simultaneously applicable to array and pointer intensive applications. Furthermore, we will demonstrate the technique is robust in the presence of loop control structures such as *for* and *while* loops.

— **Leverages existing architectural features.** Whereas previously proposed prefetching schemes [4, 5, 14, 15, 28, 30, 18] require additional, and often complex, architectural support, the methodology proposed in this paper leverages existing architectural features that are visible to the compiler via the instruction set architecture (ISA). In particular, our innovative technique effectively utilizes control and data speculation to assure very high degrees of prefetching accuracy. Speculation is a feature of modern optimizing compilers introduced to improve ILP and overcome legacy performance limitations such as long branch. When an instruction is *control speculated*, it is moved above a branch and is now unconditionally executed, whereas previously it was executed conditionally (Figures 1(a) and 1(b)). Since speculation may not always be safe and hence may cause traps or exceptions during execution, the technique requires

hardware support for error recovery. To this end, both HPL-PD and Itanium provide light-weight support to enable speculative execution.

— **Incurs little instruction overhead.** The proposed also differs from past methodologies in that it incurs negligible instruction overhead while delivering significant performance benefits. Thus, the optimization serves to ameliorate the speed gap which exists between processors and memories, and helps deliver the promise of Moore's Law to the end-user.

We begin with an overview of our methodology, and then we detail the algorithm and some heuristic variants in Section 3. Section 4 describes our evaluation framework and Section 5 provides an in-depth analysis of our main results using the experimental HPL-PD architecture. In Section 6 we report results obtained directly using an Itanium processor. Finally, Section 7 discusses related works, and Section 8 summarizes and concludes the paper.

2. OVERVIEW OF OUR METHODOLOGY

A successful compiler based approach to prefetching requires a schema that (i) carefully identifies load instructions likely to benefit from prefetching, (ii) assures the timely availability of the data address to be prefetched, and (iii) issues a prefetch instruction with negligible resource contention or overhead. A naive prefetching strategy which excessively prefetches data poorly utilizes resources and is likely to degrade performance. In other words, load addresses that are usually cached (*hit*) at lower levels of the hierarchy experience short turnaround latencies, and thus obviate the need for prefetching. On the other hand, load operations whose target addresses typically *miss* the cache require dramatically longer delivery times, and hence ought to be prefetched; we shall refer to such loads as *delinquent* loads.

2.1 Identifying Delinquent Loads

Our framework identifies delinquent load instructions in a given program $\mathcal{P}$ via application *profiling*, an increasingly popular technique in the context of feedback driven optimizations [6, 35, 18]. For the purposes of this paper, we identify loads with miss ratios greater than a chosen threshold as delinquent and hence isolate them for prefetching.

2.2 Early Address Generation

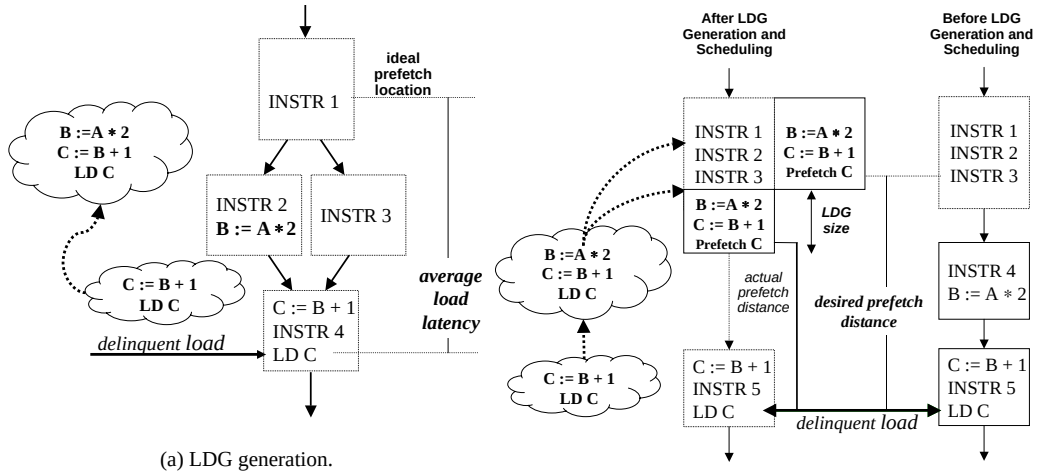
Given a framework that properly identifies delinquent loads, it is also necessary for the target data address to be available far in advance of the actual load, so as to maximize the masking of a long access latency. Unfortunately, it is often the case that the address computation immediately precedes the load operation and thus there are very few opportunities for prefetching. A comprehensive study of previous work reveals several hardware based approaches for the advanced generation of a load address [1, 25]. To a large extent, such techniques often require extensive architectural modifications. By contrast, we achieve the same stated goal of early address generation using (i) existing microarchitectural features and (ii) a light-weight compiler algorithm, with a running time linear in the size of the data dependence graph of the program.

2.3 Prefetch Initiation

Although several compiler based prefetching strategies have been proposed in the past [17, 19, 20, 22, 26], most suffer from three main limitations. Specifically, they may (i) increase the instruction overhead, (ii) increase register pressure and (iii) mispredict data usage and hence unnecessarily issue prefetch instructions. In contrast, the algorithm proposed here is precomputation-based and is therefore less prone to mispredictions. It also exploits available ILP and to avoid other undesirable effects such as increased schedule length.

2.4 The Algorithm

The general algorithm first identifies delinquent load operations. Subsequently, for each delinquent load *i*, we construct its data dependence graph of at most *N* vertices or operations. We shall refer to the graph as the *load dependence graph* or LDG. The LDG is a *program slice* [2] of the set of instruction that contribute to the computation of the address for



(a) LDG generation.

(b) Speculative scheduling of an LDG.

Figure 2. LDG generation and scheduling.

the delinquent load. In Fig. 2(a), we illustrated an example LDG. Next, we insert the operations from the slice into the program $\mathcal{P}$ such that the last LDG instruction completes at least L_i cycles prior to the delinquent load, where L_i is the average miss latency for load i. In essence, we statically insert a set of speculative operations along various control paths in the program to precompute the address of a particular load, thereby creating enough slack for inserting an effective prefetch instruction. Fig. 2(b) shows how an LDG may be statically scheduled in a program.

It is evidently important that the speculation must be carefully throttled so as not to increase resource contention. In other words, the extent to which speculative prefetching is effective is sensitive to the available resources, namely registers and functional units. In Section 4 we demonstrate that our methodology successfully exploits available ILP and does not adversely impact the computation time. More importantly, we demonstrate that the trend is consistent across the benchmarks used in this paper.

3. LDG FRAMEWORK

Throughout this section, we assume the reader is familiar with standard control and data flow analysis techniques [23]. We also assume that for each load operation i occurring in $\mathcal{F}$, the miss ratio R_i and average latency L_i are available. We perform the LDG optimization after pre-pass scheduling. The impact of compiler phase ordering on the current optimization is the subject of ongoing research and is well beyond the scope of this paper. As we assume scheduling has already taken place, we note the following:

— Each function consists of a set of blocks or regions.

— Each operation i in a block is a member of a unique instruction word or bundle w_i. The bundled operations will be issued in parallel.

— The schedule time of a bundle w is t_w, and hence the schedule time of an operation $i \in w$ is $t_i = t_w$.

We present the algorithm in three parts. First, we describe the algorithm driver. Next, we describe the LDG generator, and finally we outline the LDG scheduler.

Input: profile-annotated and scheduled function $\mathcal{F}$
Output: function $\mathcal{F}$ with incorporated LDGs

```
01. for each load i in F do
02.     if R_i ≥ miss threshold then
03.         insert i into it a new LDG
04.         mark all blocks in F as not visited
05.         Generate-and-Schedule-LDG (F, LDG, i)
06.     end if
07. end for
```

Figure 3. Top level driver for generating and scheduling LDGs.

3.1 LDG Driver

The driver algorithm for generating and scheduling LDGs is outlined in Fig. 3. To build the LDG for a load i occurring in a function $\mathcal{F}$, we process the data dependence graph for the function bottom-up and starting with operation i (Fig. 2). Since we are only interested in delinquent loads, we first check if the *relative* miss ratio R_i is greater than a predefined *miss threshold*. We define R_i as the number of cache misses for i divided by the total number of cache misses incurred in $\mathcal{F}$. When the miss ratio exceeds the threshold, we begin generating an LDG for i. The miss threshold serves mainly to throttle the amount of prefetching the framework achieves. In this paper, we use an aggressive threshold of 5%.

3.2 LDG Generator

The algorithm for generating LDGs is as follows. For each operation j occurring in the bundles preceding the current instruction word, it checks for the existence of a data-flow path from j to i. If a dependence exists, a speculative version of j is added to the LDG. As our goal is the early generation of a load address, it is likely that the operations of interest are either simple arithmetic operations, or in some pointer-intensive applications, load operations. Both the HPL-PD and the Itanium processors offer speculative versions of these instructions. The load dependence graph is maintained as a queue, and operations are inserted at the head of the queue to preserve data dependencies. While building the LDG, the algorithm may need to cross a block boundary, and hence, for each predecessor block it builds a path-specific LDG. Evidently, without path profiling or some form of path pruning, the strategy may lead to an excessive number of LDGs. To this end, we performed branch profiling and selected various branch-frequency thresholds to exclude certain control paths from the LDG generation. The results in this paper are based on a 20% control edge frequency, meaning a branch edge must have been taken at least 20% of the time in order to be considered during LDG generation. The algorithm terminates when any of the following *conditions* are satisfied:

— *Condition 1: Desired prefetching distance reached.* When the total data-flow path distance from the current bundle to w_i is equal to L_i + |LDG|, the algorithm has reached the desired prefetch insertion point. Recall that L_i is the average miss latency of the delinquent load. However, since the prefetch requires the timely availability of the target address, we require at least |LDG| additional cycles to compute it. Hence, the algorithm must continue its upward traversal of the program graph for an additional |LDG| cycles, and incorporating additional operations along the way as necessary. When this condition is satisfied, LDG generation is terminated and scheduling begins.

— *Condition 2: Out of budget.* This condition enforces a budget which prevents code explosion and minimizes resource contention. When the number of instruction words (|LDG|) in the graph has reached a predefined limit, LDG generation is terminated and scheduling begins. The results in this paper are based on a budget size of seven instruction bundles.

— *Condition 3: A branch and link (BRL) is encountered.* We do not yet consider inter-procedural optimizations. Hence when a BRL (i.e., function call site) is encountered, LDG generation is terminated and scheduling is initiated, ensuring

Input: function $\mathcal{F}$, LDG, and the operation c where scheduling is to begin
Output: function $\mathcal{F}$ with LDGs

01. perform register live range analysis
02. create a *map* and initialize to be empty
03. process each operation j in the LDG from head to tail
04. find the earliest available scheduling slot occurring
    at time $t \geq t_c$ along the *visited* blocks
05. $d \leftarrow$ destination operand of j
06. find an available register r
07. use r as the new destination register for j
08. for each source operand s of j do
09. if $s \in map$ then replace s with $map(s)$
10. $map(d) \leftarrow r$
11. end for

Figure 4. Algorithm for scheduling and register allocation LDGs.

the prefetch instruction is scheduled after the BRL. This is to avoid the working set of the callee evicting the prefetched data from the cache.

— *Condition 4: All paths exhausted.* When either all predecessor blocks are visited or the current block is the first region in a function, LDG generation is terminated, and scheduling performed. As noted earlier, we do not yet consider inter-procedural analysis, and we anticipate significant benefits from such analysis (elaborated in Section 5).

The four conditions above trade-off higher degrees of prefetching for reduced timeliness. In Section 5 we quantify how often each of the above conditions is encountered for the set of benchmarks used in this paper.

3.3 LDG Scheduling

The algorithm for scheduling the LDG is outlined in Fig. 4. Note that as part of the LDG scheduling, it may be necessary to perform simple register allocation and renaming as shown in Figure 5. In almost all cases, we successfully scheduled and register allocated the LDG instructions without (i) increasing the static schedule length of a block or (ii) increasing register pressure; should the resource allocator run out of register, then it will insert register spill and restore operations as a traditional register allocator would do. Results that support these claims will be presented in later sections. Also note that it may be necessary to cross region boundaries when scheduling the LDG. The extent to which this occurs depends on the size of the load dependence graph and the size of each candidate block along a marked control flow path. The reason for crossing region boundaries is to avoid unnecessarily lengthening the static schedule length of a block since there may be available resources in adjacent regions (and the LDG is mostly comprised of straight-line code).

3.4 LDGs and Compiler Optimizations

So far we have describe a schema for identifying delinquent loads, generating their load dependence graph and speculatively scheduling them in the original program. Many loads in an application may be, and indeed are, characterized as delinquent loads. Hence, the rate of early load generation and prefetching must be carefully managed so as not to increase resource contention and lengthen computation time. We have found however that the proposed methodology successfully schedules many LDGs without compromising computation time (to be detailed in Section 5). We have also observed that classic compiler optimizations can often be applied when several LDGs overlap. Additionally, we do not consider all LDGs to be of equal weight. That is, we prioritize the candidate LDGs based on several criteria, such as the miss profile of a delinquent load, its average latency, as well as the resource requirements and expected payoffs – as is elaborated in the following section. Thus, we may prune the set of candidate LDGs when resource contention becomes unacceptable.

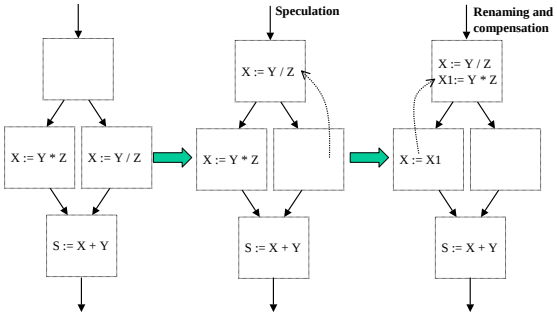


Figure 5. Register renaming to support LDG scheduling.

3.5 Pruning The List of LDGs

In order to curb the total number of LDGs that are incorporated into a function – and hence limit resource contention and the possibility of lengthening the computation time of a function – the proposed framework relies on the following heuristic to eliminate non-profitable LDGs or LDGs with substantial resource requirements. For a given delinquent load i, the heuristic computes its LDG benefit factor γ_i as follows:

$$\gamma_i = \frac{d_i \times \text{available resources}}{|\text{LDG}|} \tag{1}$$

where d_i equals the data-flow distance from the site where the LDG generation algorithm terminated to the delinquent load. Intuitively, the algorithm gives higher priority to LDGs which mask longer access latencies, since the payoff may be exceptionally great. Thus, under certain circumstances, the algorithm will lengthen the static schedule length of a function in exchange for significant latency masking capabilities. Furthermore, if there are many resources available – that is the intrinsic instruction level parallelism of the function is low – the LDG scheduling algorithm will likely succeed in finding all the resources required to schedule an LDG. The heuristic also considers the LDG size in making its pruning decisions; the LDG size is the number of instruction bundles currently in the specified slice. If the number of operations required to precompute the prefetch address is high, then it is necessary to balance the resource requirements of the LDG against its expected returns. Currently, the algorithm generates a list of all LDGs for a given function and considers each as a candidate for scheduling. Next, the listed is sorted using the respective benefit factor of each LDG. The list is then pruned to remove some of the lowest ranking LDGs (Section 5 quantifies the number of pruned LDGs per benchmark). Finally, the scheduling of each remaining LDGs is initiated.

3.6 Induction Unrolling

In this section, we introduce a technique for prefetching data across multiple loop iterations. We call this technique *induction unrolling.* An example of how this is done is shown in Fig. 6. In the baseline LDG technique, the LDG for the load instruction of the original loop in Fig. 6(a) is used to produce the additional two instructions in Fig. 6(b) (the LDG operations are shown in bold font). The additional instruction initiate prefetches one iteration ahead of the actual loop. In induction unrolling, the LDG consisting of the induction variable is unrolled[2] (see Fig. 6(c)). When constant folding is applied to the induction unrolled loop, we end up with just two more instructions being added to the loop that will prefetch data four iterations ahead of its use (Fig. 6(d)). In general, however, it is possible that too many instructions will

[2]Unlike standard loop unrolling, we do not unroll the entire loop.

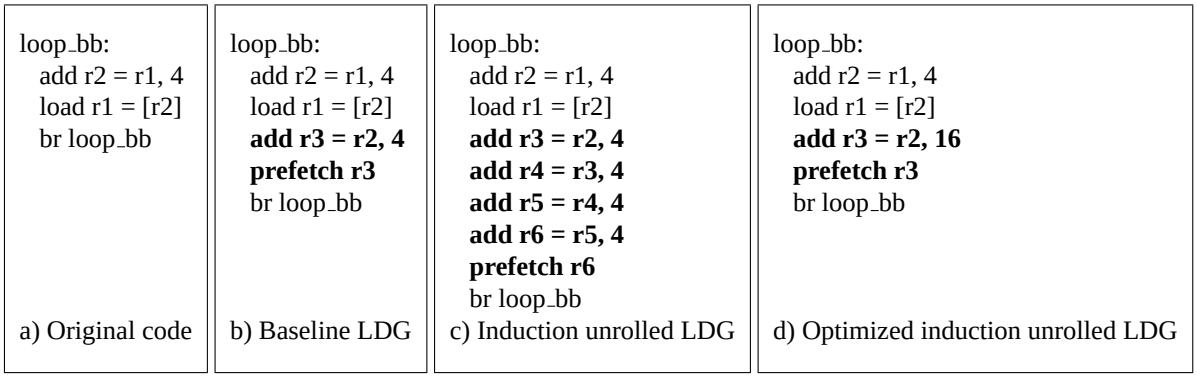


Figure 6. Optimized induction unrolling of LDGs.

Name	Profile	Input
EM3D	2000 2 50	25000 2 50
HEALTH	5 50 1	5 500 1
MST	1024 1	3407 1
PERIMETER	11	12
179.ART	Test	Train
181.MCF	Test	Train
183.EQUAKE	Test	Train

Table 1. Benchmarks and input workloads.

be introduced. To solve this problem, we constrained LDG generation such that it will only generate one LDG for each loop nest.

Induction unrolling is applied only to loop nests that are executed very frequently. However, in such loop nests, a degenerate situation can arise. Take an example of a doubly-nested loop. Suppose on the average, every iteration of the outer loop involves executing two iterations of the inner loop. It may be that the outer loop is executed a great many number of times. However, especially when the inner loop is small, it is still unprofitable to perform induction unrolling on the inner loop because the distance by which one can prefetch in the inner loop is very small. Induction unrolling the inner loop aggressively will only result in incorrect prefetches and extra overheads. In order to avoid this situation, we consider the *normalized iteration count* for the innermost loop when deciding whether or not to perform induction unrolling. The normalized iteration count of the inner loop is the average number of times the innermost loop is executed per iteration of the outer loops. For an innermost loop that has only a small normalized iteration count, only the baseline LDG generation will be used.

4. EVALUATION FRAMEWORK

For the evaluation of our proposed technique, memory intensive benchmarks from the OLDEN and SPEC2000 benchmark suites were used. The benchmarks (as well as the processor parameters) also provides a common frame of reference relative to the work of Liao et al. [18] and Collins et al. [7]. Of the applications, 179.ART and 183.EQUAKE are array based applications, whereas the others are pointer-intensive. The reason we include the former is to demonstrate that our methodology is simultaneously applicable to both array and pointer based programs. All of the benchmarks are comprised of various loop kernels, and contain delinquent loads within counted for-loops and do-while loops. We summarize the benchmarks and the input workloads in Table 1. Note that in the case of the SPEC benchmarks, we use the provided

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 128 control registers
Cache	L1 (separate I & D) : 16 Kb each cache 4-way, 2 cycle latency L2 (unified) : 128 Kb cache 4-way, 14 cycle latency L3 (unified) : 4096 Kb cache 12-way 30 cycle latency Fill buffer: 16 entries All caches have 64 byte lines
Memory	230 cycle latency

Table 2. Modeled research Itanium processor.

test input workload for profiling and the training input workload for reporting purposes; simulating the provided reference input workload was simply not practical as these benchmarks execute billions of instructions even using the training input set.

The proposed algorithms were implemented in TRIMARAN [32], a publicly available infrastructure for compiler research. It includes an optimizing compiler and a parametric EPIC processor simulator. We use the Dinero IV [9] cache simulator to simulate the cache hierarchy. The benchmarks were compiled using classic and high level optimizations which include loop unrolling, copy propagation, common subexpression elimination, dead code elimination, and aggressive register allocation. In addition, the compiler is capable of advanced region formation, including superblock [12] and hyperblock [21] optimizations. The former uses control speculation to increase program parallelism. The latter uses predication to remove branches and increase ILP. We compare the LDG performance in the context of both region formation optimizations. We also consider the interactions between LDGs and software pipelined loops [27] in the context of the Itanium processor. We found that with the exception of EM3D, the OLDEN benchmarks can not be software pipelined; this is because they are pointer intensive and perform significant pointer chasing. The SPEC benchmarks on the other hand are all candidates for software pipelining, especially since some of them are numerical applications.

In this study, we did not consider out-of-order (OOO) execution. It was recently shown that even using a long range prefetching framework which utilizes Simultaneous Multi-Threading (SMT), a negligible (5%) performance improvement is obtained (compared to an OOO Itanium processor) [18]. In large part, by scheduling instructions dynamically, OOO execution can effectively mask a substantial portion of the cache miss latency. By contrast, EPIC architectures such as the Itanium processors are in-order, and strongly warrant the need for latency masking techniques lest the memory bottleneck hinders their expected performance. Therefore, optimizations such the LDG schema proposed here are of special significance.

The results reported in this section were obtained by simulating an in-order HPL-PD processor of the configuration shown in Table 2; we assume a perfect TLB and branch predictor. The architecture parameters correspond to an Itanium processor with minor differences arising from simulation constraints. We should note that we modified the simulation environment to implement a *stall-on-use* model: suppose a load instruction is issued at cycle time t_i, and the first use of the delivered data occurs at cycle time t_j. In the stall-on-use model, we do not stall the processor unless the data is still not available in the required register at time t_j. When this happens, the processor will be stalled by $L-(t_j-t_i)$ cycles where L is the latency of the memory hierarchy in which the data is found. During a stall, no further instructions are issued until the data is available for the processor to continue.

We also varied the various thresholds and algorithm parameters – miss threshold (5–20%), budget size (7–10) and branch frequencies (10–30%) – and found little changes in overall performance.

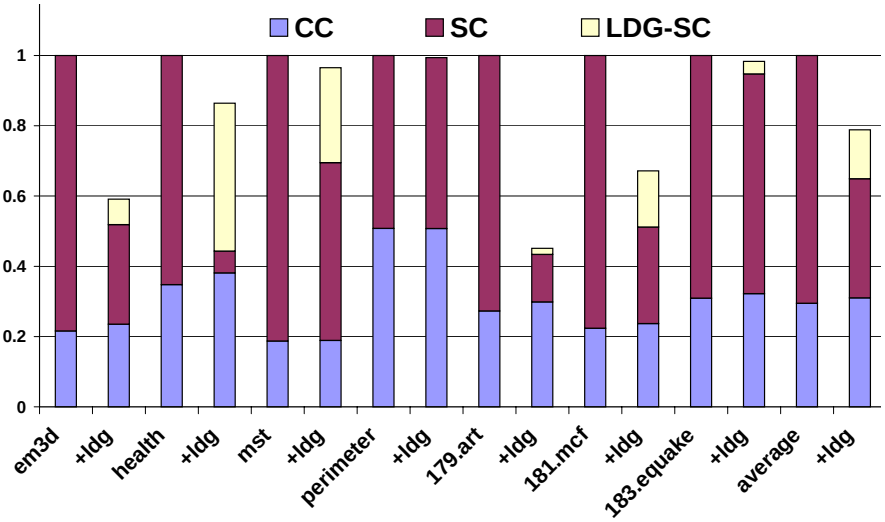


Figure 7. Summary of the total execution cycles before and after LDGs.

5. EXPERIMENTAL RESULTS

The results of the LDG optimization are highly program dependent. The measured performance speedups varied from none to more than a factor of two, with an average performance speedup of 26%. In Fig. 7, we summarize the total execution time in cycles for the benchmarks. For each benchmark, two bars are shown. The first corresponds to the TRIMARAN optimized program, and the second corresponds to the same optimized program augmented with LDGs. Each bar is divided into three parts to show the contributions of each of three factors to overall performance: namely computation cycles (CC), stall cycles incurred by the LDG itself (LDG-SC), and stall cycles incurred by the other load operations of the program (SC). In addition, we normalize the graph such that the baseline execution time is one and the execution time for the LDG augmented benchmark is relative to the baseline; hence, a number lower than one indicates performance improvement, and a number greater than one indicates degradation. By inspecting the number of computation cycles for each benchmark, before and after the LDG optimization, we observe that they are relatively the same, with LDGs incurring a 1.52% increase in computation cycles on average. This suggests that the algorithms presented earlier effectively prune and schedule LDGs. We should point out that the average IPC (instructions per cycles, assuming a perfect memory hierarchy with a two cycle latency) for the selected benchmarks ranges from 1.28 to 1.82 with an average of 1.54; this is in the context of hyperblock optimizations where the highest level of ILP is expected. Thus it is not surprising that the LDG scheduler succeeds in allocating resources for the prefetching operations.

Upon further detailed analysis and studies, we found that the efficacy of the LDG framework is restricted by the following two limitations:

(1) *Premature LDG termination due to control flow restrictions.* Most affected by this restriction are PERIMETER and MST. We observed that LDGs were often terminated prematurely because LDG generation algorithm quickly exhausted all available control flow path and reached the function boundary. While we have not yet considered interprocedural optimizations in this context, we believe they may alleviate the issue. We intend to investigate this possibility in our future work.

(2) *Stall cycles incurred by the LDG itself.* During the execution of the LDG operations, it is possible that the precomputation itself includes a load operation which may result in a cache miss. This will force the application to stall until the data is fetched. This is a disadvantage of the proposed scheme not seen in prefetching by means of a SMT thread

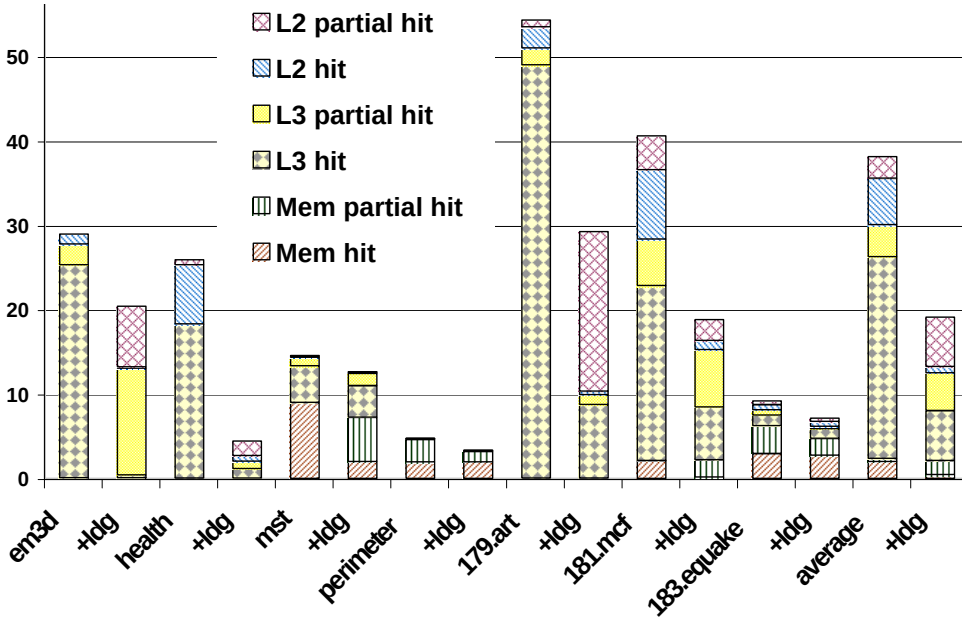


Figure 8. Load hit distribution before and after LDGs.

since a cache miss incurred by the speculative thread will not stall the main thread. Benchmarks such as 181.MCF, 183.EQUAKE, HEALTH and MST may benefit further from the LDG optimization if such stalls can be eliminated or reduced.

In Fig. 8, we report the hit distribution for the benchmarks in the absence and presence of LDGs. In this graph, we only report the hit profiles for the load instructions intrinsic to the program and did not include any hits generated from the execution of the LDGs. Each bar in the graph represents the percentage of cache hits relative to the total number of load requests initiated by the application (i.e., we do not consider loads that are initiated by a LDG in this computation). The graph shows that a substantial number of cache hits occur at lower levels of the memory hierarchy when LDGs are used to prefetch data: 80% of the loads hit in the primary cache, and the bulk of the rest in the second and third level caches. By contrast, in the absence of LDGs, a substantial amount of data is found either in main memory or in the tertiary cache; nearly 30% of loads hit in the tertiary cache. Thus, we conclude that LDGs successfully moved data closer to the processor thereby ameliorating the memory bottleneck.

5.1 Limitations of LDG

In Table 5.1 we report the relevant LDG characteristics of the various benchmarks. Of the large number of load instructions in a program, only a small number are classified as delinquent. In all, there are 31,955 loads of which only 140 (0.53%) are considered delinquent; the criteria for identifying delinquent loads was presented in Section 3.1. For each of the delinquent loads, an average of 1.6 LDGs were generated, and of which 30% were pruned using the heurisitc from Section 3.5. The table also reports the number of times each of the four conditions for termination of the LDG generation were encountered (Section 3.2). The first condition labeled *ideal* is shown in column six. This is the frequency with the LDG algorithm terminated under ideal circumstances and will likely mask the full delinquent load penalty. The second condition is labeled *budget* and it is shown in column seven. It lists the number of times LDG generation terminated because the LDG grew too large. An example of a benchmark adversely affected in this context is 183.EQUAKE. The

Benchmark	Total Loads	Delinquent Loads	Candidate LDGs	Pruned LDGs	Ideal	Budget	BRL	CF	Desired Static Latency Masking	Achieved Static Latency Masked
EM3D	1,534	12	21	1	0	0	1	20	987	308
HEALTH	1,129	17	35	15	2	0	9	24	1,475	363
MST	1,713	11	13	5	3	0	0	10	1,428	348
PERIMETER	1,257	7	13	8	0	0	0	13	780	41
179.ART	5,678	33	30	5	8	2	0	20	1,856	1,202
183.EQUAKE	16,378	21	30	6	1	3	0	26	1,995	992
181.MCF	4,266	39	83	2	4	0	3	76	3,014	1,558

Table 3. Benchmark characteristics and LDG algorithm analysis.

application uses a three dimensional data structure which requires numerous instructions for address generation. Column eight which is labeled *BRL* reports the number of times an intervening function call inhibited further LDG generation. And in column nine, labeled *CF*, we report the number of times the program control flow restricted the generation algorithm. This last condition comprised 84% of the total conditions encountered for the set of benchmarks shown. The last two columns report the desired static latency masking distance and the actual latency masking achieved via LDGs. The desired static latency masking is the sum of the average latencies for the delinquent loads of an application. The achieved static latency masking is the sum of the average distance between the prefetch operation and the delinquent load. From the data, we conclude that the LDGs statically mask 41.7% of the desired latency which translates to 39.54% savings in dynamic stall cycles on average.

5.2 ILP Optimizations and LDGs

The suite of optimizations included in TRIMARAN includes advanced region formation techniques such as superblocks and hyperblocks. Fig. 9 shows how the choice of the region formation affects the LDG performance: for the most part, there are little significant differences. Stall cycles reductions in the case for superblocks and hyperblocks is often less than the gains in the basic blocks mainly because the former increase instruction level parallelism and yield more compact schedules. Furthermore, a detailed breakdown of the execution time for each of the benchmarks in the presence of either super- or hyper- blocks showed negligible improvements in the total number of stall cycles compared to the baseline basic block case. By contrast, the LDG is specifically geared at reducing stall cycles and often the payoff is far better than ILP optimizations alone. This is particularly true since ILP enhancements are in vain if the processor is often waiting for data to be fetched from higher levels of the memory hierarchy.

6. PRELIMINARY ITANIUM RESULTS

Using the HPL-PD simulation environment gave us insights into the working of the LDGs and allowed us to implement, test and fine-tune our algorithms in TRIMARAN. However, it is rather desirable to validate the ideas on actual hardware. To this end, we carried out several experiments using an Intel Itanium processor. Our test platform is a Hewlett-Packard Itanium workstation with a 733 MHz Itanium processor with 1 Gb of DRAM. We use the ORC compiler (version 1.1) [24] to generate the IA-64 assembly code which we then instrument for memory profiling; we use the Dinero IV cache simulator to gather the cache miss profiles of the applications. Next we identify a small number of delinquent loads as described earlier and proceeded to generate and insert the corresponding LDGs into the assembly code. At this time, the LDG insertion was manually performed as the available software tools for the Itanium are not robust enough for a completely automated process. As a result, we only incorporate a few (1-4) LDGs for each of the benchmarks. It is worthy to note that the modified version of the assembly code with LDGs is exactly the same as the original, except that nop slots in a bundle and free registers are allocated to the LDG operations. When necessary, new bundles were introduced. However, under no circumstances did we make any changes to the original code except to replace nops with LDG operations. Furthermore, we repeat all of the experiments using an Itanium II 900 MHz processor.

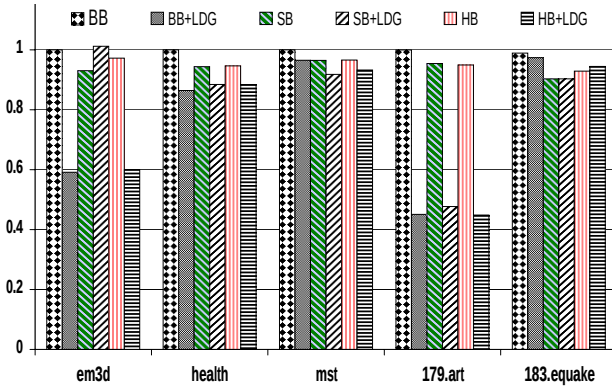


Figure 9. Results for basic blocks, super blocks and hyper blocks.

Benchmark	LDGs Inserted	Itanium Execution Time			Itanium II Execution Time		
		Before LDGs (secs)	After LDGs (secs)	Percentage Improvement	Before LDGs (secs)	After LDGs (secs)	Percentage Improvement
EM3D	1	10.22	8.99	12.03	5.107	4.164	18.5
EM3D-LI	1	10.22	8.2	19.77	5.107	4.935	3.3
EM3D-SWP	3	6.834	5.878	13.98	3.746	3.435	8.3
HEALTH	3	4.33	4.24	2.08	2.205	2.203	0.01
179.ART	3	942	834	11.46	640.71	588.62	9
179.ART-SWP	4	484.9	433	10.7	484.9	466.86	3.72
average	–	–	–	11.67	–	–	7.14

Table 4. Results of benchmarks executed on 733 MHz Itanium and 900 MHz Itanium II.

In Table 6 we report the speedups obtained on each of the two Itanium processors. On the average, LDGs reduce the Itanium execution time by 11.85% when compared to the ORC optimized code. The benchmarks we selected here are EM3D, HEALTH and 179.ART. The input arguments to EM3D are `25000 10 50`. We execute three variants of this applications. The first is the baseline application. The second applies the LDG induction unrolling technique (labeled as EM3D-LI in the table). The third enables software pipelining in the ORC compiler (labeled as EM3D-SWP in the table, ORC flag `-O3`). As it may be expected, the pipelined version of the program ran the fastest in the absence of LDGs. Incorporating the LDGs to the pipelined version of EM3D yields an additional execution time saving of 14%. The results for the other benchmarks were also encouraging, especially since we incorporated so few LDGs (one for EM3D and four for 179.ART-SWP). The input parameters to HEALTH are `5 500 1` and for 179.ART, we use the SPEC reference input set. Lastly, we enabled the automatic prefetching capabilities of the ORC compiler but noticed little differences in performance; the automatic prefetch insertion was done by specifying the `LNO:prefetch=2` option to the compiler.

6.1 Software Pipelining and LDGs

The proposed LDG schema is compatible with software pipelining as our preliminary results suggest. While software pipelining overlaps iterations of a loop, it does so assuming a typical latency for load operations; and usually an optimistic assumption is used. While there may be some effect of one iteration prefetching data for a subsequent iteration as an artifact of spatial locality, such is not its explicit goal. As currently described and in the context of software pipelined loops, the LDGs do not cross loop boundaries and are well contained within a single iteration. Therefore, it does not

interfere with the correctness of software pipelined loops provided the proper register assignment is carried out.

Although we have not implemented LDGs for software pipelined loops using the TRIMARAN compiler, we have manually incorporated a small number of them into software pipelined loops generated by ORC. Our initial results are encouraging. It shows that when the LDG mechanism is used with prudence, taking special care only to use free registers and instruction slots, it is possible for the LDG mechanism to improve the performance of software pipelining. This is on our agenda for future work with LDGs.

7. RELATED WORK

Data prefetching is the focus of a large body of work aimed at tolerating increasingly long memory latencies. While numerous prefetching strategies have been proposed, most can be classified as predictive in nature. Hence, they are generally vulnerable to irregular memory access patterns. By contrast, the LDG framework is precomputation based and thus effectively achieves better prefetching coverage. Due to the volume of research in the field, we briefly mention some of the most recent and relevant work. In general, predictive prefetching scheme first analyze the application memory access trace and subsequently make decisions based on the learned history. Some examples implemented in hardware include stride [11] and Markovian [4, 14, 30] prefetchers. While effective, many of the hardware based techniques require extensive architectural features. Examples of software based schemes include the work of Chilimbi and Hirzel who describe a runtime framework for learning and predicting data sequences [6]. In their work, the profiling or analysis phase introduces some overhead which must be overcome by the dynamic optimization to produce overall gains. On a similar note, Wu [35] describes a profile based methodology for discovering regular stride patterns in irregular programs. The LDG optimization described here extends beyond prefetching regular data streams and simultaneously targets irregular memory access patterns. This is promising since a recent study [6] suggests that many data access patterns will not be successfully prefetched using stride based techniques alone. However, it is conceivable that the LDG optimization may be successfully combined with some of the recent works on compiler directed stride prefetching and further bridge the speed gap between processor and memory system.

Also recently, several researchers have proposed the use of hardware based precomputation as a mean to conduct data prefetching. An exampled is the work of Annavaram et al. who introduce dependence graph precomputation [1]. The scope of such a strategy however is limited to short range prefetching as they perform dynamic program analysis and hence require complex architectural support. Some other related work can be found in the context of multi-threaded environments. For example, Luk [20] proposed software controlled pre-execution which leverages available threads to execute prefetching operations. Roth and Sohi [29] proposed data driven multi-threading as a mechanism to prefetch future memory requests. Collins et al. introduced dynamic speculative precomputation [7, 8], and Liao et al. [18] demonstrate how to utilize idle threads to perform precomputation and prefetching using SMT. In these works, the threads can reduce the memory latency of the main application by running ahead of the program and performing data prefetching. An advantage of thread based prefetching versus LDGs is in the context of cache misses incurred by the precomputation code. In the case of LDGs, data misses incurred by the precomputation slice will negatively impact overall execution; albeit performance improvements are still achieved in the end. However, a main disadvantage of the thread based techniques is that they require extensive hardware support and thus they may not be applicable in simpler VLIW machines. By contrast, LDGs leverage simpler instruction set architectures which are quickly becoming more common in microprocessor design.

In other related works, Zilles and Sohi [36] performed an initial characterization of backward slices as mechanisms for predicting cache misses and branch mispredictions. They showed how speculative slices may be used to mainly predict branch behaviors. The focus of the proposed optimization is an automated methodology for compiler directed prefetching of load operations. Also, Vanderwiel and Lilja [34] propose a compiler-assisted hardware extension called the data prefetch controller. The controller dispatches prefetch requests based on a compiler generated program which enables the prediction of future memory requests. The LDG achieves much of the same effects without the additional architecture investments.

8. SUMMARY AND REMARKS

In this paper, we introduce the concept of a load dependence graph (LDG), a slice of the program dependence graph that captures the computation of the address for a given load instruction. Using the LDG, we show how it may be used to precompute load addresses for timely data prefetching. We also describe algorithms for generating the LDGs and embedding the corresponding address precomputation and data prefetch into the instruction stream of application compiled for an EPIC architecture. We study in detail the conditions affecting the effectiveness of the method, and using these studies, we formulate several algorithm variants and heuristics that will maximize the efficacy of LDG as a data prefetching mechanism. As a result, we have a data prefetching schema that is (i) highly precise, (ii) efficient and robust in the context of a wide class of applications, (iii) does not require any new hardware support, and (iv) has a very low overhead. Simulation show that the method can reduce the execution time of some applications by as much as half. Furthermore, to provide real evidence in support of the proposed methodology, we validate the algorithms on two Intel Itanium processors. The lack of the complete compiler infrastructure forced us to manually apply our algorithms to a limited number of benchmarks. However, even though we only incorporate a handful of LDGs into each of the applications, we achieve a speedup of 11.67% and 7.14% on average for the Itanium and Itanium II processors respectively. We believe this is concrete evidence that the method is effective. While admittedly there is still much room for improvement, we are certain that the technique can readily contribute to the performance of EPIC processors today.

ACKNOWLEDGMENTS

The authors thank Krishna V. Palem for his constant encouragement and support. The authors also thank Hsien-Hsin Sean Lee for his comments and suggestions.

REFERENCES

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

[2] 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.

[3] A. Burks, H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Papers of John von Neumann, 1987.

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

[5] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, Oct. 1992.

[6] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–209, 2002.

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

[8] 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.

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

[10] P. Faraboschi, J. Fisher, G. Brown, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[11] 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.

[12] 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.

[13] Intel Itanium Processors. intel.com/products/server/processors/server/itanium/.

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

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

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

[17] M. Lam, E. Rothberg, and M. Wolf. The cache performance of blocked algorithms. In *Proceedings of the Fourth International Conference in Architectural Support for Programming Languages and Operations Systems*, pages 63–74, Apr. 1991.
[18] 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.
[19] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. Spaid: Software prefetching in pointer and call intensive environments. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Nov. 1995.
[20] 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.
[21] 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.
[22] 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.
[23] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, 1998.
[24] Open Research Compiler for the Intel Itanium. ipf-orc.sourceforge.net.
[25] D. Ortega, E. Ayguade, J. Baer, and M. Valero. Cost-effective compiler directed memory prefetching and bypassing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2002.
[26] 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.
[27] B. R. Rau. Iterative modulo scheduling. Technical Report Technical Report HPL-94-115, Hewlett-Packard Laboratories, 1995.
[28] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.
[29] A. Roth and G. Sohi. Speculative data-driven multithreading. In *7th IEEE High-Performance Computer Architecture*, Jan. 2001.
[30] 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.
[31] Texas Instruments high performance DSPs. http://dspvillage.ti.com/docs/allproducttree.jhtml.
[32] TRIMARAN: An infrastructure for research in instruction level parallelism. www.trimaran.org.
[33] Trimedia technologies and VLIW products. www.trimedia.com.
[34] S. Vanderwiel and D. Lilja. A compiler-assisted data prefetch controller. In *Proceedings of the International Conference on Computer Design*, pages 372–377, 1999.
[35] 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.
[36] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of 28th International Symposium on Computer Architecture*, July 2001.