Hardware Prefetching in Bus-Based Multiprocessors: Pattern Characterization and Cost-Effective Hardware

M.J. Garzarán, J.L. Briz, P.E. Ibáñez and V. Viñals
Univ. Zaragoza, España
{garzaran,briz,imarin,victor}@posta.unizar.es

Abstract

Data prefetching has been widely studied as a technique to hide memory access latency in multiprocessors. Most recent research on hardware prefetching focuses either on uniprocessors, or on distributed shared memory (DSM) and other non bus-based organizations. However, in the context of bus-based SMPs, prefetching poses a number of problems related to the lack of scalability and limited bus bandwidth of these modest-sized machines.

This paper considers how the number of processors and the memory access patterns in the program influence the relative performance of sequential and non-sequential prefetching mechanisms in a bus-based SMP. We compare the performance of four inexpensive hardware prefetching techniques, varying the number of processors. After a breakdown of the results based on a performance model, we propose a cost-effective hardware prefetching solution for implementing on such modest-sized multiprocessors.

1. Introduction

Applying data prefetching on modest-sized bus-based multiprocessors is challenging. These machines are based on a cheap, but fixed and non-scalable organization, and have to cope with the ever growing CPU and memory requirements of new applications. They need to balance processor speed, cache sizes, bandwidth and final cost carefully. In this market so sensitive to cost, yet so performance-driven, cost-effective performance optimizations have a great interest.

Data prefetching has been proposed to hide read latencies in multiprocessors and uniprocessors. Software approaches can perform well whenever either the programmer or the compiler can provide or extract information about the access pattern of the data in the application [14, 16, 22]. However, they add extra instructions to the code, and their automatic application may not be easy. Hardware approaches may not predict so accurately the usefulness of the prefetched data, but they can dynamically decide when and what to prefetch. They look for regular sequential or non sequential patterns into the stream of memory references issued by the processor, or into the stream of consecutive references issued individually by each memory instruction (e.g. [5, 6, 7]). Lately, proposals based on dependence analysis among memory instructions [4, 20] and on Markovian predictors [13] have been suggested for uniprocessors. Hybrid software/hardware schemes that decrease the instruction overhead and improve the accuracy of predictions have been proposed for multiprocessors [3, 27]. However, prefetching in shared memory multiprocessors is prone to increasing memory traffic and false sharing. Furthermore, binding prefetch is an extremely conservative approach [10]. These problems are particularly important in bus-based SMPs, due to their limited bandwidths [22, 23]. Little work has been done in recent years to study the impact of prefetching on this kind of machines.

This paper focuses on how the number of processors and the memory access patterns influence the performance of sequential and non-sequential low-cost prefetching mechanisms in a bus-based SMP. We examine a system with up to 32 processors based on a split-transaction bus, with two levels of cache (on- and off-chip) and with prefetching tied to the first level. A subset of applications and kernels belonging to the SPLASH-2 suite is used as the workload [26].

In the first part of the work we characterize memory access patterns from single load instructions as the processor count varies from 1 to 32. Results reveal the dominance of sequential traversals and, to a lesser extent, presence of stride accesses. Self-linked patterns, where a single load exhibits a recurrence relating its read data with its next address [15], are almost absent in the used workload.

In the second part we compare four sequential and non-sequential prefetchers coupled to the on-chip cache against a Base system with no prefetch, again varying the number of processors from 1 to 32. The prefetcher based on the use of a Load Cache managed in a non-conventional way (only the loads missing in the data cache are inserted in it), appears to be a suitable prefetcher.

The paper is organized as follows. In Section 2, we discuss related work while introducing key concepts on hardware prefetching. In Section 3, we study the access patterns present in the applications. In Section 4, we evaluate different hardware prefetching alternatives through a detailed simulation model. That section also introduces the performance model used to discuss the

simulation results. In Section 5 conclusions are summarized, and a cost-effective prefetching mechanism is suggested for implementation.

2. Related work

Sequential prefetching is a hardware-controlled mechanism that prefetches one or more consecutive blocks in a sequential way. *Always*, *on-miss* and *tagged* are classical variations differing in the required condition for issuing a lookup [21]. When ported to multiprocessors, it can increase false sharing and rise memory traffic [5,6]. To simultaneously remove useless prefetches (lowering the required extra bandwidth) and advance the prefetch issue if it benefits the hit ratio, a variation named *adaptive sequential prefetching* adjusts dynamically the number of consecutive blocks being prefetched *(degree of prefetching)* from zero to a maximum [5].

Other proposals try to predict both sequential and non-sequential accesses. A *Load Cache* (**LC**) mechanism consists of a table relating the PC-address of a `load` instruction with its individual addressing behavior. Whenever a `load` that is not already in the **LC** is executed, an **LC** miss arises and it is inserted into the **LC**. Data addresses issued by that `load` in successive executions are tracked down, and once a pattern (sequential or stride) is recognized, the prefetch address will be computed and issued to the data cache when the `load` be executed again. The original proposal (*Reference Prediction Table*) was introduced in [2] and evaluated in [3] along with a mechanism based on a *lookahead PC* for issuing just-in-time prefetches. Later proposals elaborate more on the topic, either evaluating, simplifying the prefetch scheduling or adding recognizable patterns [6, 9, 12, 15]. In particular, [15] introduces the concept of value-address recurrence, enabling identification of two patterns appearing in single-load chained list traversal. Always in a uniprocessor context [20] and [19] extend the recognizable patterns to value-address recurrences between load pairs.

Unlike the conventional **LC** management, a Load Cache *with on-miss insertion (***LCm)** inserts in the table *only* those loads missing in the data cache, preventing high-locality loads from polluting the **LCm** [11]. On-miss insertion net effect is twofold: it avoids useless cache lookups and achieves a very good space efficiency, usually outperforming the conventional **LC**. For instance, a **LCm** with 8 entries can perform even better than a conventional **LC** with 512 entries, but such advantages have not been verified in a multiprocessor system. The same work pointed out that activating sequential tagged prefetching in parallel with **LCm** (called **LCms**), can achieve the best results in many cases, since the spatial locality coming from an interleaving of several load streams can not be captured by an **LC**-based prefetcher.

Sequential and stride prefetching have been extensively studied on large-scale DSM [5,6,7] or bus-based SMP [3]. In most cases, processors and memory are connected through a multi-path interconnection network and therefore contention is assumed to be low. In particular, results in [6] show that sequential prefetching generally outperforms stride prefetching in a DSM environment, but it may increase memory traffic significantly. This becomes a serious issue in bus-based SMPs. There are two well-known works on bus-based SMPs which use software prefetching [22, 23]. However, they consider only a single level of cache, and their simulations were performed on previously-recorded address traces. Furthermore, all previous works use a fixed number of processors for a given application in their experiments, and they do not evaluate how its variation may affect the behavior of the prefetching algorithm. The length of memory accesses following a given pattern has been measured in [6], but little systematic work on pattern characterization has taken into account the number of processors.

Summarizing, we feel that an **LCm** equipped with stride and list pattern recognizers is worth trying on a bus-based multiprocessor, because of several reasons: it could press the cache and the memory far less than sequential prefetchers do, it can capture a rich set of patterns, and its implementation cost is low. On the other hand, when adding tagged sequential prefetching, **LCms** could outperform in particular applications requiring moderate bandwidth. Since bandwidth can become a limitation as the number of processors grows, we will consider its impact both on the addressing pattern distribution and the **LCm**/**LCms** performance.

3. Pattern Characterization

The prefetching techniques that we use strongly rely on the existence of predictable memory access patterns. As far as we know, there is no systematic study of such patterns in parallel applications as the number of processors varies. Therefore, we have analyzed `load` accesses for a subset of the SPLASH-2 suite [26], using sequential and non-sequential pattern detectors.

3.1. Workload and Methodology

Table 1 shows the selected SPLASH-2 applications and some metrics for 16 processors. The applications have been targeted to a MIPS-2 architecture and run until completion by using MINT, an execution driven simulator for multiprocessors [24]. Throughout the paper the analysis is constrained to the parallel section of the programs.

SPLASH-2 programs have been classified into three classes according to the interaction between their data structures and access patterns and the cache lines. We refer to these classes as: *a*) *Regular* programs, based on contiguously allocated data structures (Ocean, FFT, LU, Radix and Cholesky), *b*) *Particle* programs, based on particles, i.e. structures that can share the same cache line, and that are accessed by different processors (FMM, Barnes); and *c*) *Irregular* programs, with highly irregular data structures (Radiosity, Raytrace). The implementations of Ocean and LU used in [25] corresponds to those specified here as Non Contiguous: main data sets are 2-dimensional arrays where subblocks are not contiguous in memory. Alternatively, Ocean and LU Contiguous implementations allocate data as arrays of subblocks, in order to reduce or even eliminate false sharing. Moreover, in the case of Ocean contiguous, a multigrid (instead of a SOR) solver is applied.

3.2. Measured Patterns

We have looked for five read patterns which can potentially be recognized by the prefetching techniques we will use later. Recognition has been done by tracking for each single `load` the recurrence conditions indicated in Table 2, where A_i is the address

	Program	Parameters	Inst. (M)	Reads (M)	Writes (M)
regular	Cholesky	tk15.O	584.4	201.8	27.7
	FFT	64K points	32.9	8.1	5.8
	LU	512x512 matrix, 16x16 blocks	340.6	97.6	47.8
	LU non-	512x512 matrix, 16x16 blocks	340.9	97.5	47.8
	Ocean	258x258, tolerance 10^{-7}, steps 4	282.3	81.6	18.5
	Ocean-non	258x258, tolerance 10^{-7}, steps 4	477.4	101.8	17.8
	Radix	1024 K keys, radix =1024	47.9	11.2	6.6
particle	Barnes	16K particles	2569.0	861.0	575.1
	FMM	16K particles	1035.7	232.8	38.7
irregular	Radiosity	room -ae 5000.0 -en 0.050 -bf 0.10	2878.5	574.8	305.8
	Raytrace	car	994.0	228.6	117.6

Table 1: Selected subset of SPLASH-2. Statistics are for 16 processors, compiled with cc -O2 -mips2 -non_shared (MIPS SGI v.7.1 compiler). Ints refers to the total number of executed instructions across all processors. We also show the total number of Reads and Writes.

generated by a `load` during its *i-th* execution, D_i is the value read during its *i-th* execution, and *Bsize* is the block size. The first three

Pattern		Acro.	Recurrence	Param.
Regular Traversing	Scalar	SCA	$A_i = A_{i-1}$	
	Sequential	SEQ	$A_i = A_{i-1} + \mathbf{s}$	$0 < \mathbf{s} \leq$ Bsize
	Stride	STR	$A_i = A_{i-1} + \mathbf{S}$	($\mathbf{S}$ > Bsize) || ($\mathbf{S}$ < 0)
Irregular Traversing	Pointer List	PTR	$A_i = D_{i-1} + \mathbf{d}$	record displacement = **d**
	Index List	IND	$A_i = \mathbf{4}*D_{i-1} + \mathbf{K}$	index integer size = **4** base address of the index array **= K**
Not Recognized		NR	Not Recognized as any of the previous ones	

Table 2: Considered load patterns and the recurrences defining them. A_i and D_i refer to the address and value of the i-th instance of a given load instruction. The sequence length is the number of consecutive addresses observing a recurrence with a constant parameter value.

patterns (SCA, SEQ and STR) are particular cases of regular traversing, characterized by address-address recurrences, whereas the next two (PTR, IND) are the value-address recurrences introduced in [15]. The PTR pattern can appear when traversing a list of records chained by pointers (f.i. `p=p->next`). The IND pattern may appear in symbolic or numeric applications, when an array of indexes is used to access to a chained set of elements (f.i. `i=index[i]`). When an instance of a `load` matches several patterns at a time, it is first classified according to the pattern recognized in the previous instance. If several patterns arise repeatedly, the following priorities are applied: SCA, PTR, IND, STR and SEQ.

Moreover, we have measured *sequence lengths*, whose distribution is considered in detail in subsection 3.4. Sequence length is defined as the number of consecutive addresses observing a recurrence with a constant parameter. As an example, the address stream

..., 10, 4, **68**, **132**, **196**, 100, ...

contains a sequence of length 3, since the 3 bolded addresses differ from the previous ones in the constant stride 64. The minimum length we detect is 2, corresponding to 2 parameter repetitions.

3.3. Pattern Distribution

Figure 1 shows the pattern distribution of all load addresses for 16 processors. For each program we show two bars. In the left bar all sequences of length greater or equal than 2 have been included. In the right bar, the first two references of any sequence have been excluded and counted in the NR group (see explanation in 3.4).

A remarkable point is the practical absence of the PTR and IND patterns in the whole set of programs, with measured percentages always falling under 0.1%. This agree with previous characterizations available for uniprocessor applications[1]. Consequently, when experimenting later with LC-based prefetchers, we shall not include PTR and IND pattern detectors.

To a large extent, the results mirror the features of the application data structures. Thus, *regular programs* have a large fraction of SEQ pattern and -excluding Radix- a low fraction of NR accesses (9.7% in average). In contrast SEQ is very uncommon in the other two classes of programs. Moreover, contiguous implementations of LU and Ocean prove to increase the percentage of SEQ pattern: in LU this increase adds regularity (the NR group decreases) and in Ocean STR disappears and SEQ increases but the overall regularity remains unchanged..

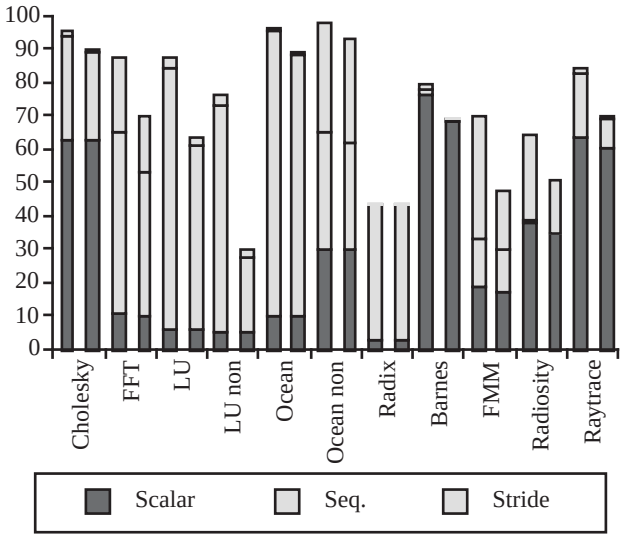


Figure 1: Breakdown of pattern distribution for 16 processors and Bsize = 32. For each program we show two bars (see text). Pointer and Index patterns are not plotted due to their negligible contribution.

Barnes and FMM are *particle programs* and most structures are linked lists. Nevertheless, a significant amount of the STR, SEQ and SCA patterns appear in FMM, because of the loops in its code that access 1-dimensional global arrays sequentially, and a 2-

1. For example, only Spice of Spec92 and health of Olden show significant percentages of self-linked patterns(21% of self-index and 27% of self-poin ter, respectively), being these patterns irrelevant in the whole set of Spec95f, Perfect and NAS-serial suites and of little importance in Spec95int and Olden (under 3% and 6%, respectively in average) [19].

dimensional global array following the STR pattern. Moreover, the compiler unrolls loops, converting many sequential accesses to arrays into STR accesses. In Barnes, integer and float global variables are frequently loaded inside a recursive function that performs the core computations, contributing in a remarkable way (76.2%) to the SCA pattern. Finally, NR accesses are significant in *particle* and *irregular* programs (25% in average), suggesting the existence of more complex patterns which will presumably not be exploitable from simple LC-based prefetchers.

In addition to 16-processors runs, we have performed simulations for 1-, 2-, 4-, 8-, and 32-processors runs. We have found that for most applications, the sum of the loads executed by all processors remains constant or increases only slightly with the number of processors. In addition, the weights of the patterns observed do not change when we vary the number of processors.

The only exceptions are Cholesky and Radix. In these applications, the sum of the loads increases with the number of processors (Figure 2). In Cholesky it can be observed a slight transfer from SEQ to STR (ending with a 2% stride for 32 proc.) and a remarkable increase in the number of scalar loads (a factor 20x from 1 to 32 proc.). In Radix, when adding processors, the additional loads split equal between SCA and SEC patterns

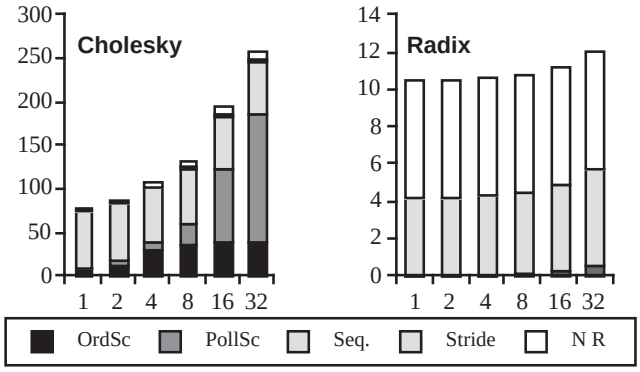


Figure 2: Pattern Distribution (million loads) for Cholesky and Radix up to 32 processors. Scalar is split in Ordinary loads (regular and synchronizing) and Polling loads.

These two applications have a high communication-to-computation ratio: each processor communicates some locally-computed data to the rest of the processors:

- In **Cholesky**, in each iteration of the outermost loop, a processor computes the pivot element of the diagonal and forwards it to the rest of processors. It has a similar structure and partitioning to the LU factorization kernel, but it is not globally synchronized between steps, and the consumer processors wait for the pivot *spinning on two scalar variables*. Consequently, processors do not synchronize only using synchronization variables: they also synchronize by polling on regular variables. In our simulations, we are able to identify the synchronization variables, and any spinning on them appears as a single load in our bars. However, all the loads caused by the polling on regular variables are counted, and appear in the bars as part of the Scalar pattern. To separate these polling loads from ordinary (regular and synchronizing) loads, we break down the Scalar category into *Poll-scalar* and *Ord-scalar*, respectively. As can be seen from Figure 2 the effect of such polling loads heavily affect the pattern distribution as Nprocs increase. Even though SEQ percentage decreases sharply (from 85.2% for 1 processor to 22.8% for 32), note that the increasing Poll-scalar accesses come from only two scalar loads, so the influence of this scalar stream over a LC-based prefetcher should be negligible.

- The goal of **Radix** algorithm is to sort N integers or keys (of B bits using a radix R. The algorithm proceeds in ((B/R) + 1) phases or iterations, and each iteration has several steps. In each iteration, a processor has to sort N/Nprocs keys. In the first step each processor uses a local data structure of R entries to compute an histogram -this step is the main responsible of the *high NR pattern*, since the visiting of entries does not follow any pattern-. Next, it traverses this local structure to accumulate the values in order to form the local accumulative histogram. In the second step, the local histograms are accumulated again into a global histogram. The parallel algorithm used requires log2Nprocs stages to compute a global histogram from the local ones. The amount of work in this second step depends on the size of the radix (which is the size of the histograms) and on the number of processors. Finally, the global histograms are used to copy the keys to the sorted position in the output array. Thus, when increasing the processor count, the instructions devoted to do the histogram reduction -which follow either the SCA or SEQ pattern- also increase: a 14.9% load increase from 1 to 32 processors.

Therefore, in applications where a significant overhead is added as the processor count grows (communication and synchronization), the relative weight of patterns can vary. In turn this can lead to applications that benefits from a prefetch predictor when executing in a few processors, but which may not benefit when executing with more processors. We feel this consideration is important, since applications obtained from sequential algorithms through automatic parallelism extraction easily could fall in this class of poor-scalability applications.

3.4. Sequence Length

LC-based prefetching is hardly useful if sequence lengths are short, due to the learning time devoted to get confidence. If we consider a state machine that only issue a prediction after two parameter repetitions, the captured pattern distribution changes as plotted in the right bar of each program in Figure 1. The overall regularity (SCA+SEQ+STR) decreases for all programs (15.1% loss in average), being noticeable the 45.9% loss in capturing the SEQ pattern of LU-non.

A finer analysis appears in Table 3, where the *dominant sequence lengths* are recorded for 16 processors, along with the load percentage that follow them. We have lumped the more than 7 together. The SEQ pattern has been split up into 4-, 8-, 16- and 32-byte sequences, as if they were different patterns. Sequences with percentages below 2.0 are excluded from the main columns and accumulated in the remaining (REM) column. The maximum length is featured in **bold**

It can be observed that usually there are a few outstanding lengths per application. Moreover, the SCA patterns are nearly always executed in long bursts (>7), and the same holds true for SEQ-4, -8 and -16. By contrast, STR and in a lesser extent SEQ-32 patterns, are executed in shorter bursts (e.g. *length 4* (34.3% STR)

	SCAL.	STRIDE	SEQUENTIAL				REM	NR
			4	8	16	32		
Cholesky	>7 (62.9%)	-	-	>7 (17.6%)	>7 (7.7%)	-	7.6%	4.3%
FFT	>7 (10.5%)	3 (7.6%) >7 (14.2%)	-	-	-	2 (5.0%) 6 (7.4%) >7 (39.9%)	3.3%	12.1%
LU	>7 (5.8%)	>7 (2.9%)	-	-	-	3 (29.5%) 7 (4.7%) >7 (44.0%)	0.8%	12.4%
LU Non	>7 (5.7%)	>7 (2.9%)	-	-	-	3 (67.1%)	0.7%	23.6%
Ocean	>7 (10.1%)	-	-	>7 (31.1%)	>7 (25.2%)	>7 (28.0%)	2.1%	3.5%
Ocean Non	>7 (29.9%)	>7 (32.6%)	-	>7 (4.9%)	>7 (21.3%)	>7 (8.8%)	0.3%	2.3%
Radix	>7 (2.6%)	-	>7 (3.8%)	-	>7 (37.4%)	-	0.2%	56.0%
Barnes	6 (2.4%) 7 (9.5%) >7 (57.5%)	-	-	-	-	-	10.3%	20.2%
FMM	>7 (17.7%)	4 (34.3%)	>7 (3.9%)	-	>7 (4.1%)	>7 (5.0%)	5.3%	29.6%
Radiosity	>7 (33.2%)	2 (3.4%) 3 (2.7%) 5 (2.8%) >7 (12.5%)	-	-	-	-	10.0%	35.4%
Raytrace	>7 (61.2%)	-	>7 (3.4%)	-	3 (11.8%)	-	8.2%	15.3%

Table 3: Dominant sequence lengths in all patterns for 16 processors. Maximum values for each application are shown in bold. Percentages below 2.0 are omitted and accumulated in the REM column.

in FMM and *length 3* (67% SEQ-32) in LU-non. As a rule, SEQ patterns perform longer sequences than the STR ones and 32B strides usually dominate over shorter ones.

Note that sequential tagged prefetching could yield good results here, because it can capture SEQ patterns coming from small-length sequences which will not be fully profited by an LC-based predictor.

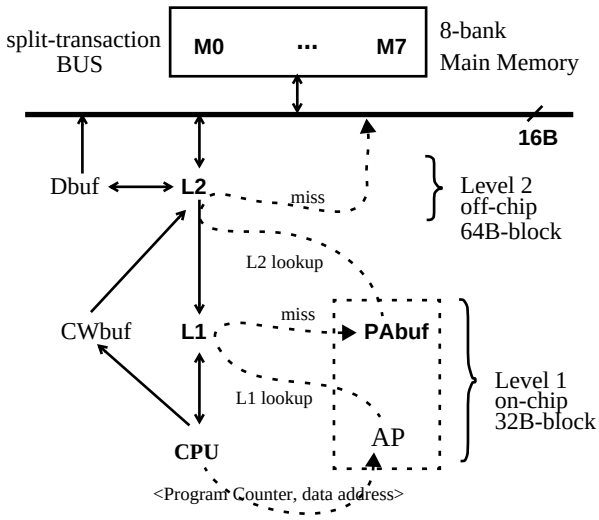
We now consider varying the number of processors. The number of loads executed *by each processor* decreases when the processor count increases. This is true even for the communication-intensive applications Cholesky and Radix. This reduction in the number of loads per processor could cause a reduction in the *sequence length*s. If this was the case, an LC-based prefetch mechanism would be less useful as we increase the number of processors executing the application. However, simulations with 1, 4, 8, 16, and 32 processors show only small differences (see next Section for details). This observation means that the length of the traversals in the data structures where prefetch is *effective* changes relatively little with the number of processors. As a result, the behavior of the prediction mechanism depends little on the processor count.

4. Hardware Prefetching Evaluation

Results in the previous Section constitute a reference, regarding the benefit we can expect from applying hardware prefetching mechanisms based on address predictors. In this Section we will evaluate four low-cost hardware prefetching mechanisms on a non-ideal bus-based system.

4.1. Base System

The system model we have considered is outlined in Figure 3, detailed for a single processor. We simulate a system with 800 MHz RISC processors, that execute a MIPS-II ISA, where each



CWbuf:	Coalescing Write Buffer	**AP:**	**Address Predictor: Ts, LC, LCm, LCms**
Dbuf:	Dirty Buffer (copy back)		
PAbuf:	Prefetching Address Buffer	**L1:**	First-Level data Cache
		L2:	Second-Level data Cache

Figure 3: Multiprocessor system used for comparing prefetching techniques. The Bases System (no prefetch) excludes the dashed components. (Address predictor and Prefetching Address Buffer)

processor has two cache levels. The bus has split transactions and is based on the DEC 7000 and 10000 AXP Family [1]. It runs at 200 MHz, with 32 bits for addresses and 128 bits for data, which yields 3.2 GBps of bandwidth. Due to the lengthy simulations that our experiments require, we simulate a single-issue processor with blocking read misses. Write latency is hide by using a Coalescing Write Buffer (**CWbuf**) operating as described in [25]. According to [17], the performance of a system with a W-issue superscalar can be roughly approximated by simulating a single-issue processor with W times higher frequency. Consequently, our speed results will roughly approximate the speed of a system with 2-issue superscalar processors cycling at 400 MHz. The pipelined bus protocol consists of four phases: *1*) Request & arbitration (4 CPU cycles), *2*) Address & command issue (4 CPU cycles), *3*) Read bank & snoop (32 CPU cycles), *4*) Transfer in chunks of 16B (4x4 = 16 CPU cycles).

The off-chip second-level cache memory (**L2)** is copy-back and write-allocate; dirty victim blocks are placed into a copy-back Dirty Buffer (**Dbuf**). **L2** supports up to three concurrent misses: a *read miss* (L1 also missed and processor stalls), a *write miss* (the top entry of **CWbuf** is not in **L2**) and a *prefetch miss*. The first level (on chip) is composed of the **CWbuf** and of a write-through, no write-allocate cache (**L1**). Table 4 shows sizing, timing and structure for all the hierarchy components. Cache sizes will be detailed in Subsection 4.3.

Coherence in **L2** and **Dbuf** is kept using an Illinois-based MESI invalidation protocol [18], which uses an additional *shared* control line in the bus. Associativity, block size and number of sets of **L2** and **L1**, along with the replacement rule of **L2**, guarantee the inclusion property between cache levels. The **CWbuf** can delay and aggregate writes belonging to the same **L1** cache blocks, and therefore other processors are not able to immediately observe

those writes. This buffer is emptied before releasing a lock or acquiring a barrier, and the processor stalls while the **CWbuf** empties. This behavior is based on the support the Alpha AXP Architecture provides for implementing release consistency [8].

Component	Timing (CPU cycles)	# Entries or size	Structure
Main Memory	32	8 banks	64B block, interleaved
Bus	4: 4: 32: 4x4	16B width	split-transaction
L2 cache	4: rd/wr hit from L1 8: fill 64B from bus	see Table 5	4-way, 64B block Data Only
Dbuf (Dirty Buffer)	L2+1: hit from L1	6 entr 64B/entry	FIFO
CWbuf (Coalescing Write Buff.)	1: rd hit from L1	6 entr 32B/entry	FIFO
L1 cache	1: rd/wr hit; 2: fill from L2	see Table 5	1-way, 32B block Data Only
PAbuf (Prefetch Address Buff.)	1	6 entr	FIFO, filter addresses already present
Load Caches: LC, LCm, LCms	1	16	1-way + data path for computing address predictions

Table 4: Default sizings, basic timing and structure for memory, bus, caches and buffers.

4.2. System with prefetch

The Address Predictor (**AP**) and the Prefetch Address Buffer (**PAbuf**) make up the prefetch subsystem. We consider the following four Address Predictors: *1*) **Ts** is the sequential tagged prefetching described in [21], *2*) **LC** stands for a *sequential* and *stride* predictor based on a conventional Load Cache [2], *3*) **LCm** is a Load Cache with on-miss insertion [11] and finally, *4*) **LCms** is a Load Cache with on-miss insertion plus the sequential tagged operating in parallel. As suggested in the previous Section we not add any list prediction ability due to the practical inexistence of the right patterns.

The modeled Prefetch Subsystem works as follows. According to the internals of each address predictor, a particular stream of prefetch lookups is issued to the first level. Dedicated ports allow CPU demands and prefetches to proceed in parallel. A hit ends up the prefetching activity, whereas a miss (both in **CWB** and **L1**) pushes the predicted address in the **PAbuf**, which will contend (with the lowest priority) for the only port of the second level. No other prefetch request is issued to the second level until the previous one is serviced. A block prefetched from the Bus is loaded in both cache levels.

4.3. Workload and Methodology

We have focused on two kernels (FFT and Radix) and four applications (FMM, Radiosity, Ocean and Ocean-non). As we concentrate on the effects of data prefetching, we disregard the time or bandwidth invested in instruction processing, as in other related works (e.g. [6, 14, 23]).

We have arranged the experiments into two sets: *Set 1* models a system that strongly presses the memory system, while *Set 2* matches cache sizes closer to current organizations (see Table 5), WS1 and WS2 are respectively the primary and secondary working sets stated in [26] for 32 processors. The size ratio between levels is 1:16 for Set 1 and 1:32 for Set 2.

	SET 1		SET 2	
	L1	L2	L1	L2
FMM	2 KB < WS1	32 KB > WS1	8 KB < WS1	256 KB < WS2
FFT	8 KB < WS1	128 KB > WS1	32 KB < WS2	1MB > WS2
Ocean				
Ocean-non				
Radiosity	2 KB < WS1	32 KB > WS1	8 KB < WS1	256 KB > WS1
Radix	8KB < WS1	128 KB < WS1	32 KB < WS1	1MB > WS1

Table 5: Selected cache sizes for the two experimental sets and their relation with the working sets.

4.4. Performance model

In order to consider the prefetching activity over two cache memory levels, we will use the performance model displayed in Figure 4. All ratios are relative to the number of `loads` instructions.

The bar on the right of the zero axis is the L1 load miss ratio, split into the misses serviced either from Main Memory or from L2 (*L1 demand miss ratio = from M + from L2*). The bar on the left is the L1 miss ratio for prefetch requests (*L1 prefetch miss ratio*). In this bar misses are first classified according to the supply source (Main Memory or L2) and further divided by their utility (useful and useless): *L1 prefetch miss ratio = from M* (useful + useless) + *from L2* (useful + useless). A *useless* prefetch means data prefetched but *a*) never demanded by the processor, or *b*) replaced or invalidated before being requested.

Note that by adding the two *from M* miss ratios (demand and prefetch) we obtain the main memory requests per read reference, or *Main Memory traffic ratio* from now on. On the other hand, the total bar length (L1 demand+prefetch miss ratio) equals the L2 requests per read reference, or *L2 traffic ratio.* Finally, the number **P/N** on the left is the number of prefetch requests per every 100 `loads`, a measure of the lookup pressure on L1.

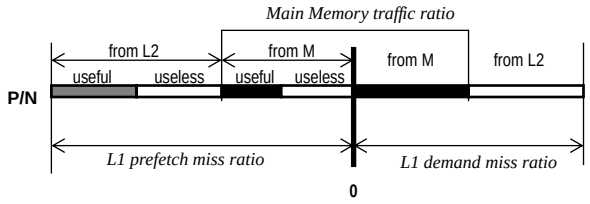


Figure 4: Performance model showing the demand and prefetch miss ratios of interest.

This performance model helps to analyze the success of a given prefetch system —decreasing execution time— by highlighting three important goals that must be balanced: *a*) to achieve a low *L1 demand miss ratio*; *b*) to keep the L2 traffic ratio low, and *c*) to keep the *Main Memory traffic ratio* low. Moreover, long left-hand bars will indicate a very active prefetcher; in that case, *useless* areas must be considered carefully. For a given prefetcher, when a right bar appears shifted to the left, with respect to the bar without prefetching, it means that the prefetcher reduces *L1 demand miss ratio*.

4.5. Results

4.5.1. Base System. Simulation results for the system without prefetching are shown in Table 6 (L1 demand miss ratio) and Figure 5 (Speedup, average read access time T_a —in CPU cycles— and bus utilization BU). The number of processors ranges from 1 to 32. It can be observed that the operating point in Set 1 has been selected in order to get substantial miss ratios. Rising the number of processors, BU almost saturates in several cases, particularly in FMM and Ocean. By contrast, miss ratios and BU are much lower in Set 2. T_a increases with the number of processors in most cases, because more processors compete for the same number of memory banks. The progressive saturation of the bus dominates the effect of having a greater global cache size when we have more processors.

We have also calculated the invalidation rates with respect to the total number of memory references, for each program and varying the number of processors. Differences are negligible when prefetching is applied, and we will not devote further attention to them.

		Number of processors				
		32	**16**	**8**	**4**	**1**
FMM	Set 1	21.8	22.3	22.4	22.7	23.8
	Set 2	8.0	8.1	8.2	8.2	8.7
FFT	Set 1	4.8	4.8	4.8	4.8	4.0
	Set 2	2.9	2.9	2.9	2.9	2.9
Ocean	Set 1	13.5	13.4	13.7	14.0	13.5
	Set 2	7.6	9.9	12.3	11.7	12.4
Ocean non	Set 1	11.0	11.4	11.2	11.8	14.5
	Set 2	8.4	9.5	9.9	10.5	13.2
Radiosity	Set 1	12.2	12.1	12.6	11.6	12.0
	Set 2	5.4	5.3	4.9	4.8	4.3
Radix	Set 1	8.5	7.8	7.4	7.1	7.1
	Set 2	5.6	5.6	5.6	5.5	5.5

Table 6: L1 demand miss ratios of loads for the Base system

The Base system miss ratios (Table 6), shows a marked stability with independence of the processor count. This allow us to concentrate in a 16 processor system to analyze the miss ratios of the whole set of prefetching alternatives.

4.5.2. Discussion of Prefetching Alternatives. Firstly, we will analyze the miss ratios of the four prefetching alternatives considered against the Base system for 16 processors (Figure 6). Let us point out some observations:

- When compared against the Base system, the *Main Memory traffic ratio* shifts to the left. Tagged- Sequential prefetchers (**Ts** and **LCms**) increase this traffic in FFT and Ocean Non.
- With respect to *L1 demand miss ratio*, the prefetch

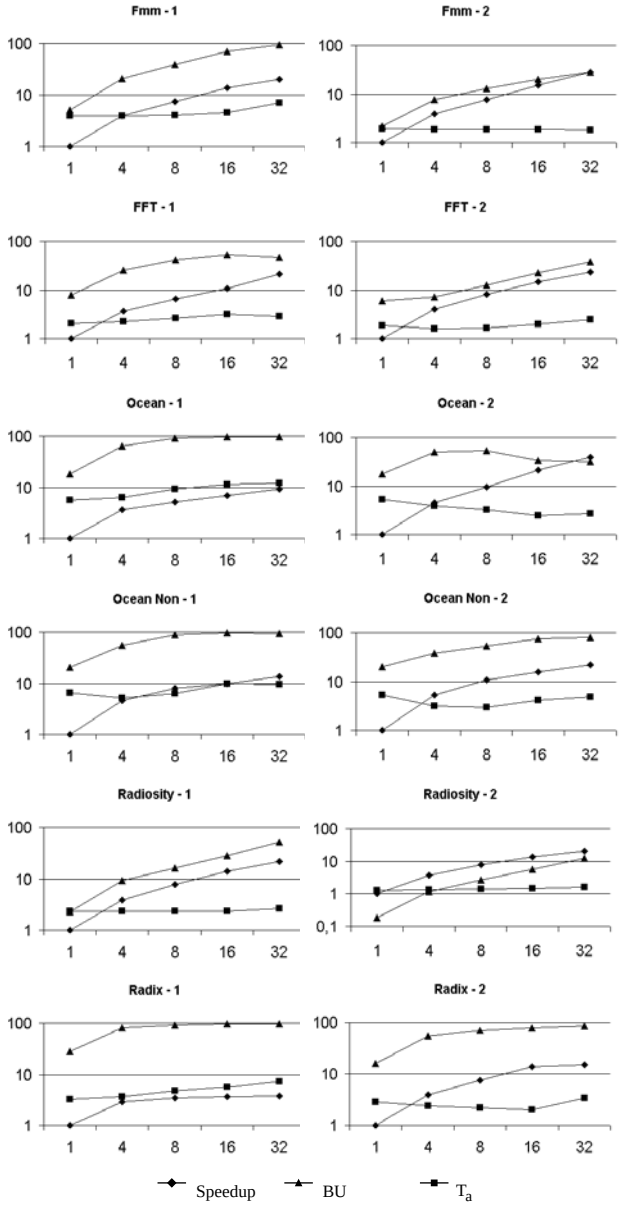


Figure 5: Speedup, T_a (cycles) and BU (in %) vs. number of processors, for the Base system.

mechanisms can be divided into two groups: *a*) **LC** and **LCm** and *b*) **Ts** and **LCms**. The second group achieves stronger reductions, except for FMM and Ocean Non, the two programs with higher percentages of STR pattern.

- **LC** and **LCm** generate a small number of *useless* prefetch requests. On the other hand, the **Ts** and **LCms** can trigger many *useless* prefetches, specially for those applications with a small fraction of sequential pattern (FMM and Radiosity in Figure 1), where traffic from **L2** to **L1** due to prefetching is similar and high for these

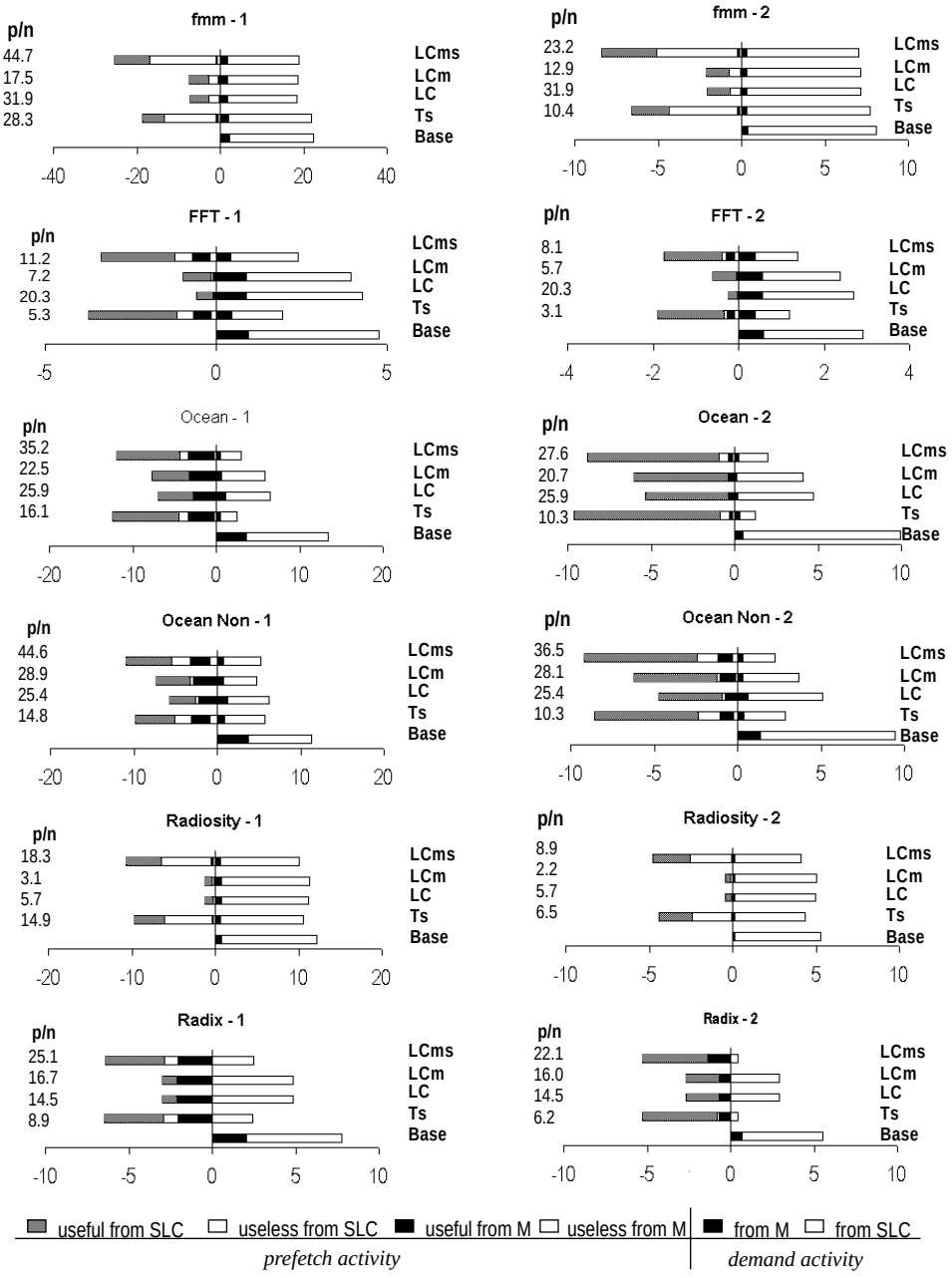


Figure 6: Demand and prefetch miss ratios, showing the behavior of the Base system and Ts, LC, LCm, LCms Prefetchers for 16 processors.

prefetchers.

• **LCms**, which combines requests from two predictors, usually makes the highest pressure on **L1** (P/N). On the other hand, P/N in **LC** *only* depends on the recognized patterns and on the number of entries. Thus, note that P/N is the same in both Sets for this prefetcher, even though cache sizes are very different. Remember, however, that in our system prefetch requests do not stall the processor, since we have a dedicated prefetch port in **L1**.

Now, let us focus on the performance of prefetchers against the Base System. Table 7 shows the speedups relative to the Base System achieved by **LCm**, **LCms** and **Ts**, for 1, 4, 8, 16 and 32 processors respectively. Note that speedups for a given number of processors are not comparable with those for other number of processors. Numbers in **bold** indicate the best speedup for a given number of processors. However, we do not show results for **LC**, since it is the best prefetcher only in Radiosity (16 processors, Set 1) and it degrades performance less than the others only in Radix (32 processors, Set 1).

		SET 1			SET 2		
		LCm	LCms	Ts	LCm	LCms	Ts
FMM	1	3.82	1.92	-1.47	1.51	1.74	0.97
	4	4.90	0.95	-1.19	1.47	1.45	0.62
	8	4.34	0.33	-2.12	1.31	0.62	-0.09
	16	4.35	-4.53	-7.04	1.07	-0.68	-0.62
	32	1.19	-17.10	-17.38	0.93	-2.04	-1.94
FFT	1	7.00	8.99	8.29	6.80	7.88	6.86
	4	1.42	5.91	5.76	1.81	5.25	4.82
	8	2.51	3.70	4.10	0.51	1.71	1.71
	16	1.10	1.10	1.76	0.31	-4.35	-3.11
	32	-0.43	-17.45	-17.02	0.98	-14.15	-14.63
OCEAN	1	16.65	22.97	23.02	18.35	24.07	24.31
	4	12.23	15.47	15.28	16.29	19.57	20.58
	8	2.90	4.85	6.13	8.42	11.26	11.03
	16	8.58	9.02	6.41	5.07	2.63	2.63
	32	-1.27	-3.68	-2.11	2.41	-2.58	-1.85
OC NON	1	28.58	25.07	19.29	24.96	28.27	24.53
	4	20.32	22.77	21.35	17.56	21.63	20.85
	8	6.84	6.36	6.54	10.86	12.57	12.24
	16	2.00	-2.41	-2.56	2.23	1.12	1.61
	32	-0.09	-4.78	-3.84	-1.44	-3.80	-2.34
RADIOSITY	1	0.79	3.33	2.41	0.39	1.41	1.05
	4	3.54	3.65	3.58	2.10	1.26	0.90
	8	1.04	2.80	1.83	0.90	1.47	0.88
	16	-0.54	0.24	-0.17	0.65	1.06	1.61
	32	0.77	-0.31	-1.40	0.63	1.02	-0.25
RADIX	1	8.67	12.74	12.39	13.65	20.10	20.15
	4	4.03	6.49	6.09	7.86	12.46	12.32
	8	2.01	3.07	1.39	0.18	6.42	4.31
	16	-0.63	-0.69	-0.85	1.66	4.66	2.33
	32	-1.28	-2.29	-2.22	0.55	2.39	2.39

Table 7: Speedups of **LCm** and **LCms** and **Ts** relative to the Base System

Relations between pattern distribution and performance can be clearly observed only in certain cases. Thus, in the two programs with the higher percentage of stride pattern (FMM and Ocean Non, Table 2), either **LCm**, **LCms** or both achieve stronger reductions in demand miss ratio and perform always better than **Ts** (Table 6). In Ocean —the application with a higher percentage of sequential pattern— **Ts** eliminates more demand misses, but outperforms **LCm** and **LCms** only in 4 over 10 cases, yet differences in the speedups relative to the Base System (Table 7) are often negligible. For the other three applications, these relations are not so clear. Looking at **LCm** and **LCms**, one of the two is the best prefetcher in 83% of cases, improving performance with respect to the Base System in 70% of cases. **LCm**, **LCms** and **Ts** yield negative speedups respectively in 8.3%, 9% and 12% of cases. There are 5 cases (3%) where all prefetchers degrade performance respect to the Base System.

Summing up, if we take the best one from **LCm** and **LCms** in each case, we obtain in average relative speedups of 9.97%, 5.03%, 2.54% and 0.43% for 4, 8, 16, and 32 processors respectively, where **Ts** yields 9.25%, 4%, 0.17% and -5.22%.

The implementation cost of **LCms** is less than 24 B per LC entry [11], totalizing less than 400 B, plus the control logic, and a bit per **L1** line for supporting **Ts**. This represents a little cost regarding the cache sizes we have dealt with in Set 1, and a negligible one considering the (more realistic) sizes used in Set 2. Consequently, we believe it is worth incorporating a **LCms** mechanism, where the sequential tagged predictor could be disconnected by software. In those cases where the sequential prefetcher gets disengaged there will be some unused hardware, but of very low cost.

5. Conclusions

We have analyzed in this paper a number of cost-effective hardware prefetching techniques that can succeed in a bus-based SMP, a platform where prefetching must be applied with care, due to the particular features of the organization. We have used a subset of the SPLASH2 suite as workload, firstly performing a characterization of the memory access patterns followed by `loads`.

The pattern breakdown reveals a practical absence of list and chained index patterns, a dominance of sequential traversals, and meaningful presence of stride accesses. The non recognized fraction of accesses is big enough (for some applications) to indicate that further research is needed on new cost-effective pattern recognizers. Although there were sparse comments in the literature, we have observed for the first time the persistence of the patterns when ranging from 1 to 32 processors, although with some exceptions in applications with a high communication-to-computation ratio. This implies that the dynamics of `loads` scarcely varies as the problem is spawned on more processors. A similar conclusion arises when studying the sequence lengths of memory accesses that follow a pattern: they concentrate on one or at most two values, which also hold when varying the number of processors.

These results give an idea of what can be expected from the use of prefetching. Sequential tagged and LC-based prefetching should improve performance in programs with long lengths of sequence and significant presence of sequential or stride patterns, scaling performance as the number of processors increase. Non-sequential predictors like those proposed in [4, 20] could be useful in *particle* and *irregular* programs, but they need a considerable amount of hardware investment in their current forms —more research is needed in this sense— and they would provide little benefit when sequential or stride patterns dominate.

We have tested four cost-effective hardware prefetching approaches against a system with no prefetch, varying the number of processors from 1 to 32. Two experimental sets have been arranged for a two-level cache system: Set 1 for modeling a system where cache memories are highly pressed; Set 2 for modeling a system where cache sizes approach those in current configurations. Memory access is not ideal because we model a split-transaction bus working with an 8-way interleaved memory.

Relative results among the different prefetchers do not vary significantly neither with the number of processors nor with regard to the cache sizes (Set 1 or Set 2). LCms appears to be a suitable

prefetcher, even though it degrades performance in some cases where bus utilization is near saturation and any moderate increase in the traffic negatively affects global performance.

Observing results from LCm and LCms together, we suggest considering an LCms mechanism where sequential prefetching can be activated or disconnected by software when compiling or by the programmer. Taking the best from the two prefetchers, we obtain in average relative speedups of 9.97%, 5.03%, 2.54% and 0.43% for 4, 8, 16, and 32 processors respectively, at a truly low cost, considering a bus with a reasonable bandwidth for a modern modest-sized bus-based multiprocessor, where a traditional sequential prefetcher yields 9.25%, 4%, 0.17% and -5.22%.

6. References

[1] B. R. Allison, C. Van Ingen. “Technical Description of the DEC 7000 and DEC 10000 AXP Family”. Digital Technical Journal Vol. 4, No. 4 Special Issue 1992: 1-11.

[2] J.L. Baer and T.F. Chen. “An Effective On-chip Preloading Scheme to Reduce Data Access Penalty”. In Supercomputing 91, 1991: 176-186

[3] T.F. Chen and J.-L. Baer, “A Performance study of Software and hardware Data Prefetching Schemes”, Proc. 21st ISCA, 1994: 223-232.

[4] C.-H. Chi, C.-M. Cheung. “Hardware Prefetching for Pointer Data References”. Procs. of the ICS98, Melbourne, Australia, 1998: 377-384.

[5] F. Dahlgren, M. Dubois and P. Stenström, “Sequential Hardware Prefetching in Shared-Memory Multiprocessors”, IEEE Trans. on Parallel and Distributed Systems, July 1995: 733-746.

[6] F. Dahlgren and P. Stenström, “Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors”. IEEE Trans. on Parallel and Distributed Systems (7) 4, April 1996: 385-398.

[7] F. Dahlgren and P. Stenström, “Performance Evaluation and Cost Analysis of Cache Protocol Extensions for Shared-Memory Multiprocessors”. IEEE Trans. on Computers (47) 10, Oct. 1998.

[8] Digital Equipment Corporation. *Alpha Architecture Handbook. Version 3*. Maynard, Massachusetts. Oct. 1996.

[9] J.W.C. Fu, J.H. Patel and B. L. Janssens. “Stride Directed Prefetching in Scalar Processors”. Proc. MICRO-25, Dec. 1992: 102-110.

[10] E. Gornish, E. Granston, A. Veidenbaum. “Compiler-directed data prefetching in multiprocessors with memory hierarchies”. Proc. ICS-90, 1990: 354-368.

[11] P.Ibáñez, V. Viñals, J.L. Briz and M.J. Garzarán. “Characterization and Improvement of Load/Store Cache-Based Prefetching”. Proc. ICS-98, Jul. 1998: 369-376.

[12] Y. Jegou and O. Temam. “Speculative Prefetching”. Proc. ICS-93, Dec. 1992: 1-11.

[13] D. Joseph and D. Grundwald. “Prefetching using Markov Predictors”. IEEE Trans. on Computers (48) 2, Feb. 1999.

[14] D.A. Koufaty, X. Chen, D.K. Poulsen and J. Torrellas. “Data Forwarding in Scalable Shared-Memory Multiprocessors”. IEEE Trans. on Parallel and Distributed Systems (7) 12, Dec. 1996: 1250-1264.

[15] S. Mehrotra and L. Harrison. “Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs”. Proc. ICS-96, 1996: 133-140.

[16] T. Mowry and A. Gupta. “Tolerating Latency through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors”. In Jour. of Parallel and Distributed Computing (12) 2, 1991: 87-106.

[17] V. S. Pai, P. Tanganathan and S. V. Adve. “The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology”. Proc. 3rd. HPCA, Feb. 1997: 72-83.

[18] M. Papamarcos y J. Patel. “A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories”. Proc. 11th ISCA, 1984: 348-354.

[19] L. Ramos, P. Ibáñez, V. Viñals and J.M. Llabería. “Modeling Load Address Behaviour Through Recurrences”. 2000 IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS 2000), April 2000:101-108.

[20] A. Roth, A. Moshovos, G.S. Sohi. “Dependence Based Prefetching for Linked Data Structures”. ASPLOS-VIII, Oct. 3- 7. San Jose, California, 1998: 115-126.

[21] A.J. Smith. “Sequential Program Prefetching in Memory Hierarchies”. IEEE Computer (11) 12, Dec. 1978: 7-21.

[22] D.M. Tullsen and S.J. Eggers. “Effective Cache Prefetching on Bus-Based Multiprocessors”. ACM Trans. on Computer Systems (13) 1, Feb. 1995: 57-88.

[23] D.M. Tullsen and S.J. Eggers. “Limitations of Cache Prefetching on a Bus-Based Multiprocessor”. Proc. 20th ISCA, NewYork, 1993: 278-288.

[24] J. Veenstra y R. Fowler. “MINT : A front end for efficient simulation of shared-memory multiprocessors”. Proc. 2nd Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecomm. Systems, 1994: 201-207.

[25] J.E. Veenstra and R.J. Fowler. “The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors”. Tech. Rep. 490. Computer Science Departament. Univ. of Rochester. Rochester, New York 14627.

[26] S. Woo, M. Ohara, E. Torrie, J. Singh, y A. Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. Proc. 22nd ISCA, 1995 : 24-36.

[27] Z. Zhang and J. Torrellas. “Speeding up Irregular Applications in Shared-Memory Multiprocessors: memory Binding and Group Prefetching”. Proc. 22nd ISCA, 1995: 188-199.