Characterization and Improvement of Load/Store Cache-based Prefetching*

Pablo Ibáñez, Víctor Viñals, José L. Briz and María J. Garzarán
Depto. Informática e Ing. de Sistemas, Univ. Zaragoza
c/ Ma. de Luna, 3 - 50015 ZARAGOZA (SPAIN)
{imarin,victor,briz,garzaran}@posta.unizar.es

ABSTRACT

A common mechanism to perform hardware-based prefetching for regular accesses to arrays and chained lists is based on a Load/Store cache (LSC). An LSC associates the address of a `ld/st` instruction with its individual behavior at every entry. We show that the implementation cost of the LSC is rather high, and that using it is inefficient. We aim to decrease the cost of the LSC but not its performance. This may be done preventing useless instructions from being stored in the LSC. We propose eliminating those instructions that never miss, and those that follow a sequential pattern. This may be carried out by inserting a `ld/st` instruction in the LSC whenever it misses in the data cache (on-miss insertion), and issuing sequential prefetching simultaneously. After having analyzed the performance of this proposal through a cycle-by-cycle simulation over a set of 25 benchmarks selected from SPEC95, SPEC92 and Perfect Club, we conclude that an LSC of only 8 entries, which combines on-miss insertion and sequential prefetching, performs better than a conventional LSC of 512 entries. We think that the low cost of the proposal makes it worth being taken into account for the development of future microprocessors.

1. INTRODUCTION

Every effort to decrease cycle time or increase Instruction Level Parallelism when designing a high performance microprocessor, may be neutralized by a slow memory subsystem [2, 3]. A large number of techniques have been developed to minimize the latency impact of a data reference, reducing either the number of misses (e.g. hardware and software based prefetching) or their cost (e.g. non-blocking caches, multithreading, decoupling).

Software based data prefetching techniques use special PREFETCH instructions to bring a block into the cache memory in advance. Most contemporary architectures include this instruction, and many compilers can transform simple loops in order to decrease their miss ratio substantially. However, these software techniques sometimes perform poorly with respect to hardware prefetching [4, 23].

Moreover, hardware based prefetching does not need to recompile, and does not increase code size. A typical approach consists in prefetching one or more consecutive blocks in a sequential way [21], and it is frequently used on the instruction stream. In recent years (91-97), alternative approaches have been proposed to predict non-sequential accesses or to issue prefetching at the right time. Nevertheless, most of the current microprocessors do not implement hardware prefetching. An exception to this is the sequential tagged prefetching mechanism between the HP-PA 7200 processor and its external cache memory.

In this paper we show a way of improving hardware-based prefetching in regular accesses to arrays and chained lists. Particularly, we focus on the Load/Store Cache (LSC), a mechanism driven by the address stream generated by each `ld/st` instruction. The LSC associates the PC-address of a `ld/st` instruction with its individual behavior. Every time a `ld/st` instruction that is not present in the LSC is executed (*LSC miss*), it is inserted into the LSC. Once a pattern has been recognized and the instruction executes again, a prefetch address will be computed and issued.

Previous work proposes a direct-mapped LSC with a number of entries between 128 and 512 [1, 10, 16, 17]. It will be shown later that the cost of a LSC with 512 entries is similar to that of a double-ported cache memory with a size between 6 and 12 KB. This may decline the use of the LSC in favor of more important resources.

Our purpose is to improve the use of the LSC entries in order to decrease their number. We want to achieve similar (or even better) performance while reducing the cost.

As far as we know, no previous reference considers both, the LSC performance [10, 16] and the behavior of the `ld/st` instructions regarding their addressing patterns [8, 17], and the workloads they consider are not very extensive. Our first contribution goes in that direction, and is based on the analysis of 25 programs (taken from SPEC92, SPEC95 and Perfect Club). From this characterization we can advance the following results: a) the miss ratio for a direct-mapped LSC with 512 entries is too high for many applications, b) most of the instructions follow scalar or sequential patterns, and c) most of the `ld/st` instructions hardly miss.

In this paper we propose preventing some instructions from being stored in the LSC in order to reduce its number of entries. By considering the former results, we identify and discard instructions that: a) induce the prefetch of blocks that are already in the cache, or b) show a sequential addressing pattern. This can be done by combining two strategies:

- Storing `ld/st` instructions in the LSC only when they miss in the data cache (*on-miss insertion*)

- Performing sequential prefetching in parallel. In particular, we use One-Block-Lookahead sequential tagged prefetching (OBLst) [21].

To understand the key aspects of prefetching, we use a performance model that takes into account the cache lookup pressure, the data CPI, the shift from CPU misses to correctly prefetched misses, and other useful characterizations of the prefetching dynamics. Through this model, we compare our approach with the conventional LSC by means of a detailed cycle-level simulation, varying parameters

ICS 98 Melbourne Australia
Copyright ACM 1998 0-89791-998-x/98/ 7...$5.00

*This work has been partially supported by the CICYT of Spain under the grants TIC98-0511-C02 and TIC96-1127.

such as LSC and data cache sizes. A noticeable point is that performance holds when the number of LSC entries are reduced: we show that an LSC of only 8 entries, managed by on-miss insertion and combined with sequential prefetching, performs better than a conventional LSC of 512 entries.

This paper is organized as follows. Section 2 reviews related works. Section 3 presents the workload, the methodology we have followed and the characterization of the `ld/st` stream behavior relevant to the LSC operation. Section 4 introduces the insertion-miss policy and the combination with sequential data prefetching in detail. Sections 5 and 6 describe the performance model, give experimental results and discuss implementation costs. Finally, Section 7 summarizes the main points of this contribution.

2. PREVIOUS WORK

Non-sequential data prefetching was firstly introduced by Baer and Chen [1] under the term Preloading. The LSC used in that work was called Reference Prediction Table (RPT). An RPT is organized as a cache indexed by a Look Ahead Program Counter (LA-PC), whose value is based on some branch prediction policy. LA-PC varies from the current PC value to a limit imposed by a constant named prefetch distance which is proportional to the latency of the following level. If LA-PC hits in the RPT, and the state of the selected entry indicates a stride pattern, the addition of the latest address issued by the corresponding `ld/st` plus the stride is prefetched. Prefetched blocks are directly added to the data cache.

Stride-directed prefetching [10] and speculative prefetching [16] use a similar table, but now indexed by the PC. Every time a `ld/st` instruction is executed, the table is searched. If the instruction is found in the table, the corresponding prefetched block will be used in the next iteration. Under stride-directed prefetching, target blocks are loaded directly into the data cache. Under speculative prefetching, the prefetched blocks are loaded into a small separate cache.

Finally, [17] suggests an addition to the previous approaches in order to detect a linear traversal of a chained list made up of records. It is assumed that each record has a next_address field that points to the next record. This mechanism detects the ld instruction that reads the next_address field, and uses that value plus/minus a constant as the prefetch address. This paper also considers the pattern that appears when traversing sparse arrays whose non-zero elements are chained by an index (e.g. spice).

Besides these papers about prefetching based on the classification of `ld/st` instructions, another big group which directly deals with the global sequence of addresses (or first-level cache misses) can be considered. The approach presented in [11] and used in [8] is based on keeping a list of common strides by calculating the strides between each reference and the previous sixteen references.

In [20] the minimum delta scheme (also used in [9]) and the partition scheme are introduced. The former calculates the stride as the minimum difference between a missed address and the last n missed addresses. The latter splits memory into zones, and calculates the stride between the last two references to the same zone. In [14] Markov Chains are used to prefetch multiple reference predictions from the memory subsystem. These schemes are proposed for off-chip environments, in which the address of the instruction to be included in the LSC is not available.

Finally, some approaches try to issue the prefetch as soon as possible, or even to determine which is the optimal time to do it, supposing very large latencies (e.g. to fill in advance off-chip caches in shared memory multiprocessors). Thus, [15] proposes the use of stream buffers, which issue several requests in sequence instead of issuing a single one and store the requests until they are referenced. The purpose of this policy is to increase the prefetch distance, i.e. to request blocks even earlier. In [20, 9] this idea is also applied to streams with stride accesses. Adaptive sequential prefetching is proposed in [6,7]. In [8] the same idea is extended to non-sequential stride prefetching, and a comparison between the sequential and not sequential cases is carried out. Adaptive prefetching modifies the prefetch distance dynamically, according to the latency of the system and to the loop size. Prefetched blocks are stored in the cache.

There are many papers dealing with software-based prefetching which we do not mention here, for they lay beyond of our scope. We should mention [19], however, because it concludes that most misses are caused by ld instructions with stride and list patterns, and this is a key idea in our work, as we have exposed in Section 1. However, it must be pointed out that [19] considers sequential prefetching as a particular case of stride prefetching. The authors propose a compilation algorithm that reorders the code to eliminate dependencies between the instructions that load values and the instructions that use those values, and analyze 12 benchmarks of SPEC92.

3. PATTERN CHARACTERIZATION AND LSC USAGE

In order to take advantage of the insertion of a `ld/st` into the LSC, the following conditions must hold:

C1. The instruction must remain some time in the LSC before being useful, because we need several executions in order to fix the prefetch condition.

C2. The instruction must access to memory following one of the regular patterns that can be recognized.

C3. The datum that is being accessed by the instruction must not reside in the data cache, since prefetching is not necessary in that case.

Previous work based on the use of an LSC [1, 10, 16, 17] does not consider the optimization of the LSC in these terms, and gives no detailed characterization of the relative importance of the different patterns recognized.

3.1 Workload and tracing methodology

The chosen workload is a set of 25 programs taken from SPEC95 (8 integer and 10 floating point), SPEC92 (spice) and from Perfect Club (6 floating point). This workload has been targeted to a SPARC V7 architecture. The user-mode execution traces have been obtained dynamically by means of Shade, a utility from SUN microsystems [5].

Figure 1 shows the complete evolution of CPI for a single issue SPARC processor executing Spice on a sample system. A transient behavior up to 4,000 million instruction can be observed. From this point on, a steady state appears with low variation in the mean value. The majority of the papers referred to in Section 2 assume a transient state of some tens of million instructions at most, and take a single observation after this point. This may not be representative of the whole program behavior.

In this section, we have simulated 2000 million consecutive instructions beyond the end of the transient state, rounded to a billion instructions. This transient state has been determined by plotting the temporal behavior of each program.

The advantage of this methodology arises when the locality in the transient state is quite different from that in the steady state. From a programmer's point of view, the transient behavior corresponds

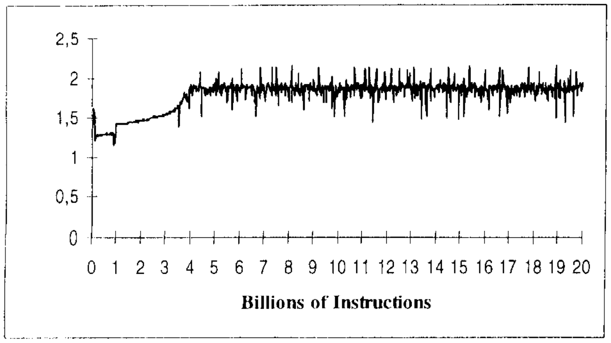

Figure 1. CPI evolution while executing Spice on a sample system.

typically to the initialization of data structures by using very simple -almost sequential- access patterns.

Table 1 shows the following data (from left to right): program input, number of instructions, total number of loads and stores of the full execution, number of instructions of the transient phase and number of instructions skipped until statistics begin to be computed. All the numbers are in billions. Programs labeled as irregular do not have a clear transient phase. To compute the means, we consider the three groups that are differentiated on the table (we include Spice from SPEC 92 inside the SPEC95-fp group).

PERFECT	Input	Instr.	Ld	St	Tran-sient	Skip.
QCD lg	perfect	1.165	0.195	0.113	~0	full ex.
MDG lw	perfect	18.593	5.754	2.207	~0	0
BDNA na	perfect	3.340	1.085	0.535	0.35	1
ARC2D sr	perfect	7.990	3.175	1.291	~0	0
FLO52 tf	perfect	1.746	0.538	0.211	0.45	full ex.
TRFD ti	perfect	2.458	0.926	0.453	~0	0
SPEC95-int						
go	5stone21	30.714	5.886	1.571	irreg	4
gcc	stmt.i	0.545	0.092	0.028	~0	full ex.
compress	in (92)	0.091	0.014	0.007	~0	full ex.
ijpeg	penguin	40.598	5.244	2.013	irreg	2
xlisp	li-input(92)	5.176	1.057	0.462	~0	0
m88ksim	ctl.big	75.885	11.337	5.295	irreg	10
perl	scrabbl.in	17.518	4.511	1.972	0.5	1
vortex	vortex.big	72.662	14.962	5.263	3.2	4
SPEC95-fp						
applu	applu.in	63.222	14.993	4.642	irreg	3
apsi	apsi.in	45.556	10.840	3.632	irreg	8
fpppp	natoms.in	291.203	66.805	24.024	irreg	1
hydro2d	hydro2d.in	58.599	13.896	3.391	0.5	1
mgrid	mgrid.in	91.587	32.074	2.445	irreg	1
spice2g6	greycode.in	21.294	4.826	0.906	4	4
su2cor	su2cor.in	50.458	12.764	3.300	6	6
swim	swim.in	40.570	12.781	3.086	0.2	1
tomcatv	tomcatv.in	50.730	12.709	3.726	4	4
turb3d	turb3d.in	259.940	24.565	12.615	irreg	1
wave5	wave5.in	42.750	11.178	4.585	2	2

Table 1: Dynamic counts, transient intervals and number of skipped instructions (x 10^9).

3.2 LSC misses

We define the miss ratio in the LSC as:

$$m_{LSC} = (\#\text{misses in LSC})/N$$

where N is the number of memory references, and a miss is counted every time the PC-address of a `ld/st` instruction is not found in the LSC directory. This miss ratio is related to the condition **C1**, because the average number of times that a `ld/st` instruction is executed before being replaced is just the inverse of m_{LSC}.

Table 2 presents the average values of m_{LSC} for a direct-mapped LSC with a different number of entries. If we analyze the individual behavior of each program, it can be observed that in order to achieve a value of m_{LSC} less than or equal to 10%, we need at least 1024 entries in 10 out of 25 benchmarks. With m_{LSC}=10%, a `ld/st` instruction remains in the LSC for ten consecutive instances in average. Whenever the instruction is replaced, 3 executions are needed to detect a pattern before triggering the prefetch. Therefore, the ratio (#prefetches/ #references) is 70%. With a miss ratio of 25%, that ratio drops to 25%.

	# entries									
	16	32	64	128	256	512	1K	2K	4K	8K
PERFECT	79.7	63.8	45.4	32.6	23.8	16.6	7.4	2.7	1.6	0.4
SPEC95-int	64.3	50.3	39.2	27.1	17.8	10.9	6.4	3.5	1.6	0.6
SPEC95-fp	62.7	43.0	23.8	16.6	14.1	11.5	8.8	6.7	4.1	1.8

Table 2: m_{LSC}, average miss ratio of a direct-mapped LSC in %

Even though we do not know the relation between m_{LSC} and the reduction of the effective access time due to prefetching, those data suggest a higher number of entries with reference to previous papers [1, 10, 16, 17].

3.3 Pattern Distribution

Even supposing an ideal behavior of the LSC (m_{LSC} = 0), some pattern is needed to trigger prefetching (condition **C2**). We have looked in our workload for five patterns that can be recognized by hardware techniques. This has been done by tracking the following equalities for each `ld/st` instruction:

SCAlar:	$A_i = A_{i-1}$	
SEQuential:	$A_i = A_{i-1} + s$	$0 < s \leq \text{Bsize}$
STRide:	$A_i = A_{i-1} + S$	$S > \text{Bsize} \parallel S < 0$
PoinTeR list:	$A_i = D_{i-1} + d$	d is a record displacement
INDex list[1]:	$A_i = 4{*}D_{i-1} + K$	4 = integer index size
		K= Base Address of the Index Array

A_i is the address generated by a `ld/st` during its execution *i*, D_i is the value read by a load instruction during its execution *i*, and Bsize is the block size. Stores can only follow the first three patterns.

The groups presented in Table 3 are disjoints. If an instance of a `ld/st` instruction matches several patterns at once, it is firstly classified according to the pattern recognized in the previous instance. If several patterns arise repeatedly, the following priorities are applied: SCAlar, PoinTeR list, INDex list, STRide or SEQuential.

Most of the accesses are scalar or sequential. Few of them follow stride or chained list patterns, and they concentrate over a few benchmarks. The higher percentages in the Remaining column are mainly due to integer programs whose access patterns we do not detect.

[1] An example of the pattern IND is accessing to a non-compressed sparse array by means of another array with the indexes of the non-zero elements; "INDex list" models the reference to the Index Array. The program spice2g6 shows a large percentage of this behavior.

PERFECT	SCA	SEQ	STR	PTR	IND	Rem.
QCD lg	51.67	20.33	1.8	0.05	0.14	26.01
MDG lw	46.18	27.25	3.65	0	0	22.92
BDNA na	69	23.07	5.12	0.02	0.03	2.76
ARC2D sr	20.33	46.62	28.11	0	0.04	4.9
FLO52 tf	14.33	71.38	5.01	0	0	9.28
TRFD ti	4.9	44.38	20.83	0	0	29.89
SPEC95-int						
go	31.62	3.52	1.45	0.08	3.87	59.46
gcc	35.42	14.8	1.45	1.33	1.01	45.99
compress	63.98	11.54	5.24	0.17	0.01	19.06
ijpeg	19.61	49.01	4.02	2.42	0.55	24.39
xlisp	37.62	7.86	5.16	4.08	0.14	45.14
m88ksim	30.49	47.66	0.14	0.07	0.6	21.04
perl	40.8	2.42	1.43	2.04	0.04	53.27
vortex	67.87	4.31	0.33	0.08	0.91	26.5
SPEC95-fp						
applu	30.86	13.83	34.35	0	0	20.96
apsi	39.87	45.71	9.6	0	0.02	4.8
fpppp	98.65	0.25	0.06	0	0	1.04
hydro2d	12.12	87.6	0.05	0	0	0.23
mgrid	18.11	78.5	0.25	0	0	3.14
spice2g6	33.44	2.79	5.73	0.08	21.17	36.79
su2cor	19.73	74.14	0.15	0	2.34	3.64
swim	0.02	99.21	0.08	0.02	0.29	0.38
tomcatv	31.06	66.8	0.17	0.05	0.01	1.91
turb3d	29.67	56.63	6.23	0	0.24	7.23
wave5	21.96	63.03	9.32	0	1.15	4.54

Table 3: Classification of accesses according to their patterns.

A lot of research on the matter reports a high number of stride accesses, because sequential accesses are considered as a particular case of stride accesses. An exception is [8], where sequential and stride patterns are separately studied in programs of the SPLASH-I suite, in a multiprocessor environment. The distributions given in that paper are very similar to those we have found in SPEC95 and Perfect Club.

The frequent SEQuential pattern, when considered as a particular case of the STRide pattern, ensures the utility of any LSC prepared for detecting strides. However, sequential prefetching is simpler (no LSC is required) and cheaper (at most one bit per block).

On the other hand, prefetching SCAlar patterns is useless when LSC is indexed through the PC, because a variable is accessed and prefetched at the same time. However, this may be useful when indexing the LSC through the LA-PC.

The sum of stride and list patterns (STR, PTR and IND columns) is small but noticeable: the average values for Perfect, SPEC95-int and SPEC95-fp are 10.8%, 4.6% and 8.35% respectively. Nevertheless, in some programs most of the accesses follow these patterns (ARC2D, TRFD, applu, spice2g6) and a high benefit from prefetching may be obtained.

3.4 Execution and miss frequencies correlation.

Whatever patterns they follow, it is of no use to keep `ld/st` instructions that never miss in the LSC (condition **C3**).

Up to now we have considered that a `ld/st` instruction is inserted in the LSC when it is executed and misses in the LSC. We call this strategy *always insertion*. Under *always insertion* and in the absence of conflicts, the probability for a `ld/st` instruction of being in the LSC is proportional to its frequency of execution. However, the set of `ld/st` instructions we are keeping in the LSC (the most frequently executed) may not be the most suitable set (the set that would cause more cache misses if prefetching were turned off).

In order to study the correlation between executions and misses we simulate a direct-mapped cache of 8KB and Bsize = 16B with tagged sequential prefetching. For each individual `ld/st` we record the number of executions and misses. Then, we put all the `ld/st` instructions into two lists: the first one ordered by number of executions, and the second one ordered by number of cache misses. Finally, we take the necessary `ld/st` to cover 90% of executions from the top of the first list, and from the second list the necessary `ld/st` to cover 90% of misses. By doing so, we can distribute `ld/st` into one of four disjoint classes:

A) `ld/st` that represent less than 10% of total executions and misses.

B) `ld/st` that represent 90% of executions, but not 90% of misses.

C) `ld/st` that represent 90% of misses, but not 90% of executions.

D) `ld/st` that represent 90% of executions *and* misses.

Table 4 presents the average number of `ld/st` instructions in each class for each workload group.

	A 10%	B 90% Exec.	D 90% Exec. 90% misses	C 90% misses
PERF. CLUB	7271	1125	194	71
SPEC95-int	5366	588	328	121
SPEC95-fp	1407	635	118	21

Table 4: Number of `ld/st` instructions for each Class.

The `ld/st` we want to store in the LSC belong to classes C and D, yet those which are really filling the table belong to classes B and D. It can be observed that B∪D is greater than C∪D by a factor of 5, 2 and 5.4 for Perfect, SPEC95-int and SPEC95-fp respectively.

Instructions belonging to class C have a low probability of being in the LSC. However, they yield many misses and in consequence, it would be convenient to keep them in the LSC. On average, they represent 25% of the set with more misses, C∪D.

On the other hand, `ld/st` instructions of class B have a high probability of being in the LSC. However, they hardly miss in the cache and it is of no use to keep them in the LSC. On average, they account for 78.6% of the most executed set, B∪D.

4. ON-MISS INSERTION PLUS SEQUENTIAL-TAGGED PREFETCHING

From the pattern distribution and the correlation between execution and miss frequencies, we propose a combined prefetching strategy, in which addresses are computed by two independent prefetching mechanisms working in parallel: a) LSC with *on-miss insertion* (LSCmi) prefetching, and b) One Block Lookahead sequential tagged (OBLst) prefetching [21].

Under *on-miss insertion*, a hit in the LSC involves the same actions as under *always insertion* (updating the state, the data field, etc.). But in case of a LSC miss, insertion is performed only when there has been a miss in the data cache too.

This way, the probability of finding a `ld/st` instruction in LSC is proportional to its miss frequency in the data cache, and not to its frequency of execution. If two or more `ld/st` instructions are mapped to the same LSC entry, they do not contend for that entry if they hit in the data cache, increasing the stability of those instructions that miss.

We add OBLst prefetching to prevent the `ld/st` instructions which follow a SEQuential pattern from contending for LSC entries. Moreover, sequential prefetching can determine sequential relations among *different* loads, adding useful prefetching streams which an LSC would not be able to generate.

By way of example, let us consider a chained list of `structs` greater than the block size. The `ld` instruction which reads the pointer field is classified by the corresponding detector. However, `ld` instructions which access the remaining fields do not follow a regular pattern; OBLst prefetching can perform successfully in this case.

In Table 5 we show the replacement ratio for an LSCmi, defined as the number of `ld/st` entries evicted divided by the total number of references. This ratio is comparable with that of Table 2 in the sense that its inverse is the average number of times that a given instruction is executed while remaining in the LSC until it is replaced. It is noticeable the one/two order of magnitude shift between the two tables.

	# entries									
	16	32	64	128	256	512	1K	2K	4K	8K
PERFECT	1,64	1,09	0,61	0,30	0,19	0,09	0,06	0,03	0,01	0,00
SPEC95-int	2,95	2,42	1,91	1,40	0,93	0,59	0,36	0,22	0,13	0,06
SPEC95-fp	9,16	5,48	2,32	0,70	0,55	0,37	0,21	0,11	0,04	0,01

Table 5: Average replacement ratio of a direct-mapped LSCmi.

5. PERFORMANCE ANALYSIS FOR A SINGLE PROCESSOR-MEMORY SYSTEM

5.1 Workload and tracing methodology

The evaluation has been carried out by using the workload described in subsection 3.1. However, we assume now a multiprogramming environment with a quantum of 1 million instructions. We also assume a multiprogramming degree high enough to empty the first-level caches between every two bursts of the same process completely.

Given the high temporal cost of the cycle level simulation and the big number of benchmarks that have been analyzed, it is not possible to proceed with the same number of instructions that was used in Section 3. A limit of 20 million instructions has been fixed. However, to improve the representativeness of sampling, we scatter the 20 quanta (observations) over a certain interval. For each application this interval starts at the end of the transient phase we showed in Table 1, and its size varies according to the benchmark behavior (e.g. a lot of benchmarks follow some periodical behavior; the size of their intervals matches the size of their periods). Similar studies confirm that this kind of sampling is much better than the contiguous selection of the observations [13].

5.2 System model

Figure 2 shows the system modeled. The processor is a single issue in-order SPARC, similar to that used in related papers [8, 12]. We consider a level one (L1) on-chip split cache memory, and a level two (L2) off-chip unified cache. Block size is 32B in both cases. In all experiments, we fix a 32KB direct-mapped L1 instruction cache with OBLst prefetching. L2 is ideal in the sense that it always hits, and has a pipelined interface for L1 block requests of 1:7:2 cycles for address transfer, access and data return, respectively.

The L1 data cache (L1dC) is also direct-mapped, but its size and prefetching capabilities are varied (sizes: 8 KB, 32 KB and 128KB; prefetching: OBLst and/or LSC, with LSC indexed by PC or LA-PC). The LSC detects PoinTeR list, INDex list, STRide and SEQuential patterns by using the policy exposed in subsection 3.3, which is similar to the one used in [17]. Demand fetches and prefetches (low priority) have to contend for a single cache port. Therefore, a high prefetching lookup pressure may degrade the system performance. When sequential and LSC prefetching work in parallel it is possible to issue up to two prefetches per reference, which are temporally held in the lookup buffer.

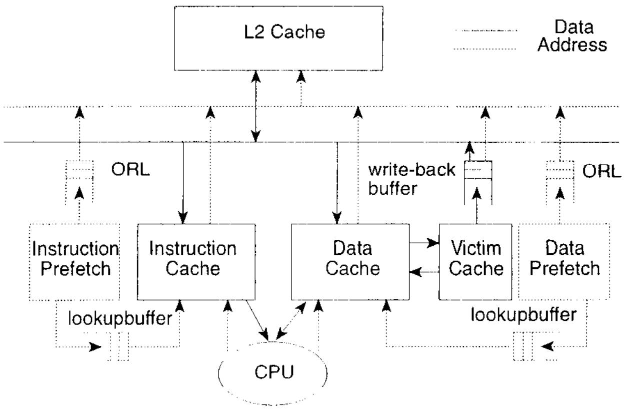

Figure 2. System model. The Data Prefetch box can include LSC-based and OBLst prefetchers.

As in other papers, a 16-entry victim cache is added [22,14]. This way, we focus on the benefits prefetching offers for the elimination of capacity and compulsory misses, and not on its ability for dealing with conflict misses. Some benchmarks experience a large fraction of conflict misses, in particular applu, apsi, su2cor, swim, tomcatv and wave5 from the SPEC95 suite.

ORL (Outstanding Request List) is an address buffer which supports pending prefetches and gives information about the blocks currently being read in L2.

5.3 Performance model

Global measures such as CPI reflect global effects, but they do not capture critical aspects of prefetching. To isolate them, we suggest a model (Figure 3) which is partially based on [22] and which considers the following quantities :

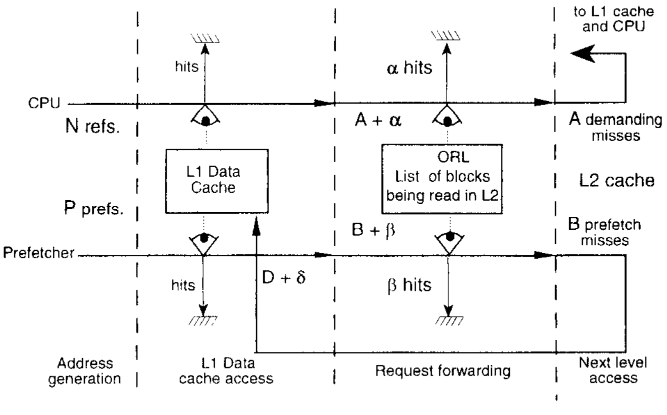

Figure 3. Quantities considered in the performance model.

***N*, *P*:** Total number of CPU data references and issued prefetches. L1dC misses are looked up in the ORL; if they are not found a search is issued on the L2.

α: Number of CPU misses that are currently being serviced in L2. They are not affected by the full miss latency: we term them *fast misses*. ***A***: Number of accesses to L2 triggered by CPU misses in L1dC. We call them *demanding misses*, since they are caused by demand, and attending them is critical.

β: Number of misses caused by the prefetcher currently being serviced in L2. ***B***: Number of accesses to L2 triggered by prefetch misses.

D, δ. Decomposition of prefetched blocks into *useful* and *useless* (i.e. replaced without reference); $B = D + \delta$.

From these quantities we define: a) Number of prefetches per reference (*P/N*); b) Full-latency miss ratio ($m_d = A/N$); c) Partial-latency miss ratio ($m_f = \alpha/N$); d) Conventional miss ratio ($m = (A+\alpha)/N = m_d + m_f$); and e) Prefetch miss ratio ($m_p = B/N$), that can be further split in *useless* and *useful* prefetch miss ratio ($m_{pl} = D/N$ and $m_{pf} = \delta/N$). Note that $m_p + m_d$ is the L2 access ratio. That is, the number of accesses to L2 per reference; this is a measure of the pressure put on L2 by L1dC and the prefetching mechanism toghether.

The generic goal of prefetching is to decrease the conventional miss ratio (m, and especially the fraction m_d) with a minimum pressure over L1 (minimum *P/N*) and a minimum L2 access ratio increment (minimum m_p)[1].

These three aspects must be balanced for each particular system. Thus, obtaining a minimum m can depend on the miss penalty at the following memory level, whereas obtaining a minimum *P/N* can be essential depending on the number of ports in the cache. Finally, obtaining a minimum m_p can be critical if the L1/L2 bandwidth is limited.

Figure 4 displays a representation of the model. The right side of the bar shows the processor activity, while the left side shows the prefetching activity. The ratio *P/N* appears numerically on the left, and the data CPI appears on the right. A good prefetch system should shift a great number of misses (*demanding* and *fast*) from the right side to the left side, increasing neither the total size of the bar (L2 access ratio), nor the number of cache accesses per reference (1 + *P/N*).

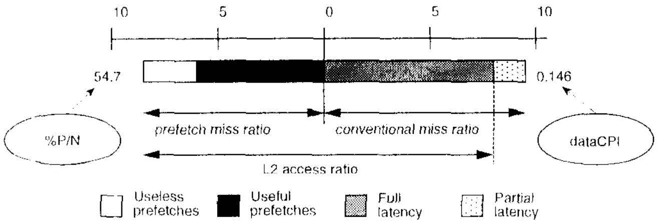

Figure 4. Graphical representation of the performance model.

5.4 Results

The performance of conventional LSC prefetching (LSCconv), LSCmi prefetching, and LSCmi prefetching combined with OBLst (OBLst + LSCmi), is interpreted from our model in Figs. 5, 6 and 7. The L1dC size is set at 32KB and, in each plot, the number of entries of the LSC is from the top towards the bottom: 8, 16, 32, 128 and 512. The two bars at the bottom show the behavior without prefetching and with OBLst prefetching only. In each figure, the average values for SPEC95-fp and SPEC95-int groups are displayed separately. The behavior of the Perfect Club group is fairly similar to that of SPEC95fp. We will omit futher references to this workload due to space limitations.

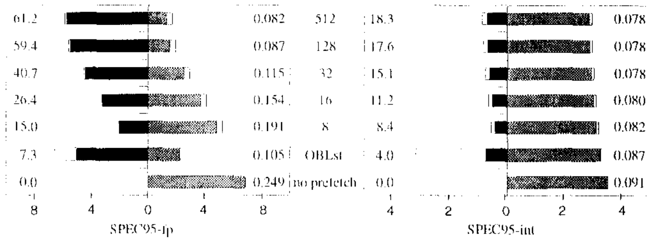

Figure 5. Conventional LSC prefetching (LSCconv).

By observing the three figures as a whole, it is noticeable the difference between floating point and integer applications. SPEC95-int is almost insensitive to the kind of prefetching, and the reduction in m (conventional miss ratio) due to prefetching is very low, varying from 7.1% (OBLst prefetching) to 19.2% (OBLst + LSCmi-512entr.). This result can be better understood if we consider the characterization given in subsection 3.3., since scalar and irregular patterns prevail here. However, SPEC95-fp is quite sensitive to the kind of prefetching and m decreases between 27.5% (OBLst prefetching) and 86.3% (OBLst + LSCmi-512entr).

Another global observation deals with the poor *accuracy* of OBLst prefetching in the integer workload. Defining *accuracy* as $D/(D+\delta)$, we can see the great waste experienced in SPEC95-int: 0.23. In SPEC95-fp the OBLst accuracy raises to 0.77.

It can be observed in Figure 5 that the floating point workload benefits from sequential prefetching, and it is quite sensitive to the size of the LSCconv. Only with a big LSC (512, 128 entries) do we obtain a better data CPI than with OBLst prefetching. The main reduction in m appears in LSCconv-512entr prefetching, which eliminates 81% of misses, whereas OBLst prefetching removes 67.4% of misses.

The lookup pressure performed by LSCconv on L1dC (i.e. *P/N*) is high. For LSCconv-512entr, *P/N* reaches 61.2%, i.e. the number of accesses to L1dC is multiplied by 1.61. On the other hand, OBLst only loads L1dC with 7.3% extra lookup activity.

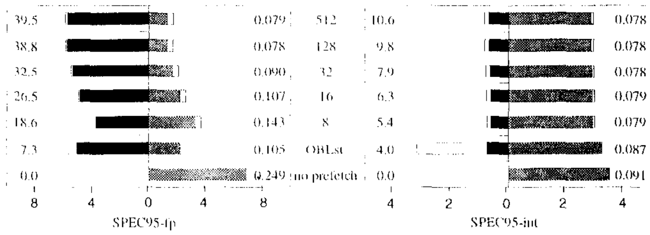

Figure 6. On-miss insertion LSC prefetching (LSCmi).

Let us now analyze separately the performance of LSCmi prefetching (Figure 6).

In SPEC95-fp on-miss insertion significantly decreases the lookup pressure (*P/N*). With an LSCmi of 128-512 entries, 35% of prefetches generated by an LSCconv prefetching are removed. With fewer entries in LSCmi, the lookup pressure increases with respect to LSCconv because in that case LSCconv hardly issue prefetches.

[1] From this model, the terms *coverage* and *accuracy* proposed in [14] could alternatively be quantified as: *coverage* = $B/(A+B)$ and *accuracy* = $D/(D+d)$.

On the other hand, the number of prefetch misses forwarded to L2 (*B*) increases with respect to LSCconv prefetching (0%, 4%, 20%, 52% and 82% for 512, 128, 32, 16 and 8 entries respectively). In the presence of a regular pattern, those prefetch misses are useful, and *m*(LSCmi) < *m*(LSCconv). Only with a 512-entry LSC does *m*(LSCmi) increase 0.8%. For the rest of cases, it always decreases: 9.8%, 27.8%, 37.1% and 28.8% for a LSC with 128, 32, 16 and 8 entries.

With regard to performance, LSCmi prefetching reduces the data CPI of a system without prefetch from 43% (8 entries) to 68% (512 entries). The behavior of LSCconv prefetching is always worse: LSCmi reduces the data CPI of a system with LSCconv from 3.6% (512 entries) to 30.5% (16 entries).

In SPEC95-int we can notice the same tendency, although the differences between LSCconv and LSCmi prefetching are smaller. Moreover, the reduction in data CPI relative to a system without prefetching is smaller (13-14%).

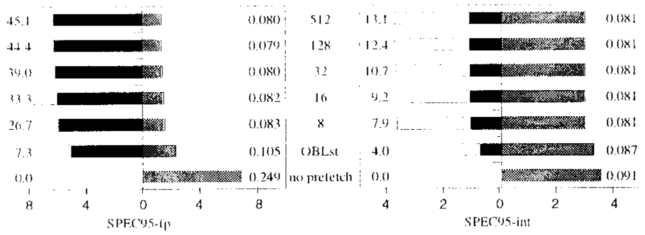

Figure 7. On-miss insertion LSC prefetching combined with OBL sequential-tagged prefetching (LSCmi + OBLst).

In Figure 7 we analyze the performance of the concurrent work of LSCmi and OBLst. In SPEC95-fp the number of lookups in L1dC (*P/N*) increases slightly with relation to LSCmi, but it is always kept below the number required by LSCconv. With an LSC of 512-128 entries, there are about 25% less prefetches than with *insertion always*. On the whole, the number of prefetch misses (*B*) increases largely with respect to a LSCconv system: from 34% (512 entr.) to 269% (8 entr.). The miss ratio decrease is also noticeable: between 21% (512 entr.) and 70% (8 entr.). Data CPI decreases are between 2.4% (512 entr.) and 56% (8 entr.).

In SPEC95-int, results are qualitatively but nor quantitatively similar. The accuracy decrease is noticeable due to the concurrent activity of OBLst prefetching.

On the whole, the most relevant fact is the very low sensitivity of the miss ratio and the data CPI with respect to the size of LSC. When moving from 512 to 8 entries the loss in performance is only 3.75% in data CPI for SPEC95-fp.

As Table 6 shows for SPEC95-fp, LSCmi-8entr + OBLst prefetching achieves better ratios in almost all the metrics with respect to LSCconv-512entr. It reduces the lookup pressure (*P/N* decreases 56.4%), increases the prefetch miss ratio (28.7% more in *B/N*), and so decreases the miss ratio (10.3%). As a negative effect, a loss of accuracy —*D*/(*D*+δ)— appears due to the use of OBLst. In both cases data CPI is almost the same, but with a cost 64 times lower.

	% P/N	% B/N	% m	data-CPI	D/(D+δ)
LSCconv-512	61.2	5.86	1.72	0.082	0.978
LSCmi-8 +OBLst	26.7	7.54	1.54	0.083	0.785

Table 6: SPEC95-fp comparison between LSCconv-512entr and LSCmi-8entr + OBLst.

5.4.1 Results on programs which follow regular patterns

To observe the performance of our proposal when applied to programs with an outstanding presence of stride and list patterns, we have carried out a selection over the whole workload.

Figure 8 shows the means for applu, apsi and wave5 from SPEC95-FP, spice from SPEC92 and arc2d and trfd from Perfect Club. Results for LSCconv appear on the left, and LSCmi + OBLst on the right.

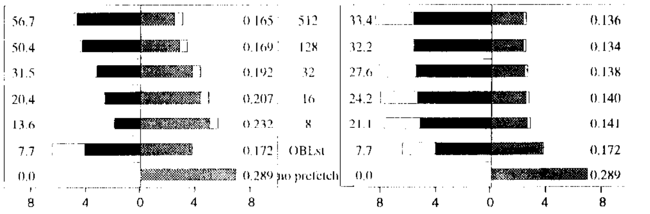

Figure 8. LSCconv (left) vs. LSCmi + OBLst (right), for applu, apsi, wave5, spice (SPEC), arc2d and trfd (Perfect).

It can be observed in Table 7 that for these benchmarks, an LSCmi-8entr.+OBLst performs better than an LSCconv-512entr. with regard to data-CPI, which decreases 14.5%.

	% P/N	% B/N	% m	data-CPI	D/(D+δ)
LSCconv-512	56.7	4.90	3.05	0.165	0.956
LSCmi-8 +OBLst	21.1	7.80	2.80	0.141	0.665

Table 7: LSCconv-512entr vs. LSCmi-8entr + OBLst for applu, apsi, wave5, spice (SPEC), arc2d and trfd (Perfect).

Although the results discussed here are based on a 32KB L1dC, we have simulated other cache sizes too (8KB and 128KB). The same conclusions are valid for these sizes, yet the advantages of our method increase with the cache size. Eventually, we issue prefetches by using an LA-PC as in [1], and with a prefetch distance equal to 1.5 times the memory latency. In this case the lookup pressure (*P/N*) increases, because the method tries to prefetch SCAlars too. Therefore, the advantage of our method is greater since most of these scalar `ld/st` hit in L1dC and are not inserted in the LSCmi.

6. COST ANALYSIS

Executing a `ld/st` instruction requires at least two accesses to the LSC. The first access reads information about that instruction which will be used later for detecting the pattern and for calculating the new state. The second access writes the updated information after the instruction has been executed. With a simple pipeline, the reading could be performed during the ALU stage, after the `ld/st` instruction has been decoded. The computations for pattern recognition and for generating the new state could be carried out during memory access. Writing should be done in the final stage.

Such a simple implementation requires a writing port and a reading port in the LSC. If prefetches are driven by LA-PC, a third additional reading port is needed in orther to check if the instruction addressed by LA-PC is a `ld/st`, and if it matches some pattern.

Every entry in the LSC contains a variable number of fields, depending on the patterns that we want to recognize. If we intend to detect strides only, four fields are required: *PC*, *Ai*, *Si* and *state*. 32-bit addressing yields 12 bytes per entry (*state* needs only a few bits). If we intend to detect accesses to lists chained by address and index, we should add three more fields (*Di*, di and *Ki*), and every entry would take 24 bytes.

To sum up, a direct-mapped LSC with 512 entries stores between 6KB and 12KB, uses a decoder with 512 entries (similar to that of a direct-mapped cache of 8KB with blocks of 16B), and requires two ports at least, since it must be tested and updated in a single cycle. Therefore, an LSC is comparable in size to a first-level data cache. Replacing such an expensive 512-entry LSCconv by a 8-entry LSCmi + OBLst divides its storage costs by a factor of 64. It is difficult to apply an area model in order to take into account the fixed cost of the control unit due to the great inaccuracy of such models when computing area for very small caches [18].

7. CONCLUSIONS

In this paper we have analyzed the performance of a load/store cache as a base for different proposals of hardware-based data prefetching with patterns other than the sequential one. We have found that in order to perform better than with sequential prefetching, it is necessary to provide as much storage area as for a first-level data cache. This is due to two key facts: a) regular patterns different from the sequential pattern are uncommon; and b) most of the instructions that occupy the LSC entries do not miss (i.e. the involved prefetching is useless).

Decreasing the cost of the LSC with no efficiency loss, implies that useless instructions must be removed from the LSC. To do that we propose applying *on-miss insertion* in the LSC (LSCmi) working in parallel with tagged sequential prefetching.

On-miss insertion introduces new instructions in the LSC only if they miss in the data cache. This way, instructions that can take profit from prefetching will be more likely included in the LSC. On the other hand, sequential prefetching reinforces this point because it prevents the `ld/st` instructions which follow this pattern from contending for the LSC entries.

For numerical workloads (SPEC95-fp and Perfect Club) our proposal achieves a great increment in performance for every cache size, specially for the small ones. We believe that the relevant point here is that the performance of an LSCmi decreases only 3.75% in terms of data-CPI when the number of entries decreases from 512 to 8. An LSCmi with 8 entries working in combination with an OBLst prefetching achieves a performance comparable to that of a conventional LSC with 512 entries (with a storage cost 64 times lower).

For non-numerical workloads (SPEC95-int) the performance of an LSC is rather limited because of the absence of recognizable regular patterns. In this context, it makes little sense improving its management. Anyway, since our method reduces the number of entries strongly, it is possible to increase the number of fields of each entry (for detecting new patterns) with little cost.

REFERENCES

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

[2] K. Bolland and A. Dollas. "Predicting and Precluding Problems with Memory Latency". IEEE Micro, vol. 14, no. 4, Aug. 1994, pp.59-67.

[3] D. Burger, J.R. Goodman and A. Kägi. "Memory Bandwidth Limitations of Future Microprocessors". In Proc. of 23th Int. Symp. on Computer Architecture, pp.78-89, May 1996.

[4] T.F. Chen and J.-L. Baer, "A Performance study of Software and hardware Data Prefetching Schemes", Proc. 21st Int. Symp. Computer Architecture, 1994, pp. 223-232.

[5] B. Cmelik and D. Keppel,"Shade: A Fast Instruction-Set Simulator for Execution Profiling". Proc. of ACM SIGMETRICS, May 1994, pp.128-137.

[6] F. Dahlgren, M. Dubois and P. Stenström, "Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors", Proc. 1993 Int. Conf. Parallel Processing, CRC Press, Boca Ratón, Fla., 1993, pp. 156-163.

[7] F. Dahlgren, M. Dubois and P. Stenström, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors", IEEE Trans. Parallel and Distributed Systems, July 1995, pp. 733-746.

[8] F. Dahlgren and P. Stenström, "Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors", Proc. first IEEE Symp. High-Performance Computer Architecture, 1995, pp. 68-77.

[9] K. Farkas, N. Jouppi and P. Chow, "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors", Proc. first IEEE Symp. High-Performance Computer Architecture, 1995, pp. 78-89.

[10] J.W.C. Fu, J.H. Patel and B. L. Janssens. "Stride Directed Prefetching in Scalar Processors". In Proc. of 25th Int. Symp. on Microarchitecture (MICRO-25), ACM, pp.102-110, December 1992.

[11] E. Hagersten. "Towards Scalable Cache Only Memory Architectures", PhD thesis, Swedish Inst. of Comp. Science, Oct. 1992.

[12] P. Ibáñez and V. Viñals. "Performance Assessment of Contents Management in Multilevel on-chip Caches". In Proc. of the 22nd Euromicro Conf. pp: 431-440, Sept. 1996.

[13] L. Jimeno, P. Ibáñez and V. Viñals. "Warm Time-sampling: Fast and Accurate Cycle-level Simulation of Cache Memory". In Proc. of the 22nd Euromicro Conf. Short Contrib. pp: 39-44, Sept. 1996.

[14] D. Josephand D. Grunwald, "Prefetching Using Markov Predictors", Proc. of 24th Int. Symp. Computer Architecture, pp.252-263, June 1997.

[15] N.P. Jouppi. "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers". In Proc. of 17th Int. Symp. on Comp. Architecture, pp.364-373, May 1990.

[16] Y. Jegou and O. Temam. "Speculative Prefetching". Proc. of ICS-93, pp.1-11, Dec. 1992.

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

[18] J. M. Mulder, N. T. Quach and M. J. Flynn. "An Area Model for On-Chip Memories and its Application". IEEE Jour. of Solid State Circuits 26 (2) Feb. 1991, pp. 98-106.

[19] T. Ozawa Y. Kimura and S. Nishizaki, "Cache miss heuristics and preloading techniques for general-purpose programs". Proc. 28th Int. Symp. Microarchitecture, 1995, pp. 243-248.

[20] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as secondary cache replacement", Proc. of 21th Int. Symp. Computer Architecture, April 1994, pp.24-33.

[21] A.J. Smith. "Cache Memories". Computing Surveys, 14(3):473-530, Sept. 1982.

[22] D.M. Tullsen and S.J. Eggers,"Effective Cache Prefetching on Bus-Based Multiprocessors". ACM Transactions on Computer Systems, Vol. 13, No. 1, February 1995, pp. 57-88.

[23] S. VanderWiel and D.J. Lilja. "When Caches Aren´t Enough: data prefetching techniques". IEEE Computer, Vol. 30, No. 7, july 1997, pp. 23-30.