Performance Potential of Effective Address Prediction of Load Instructions

Pritpal S. Ahuja, Joel Emer, Artur Klauser, and Shubhendu S. Mukherjee

VSSAD, Alpha Development Group
Compaq Computer Corporation,
Shrewsbury, Massachusetts

Abstract

Modern, deeply-pipelined, out-of-order, and speculative microprocessors are still plagued by the latency of load instructions. This latency is dominated by the latencies to resolve the source operands of the load, to compute its effective address, and to fetch the load's data from caches or main memory. This paper examines the performance potential of hiding a load's data fetch latency using effective address prediction. By predicting the effective address of a load early in the pipeline, we can initiate the cache access early, thereby improving performance.

The current generation of effective address predictors for a load instruction is based on either the history or the context of the specific load. In addition, researchers have examined load-load dependence predictors to prefetch cache misses. This paper examines the performance potential of using a load-load dependence predictor to predict effective addresses of load instructions and issue them early in the pipeline. We call this predictor the DEAP predictor.

We show that on average DEAP can improve the accuracy of effective address prediction by 28% over a perfect combination of last address, stride address, and context-based address predictors across our seven benchmarks from the SPEC95 and Olden suite. We find that an ideal hybrid of these four predictors—a predictor that always picks the right predictor for a load—can potentially achieve performance close to that of a Perfect predictor in most cases.

We use an oracle-based simulation approach to evaluate our timing results. This method allows us to measure the upper bound of the performance from effective address prediction using a mostly realistic pipeline. However, our timing simulation method does not account for penalty due to mis-prediction of an effective address and assumes a zero-cycle latency from address prediction resolution to address predictor update.

1 Introduction

The latency of load instructions—from fetch to commit—continues to plague modern out-of-order and speculative microprocessor designs. A load's latency is dominated by three components: effective address computation, store-load dependence resolution, and data fetch. We define effective address computation latency as the sum of the latency from the time a load is fetched until its source operands are ready and the latency to compute the effective address itself. The store-load dependence resolution latency is the latency to determine whether a load depends on a prior uncommitted store. Finally, data fetch latency is the latency to fetch the load's data from caches or main memory.

Unfortunately, a load's effective address computation and data fetch latencies continue to be major problems in today's microprocessors, even though recent memory disambiguation techniques (e.g., [1], [2]) have largely solved the store-dependence resolution problem. The effective address computation latency remains high because in many programs it takes a long time for a load's source operands to become data-ready. Pointer-chasing (via load-load dependence chains) is a classic example of such a code. Data fetch latency continues to be high for two reasons. First, the gap between processor and DRAM performances continues to widen, which causes cache miss latency to appear significantly worse in each generation. Second, the complexity of modern, wide-issue, and out-of-order machines increases both the width and length of result buses from the data caches to the execution units. This, in turn, increases the data fetch latency from L1 SRAM caches. Thus, on modern microprocessors, the load-to-use latency on cache hits has increased to between three and five cycles [4].

This paper examines the performance potential of hiding a load's data fetch latency, particularly on cache hits on L1 or L2 caches, using effective address prediction. Modern microprocessors, in their quest for higher clock speeds and performance, have become deeply pipelined, with 20 or more pipeline stages (e.g., Intel Willamette [12]). Because of such deep pipelining, we usually have a large number (five or more) of stages from the point an instruction is renamed until the instruction reaches the execution stage of the pipeline. Such deep pipelining potentially opens up a window of opportunity in which we can predict a load's effective address early in the pipeline, issue the load speculatively, and fetch the data from the data cache. If the prediction is correct, then we can hide a load's data fetch latency, particularly for cache hits. This allows instructions dependent on the load to issue earlier, thereby improving the number of instructions committed per cycle (IPC). Figure 1

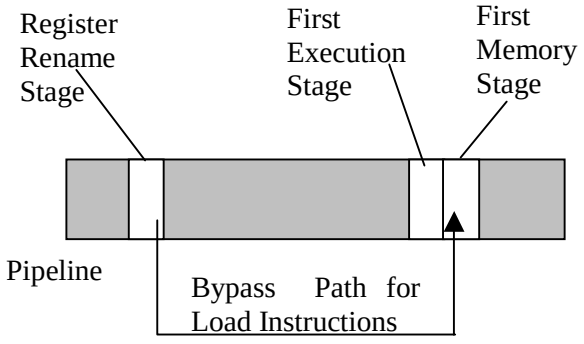

Figure 1. Effective address prediction and load issue via bypass path.

shows an example of such a pipeline in which we predict the effective address of loads and issue them via a bypass path to the memory system in the pipeline. In this figure and throughout the rest of the paper, we assume that the bypass path for loads originates after the register rename stage. This allows a load instruction to fetch its data directly into the correct physical register and not into an intermediate staging buffer.

The performance potential of effective address prediction depends on two factors—the prediction accuracy of the effective address predictors and how the effective address predictors are integrated into the pipeline. In this paper, we examine five effective address predictors—*Perfect*, *LAP*, *SAP*, *CAP*, and *DEAP*. The Perfect (and perhaps non-implementable) predictor always predicts the correct effective address for a load. LAP (Last Address Predictor) predicts that a load will use the same effective address that it had used the last time it executed [3][7]. SAP (Stride Address Predictor) predicts an effective address by using the load's last address and a stride [8][9][10]. CAP (Context-based Address Predictor) predicts effective addresses based on the history of prior addresses encountered by a load [4]. Finally, DEAP (Dependence-based Address Predictor) is an effective address predictor that predicts effective addresses based on dependences between load instructions. DEAP is a variant of Roth, et al.'s load-load dependence predictor [5]. However, unlike the Roth, et al. work, which used the dependence predictor to prefetch cache misses, we use DEAP to issue loads early in the pipeline by predicting their effective addresses.

To understand the performance potential of each of these predictors, we use a prediction from a predictor only when it is correct. We do not account for penalty for a mis-prediction of an effective address. Additionally, we assume a zero-cycle latency from address resolution to predictor update. Such a method has both advantages and disadvantages. This method allows us to study the performance potential (i.e., the upper bound of performance) of these predictors. This also frees us from worrying about the exact details and performance degradation due to the recovery mechanisms necessary when any of the predictors mispredicts. However, this method also does not account for performance difference due to different recovery costs of the different predictors.

We show that DEAP can improve the accuracy of effective address prediction by 28% over a perfect combination of LAP, SAP, and CAP across our seven benchmarks from the SPEC95 and Olden suites. Specifically, on our three pointer-intensive Olden benchmarks, between 19% – 49% of the dynamic loads' effective addresses are correctly predicted only by DEAP, and not by any other predictor.

We find that one or more of LAP, SAP, CAP, or DEAP predicts the majority of loads accurately. Across our seven benchmarks from the SPEC95 and Olden suites, the four effective address predictors make predictions on 64% – 99% of dynamically executed loads. The predictors accurately predict between 73% and 95% of these loads.

Additionally, we find that an ideal hybrid of LAP, SAP, CAP, and DEAP captures most of the performance benefit available from a Perfect predictor. The Perfect predictor can boost performance between 7% and 244% in our seven benchmarks. The ideal hybrid of LAP, SAP, CAP, and DEAP can capture 82% of the Perfect predictor's performance improvement, except for *compress*. These predictors boost performance because they allow the pipeline to prefetch both L1 cache hits and L2 cache hits. Unfortunately, for *compress*, these predictors are unable to predict the effective addresses of loads that miss frequently in the L1 data cache, but hit in the L2, and account for most of the performance degradation.

We use a method called *Oracle Simulation* for this study. Ideally, for upper bound studies, we would like to have all information about an instruction at the fetch stage itself. For example, knowing the direction of a branch at the fetch stage would allow us study the performance benefit from a perfect branch predictor. Unfortunately, in a detailed simulation model of an out-of-order and speculative pipeline, such information is not available at the fetch stage because prior instructions that the branch may depend on may not have executed.

Oracle simulation solves this problem. With Oracle simulation, we run two simulations—one that models the detailed out-of-order, multi-stage pipeline and a second one that models a single-stage pipeline. Every time the out-of-order pipe fetches an instruction, the corresponding single-stage pipe executes the instruction completely. Thus, the second pipe has complete information about the instruction. Specifically, for our case, the second pipe can return the effective address of loads when they are at the fetch stage of the first pipe. This helps us implement the Perfect effective address predictor as well as verify if the predictions from other predictors are correct.

The rest of the paper is organized as follows. Section 2 describes our four effective address predictors. Section 3 discusses our evaluation methodology. Section 4 discusses results. Section 5 describes related work. Finally, Section 6 presents our conclusions.

2 Effective Address Predictors

In this section we describe the four predictors: LAP, SAP, CAP, and DEAP. We also simulate a Perfect Effective Address Predictor, which always returns the correct effective address of a load instruction. Section 3.2 describes how we implement these predictors in our simulator.

2.1 LAP: Last Address Predictor

A Last Address Predictor (LAP) [3][7] predicts that a load will reuse the same effective address that it used the last time it executed. LAP works well for loads that access the same variables repeatedly (e.g., globals). In our evaluation LAP can accurately predict the effective addresses of almost 35% of the dynamic loads (averaged across our seven benchmarks) on which LAP makes a prediction. For our LAP implementation we use a direct-mapped, tagged cache, indexed by the PC of the load. Each LAP entry contains a tag, the predicted address (i.e., last address), and a two-bit saturating counter. LAP does not make a prediction if the PC does not exist in the tagged cache or until the saturating counter has not reached its maximum count.

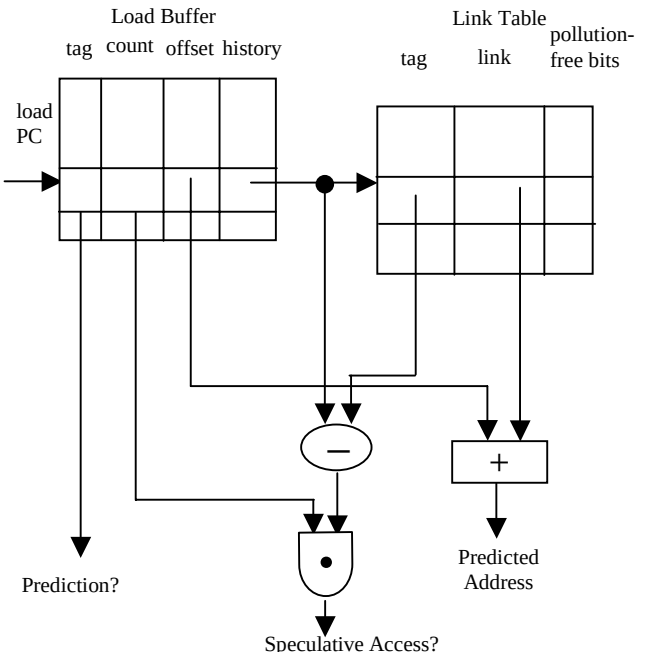

Figure 2. A Context Address Predictor (CAP)

2.2 SAP: Stride Address Predictor

A Stride Address Predictor (SAP) predicts that the effective address of a load will be fixed offset (or *stride*) from the load's effective address it produced the last time it executed. SAP is well suited for array accesses in which a load sequences through different elements of an array. Our SAP implementation accurately predicts 48% of dynamic loads (averaged across our seven benchmarks) that it makes predictions on.

For our SAP implementation we use a direct-mapped, tagged cache, indexed by the PC of the load. Each SAP entry contains a tag, the last address, the predicted stride, the last stride seen, and a confidence counter. The predicted address is the sum of the last address and the predicted stride. The last stride seen is the difference between the last effective address and next-to-last effective address used by the load. SAP changes the predicted stride only if it encounters the same stride twice in a row [9][10]. SAP makes a prediction only if the PC exists in the tagged cache and the confidence counter has reached its maximum count.

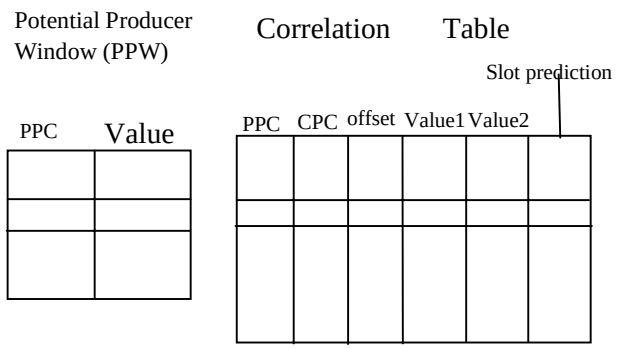

Figure 3. DEAP configuration. PPC = Producer PC, CPC = Consumer PC.

2.3 CAP: Context-based Address Predictor

A Context-based Address Predictor (CAP) predicts a load's effective address based on the history of prior addresses encountered by the load. Loads often sequence through a fixed pattern of addresses, such as in recursive data structures. CAP works well with such recursive data structures. CAP can also capture some of the effective addresses that are accurately predicted by LAP and SAP. This is because LAP is a degenerate case of CAP with no history. Similarly, CAP can capture a sequence of strided addresses, if CAP's history captures the sequence of strided addresses. Nevertheless, SAP is a more compact representation of strided addresses and, therefore, we expect SAP to have a higher accuracy for strided access patterns. For our seven benchmarks, CAP accurately predicts 52% of the loads it makes predictions on.

In this paper we use Bekerman et al.'s implementation of a CAP [4][1]. This implementation of CAP uses two tables: the *load buffer (LB)* and the *link table (LT)*, as shown in Figure 2. The load buffer is a per-static-load table that maintains the recent history of effective addresses of the associated load. The link table is a second-level table that provides the address used for effective address computation. The load buffer is indexed by part of the load's PC. Each load buffer entry contains a tag (the rest of a load's PC bits), a saturating counter, the load's offset as found in the load instruction itself, and the history. The saturating counter serves as a confidence estimator. Thus, CAP will not return a prediction unless the value of the counter is above a certain threshold. The load buffer history bits contain the history of prior base addresses of a load instruction. We update the CAP history as follows: history = (history << m) XOR (new_base_address >> 2). That is, the history is shifted left m bits and XOR-ed with the new base address shifted right by two. We shift the new base address by two bits because the lower two bits do not matter except for unaligned accesses.

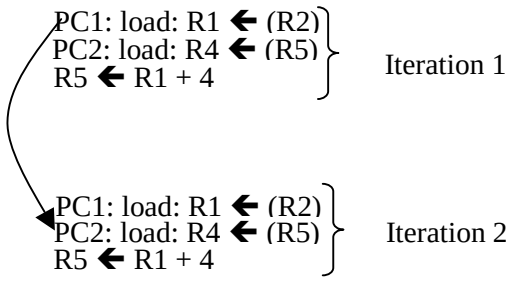

Figure 4. Load-Load Dependence Across Iterations.

We index the link table using the history bits of a load buffer entry. Because of this, CAP's history maintains only base addresses, multiple load buffer entries may map into the same link table entry. To avoid destructive aliasing, the link table also contains a tag, which is the history of the load buffer entry itself. The link table contains the link, which is the base address of the predicted load. To obtain the complete effective address, we must add the offset from the load buffer with the link address from the link table. Finally, a link table entry also contains a few pollution-free bits. The pollution-free bits record a few bits of the base address of a probing load. We update the link table entry when two consecutive accesses to the link table have the same pollution-free bits.

Like the other predictors, CAP does not make a prediction if it does not find a matching PC in the load buffer or if there is tag mismatch in the link table.

2.4 DEAP: Dependence-based Address Predictor

Our Dependence-based Effective Address Predictor (DEAP) predicts effective addresses based on dependences between load instructions. Two load instructions are dependent if the first one (the *producer load*) loads a value that is the base address for the second load *(*the *consumer load*). DEAP recognizes such dependences and predicts the effective address of the consumer load based on the value loaded by the producer. DEAP is well suited for

[1] We have validated our implementation against the prediction accuracy numbers in the Bekerman, et al. paper.

Predictor	Total Bytes	Configuration
LAP	45K	4K-entry, 2-way table
SAP	60K	4K-entry, 2-way, 2-delta, strided predictor
CAP	70K	4K-entry, 2-way Load Buffer; 4K-entry, direct-mapped Link Table
DEAP	3.8K	128-entry, depth=2, fully-associative CT; 32-entry, fully-associative PPW

Table 1. Effective Address Predictor Configurations.

predicting effective addresses in pointer-chasing programs, which have such load-load dependence patterns. Of the three Olden benchmarks [11] we examine in this paper, DEAP can accurately predict 59% of the total number of dynamically executed loads that it makes predictions on.

We have derived DEAP from Roth, et al.'s load-load dependence predictor [5], which we refer to as the RMS[2] predictor. The RMS predictor captures dependences between producer and consumer loads. However, the RMS predictor does not always capture the precise effective address of a specific instance of a consumer load. First, we describe the RMS predictor. Then, we show how we derive DEAP from the RMS predictor.

2.4.1 The RMS Predictor

The RMS predictor consists of two tables: the *Potential Producer Window (PPW)* and the *Correlation Table (CT).* The PPW maintains a list of most recently loaded values and their corresponding instructions. Thus, loads in the PPW are loads that can potentially be producer loads. The CT maintains correlations or dependences between producer and consumer loads. As a load completes, we enter its PC and loaded value into the PPW. At the same time, we do an associative search of the PPW to find if a prior load produced this load's base address. If so, then this load is a consumer load of the prior load. This establishes the load-load dependence template that we record in the CT. Subsequently, when the producer load appears again, the RMS predictor will search the CT associatively for a match. If there is a match, the RMS predictor will trigger a prefetch on the value (i.e., base address) loaded by the producer load + offset of the consumer load obtained from the CT.

[2] The acronym RMS comes from the first letter of the authors' last names: Roth, Moshovos, and Sohi.

CPU Model
128 entry instruction queue
128K 2-way set-associative Level 1 instruction cache
128K 2-way set-associative Level 1 write-through data
A bigger and more powerful 21264-like branch predictor
8 instructions maximum issued per cycle
4 D-Cache Ports (any combination of loads and stores), 2-cycle access
8M Direct Mapped, Write-Back, Unified Second Level Cache, 12 cycle access

Table 2. Base processor configuration used in all simulations. Note that our results are independent of our choice of write-back or write-through cache.

The RMS predictor works great for prefetching load misses, but may not be as precise for effective address prediction in a few cases. Figure 4 shows such an example. In this example, the RMS predictor establishes a dependence between PC1 and PC2. However, the real dependence exists between PC1 of the first iteration and PC2 of the second iteration. That is, PC1 of the first iteration loads the base address of PC2 of the second iteration. Unfortunately, if we use the RMS predictor to predict effective addresses, then we would incorrectly predict that the effective address of PC2 in the first iteration is loaded by PC1 from the first iteration. This, however, works well for prefetching load misses—the purpose Roth, et al. originally designed the RMS predictor for. This is because Roth, et al. trigger a prefetch using the base address (and adding in any necessary offset) loaded by the producer load from PC1 of the first iteration. As long as this prefetch does not cause too much cache pollution, it is irrelevant whether PC2 from the first iteration or second iteration uses the prefetched cache block. However, we do not have this luxury for effective address prediction, which must precisely predict the addresses for PC2 in every iteration.

2.4.2 Deriving DEAP from the RMS Predictor

We construct DEAP by augmenting the RMS predictor with more precise dependence information (Figure 3). Like the RMS predictor, DEAP has the PPW and the CT. Like RMS, DEAP's PPW has two fields: PPC and Value. The PPC is the potential producer PC and Value is the value (a potential base address of a subsequent load) loaded by the potential

Benchmark	Suite	Warmup	Tot Insts	Loads	Stores	Base IPC
compress	SpecInt95	1M	30M	5.8M	1.7M	1.6
gcc	SpecInt95	1M	30M	7.4M	3.3M	2.2
m88ksim	SpecInt95	1M	30M	4.1M	2.0M	2.2
mgrid	SpecFp95	1M	30M	10.8M	2.0M	6.1
em3d	Olden	1M	30M	10.6M	2.0M	1.9
health	Olden	1M	30M	10.0M	4.0M	0.8
tsp	Olden	1M	30M	7.7M	318K	1.5

Table 3 . Benchmark Characterization. We skipped between 30 million and two billion instructions to reach the interesting portion of each benchmark. M = one million. K = one thousand. Warmup = number of instructions for which we warm up the simulator without collecting statistics. Tot insts = total number of instructions simulated.

producer load. Like the RMS predictor, DEAP's CT maintains the PPC (producer PC), CPC (consumer PC), and offset of the consuming load. Unlike, the RMS predictor, DEAP's CT has several additional fields. Each entry in DEAP's CT can have one or more Value slots (Value1, Value2, ...) and a slot predictor. Also, unlike the RMS predictor, we force DEAP to have only one entry corresponding to a consumer PC to simplify the implementation of DEAP. We could have had multiple consumer PC entries like the RMS predictor, but that would have made the slot prediction more complicated.

When a load commits, we take three update actions. First, as in the RMS predictor, we enter the load into the PPW. Second, we associatively search the CT with the load's PC for a match on the PPC field of the CT. For every matching entry, we record the value read by the load into the entry. We maintain the Value1, Value2, ... etc. fields as a circular queue, so this newly loaded value will remove the oldest value loaded in every matched CT entry. Third, we associatively search the CT with this load's PC for a match on the CPC field. Since all CPC entries in DEAP are forced to be unique, this consumer load's PC will match only one CT entry. Then, we will search the Value slots in the CT entry for a match. If there is a match, we update the slot predictor to point to this slot in the CT entry.

The DEAP probe is simpler than the update. We probe the CT with a load's PC. If the probe finds a CT entry with a CPC value that matches the load's PC, we examine the slot predictor to obtain the specific slot to use in the prediction. We add the value (i.e., base address) from the predicted slot of the CT entry and the load's offset to predict an effective address for the load. However, DEAP does not make a prediction if there is no matching CPC entry in the CT.

There is another subtle difference between RMS and DEAP. The RMS predictor is probed by producer loads, so that the producer loads can trigger the prefetch of the corresponding consumer loads' data. In contrast, the DEAP predictor is probed by consumer loads to obtain their own effective addresses. This enables the pipeline to issue consumer loads early in the pipeline.

3 Evaluation Methodology

This section describes *Asim*—our simulator framework, the machine model we simulated using Asim, and the benchmarks we used for our evaluation.

3.1 Asim

Asim is a simulation framework developed by the VSSAD, Alpha Development Group, to rapidly prototype modern out-of-order and speculative microprocessors [16]. Asim is divided into two major components—a front-end instruction feeder and a back-end performance model. The front-end feeder fetches, issues, and executes instructions as well as reads and writes memory under the control of the performance model, which can direct the feeder to go down and recover from wrong speculative paths. This level of control allows us to perform detailed simulation of out-of-order and speculative microprocessors.

We have structured Asim's performance models as a bundle of modules and buffers. Modules execute algorithms with almost no notion of time, while buffers connect modules and incorporate time in them. For example, a data cache could be a module. A data cache probe would return the data via the result bus, which we model as an output buffer. However, this

	L	S	C	D	LS	LC	LD	SC	SD	CD	LSC	LSD	LCD	SCD	LSCD	None
Compress	0.0	12.1	0.2	0.3	0.1	0.0	0.0	0.1	0.0	0.0	57.9	0.0	0.0	0.0	2.4	26.8
Gcc	1.9	1.7	12.8	2.5	8.1	0.8	0.2	0.9	0.2	0.6	21.3	0.6	0.0	0.1	1.5	46.7
m88ksim	1.4	1.3	23.8	1.3	3.1	0.8	0.2	0.3	0.1	0.3	35.2	1.2	0.0	0.1	13.8	17.2
Mgrid	0.0	27.9	12.8	0.1	0.3	0.0	0.0	40.5	0.0	0.2	4.1	0.0	0.0	0.0	0.3	13.8
em3d	0.0	15.5	0.0	27.1	0.1	0.0	2.5	0.1	2.2	10.5	0.2	5.4	4.9	1.5	11.5	18.5
Health	0.8	0.7	12.5	49.1	3.1	3.4	0.3	0.1	0.2	1.2	15.5	3.2	0.0	0.1	2.6	7.1
Tsp	0.2	0.2	4.9	19.4	3.9	0.2	0.1	0.1	0.1	32.8	30.9	0.4	0.0	0.0	0.4	6.4

Table 4. Breakdown of accurately predicted dynamic loads as a percentage of covered loads. L = % loads accurately predicted by LAP alone and no other predictor. S = % loads accurately predicted by SAP alone. C = % loads accurately predicted by CAP alone. D = % loads accurately predicted by DEAP alone. LS = % load accurately predicted both by L and S, but not by L and S individually and not by another predictor either. We define various combinations of L, S, C, D in a similar way. None = no predictor accurately predicted these loads. Figure 5 shows a Venn diagram that illustrates the breakdown of accurately predicted loads in this table.

output buffer from the data cache module will capture both the time to run the data cache probe algorithm in hardware as well as the time on the result bus. Such level of detail allows us to accurately prototype a modern microprocessor.

Asim modules can be used in different contexts. For example, they can work independently (and called *Standalone Model*) or within a timing framework (called *Timing Model*). For example, the data cache module can run by itself and return cache hit or miss rates. At the same time, we can plug in the same data cache module into a Timing Model to understand the behavior and timing of the data cache in a real pipeline model.

In this paper, we use both the Standalone and Timing Models to understand the behavior of the five predictors. The Standalone Model gives us the accuracy of our predictors, while the Timing Model shows the impact of effective address prediction in a real pipeline.

3.2 Oracle Simulation

Oracle simulation allows us to perform upper bound studies of realistic pipelines. For example, in this paper using Oracle simulation, we study the upper bound of the performance of a Perfect effective address predictor with a modern, out-of-order, and speculative pipeline. Our Perfect predictor obtains the correct effective address for an instruction by querying the Oracle, which runs simultaneously with the main simulation thread. Then, Perfect sends the predicted loads down a fast path to the memory system in the pipeline (Figure 1). The entire pipeline remains unchanged except for the fast path and higher bandwidth to the memory system.

We implement the Oracle using Asim's multi-threading support. Normally, in Asim, each benchmark is run in two threads—the Feeder Thread and the Timing thread. With Oracle simulation, each benchmark is run in three threads—the Feeder thread, the Timing thread, and the Oracle thread. The Feeder thread feeds instructions to the Timing Thread. The Timing thread corresponds to the conventional detailed pipeline model simulation with out-of-order issue and speculative execution. The Oracle thread, however, fetches the same instruction stream as the Timing thread, but executes each instruction immediately upon fetching.[3]

In Oracle mode, the simulator keeps track of the relationship between corresponding instructions in the Timing and Oracle threads. During the simulation we can find the Oracle thread counterpart to each instruction in the Timing thread. The simulator allows us to query this *Oracle Instruction* for its input values, computed output values, and internal state like branch direction and effective address. This query is possible

[3] Unfortunately, we cannot run the Timing Thread in the same way—that is, execute an instruction completely on fetch—because we need to accurately model the relative occurrence of instruction-related events. For example, we cannot correctly model the impact of speculation and wrong-path instructions without a detailed pipeline model.

even before the respective instruction in the Timing thread has issued, since the corresponding instruction in the Oracle thread has already executed with the correct input values and produced the correct output values. Note that the Oracle thread follows the Timing thread down all paths of execution, including speculative paths. In case of a misprediction recovery, both the Timing thread and the Oracle thread restore their state to the same point, i.e. the killed instruction, and continue to execute along the correct path.

Although other researchers have used the separation of Feeder and Timing threads to model modern out-of-order and speculative processors (e.g., Bechem, et al. [17]), we are not aware of any prior work that used Oracle threads for upper bound studies of the nature explored in this paper.

3.3 Simulated Machine Model

Table 1 shows the configurations of the four effective address predictors we implemented in Asim. Table 2 shows the base processor configuration we use in all our simulations. Figure 1 shows how we integrated our predictors with our realistic pipeline model. All the five predictors—Perfect, LAP, SAP, CAP, and DEAP—make predictions right after the register rename stage of the pipeline. However, to study the upper bound of performance available from effective address prediction with a realistic pipeline, we made the following assumptions:

- We probe and update the predictors in the same cycle. Perfect is always correct, so it does not need to be updated. However, we update the other four predictors using the correct effective address available from the Oracle. Additionally, DEAP requires producer loads to update the DEAP tables with the value it loaded. We assume that this update happens immediately in the same cycle, and not when the producer load commits. In a real implementation, such update may be delayed and must happen speculatively.
- We allow loads for which the predictor makes a correct prediction to go down a *fast path*. The fast path is a bypass from the front-end of the pipe (after the rename stage) to the memory access stage of the pipeline. The fast path never stalls, so the fast path must provide 8 load ports (because we can issue up to 8 instructions, and hence 8 loads, per cycle). [4]

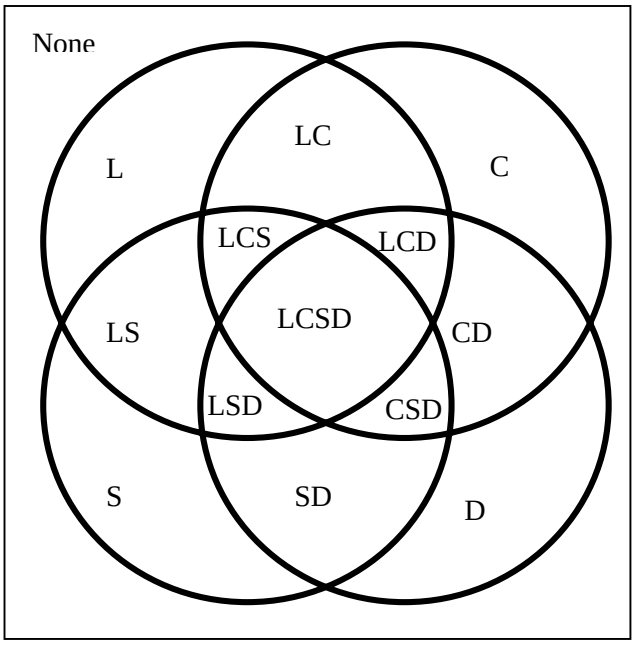

Figure 5. Explanation of columns in Table 4. This Venn diagram shows how we break down the accurately predicted loads into different categories.

- To avoid any unwanted interaction of store-load dependence predictor with our effective address predictors, we use a perfect memory dependence predictor for all our simulations.
- Finally, we have not implemented any recovery mechanism. This allows us to study the upper bound of performance of effective address prediction without having to worry about quirks in individual recovery mechanisms for different predictors.

3.4 Benchmarks

We use a mix of seven benchmarks in our evaluation—three from SpecInt95, one from SpecFp95, and three from the Olden benchmark suite [11]. The SpecInt95 and SpecFp95 benchmarks are from the SPEC suite (http://www.spec.org), while the Olden benchmarks are pointer-intensive benchmarks from Princeton University. We believe these provide a good mix of benchmarks because they represent programs from widely different application areas. We added Olden to our evaluation because the SPEC benchmarks are not particularly pointer-rich. Table 3 summarizes the characteristics of our seven

[4] This increases the total number of load ports in our machine to 12 (4 in the regular path and 8 extra ports on the fast path). However, our experiments, not shown here, reveal that our base machine model (with no fast path) has almost no performance gains with greater than 4 load ports on the seven benchmarks we evaluated in this paper.

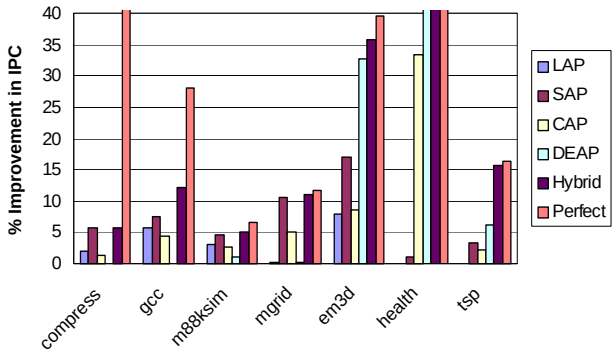

Figure 6. Potential IPC Improvement from Effective Address Predictors. The vertical axis is % percentage improvement in IPC over the base IPCs reported in Table 3. Four bars that are cut-off from the top are: compress-Perfect = 60%, health-DEAP = 218%, health-Hybrid = 234%, health-Perfect = 244%.

benchmarks. *Compress* compresses large text files using adaptive Limpel-Ziv coding. *Gcc* compiles pre-processed source into optimized SPARC assembly code. *M88ksim* simulates the Motorola 88100 processor running Dhrystone and a memory test program. *Mgrid* calculates a 3D potential field**.** *Em3d* simulates the propagation of electro-magnetic waves in a 3D. *Health* simulates the Columbian health care system. Finally, *Tsp* computes an estimate of the best hamiltonian circuit for the Traveling-salesman problem. Em3d, health, and tsp use a variety of pointer-based data structures, such as lists, binary trees, and quadtrees.

We compiled all the above benchmarks using the Compaq GEM compiler tuned for the Alpha 21264 processor at peak optimization levels. Also, we skipped between 30 million to two billion instructions to get to the interesting part of each benchmark.

4 Results

In this section, we discuss our results from the Standalone and Timing Models. We also perform some sensitivity tests with the effective address predictors.

4.1 Standalone Model Results

Asim's Standalone Models allow us to study the prediction coverage and accuracy of our effective address predictors. Perfect is an ideal effective address predictor, whose coverage and accuracy is

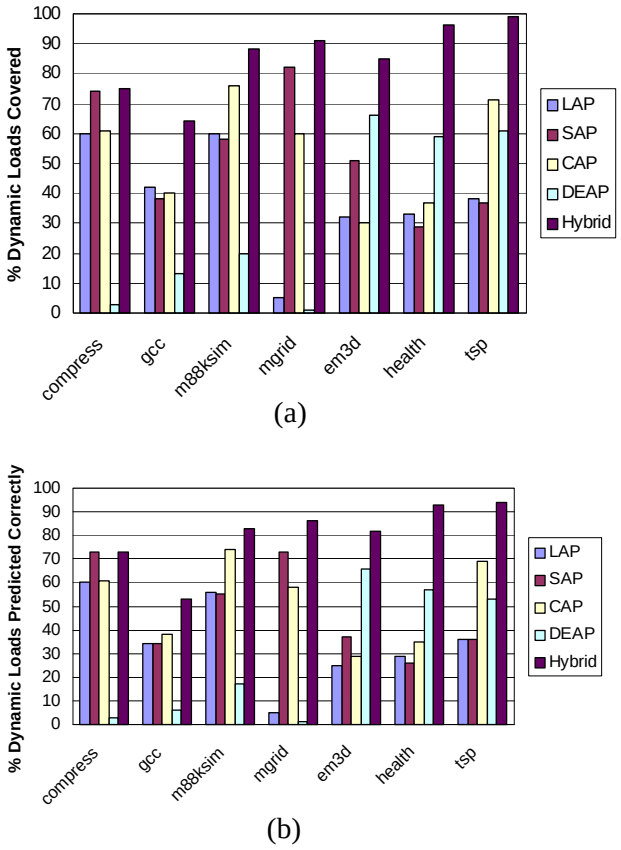

Figure 7. (a) % Dynamic loads covered by all predictors. (b) % Dynamic Loads predicted correctly by predictors. Hybrid means that at least one predictor tried to predict (for graph a) or predicted accurately (for graph b) the effective address of a dynamic load.

100% in the Standalone Model.[5] Figure 7 shows the coverage and accuracy of the other four predictors—LAP, SAP, CAP, and DEAP. Figure 7a shows that a hybrid of the four predictors—LAP, SAP, CAP, and DEAP—could cover (i.e., make a prediction on) a large percentage of dynamically executed loads in our seven benchmarks. Individually, the predictors have widely different coverage. LAP, SAP, CAP, and DEAP cover 39%, 53%, 54%, and 32% respectively of all loads. However, the hybrid predictor on average covers 85% of loads across the seven benchmarks.

Figure 7b shows that an *ideal* hybrid, which can always pick the best predictor for each load, can

[5] Perfect's coverage is, however, not 100% in the Timing model because Perfect does not make a prediction on loads that depend on a recent prior uncommitted store. In contrast, the Standalone model simply executes instructions without a pipeline model and, hence, Perfect makes predictions on, and therefore covers, all loads in the Standalone model.

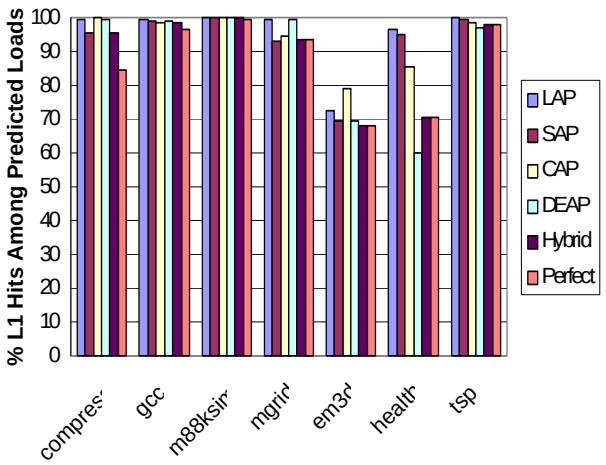

Figure 8. % Predicted Loads That Hit in L1.

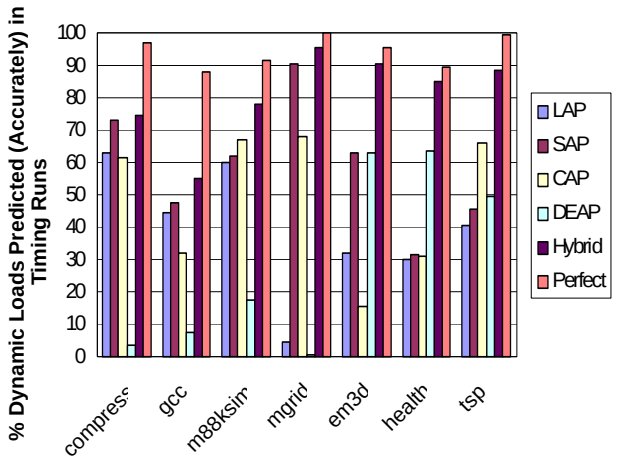

Figure 9. % Dynamic Loads Predicted Accurately in the Timing Runs.

provide very high prediction accuracy. Like the coverage results, the individual accuracy results vary between 29% - 52% across the predictors. However, an ideal hybrid could correctly predict the effective addresses of 81% of dynamically executed loads that it makes predictions on.

Table 4 explains why the ideal hybrid can predict the effective addresses with such high accuracy. Table 4 shows a breakdown of correctly predicted loads for each predictor and

Figure 5 shows the corresponding Venn diagram of predictor coverage. For example, the column S shows the percentage of loads that were correctly predicted only by SAP. Similarly, the SC column shows the percentage of loads that were correctly predicted by both SAP and CAP, and not by any other predictor. The S number does not include the SC number.

As Table 4 shows, SAP, CAP, and DEAP appear to accurately predict different loads. For example, only SAP correctly predicts the effective addresses of 27.9% of the loads in mgrid, while CAP alone correctly predicts 23.8% of the loads in m88ksim. On the other hand, only DEAP accurately predicts between 19% – 49% of the loads in the Olden benchmarks. Thus, an ideal hybrid of these predictors can result in very high prediction accuracy.

4.2 Timing Model Results

Asim's Timing Model combined with Oracle Simulation allows us to study the upper bound of the performance from effective address prediction. As Figure 6 shows, the potential performance improvement from effective address prediction varies widely among the benchmarks. The potential performance improvement (as shown by the Perfect predictor) ranges from as low as 7% to as high as 244%. Interestingly, however, the ideal hybrid—the predictor that always picks the correct predictor for each load—performs very close the Perfect predictor for most benchmarks. Individually, on average, LAP, SAP, CAP, and DEAP result in 3%, 7%, 8%, and 36% performance improvement respectively. The ideal hybrid results in 44% improvement, while the Perfect predictor performs 58% better. Note that the Perfect predictor makes predictions on all loads, except those that are dependent on recent prior stores in flight. Similarly, the other predictors do not make predictions on loads that depend on recent prior stores in flight.

Several interesting results stand out:

- The ideal hybrid performs very well for all benchmarks, except compress. This is because none of the predictors can accurately predict a large fraction (25%) of the dynamically executed loads in compress (Figure 9). Thus, for the hybrid predictor, only 22% of all loads that miss in the L1 cache go down the fast path. In contrast, the Perfect predictor, which can issue the loads that miss earlier in the pipe and thereby prefetch the L1 misses (and L2 hits) ahead of time, predicts accurately 97% of the loads.
- DEAP is extremely effective for the Olden benchmarks and provides a performance improvement between 49% - 63%. This is because DEAP can predict loads that are in pointer-chasing codes, such as the Olden benchmarks. This is because of two reasons. Like the RMS predictor, DEAP can prefetch L2

hits that miss in the L1 cache. However, more importantly, unlike the RMS predictor, DEAP helps prefetch L1 hits. This is critical to performance improvement because a large percentage of the predicted loads actually hit in the L1 cache (Figure 8).

- Finally, we found that increasing the DEAP size had very little impact on performance of these benchmarks. Increasing the DEAP size by roughly seven times gave a performance improvement of less than 1%.

5 Related work

In this paper we draw upon a huge body of prior research on effective address prediction. The predictors LAP, SAP, and CAP are based on work in several papers, such as [4], [7], [8], [9], [10], [13], and [15]. Additionally, many papers, such as [3], [4], and [15], performed a comparative study of some of the predictors and their hybrids.

We improve upon this prior body of research by evaluating DEAP and understanding its prediction rates and performance. DEAP is variant of the RMS predictor [5]. However, unlike the RMS predictor, which was used by Roth, et al. to prefetch cache misses, we use DEAP to prefetch L1 cache hits and issue loads early in the pipeline. We compare DEAP with LAP, SAP, and CAP in terms of their prediction rates and upper bound of performance achievable from these predictors. We also show that on six of our seven benchmarks an ideal hybrid predictor, that picks the correct predictor for each load, can achieve performance close to a Perfect Effective Address Predictor.

Although we focussed on effective address prediction, there are other ways to generate effective addresses earlier in the pipeline. Austin and Sohi [14] proposed overlapping effective address computation with cache access with the help of special circuits and software support. Bekerman, et al. [6] proposed tracking certain registers and immediate values to calculate a load's effective address earlier in the pipeline. We believe that a combination of these techniques along with the effective address predictors we studied in this paper will lead to good effective address prediction rates.

6 Conclusions and Future Work

Modern, deeply-pipelined, out-of-order, and speculative microprocessors continue to be plagued by the latency of load instructions. This latency is dominated by the latencies to resolve the source operands of the load, to compute its effective address, and to fetch the load's data from caches or main memory. This paper examined the performance potential of hiding a load's data fetch latency using effective address prediction. By predicting the effective address of a load early in the pipeline, we could initiate the cache access early, thereby improving performance.

The current generation of effective address predictors for a load instruction is based on either the history or the context of the specific load. In addition, researchers had examined load-load dependence predictors to prefetch cache misses. This paper examined the performance potential of using a load-load dependence predictor to predict effective addresses of load instructions and issue them early in the pipeline. We called this predictor the DEAP predictor.

We showed that on average DEAP could improve the accuracy of effective address prediction by 28% over a perfect combination of last address, stride address, and context-based address predictors across our seven benchmarks from the SPEC95 and Olden suite. We found that an ideal hybrid of these four predictors (including DEAP), which always picked the right predictor for a load, could potentially achieve performance close to that of a Perfect predictor in most cases.

We used an oracle-based simulation approach to evaluate our timing results. This method allowed us to measure the upper bound of the performance from effective address prediction using a mostly realistic pipeline. However, our timing simulation method did not account for penalty due to mis-prediction of an effective address and assumed a zero-cycle latency from address prediction resolution to address predictor update.

This work can be extended in several ways in future. To accurately reflect pipeline effects, one must model the penalty due to the mis-prediction of the various effective address predictors as well as realistic latencies for address prediction resolution to address predictor update. Also, it will be interesting to understand the combined impact of load-load dependence prediction on cache misses (or prefetching) as well as cache hits (as studied in this paper) for long-latency pipelines. An analytical model may help in this effort.

Acknowledgements

We would like to thank Rick Kessler, Geoff Lowney, and Paul Rubinfeld for their valuable feedback on early drafts of this paper.

References

[1] George Chrysos and Joel Emer, "Memory Dependence Prediction Using Store Sets", *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, June, 1998.

[2] Andreas Moshovos, Scott E. Breach, T. N.Vijaykumar, and Gurindar S. Sohi, "Dynamic Speculation and Synchronization of Data Dependences," *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, May, 1997.

[3] Glenn Reinman and Brad Calder, "Predictive Techniques for Aggressive Load Speculation," *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO)*, December, 1998.

[4] Michael Bekerman, Stephan Jourdan, Ronny Ronnen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser, "Correlated Load-Address Predictors," *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, May, 1999.

[5] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi, "Dependence Based Prefetching for Linked Data Structures," *Proceedings of the 8th International Conference on Archtiectural Support for Programming Languages and Operating Systems (ASPLOS)*, October, 1998.

[6] Michael Bekerman, Adi Yoaz, Freddy Gabbay, Stephan Jourdan, Maxim Kalaev, and Ronny Ronen, "Early Load Address Resolution via Register Tracking," *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, June, 2000.

[7] M.H.Lipasti, C.B.Wilkerson, and J.P.Shen, "Value Locality and load value prediction," *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 138-147, October, 1996.

[8] T-F. Chen and J-L Baer, "Effective Hardware-Based Data Prefetching for High Performance Processors," *IEEE Transactions on Computers*, 5(44):609-623, May, 1995.

[9] R.J.Eikermeyer and S. Vassiliadis, "A Load Instruction Unit for Pipelined Processors," *IBM Journal of Research and Development*, 37:547-564, July, 1993.

[10] Y. Sazeides and James E. Smith, "The Predictability of Data Values," *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pp 248-258, December, 1997.

[11] Martin C. Carlisle and Anne Rogers, "Software Caching and Computation Migration on Olden," *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, July, 1995.

[12] James Smith in "Slow Wires, Hot Chips, and Leaky Transistors: New Challenges in the New Millennium," Panel at the International Symposium on Computer Architecture (ISCA), June, 2000. Moderator: Shubhendu S. Mukherjee. Panelists: Robert Colwell, Dirk Grunwald, Mark Horowitz, Norm Jouppi, James Smith, and T. N.Vijaykumar.

[13] J. Gonzalez and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," *Proceedings of the 11th International Conference on Supercomputing (ICS)*, pages 196 – 203, July, 1997.

[14] T.M.Austin and G.S.Sohi, "Zero-cycle Loads: Microarchitecture Support for Reducing Load Latency," *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO)*, pages 82-92, December, 1995.

[15] B. Black, B. Mueller, S. Postal, R. Rakvie, N. Tamaphethai, and J.P.Shen, "Load Execution Latency Reduction," *Proceedings of the 12th International Conference on Supercomputing (ICS)*, June, 1998.

[16] Shubhendu S. Mukherjee, "The Asim Manual," *Compaq Confidential Document.*

[17] Candice Bechem, Jonathan Combs, Noppanun Utamaphethai, Bryan Black, R.D.Shawn Blanton, and John Paul Shen, "An Integrated Functional Performance Simulator," IEEE Micro, pp. 26 – 35, May/June, 1999.