Effective Hardware-Based Data Prefetching for High-Performance Processors

Tien-Fu Chen, *Member, IEEE*, and Jean-Loup Baer, *Fellow, IEEE*

Abstract—**Memory latency and bandwidth are progressing at a much slower pace than processor performance. In this paper, we describe and evaluate the performance of three variations of a hardware function unit whose goal is to assist a data cache in prefetching data accesses so that memory latency is hidden as often as possible. The basic idea of the prefetching scheme is to keep track of data access patterns in a Reference Prediction Table (RPT) organized as an instruction cache. The three designs differ mostly on the timing of the prefetching. In the simplest scheme (*basic*), prefetches can be generated one iteration ahead of actual use. The *lookahead* variation takes advantage of a lookahead program counter that ideally stays one memory latency time ahead of the real program counter and that is used as the control mechanism to generate the prefetches. Finally the *correlated* scheme uses a more sophisticated design to detect patterns across loop levels.**

These designs are evaluated by simulating the ten SPEC benchmarks on a cycle-by-cycle basis. The results show that 1) the three hardware prefetching schemes all yield significant reductions in the data access penalty when compared with regular caches, 2) the benefits are greater when the hardware assist augments small on-chip caches, and 3) the *lookahead* scheme is the preferred one cost-performance wise.

Index Terms—**Prefetching, hardware function unit, reference prediction, branch prediction, data cache, cycle-by-cycle simulations.**

I. Introduction

PROCESSOR performance has increased dramatically over the last few years and has now surpassed the 100 MIPS level. Memory latency and bandwidth have also progressed but at a much slower pace. Caches have been shown to be an effective way to bridge the gap between processor cycle and memory latency times. However, caches, while eliminating many main memory accesses, do not reduce the memory latency. It is therefore essential that we investigate techniques to reduce the effects of the imbalance between processor and memory cycle times.

There have been several techniques that have been used or proposed for tolerating high memory latencies. Among the hardware-based, we can list:

- Cache hierarchies [2] (now present in many medium to high-performance systems) and cache assists [11] (write buffers, victim caches);
- Lock-up free caches [13], [5] (with various degrees of sophistication);
- Hardware-based prefetching [24], [1], [7];
- Relaxed memory consistency models in the case of shared-memory multiprocessors (outside the scope of this paper).

On the software side, we can mention:

- Prefetching and poststoring [12], [18], [21];
- Software cache coherence schemes;
- Data placement (increasing locality and reducing false sharing);

with the last two items being of importance for multiprocessors.

In this paper we focus on one of these techniques, namely hardware-based prefetching. As we briefly discuss in Section II, both hardware and software prefetching schemes have their advantages and their drawbacks. The intent of this paper is to demonstrate that a simple hardware assist, on-chip, can reap important benefits in reducing the data access penalty between an on-chip data cache and the next level in the memory hierarchy when a miss in the first level and a hit at the second-level has a penalty of roughly one order of magnitude. Any mechanism that is meant to reduce the latency between these two lowest levels of the memory hierarchy should be non-intrusive. This calls for a hardware scheme that is not on the critical path and that does not "steal" cycles from the execution of the instruction stream.

Any prefetching scheme has for goal to reduce the processor stall time by bringing data into the cache before its use so that it can be accessed without delay. However, if data were prefetched too far in advance we would run the risk of polluting the cache. Ideally, a perfect prefetching scheme would totally mask the memory latency time; practically the latency can only be reduced since there are many impediments that prevent a perfect prediction of both the instruction stream, e.g., imperfect branch prediction, and of the data stream, e.g., data dependent addresses. The basic idea of the hardware-based prefetching scheme that we introduced in [1] is to keep track of data access patterns—effective address and stride—in a Reference Prediction Table (RPT). The design strives for simplicity. It combines the simplest look-up approach found in direct-mapped caches and an unsophisticated two-bit (four states) state transition mechanism as implemented in branch

Manuscript received Mar. 19, 1993; revised Aug. 13, 1993.

T.-F. Chen is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, Republic of China; e-mail chen@cs.ccu.edu.tw.

J.-L. Baer is with the Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195; e-mail baer@cs.washington.edu.

IEEECS Log Number C95032.

0018-9340/95$04.00 © 1995 IEEE

target buffers. In Section III, we describe three variations on this basic design. They differ mostly on the timing of the prefetching. In the simplest scheme, called *basic*, prefetches can be generated one iteration ahead of actual use. The *lookahead* variation takes advantage of a lookahead program counter that stays as much as possible one memory latency time, i.e., potentially several iterations, ahead of the real program counter. The lookahead program counter serves as the control mechanism to generate the prefetches. Finally, the *correlated* scheme uses a more sophisticated design to detect patterns across loop levels.

These three variations are evaluated by simulating the ten SPEC benchmarks cycle-by-cycle. The evaluation methodology as well as the parameters of the simulated processor and memory architectures are described in Section IV. The results, analyzed in Section V, show that 1) the three hardware prefetching schemes all yield significant reductions in the data access penalty when compared with regular caches, 2) the benefits are greater when the hardware assist augments small on-chip caches, and 3) the *lookahead* scheme is the preferred one cost-performance-wise.

Finally, concluding remarks are given in Section VI.

II. Related Work

We define data prefetching in the context of this work as the asynchronous action of bringing data in the data (or mixed instruction-data) cache before it is directly accessed by a memory instruction (such as a load or a store). We use the following terminology to review previous work on prefetching and to motivate our hardware scheme,

Consider a program segment with m-nested loops indexed by $I_1, I_2, \cdots, I_m$. Let LP_{I_i} be the set of statements with data references in the loop at level i. Given a data reference r, we can divide the memory access patterns into four categories: *scalar*, *zero stride*, *constant stride*, and *irregular* as shown in Table I.

TABLE I
MEMORY ACCESS PATTERNS

Pattern	Description	Examples
scalar	simple variable reference	index, count
zero stride	$r \in LP_{I_i}$ with subscript expression unchanged w.r.t I_i	A[I_1, I_2] in LP_{I_3} TAB [I_1].off in LP_{I_2}
constant stride	$r \in LP_{I_i}$ with subscript expression linear w.r.t. I_i	A[I_1] in LP_{I_1} A[I_1, I_2], A[I_2, I_1] in LP_{I_2}
irregular	none of the above	A[B[I]] in LP_I A[I, I] in LP_I Linked List

The difference between *scalar* and *zero stride* is that the latter is a reference to a subscripted array element with the subscript being an invariant at the inner loop level but modifiable at an outer level.

Obviously, standard caches work well for *scalar* and *zero stride* references. Caches with large block sizes can slightly improve the performance for the *constant stride* category if the stride is small but will be of no help if the stride is large. The goal of prefetching techniques is to generate prefetches in advance for uncached blocks in the *scalar*, *zero stride*, and *constant stride* access categories independently of the size of the stride. At the same time, erroneous prefetches for *irregular* accesses as well as overhead caused by prefetches for already cached data should be avoided.

Prefetching can be triggered either by a hardware mechanism, or by a software instruction, or by a combination of both. The hardware approach detects accesses with regular patterns and issues prefetches at run time, whereas the software approach relies on the compiler to analyze programs and to insert prefetch instructions. Most of the hardware solutions proposed in the literature require little compiler support.

A. Hardware-Based Prefetching

Hardware-based approaches can be classified into two categories: Spatial, where access to the current block is the basis for the prefetch decision, and temporal, where lookahead decoding of the instruction stream is implied.

In the spatial schemes, prefetches occur when there is a miss on a cache block. Smith [24] studied variations on the one block lookahead (OBL) policy, i.e., when block i is referenced, block $i + 1$ could be prefetched. Jouppi [11] proposed an extension to OBL where several consecutive data streams are prefetched in FIFO *stream buffers*. In OBL and extensions, miss rates can be reduced, mostly for direct-mapped caches, at the expense of some increase in memory traffic. These schemes take advantage of limited (sequential) spatial locality by prefetching data separated by small constant strides but are not able to deal with large strides. The use of stride information carried by vector instructions led Fu and Patel [7] to propose prefetch strategies for vector processors. They subsequently derived a similar approach for scalar processors [8]. The major mechanism is to record the previous memory address in a history table and to generate prefetch requests by calculating a stride between the current address and the previous address if the stride is nonzero. The approach, which lacks the control of preventing unnecessary prefetches on irregular accesses, corresponds to a degenerated version of our *basic* scheme, i.e., one without state transition mechanism. Sklenar [23] has presented the same general idea of a hardware assist, but without any performance or cost evaluation. Note that in spatial schemes, the opportune time to initiate a prefetch is not linked closely to the time of next use.

Temporal mechanisms attempt to have data be in the cache "just in time" to be used. Data prefetching by instruction lookahead [16] and the implicit prefetching used in decoupled architectures [25] fetch speculatively those data operands most likely needed in the near future. The time window where this prefetching can occur is limited by the instruction decoding buffer size and is not wide enough for large memory latencies. Also, the address of the data to be prefetched is based on the values of the speculated operands and is not related to either the current locality in the cache or the patterns described above.

As will be seen, the *lookahead* scheme that we describe combines the advantages of both the spatial and temporal approaches. The spatial mechanism is realized by capturing dynamically the data access patterns in a Reference Prediction Table (RPT). A lookahead program counter (LA-PC, Section III.C) dynamically controls the arrival time of the prefetched data. A branch prediction mechanism is used in lieu of a more complex lookahead instruction decoding.

B. Software-Directed Prefetching

A totally different approach to prefetching is to use software-directed techniques that rely on static program analysis to detect regular data access patterns. An intelligent compiler inserts data prefetch instructions several cycles before their corresponding memory instructions. These prefetch instructions are explicitly executed by the processor to initiate prefetch requests.

Porterfield [21] examined the effect of prefetching all array references in the inner loop inserting prefetches one iteration ahead. He recognized that this led to too much overhead since many prefetch instructions were directed to data already in the cache. Gornish et al. [10] proposed an algorithm to find the earliest point before a loop that an entire subarray could be prefetched. The approach focuses on fetching block data, rather than a single cache line at a time. Klaiber and Levy [12] showed that the time to prefetch should depend on memory latency and loop execution time, a feature present in the hardware *lookahead* scheme. They proposed prefetching into a separate fetch buffer instead of a unified cache. A study of software prefetching for non-scientific code by Chen et al. [6] found that it is more difficult to generate prefetch addresses early when the access patterns are irregular. Mowry and Gupta [17] showed (by manually inserting prefetch instructions) that prefetching can be effective for tolerating large memory latencies in a shared-memory multiprocessor environment. They considered prefetch instructions for both read and write accesses. They further developed a compiler algorithm [18] that automatically inserts the prefetch instructions. The algorithm is based on locality analysis and on loop transformations with proper prefetch predicates. Prefetches are inserted selectively for those references in the constant stride pattern that are likely to cause cache misses.

Hardware-based techniques and software-directed approaches are both successful in identifying prefetches of data with simple constant stride patterns. Both schemes require lockup-free caches as architectural support. The only other architectural requirement of the software approach is the availability of a prefetch instruction. In efficient hardware schemes, a non-trivial hardware mechanism is needed to detect the items to be prefetched and to initiate the prefetching.

From a performance viewpoint, the software approach can identify more prefetches for accesses with complex patterns, primarily on loop-domain references. In addition, software prefetching in a multiprocessor environment can take into account factors such as data coherence, task scheduling, and task migration. On the other side of the ledger, software prefetch instructions introduce an overhead, at the very least the execution of the prefetch instruction and possibly other computations for the effective addresses and the prefetch predicates. Code often needs to be expanded, e.g., by loop unrolling, so that the prefetch can occur sufficiently in advance. Side-effects, such as increased register pressure, can result from these prefetch optimizations. Although an intelligent compiler may be able to reduce much of the unnecessary overhead, the remainder could still be significant, especially when the memory latency is not too large. Moreover, the software approach does not cater to the prefetching of data that has been replaced because of conflict or capacity cache misses that can be frequent if the cache size is small. Similarly, estimates of execution time for loops calling subroutines may not be uncovered at compile time, thus preventing prefetch at the right time. Finally, the optimizations are language and compiler dependent while the hardware schemes do not require any change in the executable code.

The references cited above make the case for software prefetching. The goal of this paper is to show that a hardware-based prefetch mechanism can be a cost and performance efficient mechanism when placed in the context of a (small) on-chip cache and a not too large memory latency. It is not the intent of this paper to compare hardware-based and software-directed prefetching techniques. A report of a preliminary study of this type, for shared-data in multiprocessor environments, can be found in the dissertation of the first author [4]. In addition, it is clear that compiler interaction can help the hardware scheme [9] and that a combination of hardware and software approaches should certainly be investigated thoroughly.

III. Hardware-Based Data Prefetching Schemes

A. Motivation

Sequential prefetching can be successful for the optimization of I-caches, but much less so for D-caches. In this section, we describe three variations, in increasing order of complexity *basic*, *lookahead*, and *correlated* of a hardware-based prefetching scheme for D-caches. The common basis of these schemes is to predict the instruction execution stream and the data access patterns far enough in advance, so that the required data can be *prefetched* and be in the cache when the "real" memory access instruction is executed.

The three schemes have in common the following goals:

- generate prefetches in advance for uncached blocks in the *scalar*, *zero stride*, and *constant stride* access categories independently of the size of the stride;
- avoid unnecessary prefetching for the *irregular* accesses;
- incur no execution time penalty for prefetch requests for data that is already cached;
- design the hardware assist so that it does not increase the processor cycle time, i.e., does not interfere with critical path timing.

The basis for the three designs is an RPT that holds data access patterns of load/store instructions. The RPT is organized as an instruction cache. Minimally, each entry in the table will contain a tag related to the instruction address, fields to record the memory operand address and its stride, and a state transition field.

To illustrate the concept, we consider the usual matrix multiplication loop (for more detail, see Section III.B.3) and the pseudo-assembly RISC-like code version of the computational part of the inner loop shown in Fig. 1. In the code we assume that the subscripts are kept in registers. At steady state, the RPT will contain entries for the three load *lw* and the store *sw* instructions. Since each iteration of the inner loop accesses the same location of A[i, j] (*zero stride*), no prefetch will be requested for it. Depending on the block size, references to B[i, k] (*constant stride*) will either be prefetched at every iteration (block size = 4), or every other iteration (block size = 8), and so on. Load references to C[k, j] (*constant stride* with a stride larger than the block size) will generate a prefetch instruction every iteration.

```
int A[100,100],B[100,100],C[100,100]
for i = 1 to 100
    for j = 1 to 100
        for k = 1 to 100
            A[i,j] += B[i,k] × C[k,j]
```

(a) A Matrix Multiplication

```
addr  instruction         comment
500   lw   r4, 0(r2)      ; load B[i,k]   stride 4 B
504   lw   r5, 0(r3)      ; load C[k,j]   stride 400 B
508   mul  r6, r5, r4     ; B[i,k] × C[k,j]
512   lw   r7, 0(r1)      ; load A[i,j]   stride 0
516   addu r7, r7, r6     ; +=
520   sw   r7, 0(r1)      ; store A[i,j]  stride 0
524   addu r2, r2, 4      ; ref B[i,k]
528   addu r3, r3, 400    ; ref C[k,j]
532   addu r11, r11, 1    ; increase k
536   bne  r11, r13, 500  ; loop
```

(b) assembly code

Fig. 1. Example of matrix multiplication.

Although a compiler has no difficulty identifying those access patterns for prefetching, the hardware scheme provides the advantages of dynamic prefetching detection and code compatibility. In particular, as can be shown in the following sections, our scheme will be most appropriate for high-performance processors with small first-level caches with a small block size, where software-directed approaches may have significant instruction overhead.

B. Basic Reference Prediction

The basic idea of reference prediction is to predict future references based on the past history for the same memory access instructions. The most intuitive prediction scheme is to have prefetches for the $(i + 1)^{st}$ iteration be generated when the *i*th iteration is executed. Thus, when the program counter (PC) decodes a load/store instruction, a check is made to see if there is an entry corresponding to the instruction in the RPT. If not, it is entered. If it is there and if the reference for the next iteration is predictable (as defined below), a prefetch is issued. This *basic* scheme involves only the PC and the RPT. As shown in Fig. 4, the hardware requirement for the *basic* design is a subset of the more complex *lookahead* variation that will be described in Section III.C. We now introduce the design and use of the RPT under the *basic* scheme.

B.1. Reference Prediction Table—RPT

The RPT, organized as an instruction cache, keeps track of previous reference addresses and associated strides for load and store instructions. The RPT entry, indexed by the instruction address, not only includes the effective address of data access and the stride information, but also the regularity status for the instruction. Each entry has the following format (see Fig. 2):

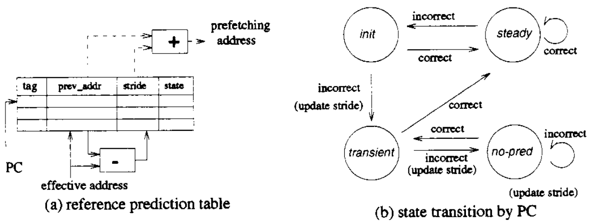
- *tag:* corresponds to the address of the Load/Store instruction
- *prev_addr:* the last (operand) address that was referenced when the PC reached that instruction.
- *stride:* the difference between the last two addresses that were generated.
- *state:* a two-bit encoding (four states) of the past history; it indicates how further prefetching should be generated. The four states are:

  —*initial:* set at first entry in the RPT or after the entry experienced an incorrect prediction from *steady* state.

  —*transient:* corresponds to the case when the system is not sure whether the previous prediction was good or not. The new stride will be obtained by subtracting the previous address from the currently referenced address.

  —*steady:* indicates that the prediction should be stable for a while.

  —*no prediction:* disables the prefetching for this entry for the time being.



(a) reference prediction table
(b) state transition by PC

Fig. 2. Reference prediction.

B.2. RPT Mechanism

The basic mechanism in the RPT is to record the effective address of the memory operand, compute the stride for that access, and set a state controlling the prefetching by comparing the previously recorded stride with the one just computed. The stride information is obtained by taking the difference between the effective addresses of the two most recent accesses for the same instruction.

When a load/store instruction is encountered for the first time, the instruction is entered in the RPT in the *initial* state. When the stride is obtained for the first time, i.e., at the second access, the state is set to *transient* since it is not known yet whether the pattern will be regular or irregular. When a further stride is computed and, if it is equal to the one previously recorded, i.e., if the same stride has occurred twice in a row, then the entry will be set to the *steady* state. It will remain in the *steady* state until a different stride is detected, e.g., at the end of an inner loop. At that point, the state is reset to *initial.* If a different stride is computed while in the *transient* state, the entry is set in the *no-prediction* state since the pattern is not regular and we want to prevent erroneous prefetching. In the presence of irregular patterns, the state of the entry will either stay in the *no-prediction* state or oscillate between the *transient* and *no-prediction* states until a regular stride is detected.

Hence, when the PC encounters a load/store instruction with effective operand address *addr*, the RPT is updated as follows: (To make it clear, we denote *correct* by the condition: *addr* = (*prev_addr* + *stride*) and *incorrect* by the condition: *addr* ≠ (*prev_addr* + *stride*).)

- A.1. There is no corresponding entry. The instruction is entered in the RPT, the *prev_addr* field is set to *addr*, the *stride* to 0, and the *state* to *initial.*
- A.2. There is a corresponding entry. Then:

  a) Transition –
  When *incorrect* and *state* = *initial*:
  Set *prev_addr* to *addr*, *stride* to (*addr* - *prev_addr*), and *state* to *transient.*

  b) Moving to/being in steady state –
  When *correct* and (*state* = *initial, transient,* or *steady*):
  Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *steady.*

  c) Steady state is over; back to initialization –
  When *incorrect* and *state* = *steady*:
  Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *initial.*

  d) Detection of irregular pattern –
  When *incorrect* and *state* = *transient*:
  Set *prev_addr* to *addr*, *stride* to (*addr* - *prev_addr*), and *state* to *no prediction.*

  e) No prediction state is over; back to transient –
  When *correct* and *state* = *no prediction*:
  Set *prev_addr* to *addr*, leave *stride* unchanged, and set *state* to *transient.*

  f) Irregular pattern –
  When *incorrect* and *state* = *no prediction*:
  Set *prev_addr* to *addr*, *stride* to (*addr* - *prev_addr*), and leave *state* unchanged.

Following the update, a prefetch request can be generated based on the presence and state of the entry. Note that the generation of a prefetch does not block the execution of the instruction stream and, in particular, the increment of the PC. There are two mutually exclusive possibilities:

- B.1. No action.
  There is no existing entry or the entry is in state *no prediction.*
- B.2. Potential prefetch.
  There is an entry in *init*, *transient*, or *steady* state. A data block address (*prev_addr* + *stride*) is generated. A prefetch is initiated if the block is uncached and the address is not found in an Outstanding Request List (ORL) (see Section III.C). This implies sending a request to the next level of the memory hierarchy, or buffering it if the communication channel is busy. The address of the request is entered in the ORL.

B.3. Example and Discussion

Fig. 3 illustrates how the RPT is filled and used when the inner loop of the matrix multiplication code shown previously is executed. We restrict our example to the handling of the three load instructions at addresses 500, 504, and 512. We assume that the base addresses of matrices A, B, and C are, respectively, at locations 10,000, 50,000, and 90,000.

tag	prev_addr	stride	state

Initially empty
(a)

tag	prev_addr	stride	state
500	50,000	0	*init*
504	90,000	0	*init*
512	10,000	0	*init*

After iteration 1
(b)

tag	prev_addr	stride	state
500	50,004	4	*transient*
504	90,400	400	*transient*
512	10,000	0	*steady*

After iteration 2
(c)

tag	prev_addr	stride	state
500	50,008	4	*steady*
504	90,800	400	*steady*
512	10,000	0	*steady*

After iteration 3
(d)

Fig. 3. Example: filling RPT entries.

Before the start of the first iteration, the RPT can be considered empty since there will not be any entry corresponding to addresses 500, 504, and 512 (Fig. 3(a)). Let us assume also that no element of A, B, or C has been cached.

When the PC executes for the first time the load instruction at address 500, there is no corresponding entry. Therefore, the instruction is entered in RPT with its *tag* (500), the *prev_addr* field set to 50,000, the address of the operand, the *stride* set to 0, and the *state* to *initial* (A.1 above). Similar actions are taken for the other two load instructions (Fig. 3(b)). In all three cases, there will be cache misses and no prefetches.

When the PC executes the load instruction at address 500 at the beginning of the second iteration, we are in the situation described as "transition" (A.2.a). The following three actions take place:

1) Normal reference access to address 50,004. This results in a cache hit if the block size is larger than four and in a miss otherwise.

2) Update of the entry in the RPT. The *prev_addr* field becomes 50,004, the *stride* is set to 4, and the *state* to *transient*.
3) Potential prefetch of the block at address (50,004 + 4) = 50,008. A prefetch occurs if the block size is less than eight.

Similar actions take place for the load at address 504 with, in this instance, the certainty that a prefetch will be generated (Fig. 3(c)). For the load at instruction 512, we are in the situation "moving to steady state" (A.2.b). The *prev_addr* and *stride* fields are unchanged and the *state* becomes *steady*. Of course, we have a cache hit and no prefetch.

During the third iteration, all three loads should result in cache hits, or in indications that prefetches for the referenced items are in progress. The RPT entries are updated as shown in Fig. 3(d) (note the *transient* to *steady* transitions); prefetches are generated for blocks at addresses 50,012 (if needed) and 91,200. Subsequent iterations follow the same pattern.

As can be observed in Fig. 2, *scalar* and *zero stride* references will pass from *initial* to *steady* state in one transition (instruction 512). The *constant stride* references will pass through the *transient* state to "obtain" the stride and then stay in *steady* state (instructions 500 and 504). References with two wrong predictions in a row (not shown in the example) will be prevented from being prefetched by passing to the *no prediction* state; they could re-enter the *transient* state, provided that the reference addresses become predictable. For instance, accesses to elements of a triangular matrix may follow such a pattern. Note that the *stride* field is not updated in the transition from *steady* to *initial* when there is an incorrect prediction.

C. Lookahead Reference Prediction

The *basic* scheme has a potential weakness associated with the timing of the prefetch, that is, the hiding of the memory latency by prefetching depends on the execution time of one loop iteration. If the loop body is too small, the prefetched data may arrive too late for the next access, and if the loop body is too large, an early arrival of prefetched data may replace (or be replaced by) other useful blocks before the data is used. Although a compiler may easily solve the former problem by loop unrolling, we assume that the hardware scheme does not rely on software support and preserves code compatibility across various hardware implementations. The *lookahead* reference prediction scheme seeks to remedy the drawback of the basic scheme.

Assuming no contention in the interconnect or in access to the cache, the ideal time to issue a prefetch request is δ cycles ahead of the actual use, where δ is the latency to access the next level in the memory hierarchy. The data would then arrive "just in time" to prevent a cache miss and would not displace potentially useful data. Instead of prefetching one iteration ahead, the *lookahead* prediction will approximate this ideal prefetch time with the help of a pseudo program counter, called the Look-Ahead Program Counter (LA-PC), that will remain as much as possible δ cycles ahead of the regular PC and that will access the RPT to generate prefetches. The LA-PC is incremented as the regular PC. It is used with a Branch Prediction Table (BPT) to take full advantage of the *look-ahead* feature.

A block diagram of the target processor is shown in Fig. 4. The bottom part of the figure abstracts a common high-performance processor with on-chip data and instruction caches. The upper-left part shows the RPT and the ORL that keeps track of the addresses of fetches in progress and of outstanding requests. Under the *basic* scheme, the RPT is accessed by the PC. To implement the *lookahead* mechanism, a LA-PC and its associated logic are added to the top part (on the right in the figure). The LA-PC is a secondary PC used to predict the execution stream. In addition, we assume that a BPT such as a branch target buffer (BTB), a branch prediction mechanism for the PC in a high-performance processor, is used for modifying the LA-PC.

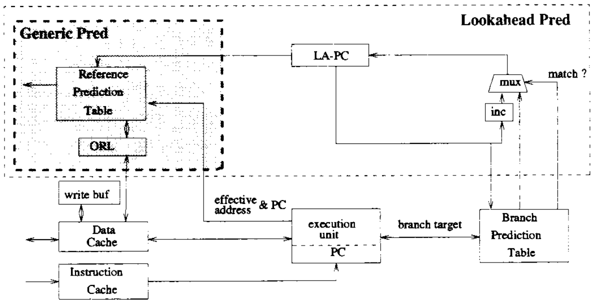


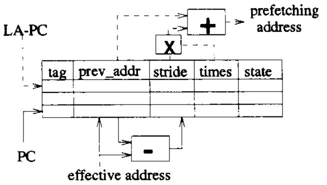
Fig. 4. Block diagram of data prefetching.

Entries in the RPT and BPT, are initialized and updated when the PC encounters the corresponding instruction. In the *lookahead* scheme, in contrast to the *basic* prediction, it is the LA-PC rather than the PC that triggers potential prefetches according to rules B.1 and B.2 of the previous section. At each cycle, the LA-PC is simply incremented by one. When the LA-PC finds an entry in the BPT, it indicates that the LA-PC points to a branch instruction. In that case, the prediction result of the branch entry in the BPT is provided to modify the LA-PC. Note that, unlike the instruction prefetch structure in [15] or decoupled architectures [25], the LA-PC does not need to decode the predicted instruction stream. Instead, the lookahead mechanism is based on the history information of the execution stream, since the LA-PC is just a pointer to detect the prefetching in the RPT.

C.1. Lookahead Program Counter (LA-PC) and RPT

In the *basic* prediction scheme, prefetching can occur only one iteration ahead and thus, as mentioned earlier, the prefetched data might not yet be in the cache when the real access takes place. This situation will occur when the loop iteration time is smaller than the memory latency. With the help of the *lookahead* mechanism, the LA-PC may wrap around the loop and revisit the same load instruction when the execution time of a loop iteration is smaller than the memory latency. In this way, we may have multiple iterations lookahead.

An extra field (*times*) in the entries of the RPT will keep track of how many iterations the LA-PC is ahead of the PC (Fig. 5). In the *lookahead* design, the LA-PC detects and generates prefetch requests and the PC accesses the RPT when an effective address is obtained. As a result, the RPT should be dual-ported, which allows simultaneous accesses of PC and LA-PC. Now, when the LA-PC hits an instruction with a corresponding entry in the RPT, the address of a potential prefetch is determined by computing (*prev_addr* + *stride* × *times*). The *times* field is incremented whenever the LA-PC hits the entry, while it is decremented when the PC catches up with the entry. The *times* field is reset when it is found that the reference prediction of the corresponding entry is incorrect.



- **When hit by the LA-PC** :
  times++
- **When hit by the PC** :
  If the reference prediction is
  - Correct ⟶ *times-- --*
  - Incorrect ⟶ *times = 0*

Fig. 5. RPT with Lookahead mechanism.

C.2. Lookahead Distance and Limit

The ideal Look Ahead distance (LA-distance), i.e., the time between the execution of the instruction pointed to by the PC and that of the instruction pointed to by the LA-PC, is equal to the latency δ of the next level in the memory hierarchy. Clearly this can only be approximated since the LA-distance is variable. Initially, and after each wrong branch prediction, the LA-distance will be set to one, with the LA-PC pointing to the instruction following the current PC. When a real cache miss occurs or when a prefetch is not completed by the time the data is needed, the current execution is stalled; the value of PC does not change, while the LA-PC can still move ahead and generate new requests (recall the role of the ORL).

As shown in Fig. 4, the LA-PC is maintained with the help of a branch prediction mechanism BPT. BPT designs have been thoroughly investigated [14], [20] and we will not repeat these studies here. In our experiments we use the Branch Target Buffer (BTB) with two-bit state transition design described in [14] and we assume that the BTB has been implemented in the core processor for other purposes.

As the LA-distance increases, the data prefetch can be issued early enough so that the memory latency can be completely hidden. However, the further PC and LA-PC are apart, the more likely the prediction of the execution stream will be incorrect because the LA-distance is likely to cross over more than one basic block. Moreover, we do not want some of the prefetched data to be cached too early and displace other needed data. Therefore, we introduce a system parameter called Look Ahead Limit (LA-limit *d*) to specify the maximum distance between PC and LA-PC. Thus, the LA-PC is stalled (until the normal execution is resumed) in the following situations: 1) The LA-distance reaches the specific limit *d*, or 2) the ORL is full.

C.3. Handling Cache Misses

On a cache read miss, the cache controller checks the ORL. If the block has already been requested, a "normal," but less lengthy, stall occurs. We call *hit-wait cycles* those cycles during which the CPU waits for the prefetched block to be in the cache. If the ORL has no such block entry either, a regular load is issued with priority over the buffered prefetch requests.

Since we are using a write-back, write-allocate strategy, a write miss in the data cache will cause the system to fetch the data block and then update the desired word. If the block size is larger than a single word, we can initiate prefetching as for a read miss. When the block size is one word, no prefetch needs to be issued but a check of the ORL is needed for consistency purposes. If there is a match, the entry in the ORL must be tagged with a discard status so that the data will be ignored when it arrives.

When the LA-PC has to be reset because of an incorrect branch prediction, the buffered prefetch requests are flushed. Finally, when a prefetch raises an exception, e.g., a page fault or an out-of-range violation, we ignore the prefetch. The drawbacks of a wrong page fault prediction would far outweigh the small benefits of a correct prefetch.

D. Correlated Reference Prediction

In the previous two designs, reference prediction was based on the regularity between *adjacent* data accesses. In general, the schemes work well for predicting references in inner loops. However, the results are less significant for those execution segments with small inner-loop bodies or triangle-shaped loop patterns because of the frequent stride change in the outer iterations. For example, let us look at Livermore Kernel Loop 6 as shown in Fig. 6.

```
    int B[100,100], W[100]
    DO 6 i=1,n
    DO 6 k=0,i
        W(i) = W(i) + B(i,k)*W(i-k)
6   CONTINUE
```

(a) Code

```
1,0  1,1
2,0  2,1  2,2
3,0  3,1  3,2  3,3
4,0  4,1  4,2  4,3  4,4
```

(b) Access pattern of Matrix B

Fig. 6. Livermore Loop 6.

While executing the inner loop, accesses to the B matrix have regular strides; for example, B[3, 0], B[3, 1], B[3, 2], and B[3, 3] have a *stride* of 4. This pattern will be picked up by the two schemes presented above. However, an incorrect prediction will occur each time the *k* loop is finished, e.g., when accessing B[4, 0] after B[3, 3]. We can observe though that there is a correlation between the accesses at the termination of the inner loop (B[1, 0], B[2, 0], B[3, 0], etc. have a *stride* of 400). Correlation that has led to the design of more accurate branch prediction [19], [26] can be equally applied to data reference prediction.

The key idea behind *correlated* reference prediction thus is to keep track not only of those *adjacent* accesses in inner loops, like in the above two schemes, but also of those *correlated* by changes in the loop level. Since branches in the inner loop are taken until the last iteration, a nontaken branch will trigger the correlation to the next level up.

D.1. Implementation of Correlated RPT

The implementation of a *correlated* scheme would bring two additions to the *lookahead* mechanism: a shift register to record the outcome of the last branches and an extended RPT with separate fields for computing the strides of the various correlated accesses. In the most general case, an *N-bit* shift register can be used to keep track of the results of the last *N* branches and to serve as a mask to address the various fields in the extended RPT.

Since prefetching too far in advance might be detrimental, we restrict ourselves to the correlation in two-level nested loops. The RPT is extended (Fig. 7), with a second pair of *prev_addr* and *stride* fields for recording the access patterns of the outer loop. (Note that at the outer loop level the *times* and *state* fields are no longer relevant.) We also have now a two-bit shift register for recording the outcome of loop-only branches. We assume that a bit '1' encodes a taken branch; then the steady state encoding while executing the inner loop will be '11.' In that case prefetching will be based on the entry in the RPT corresponding to the inner loop (the "right" part in the figure). When the branch is not taken, the shift register will contain '10' (because a nontaken inner-loop branch has been followed by a taken outer-loop branch) and prefetching will be based on the part of the entry corresponding to the outer loop (the "left" part). Updating of the left *prev_addr*0 and *stride*0, as well as of the right *prev_addr*1, fields will take place at the beginning of an outer iteration (when the shift register contains '10' or '00') while all the right fields will be updated for consecutive inner iterations (when the register contains '11' or '01').

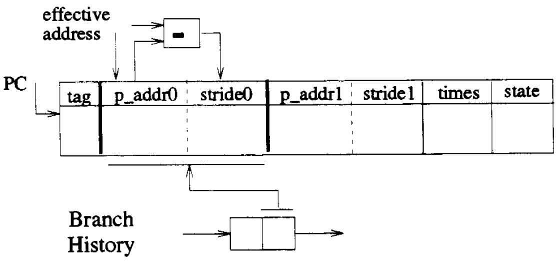


Fig. 7. Correlated RPT.

Fig. 8 shows how the RPT entry for B[i, k] would be filled and updated during execution of the first three iterations of the outer loop of Kernel Loop 6. We have left out the *times* field for ease of explanation. Without loss of generality, we can assume that the initial content of the shift register is '10' and that the entry in the RPT is empty. At the initial access of B[1, 0], all fields are filled as in the previous schemes (first row of left table in Fig. 8). At the second access (first row of right table) only the right fields are modified as they would be in the previous schemes (the shift register contains '11'). At the beginning of the second outer iteration, i.e., first access to B[2, 0], the shift register will again contain '10' (branch nontaken). Thus, prefetching of B[3, 0] and B[2, 1], if needed, will be done for accesses in both levels of the loop, and updating of the first pair of fields and of *prev_addr*1 will be performed (second row of top table). On subsequent accesses to B[2, i], prefetching and updating will be based on the right fields (second row of bottom table). At the beginning of the third outer iteration, we are in steady state (last row of the table). By that time B[3, 0] should have been prefetched (at the end of the second iteration). A prefetch to B[3, 1] would be generated and most likely not activated if the line size is large enough, i.e., if B[3, 0] and B[3, 1] are in the same line.

	1st inner iteration					2nd inner iteration				
outer iteration	prev addr0	stride0	prev addr1	stride1	state	prev addr0	stride0	prev addr1	stride1	state
1	B[1,0]	0	B[1,0]	0	*init*	B[1,0]	0	B[1,1]	4	*transient*
2	B[2,0]	400	B[2,0]	4	*transient*	B[2,0]	400	B[2,1]	4	*steady*
3	B[3,0]	400	B[3,0]	4	*steady*	B[3,0]	400	B[3,1]	4	*steady*

Fig. 8. Example: correlated RPT entries.

Three issues regarding the implementation of a correlated reference prediction scheme need to be addressed. In all three cases, we make reasonable assumptions to keep the design as simple as possible. The first assumption is that since it is easy for a compiler to distinguish between end of loop branches and other branches, the former will be flagged, e.g., in the branch prediction table, and the shift register will be modified only when they are encountered. This assumption could be removed by letting the shift register be modified on every branch as in [19]. While some predictable patterns might emerge the complexity of implementation might not be warranted. Second, we assume that the loop iterations are controlled by backward branches.[1] Third, we assume that prefetches for the correlated references, across outer iterations, are issued as in the *basic* case, since the LA-PC will be the same as the PC on the incorrect branch prediction. Those prefetches are generated without any attempt to control the prefetch issue δ cycles ahead of data use. This last assumption is reasonable since accesses in the outer loop are separated by the execution of all the iterations of the inner loop which, in most likelihood, will take longer than δ cycles.

IV. Evaluation Methodology

A. Trace-Driven Simulation

We evaluated our proposed architectures using cycle-by-cycle trace generation combined with on-the-fly simulation. For comparison purposes, a baseline architecture consisting of a processor with perfect pipelining and direct-mapped D-cache with 32K bytes and a block size of 32 bytes, unless otherwise specified, was also simulated. Benchmarks were instrumented

[1] A backward branch always passes execution control to a location that is before the address of the branch instruction. A transformation by a compiler is required for these programs in which forward conditional branches are used to control the loop iterations [3].

on a DECstation 5000 (R3000 MIPS CPU) using the *pixie* facility. Traces include data and instruction references so that the simulator can emulate the detail behavior of overlapping computation with data access. The experiment results were collected at the clock cycle level from the individual configurations.

We used 10 applications from the SPEC[2] Benchmark, compiled with the MIPS C compiler and the MIPS F77 compiler, both with optimization options. The traces captured at the beginning of the execution phase of the benchmarks were discarded because they are traces of initial routines that generate the test data for the benchmarks. No statistical data was recorded while the system simulated the first 500,000 data accesses. However, these references were used to fill up the cache, the BPT, and the RPT to simulate a warm start. After the initialization phase and the warm-start period, simulations results were collected for the first 100 million instructions for all programs.

Table II shows the dynamic characteristics of the workload. These statistics were collected by simulating the program with an infinitely large RPT. The columns below *data references* show the proportions of data references (weighted by their frequency) that belong to the categories mentioned previously. They are one indication of the reference predictability of the 10 programs. Scalar or zero stride references are beneficial to the data cache; the prefetching schemes can be useful to bring back in advance blocks that were displaced because of a small cache size (capacity misses) or a small associativity (conflict misses). Constant stride references, which may substantially contribute to cache misses, should be helped by the RPT schemes. Prefetching should be avoided for unpredictable irregular references. The column *branch prediction miss ratio* shows the outcome of branch predictions with a 512-entry BPT, that functions like a 2-state-bit Branch Target Buffer [14]. This is a second indication of the reference predictability, illustrating the possible benefits exploited by the lookahead approach.

TABLE II
CHARACTERISTICS OF BENCHMARKS

Name	data references			branch pred. miss ratio
	scalar, zero stride	constant stride	irregular	
Tomcatv	0.312	0.682	0.006	0.005
Fpppp	0.981	0.006	0.014	0.110
Matrix	0.059	0.921	0.021	0.073
Spice	0.581	0.239	0.180	0.060
Doduc	0.692	0.154	0.154	0.120
Nasa	0.006	0.989	0.003	0.008
Eqntott	0.338	0.574	0.088	0.069
Espresso	0.460	0.424	0.116	0.055
Gcc	0.516	0.120	0.365	0.204
Xlisp	0.440	0.078	0.482	0.156

We experimented with the three architectural choices, and varying architectural parameters, described previously. The results of the experiments are presented with the "cycle per instruction contributed by memory accesses" (MCPI) as the main metric. Since we assume that the processor can execute each instruction in one cycle (perfect pipelining) and that we have an ideal instruction cache, the only extra contribution of CPI stems from the data access penalty. Hence, the MCPI caused by the data access penalty is obtained as:

$$MCPI_{data\ access} = \frac{total\ data\ access\ penalty}{number\ of\ instructions\ executed}$$

The reason for choosing MCPI as a metric instead of the miss rate or the average effective access time is that MCPI can reflect the actual stall time observed by the processor, taking both processor execution and cache behavior into account. In the figures, we also give the percentage of the data access penalty reduced by the prefetching scheme. This percentage number is computed as:

$$\%\text{ of penalty reduced} = \frac{\text{data penalty}_{cache} - \text{data penalty}_{prefetch}}{\text{data penalty}_{cache}} \times 100$$

B. Memory Models

Data bandwidth is an important consideration in the design of an architecture that allows overlap of computation and data accesses since several data requests, such as cache misses or prefetching requests, can be present simultaneously. Thus, we have associated an ORL with the prefetching caches. A requirement for this list is that it can be searched associatively. We will assume, conservatively, that a fetch in progress cannot be aborted. However, a demand cache miss will be given priority over buffered prefetch requests.

Three memory interfaces with increasing capabilities of concurrency were simulated. They are (see Fig. 9 for timing charts and block diagrams):

—**Nonoverlapped (1):** As soon as a request is sent to the next level, no other request can be initiated until the (sole) request in progress is completed. This model is typical of an on-chip cache backed up by a second level cache.
This interface supports only one cache request at a time.

—**Overlapped (C, N):** The access time for a memory request can be decomposed into three parts: request issue cycle, memory latency, and transfer cycles. We assume that during the period of memory latency other data requests can be in their request issue or transfer phases. However, no more than one request issue or transfer can take place at the same time.
This model represents split buses and a bank of *C* interleaved memory modules or secondary caches. An ORL with *N* entries is associated with each module.

—**Pipelined (N):** A request can be issued at every cycle. This model is representative of processor-cache pairs being linked to memory modules through a pipelined packet-switched interconnection network. We assume a *load through* mechanism [24], i.e., the desired word is available as soon as the first data response arrives. An *N*-entry ORL is associated with the cache.

[2] SPEC is a trademark of the Standard Performance Evaluation Corporation.

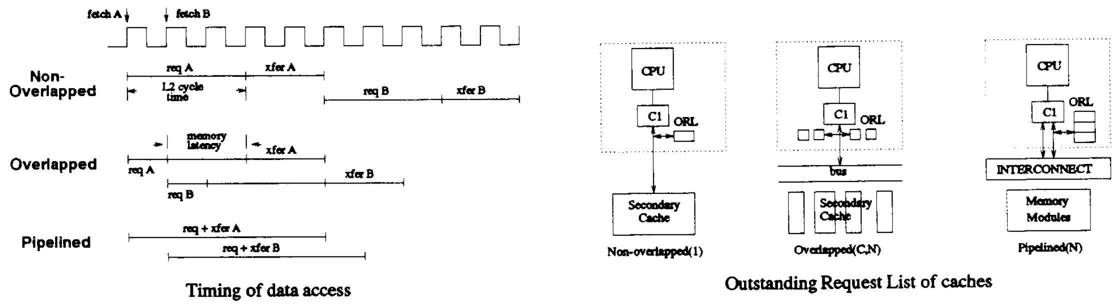


Fig. 9. Three memory models.

The configurations of the ORLs used in our experiment are **Nonoverlapped (1)**, **Overlapped (8, 2)**, and **Pipelined (8)**, respectively. The memory latency δ is usually equal to 30. The *Overlapped* model is the default model to show general results since it is the most likely implementation for future high performance processors. The cycle time of the three phases (requesting, accessing memory, and transferring) are 2, 20, and 8 cycles, respectively.

V. Simulation Results

In this section, we present experimental results that show the benefits of the prefetching schemes. We first compare an architecture with a baseline cache with the same architecture augmented by each of the three prefetching schemes. These comparisons are performed on all 10 SPEC benchmarks. In the remainder of the section, for brevity's sake, we restrict ourselves to reporting on benchmarks with the most salient features. We examine the impact of the memory models and associated latencies, the effect of block size variations, and means to improve the efficiency of the RPT. Finally, we discuss the setting of the Lookahead Limit *d* that is relevant to the *lookahead* prefetching scheme.

A. General Results

Figs. 10(a) and 10(b) show the results of the simulation of the four architectures with the data access penalty MCPI as a function of the cache size. The *Overlapped* memory model is used, the block size is 32 bytes, and the RPT and the BPT in the prefetching schemes have 512 entries. The results show that the prefetching organizations are always superior to the pure cache scheme since they have the same amount of cache and, in addition, the prefetching component. When the cache is too small to contain the working set of the application, the best prefetching scheme can reduce the data access penalty from 16% up to 97%. The additional cost paid for prefetching is justified by the significant performance improvement. This additional cost (RPT and logic) is approximately equivalent to a 4K-byte D-cache (Section V.D).

We examine further the performance curves by dividing the ten benchmarks into three groups: 1) prefetching performs extremely well, 2) prefetching yields a good or moderate improvement to the performance, and 3) prefetching's contribution to the reduction in data access penalty is slight.

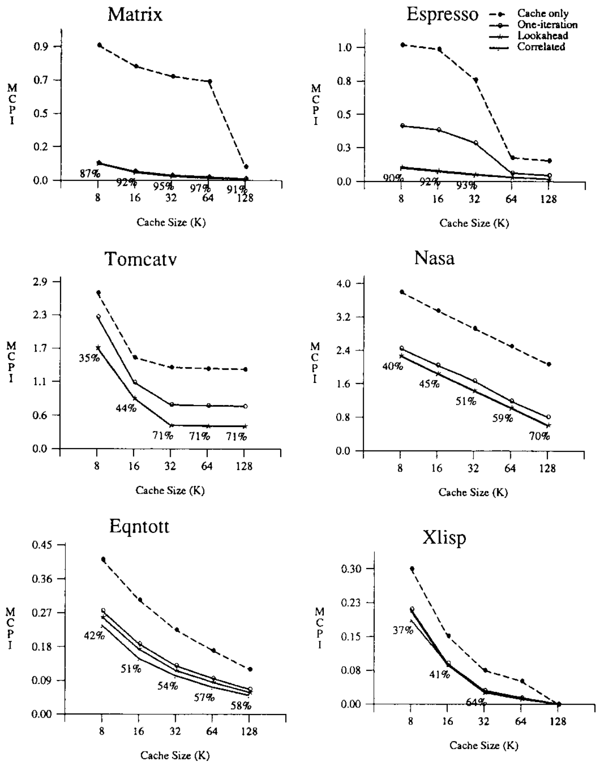


Fig. 10(a). Simulation results for δ = 30 (*Overlapped*).

The first group is formed by the Matrix and Espresso benchmarks, in which the data access penalty has been reduced by over 90%. For all practical purposes, the MCPI is almost completely eliminated. With good reference predictability in these two programs, the flat performance curves of the prefetching illustrate that a cache of small size is sufficient to capture most of the locality when compulsory cache misses have been eliminated by the prefetching.

The second group includes Tomcatv, Nasa, Eqntott, and Xlisp in which the prefetching yields a good performance improvement, a reduction in data access penalty in the range of 35% to 70%. In Eqntott and Xlisp, the MCPI penalty is about a quarter of a cycle even for a very small cache. Seeking further improvement is not worthwhile. In Tomcatv and Nasa, as the cache size increases, the miss penalty of both the pure and prefetching caches is minimized until the working set is captured (32K for Tomcatv). Notice that the absolute reduction in the MCPI is significant, over one cycle in both cases, and independent of the cache size. Based on the results of the first two groups, it can be seen that the performance data at moderate cache size, 16K and 32K, argues forcefully for spending some cache real estate on the RPT and BPT rather than increasing the cache size.

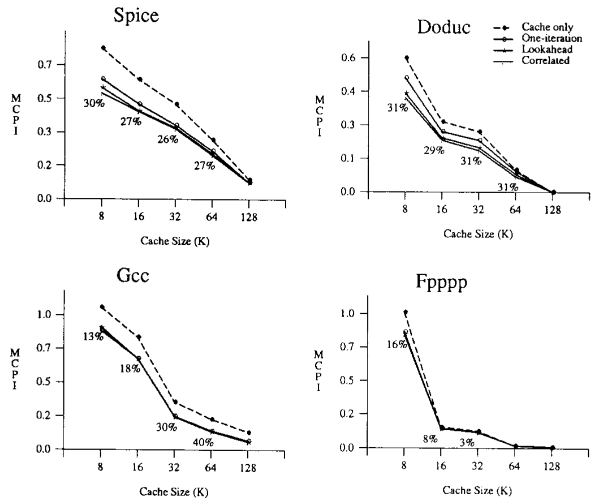


Fig. 10(b). Simulation results for δ = 30 (*Overlapped*).

The third group consists of Spice, Doduc, Gcc, and Fpppp. For Spice and Doduc prefetching is still valuable: The data access penalty is reduced by about 30%. For Fpppp, and to a lesser extent Gcc, a pure cache of 16K has almost captured the working set; prefetching cannot help much. Several factors are responsible for the small advantage brought by prefetching. First, because the fraction of references in scalar or zero stride categories dominates (98% in Fpppp and over 50% in the other three benchmarks, Table II), the performance contribution by prefetching accesses with nonzero strides become less significant. Second, the significant branch prediction miss ratio, e.g., 20% in Gcc, precludes successful prefetching. And, third, the RPT may not be capable to hold all active memory instructions at the same time because of either its limited associativity or its small number of entries. We examine this last issue later in this section.

Finally we compare the relative performances of the three reference prediction schemes. As could be expected, the increased level in hardware complexity pays off. However, the difference between the *lookahead* and *correlated* variations is always small, less than 2%, with Eqntott being the only exception with one data point showing a 10% improvement. The difference between *lookahead* and *basic* is more significant. It is most notable, differences of over 40%, in the benchmarks Tomcatv (large loop body so in the *basic* scheme the prefetched data will arrive too early displacing other useful data or being replaced before its use) and Espresso (small basic block so in the *basic* scheme the data will arrive too late, generating *hit-wait* cycles). These results show that the *lookahead* logic is worth implementing since it allows the flexibility to prefetch at the correct time while the complexity required to help data accesses in outer loops as in the *correlated* scheme plays a much less significant role.

In summary, the prefetching schemes are effective in reducing the data access penalty. A prefetching hardware unit is particularly worthwhile when the chip area is limited and a choice has to be made between the added unit and increasing slightly the on-chip cache capacity.

B. Effect of Memory Models and Latencies

Fig. 11 presents the data access penalties of the baseline cache and the *lookahead* scheme for the three memory models and memory latency varying from 10 to 50 cycles for four of the benchmarks (Tomcatv, Espresso, Eqntott, and Xlisp). Each bar corresponds to one architecture and one memory latency, with the MCPI for the *Pipelined* access and the overhead coming from the *Overlapped* and *Nonoverlapped* models stacked on top of each other. The two numbers inside the bars of the lookahead prefetching give the percentages of the penalties reduced by the prefetching for the *Nonoverlapped* model (worst) and the *Pipelined* model (best), respectively. The overhead in the case of the baseline cache comes from the waiting time incurred by a cache miss when a write back is in progress since we assume that a request in progress cannot be aborted. Similarly, the overhead in the prefetching scheme includes the stall time of a "real" demand cache miss waiting for prefetching or write-back requests in progress. Note that it is not meaningful to have a large access time, say 50 cycles, for the *Nonoverlapped* model and a small latency of 10 cycles for the *Pipelined* model. We simply intend to show the effect of the stall penalty for a large spectrum of memory bandwidth.

As could be expected, a memory interface with restricted bandwidth like that of the *Nonoverlapped* model will result in poorer relative performance improvements with longer memory latencies. This is noticeable in the benchmarks Espresso and even more so in Tomcatv, where the MCPIs are larger than in the other benchmarks. A large portion of busy time is eliminated when passing from the *Nonoverlapped* model to the *Overlapped* model and then even more with the *Pipelined* model. For all four benchmarks, the difference between the latter two models is less significant than that between the first two models. This is because much of the required parallelism can be exploited by the *Overlapped* model. The results shown in Fig. 11 indicate that an adequate interface is necessary to meet the memory bandwidth demand of prefetching techniques that exploit the parallelism among several memory requests. Cache miss reduction by itself is not sufficient to assess the value of a prefetching scheme.

As the memory latency increases, the relative access penalty of the prefetching scheme in all three models also increases.

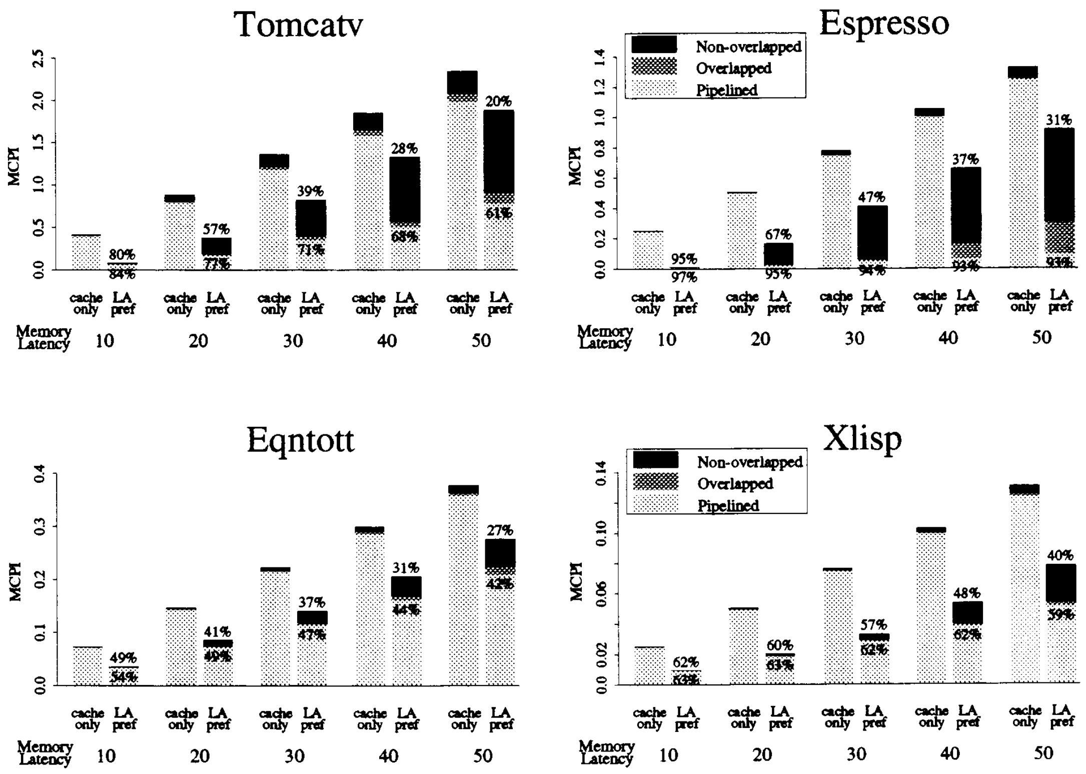


Fig. 11. Effect of memory models and latencies.

The main reason in the *Nonoverlapped* model is the lack of concurrency in the requests, resulting both in *hit-wait* cycles and in the ORL being full more often. Another reason, common to all three models, is that the *lookahead* scheme relies on the branch prediction for the LA-PC. The correctness of the prediction is sensitive to a large latency (see also Section V.E) and therefore wrong prefetches using the interface can occur more often with larger latencies. The better results obtained with small memory latencies reinforce our previous claim that the *lookahead* scheme is beneficial to high-performance processors with a limited on-chip cache. Such benefits do not degrade too much even with an interface to a secondary cache with limited concurrency such as the *Nonoverlapped* model.

C. Effect of Block Size

It is well known that for a cache of given capacity and associativity, the block size that leads to the best hit ratio is a compromise between large sizes to increase the spatial locality and small sizes to reduce conflict misses [22]. Given that a prefetching scheme will increase the spatial locality, we can predict that the best block size for a prefetching scheme should be smaller than or equal to that of the pure cache.

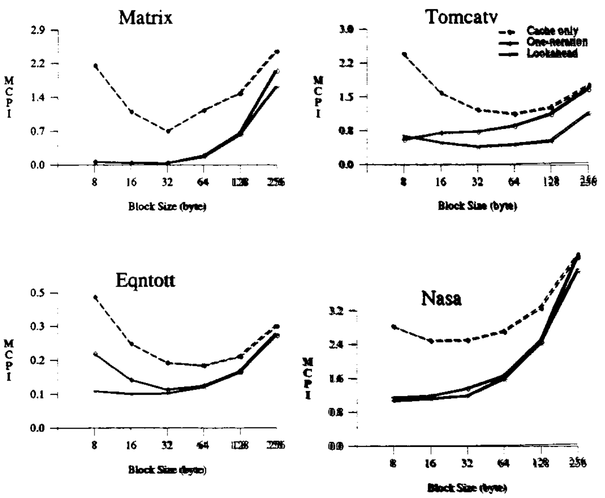


Fig. 12. MCPI vs. block size for 32K cache (*Overlapped*).

Fig. 12 presents the performance of the various architectures as a function of the block size. The baseline is a 32K-byte direct-mapped cache. The prefetched blocks are of the same size as the blocks fetched on real misses. Our experiments are based on the *Overlapped* model with a transfer rate of eight bytes per cycle, and request and memory latency of two and 20 cycles, respectively. As can be seen in the figure, the best block size for the baseline architecture is either 32 or 64 bytes and the choice can lead to significant improvements, for example a reduction in MCPI by a factor of three in Matrix and a factor of two in Eqntott when passing from a

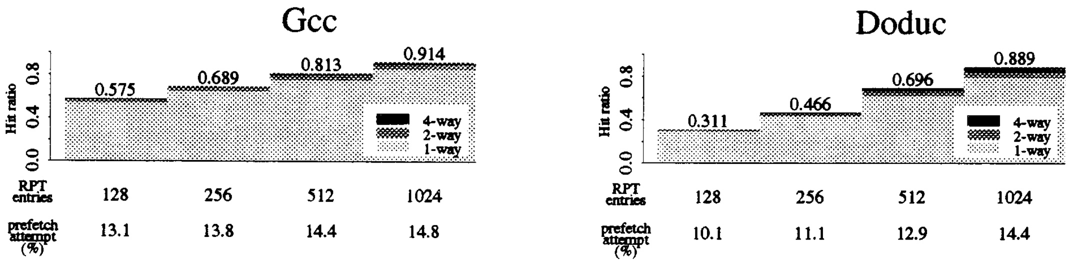


Fig. 13. Hit ratio and attempted prefetch of RPT.

block size of eight to a block size of 32. By contrast, the prefetching scheme is much less sensitive to the block size and the best results are obtained for a block size of 32 or less. This result one more time argues for the hardware prefetching being associated with an on-chip cache since limited bandwidth (small number of pins, i.e., small block size) is not an impediment to its performance.

D. Organizing the Reference Prediction Table

As discussed in Section V.A, the benefits incurred by the prefetching schemes depend primarily on program behavior, more specifically on the amount of predictable references. A second factor is the organization of the RPT, i.e., its size and its associativity.

We have stated earlier that the cost of the 512-entry RPT used in our experiments was equivalent to that of 4K-byte data cache. In each entry of the basic scheme, the *prev_addr* and *stride* fields need four bytes each, and the tag and state transition bits require the same amount of memory as the tag directory and status bits in a cache. For a lookahead scheme, we have to add a few bits per entry for the *times* field. The correlated scheme requires significantly more space (maybe 50% more). Therefore, the memory cost of a 512-entry RPT for the lookahead scheme is roughly equivalent to that of a 4K-byte data cache with block size of eight bytes. The RPT requires a comparator for the tag, as in a cache, an adder and shifter for the computation of the stride and of the effective address to be prefetched, and minimal logic for the state transition table. However, there is no need to implement write strategies when there is a conflict in the RPT nor is there a need to differentiate between clean and dirty blocks. Thus, the RPT logic might be slightly more complex than in a cache but it uses standard techniques. Similarly, the logic needed to implement the LA-PC is simple although its presence requires that the D-cache, BTB, and RPT be dual-ported.

The hit ratio of instructions referencing the RPT is over 90% in seven out of the 10 SPEC benchmarks. In an eighth benchmark, Fpppp, the hit ratio is very low, as low as 10%, even when we double the size of the RPT. This is primarily because the main part of the program consists of a large loop body with a long sequence of scalar accesses. Most of the references recorded in the RPT have been replaced when the loop starts its next iteration. For the two remaining benchmarks, Gcc and Doduc, Fig. 13 shows the fractions of instructions that hit in the RPT as a function of the size and associativity of the RPT. In addition, the line entitled "prefetch attempt %" (below the number of RPT entries) shows the percentage of accesses hitting entries not in the *no-prediction* state and with a nonzero stride for which prefetching was attempted.

Increasing the associativity of the RPT has minimal effect (Fig. 13). The sequential nature of instructions is the reason for this lack of improvement. On the other hand, increasing the size of the RPT improves the hit ratio since a small RPT cannot hold the referencing instructions of the most frequently executed loops in these two benchmarks. Note also, that the "prefetch attempt %" increases with the larger hit ratio. It is because it takes two or three accesses to regain the necessary stride information for those instructions that have been replaced. When an adequate hit ratio is obtained, the percentage of accesses with prefetches is roughly equal to that of data accesses in the *constant stride* category (Table II).

A question that might arise is given extra chip capacity, should it be devoted to a larger or more complex D-cache or a larger or more complex RPT. On one hand, a good hit ratio in the RPT may not be directly translated into a smaller miss ratio in the data cache (depending on the fraction of nonzero stride accesses). On the other hand, adding complexity to the data cache may yield a better performance, but care should be taken not to increase the basic cycle time of the cache, for example because of extra gate delays from comparators and multiplexors. However, on the basis of our experiments we would not argue for a larger or more sophisticated RPT. A possible solution for improving the RPT hit ratio without enlarging the table is to not replace those entries with nonzero strides. While useful patterns might be preserved, we are needlessly locking in the RPT entries corresponding to instructions that will be never executed again. A better approach is to enter in the RPT only those instructions that may have a nonzero stride. A compiler can easily provide this information. In an experiment on the DECstation 5000, we simply excluded memory instructions that use the stack pointer or a general register (*sp* or *gp*) from being entered in the RPT, since in general a non-scalar reference does not use these two registers as base register. The hit ratios for Fpppp and Doduc were increased up to 91% for an RPT of 512 entries.

In summary, in most cases, a moderate sized RPT, of the order of 512 entries, is sufficient to capture the access patterns for the most frequently executed instructions. A possible optimization that would be useful for programs with large basic blocks is to be selective in storing entries in the RPT.

E. Varying the Lookahead Limit

In the *lookahead* and *correlated* schemes, the LA-PC controls the timing of the prefetches. Its forward progress is bounded by the Lookahead limit *d*, the maximum number of cycles allowed between LA-PC and PC. Setting *d* must take into account two opposite effects. We should certainly issue prefetches early enough and therefore *d* must be greater than δ so that the number of *hit-wait* cycles is reduced. This is even more crucial when the data misses are clustered and the memory model is restrictive like in the *Nonoverlapped* and to a lesser extent the *Overlapped* model. On the other hand, *d* should not be too large and cross over too many basic blocks because the branch prediction mechanism loses some of its reliability with increased *d*. Also, we do not want prefetched data to replace (or to be replaced by) other useful blocks.

Fig. 14 shows the performance of the *lookahead* scheme under the *Overlapped* model as a function of the Lookahead limit *d* for two representative programs. When *d* is less than the memory cycle time δ (30 cycles in the figure), each access to the prefetched block in progress will be a *hit-wait* access and thus contributes *hit-wait* cycles to the total access penalty. The contributed *hit-wait* cycles are decreasing as *d* approaches δ. A local minimum for the MCPI happens around *d* = 35. A further increase in *d* will result in a slight MCPI increase because of the two aforementioned factors (incorrect branch prediction and data replacement). For the *Overlapped* model, it appears that setting *d* to a value slightly above δ will give the best results.

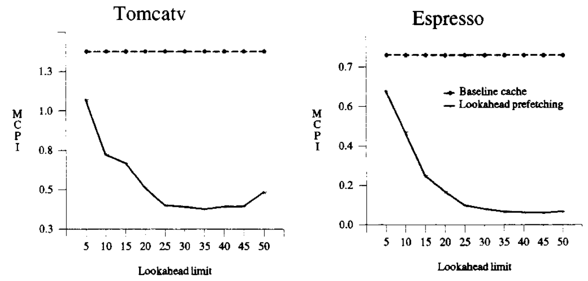


Fig. 14. MCPI vs. LA-limit for δ = 30 (*Overlapped*).

VI. Conclusions

In this paper we have described and evaluated a design for a hardware-based prefetching scheme. The goal of this support unit is to reduce the CPI contribution associated with data cache misses. The basic idea of data prefetching is to predict future data addresses by keeping track of past data access patterns in a Reference Prediction Table. Based on the various times when a prefetch is issued, we have investigated three variations: basic, lookahead, and correlated prefetching schemes.

We have evaluated these three schemes by comparing them with a pure cache design at various cache sizes. We performed these comparisons using cycle by cycle simulations of the ten SPEC benchmarks. The results show that the prefetching schemes are generally effective in reducing the data access penalty. The cost of the hardware unit is not prohibitive; a moderately sized RPT, of cost equivalent to that of a 4K D-cache, can capture the access patterns for the most frequently executed instructions. We observed that the lookahead scheme is moderately superior to the basic scheme, while the performance difference between the lookahead and correlated schemes is small.

We have also examined the performance of the prefetching scheme when we vary architectural parameters such as block size, memory latency, and memory bandwidth. The main results are that the performance of the lookahead prefetching is best for small blocks (eight or 16 bytes) and that its effectiveness is significant with a small memory latency even when assuming a restricted bandwidth interface to the next level of the memory hierarchy. These observations lead us to argue that a hardware-based prefetching scheme would be valuable and cost-effective as an assist to an on-chip data cache backed-up by a second-level cache with an access time an order of magnitude larger.

We advocate an effective prefetching hardware support unit, which is designed close to the processor without introducing extra gate delays to the critical path. However, hardware prefetching schemes will not be as effective in higher levels of the memory hierarchy. In that case, when the latencies are two orders of magnitude larger than the processor cycle time, prefetching data by software-directed techniques may be more beneficial. The software approach might also lend itself better to multiprocessor environments. Our future research will look at possible ways to combine hardware and software prefetching.

Acknowledgments

This work was supported in part by NSF Grants CCR-91-01541 and CDA 91-23308 and by Apple Computer, Inc.

References

[1] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," *Proc. Supercomputing '91*, pp. 176-186, 1991.

[2] J.-L. Baer and W.-H. Wang, "Multi-level cache hierarchies: Organizations, protocols, and performance," *J. Parallel and Distributed Computing*, vol. 6, no. 3, pp. 451-476, 1989.

[3] T. Ball and J.R. Larus, "Branch prediction for free," Technical Report #1137, Computer Science Dept., Univ. of Wis.-Madison, Feb. 1993.

[4] T.-F. Chen, "Data prefetching for high-performance processors," PhD thesis, Dept. of Computer Science and Engineering, Univ. of Wash., 1993.

[5] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 51-61, 1992.

[6] W.Y. Chen, S.A. Mahlke, P.P. Chang, and W.-M. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," *Proc. 24th Int'l Symp. Microarchitecture*, 1991.

[7] J.W.C. Fu and J.H. Patel, "Data prefetching in multiprocessor vector cache memories," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, pp. 54-63, 1991.

[8] J.W.C. Fu and J.H. Patel, "Stride directed prefetching in scalar processors," *Proc. 25th Int'l Symp. Microarchitecture*, pp. 102-110, Dec. 1992.

[9] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel and Distributed Computing*, vol. 5, no. 5, pp. 587-616, Oct. 1988.

[10] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," *Proc. 1990 Int'l Conf. Supercomputing*, pp. 354-368, 1990.

[11] N.P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, pp. 364-373, May 1990.

[12] A.C. Klaiber and H.M. Levy, "An architecture for software-controlled data prefetching," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, pp. 43-53, 1991.

[13] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Ann. Int'l Symp. Computer Architecture*, pp. 81-87, 1981.

[14] J.K.F. Lee and A.J. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, pp. 6-22, Jan. 1984.

[15] R.L. Lee, P-C. Yew, and D.H. Lawrie, "Data prefetching in shared memory multiprocessors," *Proc. Int'l Conf. Parallel Processing*, pp. 28-31, 1987.

[16] R.L. Lee, P-C. Yew, and D.H. Lawrie, "Multiprocessor Cache design considerations," *Proc. 14th Ann. Int'l Symp. Computer Architecture*, pp. 253-262, 1987.

[17] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *J. Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87-106, June 1991.

[18] T. Mowry, M.S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.

[19] S.-T. Pa, K. So, and J.T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 76-84, 1992.

[20] C.H. Perleberg and A.J. Smith, "Branch target buffer design and optimization,," Technical Report UCB/CSD 89/552, Univ. of Calif., Berkeley, Dec. 1989.

[21] A.K. Porterfield, "Software methods for improvement of cache performance on supercomputer applications," Technical Report COMP TR 89-93, Rice Univ., May 1989.

[22] S. Przybylski, "The performance impact of block sizes and fetch strategies," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, pp. 160-169, May 1990.

[23] I. Sklenar, "Prefetch unit for vector operations on scalar computers," *Computer Architecture News*, vol. 20, no. 4, pp. 31-37, Sept. 1992.

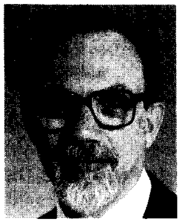
[24] A.J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.

[25] J.E. Smith, "Decoupled access/execute computer architectures," *Proc. Ninth Ann. Int'l Symp. Computer Architecture*, pp. 112-119, 1982.

[26] T.Y. Yeh and Y.N. Patt, "Alternative implementation of two-level adaptive branch prediction," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 124-134, 1992.

Tien-Fu Chen (S'90-M'93) received the BSc degree in computer science from National Taiwan University, Taiwan, in 1983, and the MS and PhD degrees in computer science from the University of Washington, Seattle, in 1991 and 1993, respectively. Since August 1993 he has been a faculty member at National Chung Chen University, Taiwan.

Dr. Chen's research interests include design and performance evaluation of cache memory systems, parallel processing, and distributed systems.



Jean-Loup Baer (S'66-M'69-SM'87-F'92) received a Diplome d'Ingénieur in electrical engineering and a Doctorat 3^e cycle in computer science from the Université de Grenoble (France) and a PhD from the University of California, Los Angeles, in 1968.

Dr. Baer is professor of computer science and engineering and adjunct professor of electrical engineering at the University of Washington where he has been since 1969. His current research interests are in computer systems architecture with a concentration on the design and evaluation of memory hierarchies, and in parallel and distributed processing.

Dr. Baer has served as an IEEE Computer Science Distinguished Visitor and was an ACM National Lecturer. He is a Guggenheim Fellow, editor of *IEEE Transactions on Parallel and Distributed Systems*, editor of the *Journal of Parallel and Distributed Computing*, and an asociate editor of the *Journal of Computer Languages*. He served as program chair for the 1977 International Conference on Parallel Processing, as program co-chair for the 10th International Symposium on Computer Architecture, and as general co-chair for the 17th International Symposium on Computer Architecture.