Correlated Load-Address Predictors

Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim,
Lihu Rappoport, Adi Yoaz, and Uri Weiser

Intel Corporation

Intel Israel (74) Ltd., Haifa 31015, Israel

{bekerman,ronen,rlihu,ayoaz,weiser}@iil.intel.com, sjourdan@ichips.intel.com

Abstract

As microprocessors become faster, the relative performance cost of memory accesses increases. Bigger and faster caches significantly reduce the absolute load-to-use time delay. However, increase in processor operational frequencies impairs the relative load-to-use latency, measured in processor cycles (e.g. from two cycles on the Pentium® processor to three cycles or more in current designs). Load-address prediction techniques were introduced to partially cut the load-to-use latency. This paper focuses on advanced address-prediction schemes to further shorten program execution time.

Existing address prediction schemes are capable of predicting simple address patterns, consisting mainly of constant addresses or stride-based addresses. This paper explores the characteristics of the remaining loads and suggests new enhanced techniques to improve prediction effectiveness:

- *Context-based prediction to tackle part of the remaining, difficult-to-predict, load instructions.*
- *New prediction algorithms to take advantage of global correlation among different static loads.*
- *New confidence mechanisms to increase the correct prediction rate and to eliminate costly mispredictions.*
- *Mechanisms to prevent long or random address sequences from polluting the predictor data structures while providing some hysteresis behavior to the predictions.*

Such an enhanced address predictor accurately predicts 67% of all loads, while keeping the misprediction rate close to 1%. We further prove that the proposed predictor works reasonably well in a deep pipelined architecture where the predict-to-update delay may significantly impair both prediction rate and accuracy.

Keywords

Load-address prediction, context-based predictor, global correlation, predictor implementation, recursive data structures.

1. Introduction

One of the key factors in microprocessor performance is the load-to-use delay, i.e. the time required for a load instruction to fetch data from the memory hierarchy and to deliver it to the dependent instructions. Current microprocessors exhibit the following trends:

- They operate at higher frequencies, increasing the average load-to-use delay cycle-wise (load-to-use latency).
- They reduce the overall load-to-use delay mainly by using larger on-chip caches. This improves the on-chip cache hit ratio, eliminating many cycles previously consumed in off-chip memory accesses.
- They feature multi-level on-chip caches to conciliate high frequency of operation and larger on-chip caches.

Despite the use of multi-level on-chip caches, higher frequency and larger caches negatively affect the load-to-use latency of the first-level cache. First-level cache accesses are expected to consume two to five cycles in next-generation processors. The increased latency is needed to perform several activities under a shorter cycle time (lower number of logic levels). These activities include load-address generation, linear-to-physical address translation, tag and data array accesses, tag matching, memory disambiguation, and data delivery. Longer on-chip cache access impacts both the performance and the micro-architecture:

- Load instructions become more likely to be on the critical paths.
- Memory access latencies are more difficult to hide, requiring new techniques to expose more instruction-level parallelism.

Since the above activities are dependent on the load-address generation, one possible way to reduce the load-to-use latency is to predict load addresses ahead of time. Remaining activities, including the cache access, can be processed speculatively early in the pipeline. If the prediction is done early enough, these activities may be overlapped with other front-end processing. If the data delivery phase is also processed speculatively, dependent instructions are executed based on the speculated loaded data. Performing address prediction early in the front-end pipeline may be useful to partially hide load-to-use latencies in accessing lower levels of the memory hierarchy. On the down side, an early address predictor may reduce the correct prediction rate and increase recovery time. Load address predictions are verified when the actual effective addresses are generated.

Load-address prediction reduces the overall load-to-use latency, significantly shortens critical paths, and boost performance. Load-value prediction may be used as an alternate option to reduce load-to-use latency. However, its lower predictability makes this option less attractive.

The concept of address prediction has been extensively investigated. However, all studies have been restricted to the prediction of simple patterns such as constant or stride-based address sequences. Last-address predictor speculates that addresses remain unchanged over several invocations of the same static load (it predicts that $A_{N+1}=A_N$). Last address predictor performs well on global scalar variables, read-only constants, simple, reoccurring, stack references, etc. Stride-based address predictor speculates that consecutive addresses of a given load differ by some constant delta (it predicts that $A_{N+1}=A_N+(A_N-A_{N-1})$). This technique performs well on regular data structures (e.g. arrays) that are linearly traversed. Last-address predictors surprisingly handle an average of 40% of all load addresses, whereas stride-based predictors add an additional 13%. This leaves almost half of the load sequences that exhibit more complex patterns than constant or stride. In this paper, we focus on part of these remaining patterns. We first try to understand their behavior and then depict specific address prediction schemes.

A significant portion of the remaining patterns consists of short sequences that repeat themselves (*repetition property*). Examples for such sequences include (1) repeated traversals of a short recursive data structure (e.g. linked list), and (2) access patterns of load instructions in functions called from different places in a regular and repetitive manner, which are highly dependent on the call sites. Furthermore, many of these patterns also exhibit some global correlation among static load instructions, i.e. address sequences belonging to different static loads, which only differ by some constant offset (*correlation property*).

The repetition property suggests the use of a context-based predictor, which performs well on such short repetitive sequences[1]. We devised an enhanced correlated Context-based Address Predictor (CAP) that includes mechanisms to:

- Efficiently implement context-based predictor.
- Take advantage of the correlation property.
- Avoid mispredictions.

The proposed address predictor can be used as a stand-alone predictor since it can predict stride-based accesses as well. However, it cannot handle long stride-based sequences efficiently. In practice, the CAP predictor should be coupled with a stride-based predictor to achieve high-performance. The hybrid stride/CAP predictor handles 67% of all loads on average; this is about 14% more than what the stride-based predictor achieves. Worth noting that the hybrid CAP helps more in cases where a stride-based predictor performs poorly.

The remainder of this paper is organized as follows. Section 1 discusses related work. Section 2 studies some examples of loads that are not predicted by current predictors, explains their origins and analyzes their address patterns. Section 3 describes the CAP predictor structure and discusses several configuration options. Section 4 presents the simulation methodology and shows the predictor potential by reporting simulation results assuming a hypothetical predictor capable of immediate updates. Section 5 discusses the effect of pipelining on address prediction schemes, and extends the simulation results to include pipeline effects. Some concluding remarks are provided in section 6.

1.1 Prior Art

Related work has been conducted in three different directions:

Reducing the load-to-use latency by performing data prefetching. Data prefetching cuts the overall load-to-use latency by bringing ahead of time, likely-to-be-used data from distant memory to an upper level of the memory hierarchy. The cache block is the atomic unit of work. Unlike address prediction, data prefetching does not require any recovery schemes since predictions are performed for future references. Baer and Chen defined the concept and introduced last-address and stride-based prefetchers [Baer91] [Chen95].

Performing data speculation by predicting result values (*value prediction*). Value locality and data value prediction were introduced by Lipasti and Shen. They studied, first, value prediction for loads [Lipa96a] and generalized it later to include all computed values [Lipa96b]. Those studies used only a last value predictor but they did mention stride-based and control-flow based predictors. Further studies include the investigation of the potential of ideal context-based value predictor [Saze97] and the description of a practical implementation of a hybrid stride and context-based value predictor [Wang97].

Performing data speculation by predicting the address of load instructions (*address prediction*). This is the topic of this paper. [Eick93] proposed a stride-based predictor to speculate on the address of a load during the early stage of the pipeline. The objective was to hide the memory hierarchy latency. This scheme was not speculative since the data delivery was performed once the address prediction was verified. Austin performed similar work to hide the whole load-to-use delay but with operand-based predictors [Aust95]. [Lipa96a] extended the work by proposing the use of last-address predictors to achieve this very same goal. This study also mentioned stride-based address predictors. Finally, [Gonz97] proposed to share the same stride-based prediction structures to perform address prediction and data prefetching (next invocations) simultaneously.

Both value and address predictors employed confidence mechanisms to avoid forlorn accesses. These mechanisms usually consist of simple saturated counters. Recently, [Mora98] devised a mechanism to prevent random accesses from polluting the prediction table.

The contribution of our study over the prior art consists of:

- Further analysis and understanding of the more complex address patterns.
- Design of a practical, efficient context-based address predictor.
- Use of correlation among loads to improve address prediction.
- Trading mispredictions with no prediction by using enhanced confidence mechanisms.
- *Pollution-Free* mechanism to prevent large or non-recurring sequences from polluting the address predictor data structures.
- Discussion and analysis of a pipelined environment.

2. Load Behavior

In this section, we describe two very common patterns that are not predictable by current address predictors. Discussions are illustrated with examples taken from real programs.

2.1 Recursive Data Structures

Recursive Data Structures (RDS) are common structures used in pointer-chasing programs. Simple examples are linked lists and binary trees. A RDS is a collection of elements, each respecting the same format consisting of one or more fields. At least one of these fields is a pointer to another element. In a simple link list for example, there is only one such field and it is often called `next` (figure 1). Another example is a binary tree featuring two such fields called `left` and `right`. These pointer fields are used to traverse the associated RDS. Other fields may record any sort of data attached.

Code sections that access a RDS typically store a pointer to the element currently being processed in a pointer variable. In the simple link list example, switching to the next element, in C, is performed via an instruction of the form `p=p->next;` where `p` is the pointer variable. Such an instruction is compiled into a load instruction that is highly unpredictable by any stride-based address predictor. Note that since this instruction loads a pointer from

[1] Performing address prediction on these sequences results in a bigger performance gain than on stride-based sequences. Indeed, the code associated to stride-based sequences can often be pipelined over successive iterations, resulting in performance gains only at the beginning and at the end. By contrast, in recursive data structures, successive load addresses depend on each other. The address prediction technique is the enabler for parallel execution.

memory, the loaded address must not be confused with the effective address of the access itself. For example in figure 1, if `10-80-40-20` are the base addresses of successive elements, and `next` has an offset of `8` from the base, the pattern of the load associated to `next` is `18-88-48-28`. Furthermore, if the linked list features some data fields, accessing these fields is as unpredictable as for the `next` field. In fact a C-instruction of the sort `sum=sum+p->val;` is compiled into a set of instructions with a load which may follow a pattern close to the load operating on the `next` field.

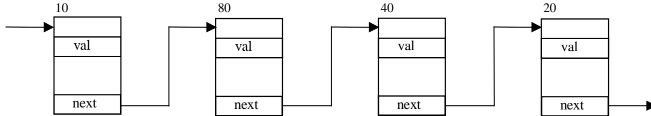

Figure 1. A Simple Linked List.

`Xlisp` (of SPECint95) features some RDS in its code. One of them is the record `NODE`, which specifies two pointers (`car` and `cdr`) and several data fields like `n_type`. The function `xlevarg` operates on this RDS. Its input is a global `NODE` pointer (to the current element). It first verifies that the current element is of a certain type (`n_type==LIST`), and then evaluates the element pointed to by `car`. The function returns the result of the evaluation. The current element is updated to be the one pointed to by `cdr` (next element). This function is called to traverse the RDS and to evaluate each of its elements. The compiled version uses three loads, which access `NODE` for `car`, `cdr`, and `n_type`. We list below a portion of the compiled code where `%ebx` records the memory address where the pointer to the current element is stored:

```
movl  (%ebx),%eax     ; move ptr into eax
cmpb  $0x3,(%eax)     ; load n_type and cmp to LIST
jne   <xlevarg$+4e>   ; exit if different
movl  0x8(%eax),%edx  ; load cdr into edx
movl  0x4(%eax),%eax  ; load car into eax
movl  %edx,(%ebx)     ; store cdr (move to next)
```

A study of the dynamic patterns of these three loads shows that they have small, simple, and recurring sequences of addresses. The sequences are highly stable over time. If `A`, `B`, `C`, `D`, `E`, `F` are different addresses, one fingerprint observed for each of these loads is:

```
A B
A C D E F B
A C D E F B
A C D E F B
A C D E F B
```

This sequence is completely unpredictable by any stride-based predictor.

`Go` (SPECint95) also provides examples of RDS. For instance, its code uses a set of related functions performing various operations on linked lists (functions `addlist`, `mrglist`, `cntlist`, etc.). Most loads in these functions are highly correlated when they work on the same linked list[2]. Recurring patterns are small and fairly stable. `Addlist` exhibits such a behavior as shown below (`A` to `P` are different addresses):

```
A B C D E F G H I J K
A B C D E F G H I J K L
A B C D E F G H I J K L
A B C D E F G H I J K L M
A B C D E F G H I J K L M N
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
A B C D E F G H I J K L M N O
```

[2] The coding of the RDS used in these functions is unusual. It consists of several arrays, each recording one field of the RDS. One of them holds next pointers, which are indices to the arrays.

2.2 Control Correlation

Input parameters may be passed through the stack or via registers. Some of them are highly dependent on the call site. Loads operating on pointers passed via registers or on any type of values passed through the stack may reflect a correlated behavior. If the control-flow is regular, such loads may exhibit recurring and stable patterns.

Many such examples can be found in `xlisp`. For instance, the function `xlmatch` is called from three different functions: `xcond` (`c`), `doupdates` (`u`), and twice in `xaref` (`a`). The pattern of the calls is regular and consists of a recurring sequence: `a-c-u-a`. `xlmatch` features many loads that are highly dependent on the call sites. Since `xlmatch` is called twice in `xaref`, a fingerprint of these loads is:

A_1 A_1 C U A_2 A_2
A_1 A_1 C U A_2 A_2
A_1 A_1 C U A_2 A_2
A_1 A_1 C U A_2 A_2

Where A_1 and A_2 are addresses related to `xaref`, `C` to `xcond`, and `U` to `doupdates`.

Another example is the function `xllastarg` that is called from `xaref` (`a`), `unary` (`u`), `compare` (`c`), and `xbquote` (`b`). The control-flow pattern is `a-a-u-c-b`. Correlated loads in `xllastarg` follow the following pattern:

A_1 A_2 U C B
A_1 A_2 U C B
A_1 A_2 U C B
A_1 A_2 U C B

Where A_1 and A_2 are addresses related to `xaref`, `U` to `unary`, `C` to `compare`, and `B` to `xbquote`.

3. Advanced Predictor

3.1 Context-based Predictor

Context-based predictors successfully handle the recurring and stable sequences described in the previous section. This is regardless of the field targeted. [Saze97] provided a good definition of such predictors as:

Context-based predictors learn the value(s) that follow a particular context – a finite ordered sequence of values – and predict one of the values when the same context repeats. This enables the prediction of any repeated sequence, stride or non-stride.

Such predictors record the history of recent past addresses for each static load. Predictions are performed based on the history values. This suggests a two-level mechanism for predicting such loads (figure 3):

- The first level is a per-static-load table, called the *load buffer* (LB), in which each entry records the history of recent addresses exhibited by the associated load. Other LB fields are described later in this section.
- The history found in the LB is used to index a second level table, called the *link table* (LT), which provides the predicted address.

3.2 History Length

For a given static load, the history of recent past addresses is used to predict the current address. For instance, in figure 1, for the load operating on the `next` field, the address `18` is used to predict `88`, which is used to predict `48`. There are two questions that need to be answered concerning the history:

How many past addresses does the history need to record? To predict RDS accesses correctly, the history may need to record more than one address. Figure 2 gives an example of a double linked list. In this example, the pointer fields, `next` (offset `6`) and `previous` (offset `8`), require the recording of only one address in the history. Indeed for next, `86` is usually followed by `46`, which is followed by `26`. For previous, `28` is followed by `18`. On the other hand for `val` (offset `2`), `82` may be followed either by `12` or `42`. It requires a history of two addresses to figure out the direction of the traversal: `12` and `82` are followed by `42`, while `42` and `82` are followed by `12`. RDSs typically feature not more than two pointer fields. Keeping the last two or three addresses in the history should be sufficient to achieve adequate prediction rates. However, predicting control-correlated addresses requires longer histories. This is due to the fact that the function may be called several times in a row with the same input parameters. Typically, such sequences do not exceed four to five repetitions, which means that keeping the four last addresses in the history should be sufficient.

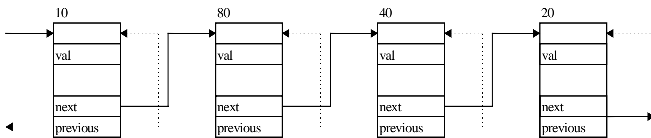

Figure 2. A Double Linked-List.

Which scheme is used to compress the history into a vector small enough to index the LT while preventing most aliasing? Since addresses are long, we cannot index the LT with the concatenation of the entire addresses of past accesses. We found out that an efficient scheme to compact the past addresses in the history is the `shift(m)-xor` scheme. When updating the current history with the last address, this scheme first shifts left the history by `m` bits and then xors it with a subset of the new address. The subset includes all least-significant bits except for the last two, which only matter on unaligned accesses. The result is then truncated to fit the history size, and stored as the new history. The `shift(m)-xor` scheme performs well since it naturally ages past addresses.

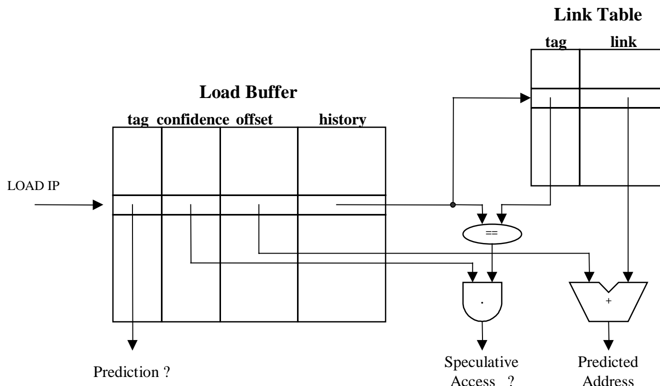

Figure 3. CAP Structure.

3.3 Global Correlation

As mentioned in section 2, the sequences that we want to predict exhibit a high-level of global correlation. The loads associated to different fields of an RDS follow patterns that differ only by a small constant offset. Currently, links are independently recorded for every field in the RDS. To address this global correlation, we propose to record base addresses instead of real addresses in both the LB and the LT. A base address is defined as the real address minus the immediate offset as specified in the load instruction opcode. All loads associated with the same RDS share the same set of base addresses. Histories in the LB keep track of past base addresses while predicted base addresses are recorded in the LT. The output of the LT now needs to be added to the immediate offset recorded in the LB for any given static load, to form the predicted address as depicted in figure 3. Using base addresses gives the following benefits:

- It dramatically reduces the number of links recorded in the LT since loads associated with different fields of an RDS now share the same entries.
- It decreases the number of mispredictions. Indeed, whenever a load is mispredicted, all the fields are updated at once. This prevents further mispredictions on other fields.
- It increases the prediction rate. For instance in the linked list in `go`, described in the previous section, any update to any field will benefit all the loads that access this list regardless of the containing function or the field accessed. This is highly beneficial when the update is performed before a related load is predicted. Under certain circumstances, this may lead to a perfect prediction of some loads that are otherwise completely unpredictable.

The drawbacks of such a scheme are:

- It may require recording more addresses in the history since all pointers from the same RDS now use the same indexing information.
- It lengthens the predictor access time since it adds ALUs past the LT access. These ALUs may compute only a few bits as shown below.
- It may create much more aliasing in the LT. For instance in `go`, in the linked list described in the previous sections, the base addresses are in fact indices to the arrays. This results in aliasing between different linked lists since only the offsets distinguish between them (the offsets are the base addresses of the arrays). The same is correct for loads accessing hash tables.

In order to prevent this LT aliasing we record in the LB only the least significant bits of the load instruction offset (typically the 8 LSBs of the original offset) and keep all the MSBs of the load address in the base address. This does not significantly impair the performance of the base address scheme since RDSs are aligned and their typical size is under 256 bytes. Recording only part of the offset in the LB also results in a significant space saving (3 bytes per LB entry) and a faster predicted address computation (no carry propagation past the 8 LSBs).

A potential alternative to the base address scheme depicted so far is to record deltas between successive accesses instead of base addresses both in the history patterns and the LT. Such a scheme may be highly efficient especially when dealing with stack references in control-dependent loads, and it takes advantage of any kind of global correlation. However, the amount of additional aliasing due to false global correlation makes this option less attractive.

3.4 Enhanced Confidence Techniques

The penalty for performing a speculative access with a mispredicted address is higher than the penalty of not speculating the load. This is highly dependent on the aggressiveness of the recovery mechanism, but in the best case, mispredictions lead to:

- Wasted resources due to both the speculative cache access and the unnecessary execution of dependent instructions.
- A longer load-to-use latency since at least the address verification latency is added.

On a LB hit, a load-address prediction is always performed. A speculative cache access is initiated only if the prediction is carried out with high enough confidence. We use three mechanisms to determine the confidence, and we execute a speculative access only when all agree. The first two schemes are per-load, i.e. the confidence information is recorded in a confidence field added to each entry in the LB, as shown in figure 3. The last scheme is per-link, i.e. saved in each LT entry. The schemes are:

- **Saturating Counters.** A saturating counter which is incremented on a correct prediction and reset on a misprediction. The counter saturates at some threshold value (typically 2 or 3). Only when the counter reaches this threshold value, it indicates that the speculative access can be performed. A more advanced scheme may provide some hysteresis behavior via the use of an extra bit.
- **Control-Flow Indications.** When a speculative access is performed and the predicted address is incorrect, the `n` (typically 1 to 4) LSBs of the global branch history register (GHR) are recorded in the LB. Subsequent predictions for which the recorded pattern matches the current value of the GHR will not access the cache. This scheme records only information related to the last misprediction. A more advanced scheme may record 2^n bits to deliver better performance by encoding the prediction correctness information on 2^n different paths (resulting from the last n branches). Each of the bits is associated with a single path, and it records the accuracy of the last speculative access done on this path.
- **LT Tags.** The LT is currently indexed with the history values. We propose to extend the size of the history values, and to index the LT with enough LSBs of the history while the remaining MSBs are used for tag matching. The new tag field is therefore updated with the bits not used for set selection. Speculative accesses are performed only on tag matches as depicted in figure 3. This also gives the opportunity to implement a set-associative LT.

3.5 Reducing Pollution

Evicting entries from the LT can have devastating effects on overall predictability. Since CAP tries to predict all loads (we do not rely on any static classification scheme), the LT needs to be updated after any resolution. However, since many loads are completely unpredictable by nature, they may trash the LT.

A very large LT can overcome this problem but this option is not attractive. An alternative is to update the entry only when the pattern is recurring. A simple implementation is to extend each LT entry to record a few bits (PF – pollution-free bits) of the last base address to update (typically bits 2 to 5). The PF bits are always updated. By contrast, all the other fields of a LB entry are updated only if the PF bits of the new update match those recorded. This means that to be recorded in the LT, a link must be seen twice in a row, without any other load touching this entry in the meantime. This prevents irregular loads from updating the LT since it is unlikely that such links repeat in a short enough time. This also prevents very long sequences that would have not fit into the LT anyway, from updating the LT. The drawback of such a scheme is the increase in training time. However, the eviction problem is much more important.

Another benefit of the PF bits is the hysteresis behavior added to the LT. Indeed, a change in the load behavior now needs to be experienced twice before actually updating the LT.

Finally, the PF field does not need to be linked to the LT. We may use a direct-mapped table that features more entries to record them [Mora98]. The extended index is formed with the history bits used to tag the LT. This enables a finer granularity in preventing harmful LT updates.

3.6 Control-Based Address Predictors

To tackle control-dependent loads, an alternate option is to predict addresses with structures similar to branch predictors. For instance, a *g-share* scheme can be used to `xor` the instruction-pointer of the load with the current global-history register and to use the result as an index to a table recording predicted addresses. Such a scheme gives poor results mainly because the loads are not well correlated to all the individual conditional branches. Using a history formed by the instruction pointers of the recent callers (path history over recent call sites) gives better results. However, performance highlighted by initial simulation results, does not seem good enough to consider a control-based address predictor as a viable substitute. Using such a scheme on top of the hybrid CAP may help, but such a study is beyond the scope of this paper.

3.7 Hybrid CAP

The CAP predictor does not handle constant or stride-based address sequences cost-effectively. This is because of the high overhead associated with the LT and the presence of non-recurring or very long address sequences. A hybrid CAP/stride address predictor should overcome both issues.

The proposed hybrid predictor does not require extra prediction structures since the LB can be shared between the two prediction components. The extra information recorded in the LB consists of the regular fields used in stride-based prediction schemes, namely the last address, the stride delta, and some state bits. The number of LB entries required by any address predictor (CAP, stride, or hybrid) to achieve the same performance is mostly the same. Both stride and context-based predictions are performed for each dynamic load and the prediction tables are updated accordingly. The LB is always updated (for both stride and CAP) whereas the LT may be updated only on certain conditions as studied in section 4.3. The LT may feature fewer entries when using a selective update policy.

Each of the two components of the hybrid predictor features its own confidence bits. Speculative accesses are performed only when at least one predictor is confident enough to deliver a predicted address. A *selector* mechanism is required when both components are sufficiently confident. This selector can be fully static, always giving priority to the same component. An alternative is a dynamic scheme using 2-bit counters to record the past performance of each predictor with respect to the other. The next section reports on prediction result with this dynamic selection scheme where selection counters are attached to every LB entry. Figure 4 depicts a block diagram of our hybrid CAP/stride address predictor.

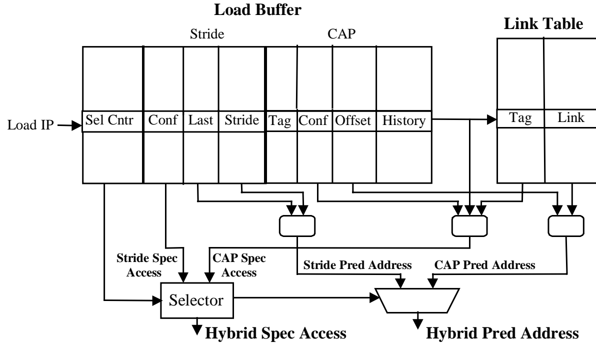

Figure 4. Hybrid CAP/Stride Predictor Block Diagram.

4. Simulation Results

4.1 Simulation Methodology

Results provided in this paper were collected from an IA-32 trace-driven simulator. Each trace consists of 30 million consecutive IA-32 instructions. Traces are representative of the entire execution, and they record both user and kernel activities. Results are reported for 45 traces grouped into 8 suites:

- SPECint95 (**INT**) - 8 traces,
- CAD programs (**CAD**) - 2,
- multimedia applications using MMX™ instructions (**MM**) - 8,
- games, e.g. Quake (**GAM**) - 4,
- programs written in JAVA (**JAV**) - 5,
- some TPC benchmarks (**TPC**) - 3,
- common programs running on NT, e.g. Word (**NT**) - 8,
- and common programs running on Windows 95 (**W95**) - 7.

In this study we used a detailed performance simulator modeling an 8-wide 128-deep out-of-order processor featuring 10 functional units and 4 data cache ports, an aggressive instruction fetch mechanism, and a hybrid branch predictor. The L1 on-die data cache is 32KB while L2 is 1MB. Instruction latencies are common to Intel's processors. An efficient dynamic memory disambiguation scheme is used to order load and store accesses.

Address prediction is performed in an early stage of the pipeline, partially hiding the longer load-to-use latency of accessing lower levels of the memory hierarchy. The predicted address is verified in the memory-ordering buffer past address generation. We simulated a non-aggressive **selective** recovery mechanism to handle address mispredictions, i.e. only the dependent instructions already scheduled are re-executed.

4.2 Predictor Performance

This section presents performance results of both a stand-alone CAP predictor and a hybrid CAP/enhanced stride predictor. Results are provided for an immediate table-update policy similarly to all prediction papers published up to now. The baseline table configuration is a 4K-entry 2-way set associative Load Buffer and a 4k-entry direct-mapped Link Table. As explained in section 3, the LT records base addresses. Control-flow indications, PF bits, and LT tags are all used. The enhanced stride-based predictor features the control-flow indications and the *interval* technique described in section 5, to trade mispredictions with no-predictions.

Figure 5 shows the prediction performance results for stand-alone CAP, enhanced stride-based, and hybrid address predictors. The prediction rate metric is defined as the percentage of loads for which speculative accesses were performed, including both correct and incorrect predictions, out of all dynamic loads (see section 3.4). The accuracy, i.e. the correct prediction rate, is defined as the percentage of correct predictions out of all speculative accesses. CAP predicts 61% of all dynamic loads on average. Except for MM, this is 5 to 13% higher than the enhanced stride-based predictor, while misprediction rate decreases by 40% on average. MM results differ since these applications mainly process large arrays which CAP, with its limited storage, can hardly handle. Overall results clearly show the potential of the CAP predictor and its importance in any address prediction scheme. The hybrid CAP/enhanced stride predictor succeeds to predict 67% of the loads with 98.9% accuracy (slightly less than CAP stand-alone).

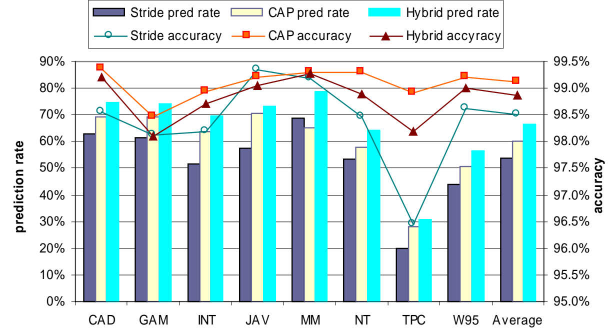

Figure 5. Prediction Performance of the Different Predictors.

In figure 5, we simulated a hybrid CAP/stride predictor with a dynamic 2 bit-counter selector initially biased towards weak CAP selection (CAP base misprediction rate is lower). The counters are updated after the address verification phase based on the relative performance of each prediction component. Figure 6 shows both the prediction rate and the accuracy of the hybrid predictor as a function of both the number of LB entries and the associativity. The prediction rate of the CAD, JAVA, NT, TPC, and W95 suites steadily increases with bigger sizes. These applications typically feature a higher number of static loads. As shown in the figure, a 2-way LB is definitely a win while higher associativity is less cost-effective. We listed accuracy only for a 4k-entry 2-way LB as it remains constant over the different configurations. Accuracy is very high across all benchmarks (98.9% on average), and is higher than the accuracy of any published address predictor.

Prediction rate sensitivity to LT sizes (not shown on the graph) is significant for applications with high volatility of load addresses, like CAD, INT, JAV and MM. On average the hybrid prediction rate steadily increases from 63% for 1K-entry LT to about 68% for 8K LT. On the other hand, the LT associativity has low impact on performance, since the history distribution is quite even.

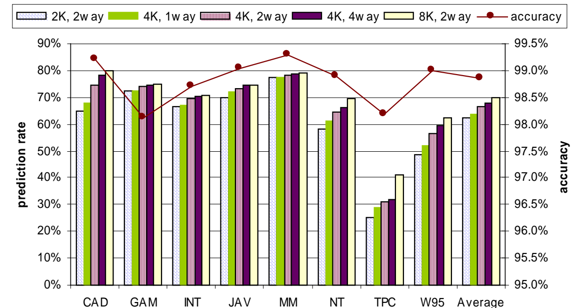

Figure 6. Prediction Performance of Hybrid CAP/Enhanced Stride as a Function of the LB Number of Entries.

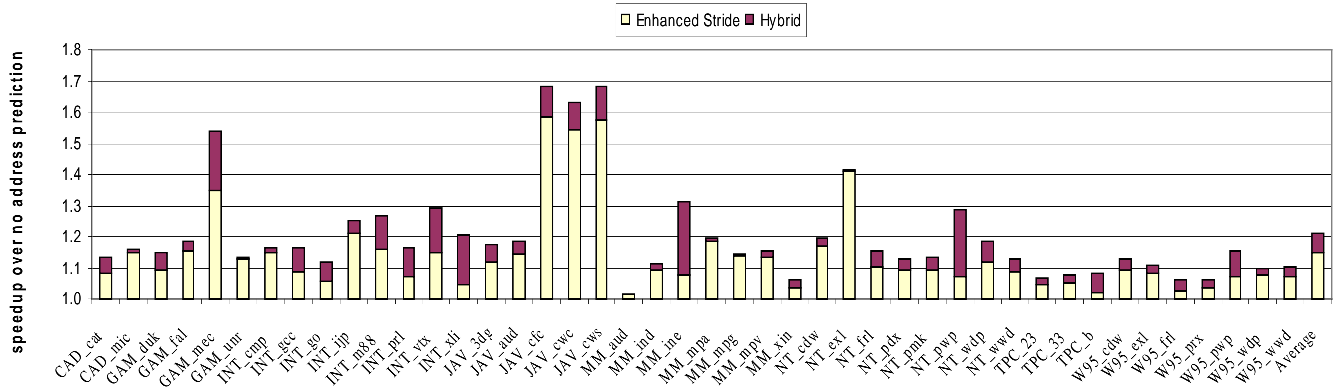

Figure 7. Relative Performance of Enhanced Stride-based and Hybrid Address Predictors

Finally, figure 7 highlights the processor performance improvement achieved by an address prediction scheme. Our baseline processor configuration was depicted in section 1.2. Results are given for both the enhanced stride-based and the hybrid CAP/stride predictors. Most traces exhibit a speedup in the range of 10-25%, with an average of 21%. As expected, the performance improvement is lower on TPC and W95 traces where prediction rates are lower due to relatively high contention rate on the LB. The unusually large speedup reported for JAVA applications can be explained by the large number of memory operations. This is due to the stack-based model and short procedures used in JAVA bytecode, and to the lack of optimizations performed by JAVA JIT compilers. Overall, the hybrid predictor is clearly a win, improving performance by 6.3% on average over the enhanced stride-based predictor. It is interesting to see that non-stride loads (loads which are predicted correctly by the CAP predictor and not handled by the stride predictor) are more critical to performance. The 20% non-stride loads (14% out of 67%) contribute 30% to the performance (6.3% out of 21%). Note that performance results are presented mainly to give a flavor of the potential benefit. This is because actual performance benefits are highly dependent on the implementation and may vary greatly. An arbitrary configuration (whether current or futuristic) may give biased results of questionable significance.

4.3 LT Update Policy

As mentioned earlier, we may omit updating the LT on certain events to reduce LT interference and save predictor space. Note that the PF bits mechanism already filters out LT updates from irregular loads. We consider three update policies:

- Update always
- Update unless the stride component predicts correctly
- Update unless the stride component predicts correctly and its prediction was selected to perform the speculative access.

Surprisingly enough, the *update always* option results in slightly better prediction results on almost all traces. An inspection of address sequences in the traces revealed multiple cases of unstable stride-like behavior, as in the JAVA inner loop address sequence shown below (we present only the 16 LSBs of each address for the sake of simplicity):

```
939a, 939c, 939e, 93a0, 93a2, 93a4, 93a6
9eb9, 9ebd, 9ec1
9eb9, 9ebd, 9ec1
9ecd, 9ecf, 9ed1
```

This inner loop is executed multiple times, resulting in a very low stride prediction rate. On the other hand, recording all the links in the LT results in 100% correct predictions once the warm-up is over. In other words, updating or not updating the LT on all loads represents a trade-off between additional CAP predictions and a potentially lower number of predictions on applications exhibiting many LT conflicts. Simulations showed that for the LT size we considered (4K-entry) updating the LT on all loads is more important.

4.4 Selector Performance

Around 80% of the speculative accesses are loads predicted by both the stride and the CAP components of the hybrid predictor. The above example illustrating a JAVA inner loop strongly suggests a dynamic selector. Figure 8 shows the distribution of the selector counter states for all loads predicted by both prediction components. Almost 90% of these loads are predicted while the counters are in one of the two states selecting the context-based predictor. The *always update* policy causes most predictions to be performed by the CAP predictor. The loads predicted by the enhanced stride-based predictor are mainly the long stride sequences. Figure 8 also highlights the performance of the selection mechanism by reporting the correct selection rate. A miss-selection is defined on a misprediction when reversing the selection would have given a correct prediction. The very high correct selection rate clearly shows that the 2-bit counter selection algorithm is quite close to perfect.

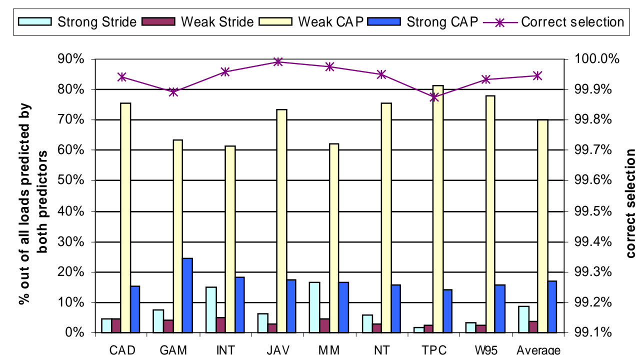

Figure 8. Selector Performance.

4.5 Individual Features

Global Correlation. The ability to share prediction information among multiple static loads (*global correlation*) is one of the most

important features of the CAP predictor. Figure 9 emphasizes the potential of such a feature in the working example of a stand-alone CAP predictor. Note that no confidence mechanism was used to isolate the influence of the correlation. The metric used in this subsection is the number of correctly predicted speculative accesses out of all dynamic loads. We varied the number of past addresses included in the history values (history length). The potential of global correlation is estimated around 10% of all dynamic loads. In section 3, we claimed that global correlation should require a larger history length. Figure 9 shows the optimal history length to be 2 without global correlation, and either 3 or 4 with global correlation. A history length of 4 was our default configuration.

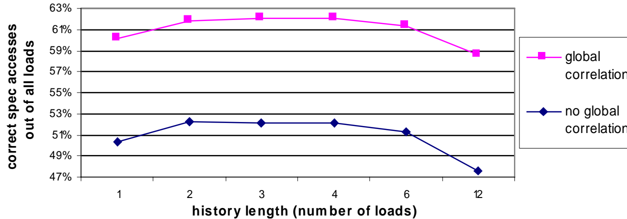

Figure 9. Correct Prediction as a Function of the History Length.

LT Tags. Figure 10 shows the prediction performance of the CAP predictor for different confidence mechanisms. The simplest scheme with no special confidence schemes beside the regular state bits achieves prediction rate of 64.2% with a misprediction rate of 3.3%. Such a high misprediction rate would require a very aggressive recovery mechanism to achieve significant performance improvement with address prediction. The addition of LT tags consisting of several bits from the history pattern, drastically reduces the number of wrong predictions. Using only 4 bits of tags reduces the misprediction rate by as much as 57% while predicting only 2% less dynamic loads. Doubling the tag size to 8 bits further decreases the misprediction rate by an additional 26%. This clearly shows that incorporating tags in the LT is an extremely efficient confidence scheme: it substantially reduces the number of wrong predictions while only marginally decreasing the number of correct address predictions. Note that the history length is increased when adding tags to the LT. One may assume that increasing the history length contributes to the low misprediction ratio. Simulations show that using a history length of 6 addresses reduces the misprediction rate by only 6%. This means that the longer history effect is in fact marginal compared to the gain provided by the tags.

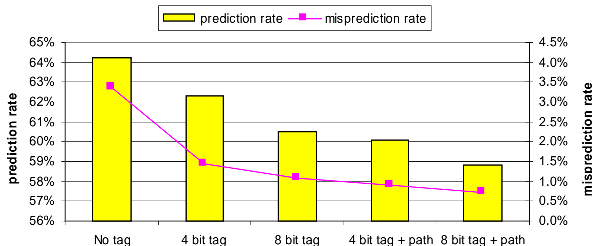

Figure 10. Influence of LT tags on the CAP predictor performance.

Control-Flow Indications. Figure 10 also shows the usefulness of the control-flow indications to achieve further decrease in the misprediction ratio. Using path information with 4 bits of tags reduces the misprediction rate by 39% (from 1.5% to 0.9%), while using it with 8 bits of tags reduces the misprediction rate by 33%, to only 0.7% of all speculative accesses.

5. Pipeline Issues

5.1 Address Prediction in a Pipelined Machine

The discussion of address prediction so far assumed a simplified machine model in which a given prediction can be individually carried out and resolved before any new prediction is made. To date, all published predictor evaluations made use of such a model. In reality, today's processors are pipelined, super-scalar, out-of-order, and speculative. These features affect the performance of address predictors (both stride and context-based predictors) and impose several problems on their implementations. One cannot ignore these aspects when designing an address predictor[3].

In our studies of address predictors we tuned the existing schemes, devised new mechanisms to deal with such real-life implications, and studied their effect on performance. These mechanisms are required to ensure correct operation and reasonable performance with a realistic processor pipeline. In this section, we briefly explain the issues involved with address predictor in the context of a heavily pipelined microprocessor, and we present simulation results taking pipelining into consideration. The results presented in this section show that despite the negative impact of pipeline issues on address prediction, address predictors, both stride and context-based, still boost performance significantly.

5.2 Multiple Pending Predictions

In a pipelined machine, the resolution of a prediction is performed only some number of cycles after the prediction itself is made. In a heavily pipelined machine this number may be quite big (dozens of cycles). During the period between prediction and resolution, new predictions may be carried out. In particular, several predictions may be done for the same static load instruction. To allow multiple pending predictions, the predictors maintain some speculative states. Once a prediction is resolved, the speculative states are checked and fixed in the case of a misprediction. Since those predictions are correlated, the distance between prediction and resolution enables a single wrong prediction to propagate and to cause additional mispredictions later down the pipe. The level of the misprediction propagation depends on the prediction scheme used and the pipeline depth. Misprediction propagation issues are explained below for both stride and context-based address predictors.

A stride-based address predictor, once it mispredicts, will most probably mispredict all the pending predictions that were made for the same static load instruction prior to the resolution of the offending prediction. The number of these pending load instructions depends on the pipeline length. However, in cases where the misprediction is due to a single wrong stride (like skipping over a single array element or going back to the first element of the array), the stride predictor may catch up easily once the misprediction is found. In other words, once the mispredicted load is resolved, the stride predictor starts predicting subsequent instances of the same load correctly even when pending instances are still unresolved. Such predictions are performed by extrapolating the currently known information, i.e. the stride is multiplied by the number of pending unresolved loads and added

[3] In a broader sense, this pertains to almost all kinds of predictors and is not unique to address prediction (with the exception of branch prediction where only the recovery problem remains).

to the new base address. In addition, a stride-based predictor implementation may choose to trade mispredictions with no-predictions by recording the number of array elements (the *interval*) and stop issuing speculative accesses once this number is exceeded for a given static load instruction. The *interval* value represents the number of consecutive correct predictions.

The misprediction propagation effect in a context-based predictor is more acute. Consider, for example, a tight loop that traverses a linked list. Any single misprediction has a *domino* effect. Subsequent predictions of the same static load instruction use a wrong history, and thus wrongly predict the new base. This chain of wrong address predictions occurs as long as a new instance of the load instruction is fetched before all previous pending predictions of that static instruction are resolved. The reason for this is the absence of any catch-up mechanism in context-based predictors. Note that speculative accesses are no longer performed once the faulting load has been resolved. The absence of a catch-up mechanism also lengthens the warm-up time in a context-based predictor.

The misprediction chain terminates only when the time gap between two instances of the same static load instruction is large enough to allow all previous address references to be resolved and updated in the prediction tables. In practice, this may be forced by some dynamic events like a branch misprediction or an instruction cache miss. In the case of a linked list traversal, a branch misprediction is likely to happen when the traversal is over. Correct context-based predictions should resume on the next traversal.

5.3 Simulation Results

In this section, we present simulation results for a hybrid CAP/enhanced stride predictor with realistic pipeline effects. The address predictions and the speculative accesses are performed in the front-end pipeline, while the verifications are done only after the actual load addresses have been generated. We call the number of pipeline stages between an address prediction and the corresponding verification the *prediction gap*. The prediction gap defines the minimum time between prediction and resolution. The actual gap for a given load is variable. It depends on the processor dynamic states. Performing the prediction earlier in the front-end pipeline increases the gap, potentially worsening the pipeline influence, but at the same time increases the gain from correct predictions.

The predictors are optimized to reduce negative pipeline effects by using the mechanisms described above:

- The stride predictor uses interval counters to record array length.
- The stride-based predictor and the CAP context-based predictor stop speculating following a misprediction. The stride-based predictor features a catch-up mechanism.

‘Figure 11 shows the average address prediction rate and prediction accuracy over all traces as a function of the prediction gap. *Immediate* means that each prediction is verified before any additional prediction is performed, as in Section 4. Results are recorded for an enhanced stride predictor (lower bar) and for a hybrid CAP/stride predictor (upper bar).

The prediction rate of a hybrid predictor decreases by about 7% with a realistic pipeline. As the graph shows, most of this decrease comes from the CAP prediction component. However, the prediction rate almost does not change when increasing the prediction gap. A longer prediction gap causes slower prediction catch-up, but its influence is quite low, as the graph indicates.

On the other hand, prediction accuracy is significantly hurt by high prediction gap values. The longer the pipeline between prediction and verification, the more loads are predicted with outdated information, and in the case of an address misprediction, the longer this misprediction is propagated. Even for a short prediction gap of 4 cycles the accuracy of the hybrid predictor drops from 98.9% to only 96.6%, and to a low 96.1% with a prediction gap of 12 cycles.

Overall, for the hybrid predictor, the percentage of **correct** speculative accesses decreases from 65.9% of dynamic loads with immediate update to 57.9% with a prediction gap of 4 and to 57.4% with a gap of 8. The correct prediction rate of the hybrid predictor is 8.6% higher than the one exhibited by the enhanced stride predictor. Despite pipelining effect, both the enhanced stride and the hybrid predictor succeed in predicting successfully a significant part of the loads.

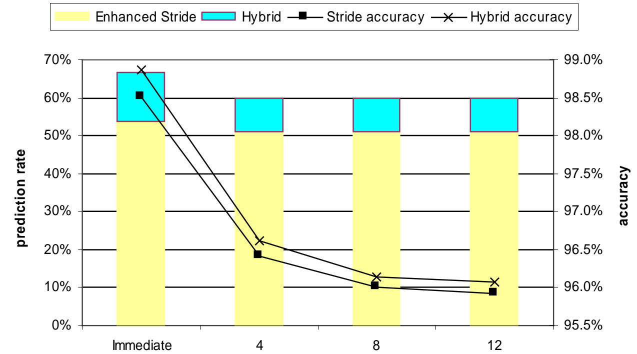

Figure 11. Influence of the Prediction Gap on the Predictor.

Figure 12 shows the performance improvement achieved by the address predictors for both an immediate update (left bar) and a prediction gap of 8 cycles (right bar). In both cases we simulated a realistic aggressive out-of-order processor configuration, summarized in section 4.1. The same pipeline depth was used for all simulations. While the speedup decreases for most of the benchmarks, it is still quite significant. The average speedup of hybrid predictor with a gap of 8 cycles becomes 14.1%. This is 3.9% more than the enhanced stride predictor alone. Note again, that performance benefits are highly dependent on the implementation and may vary a lot.

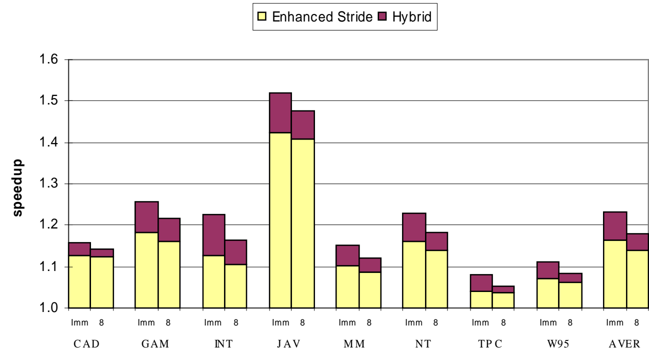

Figure 12. Relative Performance of the Enhanced Stride and the Hybrid predictors for a Prediction Gap of 8 Cycles.

5.4 Other Implications on Micro-Architecture

In addition to pipeline issues, there are other implementation effects which affect the performance of an address predictor (and in a way, any predictor). These are not new and are presented here to highlight their complexity and to give the feeling of their magnitude.

- **Speculative Control-Flow.** Address prediction is made on a speculative control path. Once in a while, a branch is mispredicted and address predictions are made on the wrong path. The branch misprediction recovery should ensure, as possible, that future address predictions (on the correct path) use information which pertains only to the correct path, ignoring any information that is related to the wrong, mispredicted path. One should not confuse this with the *multiple pending predictions* problem mentioned above: predictions may be control-speculated and eventually flushed even if the predicted load address was verified and found correct. A reorder buffer-like or history buffer recovery mechanism is required to prevent destructive updates.
- **Several Predictions in a Cycle.** In a super-scalar machine, several load instructions may be fetched/decoded in the same cycle. The prediction mechanism should allow for several predictions and verifications within a cycle. An extreme case of this problem is performing several predictions/verifications of the same static instructions in the same cycle. In particular, performing several context-based predictions in the same cycle for the same static load instruction is quite complex. It means that the LT should be iteratively scanned within a single cycle. Context based prediction of multiple addresses ahead can be done by applying a mechanism similar in concept to the two-block ahead branch predictor [Sezn96].

6. Concluding Remarks

Summary. In this paper we presented novel mechanisms to improve address prediction. We introduced a practical, efficient, *context-based* predictor to predict a portion of the traditionally difficult-to-predict load addresses. We optimized the address predictor structure and increased the prediction rate by taking advantage of *global correlation* among different static loads. We reduced the number of mispredictions and prevented table pollution by applying smart *confidence mechanisms* and *pollution eliminating* structures. We constructed a *hybrid stride/context based* address predictor to achieve better prediction rate while keeping structures to a reasonable size.

We conducted extensive simulations to demonstrate the potential of the proposed mechanisms. The simulations clearly show the added value of the hybrid address predictor. The prediction rate of the hybrid predictor is about 14% higher than the prediction rate achieved by the stride predictor, while the misprediction rate is 27% lower. We extended our simulation to understand the implication of a more realistic, *pipelined environment* on address prediction. We showed that the suggested hybrid address predictor, although exhibiting some performance degradation and lower correct prediction rate in a pipelined environment, is still beneficial.

Future Work. The work on address prediction in general, and on hybrid stride/context based predictor in particular is in its early stage. Further work may continue in several dimensions.

The first and most important area, in our opinion, is to acquire deeper understanding of the load address patterns in order to trigger ideas for newer, different, predictors. Our observations of load addresses resulting from RDS and control correlation are a step in the right direction. Two examples of such patterns are stack and control flow related accesses. There are still about one third of all load addresses that we do not attempt to predict - we should strive to reduce this number.

Other potential future directions follow the work done on other predictors, mainly branch predictors. This includes:

- Profile feedback/Software assist: to ease the hardware work by letting the compiler/profiler classify loads according to the expected address pattern: last value, stride, context based, unknown, etc… This reduces warm-up time, helps reducing predictor size, and eliminates prediction table pollution. Other supporting information like array/list count may be considered.
- Improving the predictor by applying novel ideas like variable history length, history correlation, etc. These ideas were tried on branch prediction and they seem promising. They may be useful for address prediction as well.
- Tuning the predictor parameters to increase predictor performance. In particular, smart history recording and better confidence tracking should be considered. As in other predictors, determining the right amount of information is an art unto itself: using longer histories and more conservative confidence level reduce the number of attempted predictions but improve the prediction accuracy rate.

7. Acknowledgements

We express our gratitude to Laura Cane for her help in polishing the paper, and to the anonymous reviewers for their insightful comments.

8. References

[Aust95] T. M. Austin and G. S. Sohi, "Zero-cycle Loads: Microarchitecture Support for Reducing Load Latency," in *MICRO-28*, 1995.

[Baer91] J. Baer and T. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," in *Supercomputing'91*, 1991.

[Chen95] T. Chen and and J. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," in *IEEE Transactions on Computer*, V.45 N.5, 1995.

[Eick93] R. J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," in *IBM Journal of Research and Development*, 1993.

[Gonz97s] J. Gonzalez and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," in *Supercomputing'97*, 1997.

[Lipa96a] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *ASPLOS-VII*, 1996.

[Lipa96b] M. H. Lipasti and J. P. Shen, "Exceeding the Dataflow Limit via Value Prediction," in *MICRO-29*, 1996.

[Mora98] E. Morancho, J. Llaberia, and A. Olive, "Split Last-Address Predictor," in *PACT'98*, 1998.

[Saze96] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing," in *MICRO-29*, 1996.

[Saze97] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *MICRO-30*, 1997.

[Sezn96] A.Seznec, S.Jourdan, P.Sainrat, P.Michaud, "Multiple Block Ahead Branch Predictors," in *ASPLOS-VII*, 1996.

[Wang97] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," in *MICRO-30*, 1997.