A victim cache for vector registers

Roger Espasa Mateo Valero
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya–Barcelona
e-mail: {roger,mateo}@ac.upc.es
http://www.ac.upc.es/hpc

Abstract

In this paper we introduce the concept of a victim cache for the register level of a vector processor. This small cache holds full vectors "evicted" from the register level through store instructions. The goal of this cache is to filter out the abundant spill code present in vector programs. Using a trace driven simulation technique we compare the performance of a conventional vector implementation with and without the addition of the victim cache proposed. We study a write-through victim cache that eliminates around a 5–20% of load spill traffic and provides speedups in the range 1.06–1.20. In order to allow elimination of store spill traffic, we also present a write-back victim cache that increases the amount of traffic eliminated up to the 10–50% range. Finally, we study the behavior of the victim cache under different memory latencies and show that the victim allows a good tolerance of large memory latencies and can provide speedups as large as 1.24.

1 Introduction

Spill code is a serious problem on architectures that have a limited number of registers at the ISA level. The small number of logical registers available to the programmer or compiler induce extra traffic between the processor and the next memory level in four different ways. First, when the compiler runs out of registers it must insert store and load instructions to spill and reload a register to the stack. Second, a register may be loaded from memory, but the data can not be saved for long because the register is needed for some other instruction. When the memory data is needed again, it must be re-loaded. Third, variables that are reused between close iterations of a given loop are difficult to keep in the register file due to the high register pressure. Optimizations such as scalar replacement [1] are hardly useful when the compiler has so few logical registers to use. Fourth, call-return sequences might involve saving and restoring of vector registers.

This work was supported by the Ministry of Education of Spain under contract 0429/95 and by the CEPBA (European Center for Parallelism of Barcelona).

In the context of traditional vector architectures, it is typical to find only a few vector registers available to the programmer. The successive Cray machines [12, 6] and mini-supercomputers such as Convex C3-Series [2] and Alliant [11] only have 8 logical vector registers. In this type of machines spill code can be very detrimental to performance due to two main reasons: first, when a vector register is spilled it involves a very large amount of data movement and, second, the distance, in terms of processor cycles, between a vector register and main memory can be rather large. Moreover, vector processors are generally found in multiprocessor systems. On such types of machines, an expensive interconnection network between the processors and a highly interleaved main memory system is needed. The more traffic the network has to support, the more likely it is that network degradation occurs. Every bit of traffic added to the memory system due to spill code problems adds to the average memory latency seen by each individual vector processor.

The obvious solution to this register limitation is to add more physical and logical registers to a given machine. Alas, such solution is not always feasible since binary compatibility is often more important than a performance improvement. Also, instruction sets that were not designed with an extension of the vector register file in mind might have opcode space problems to specify the extra registers. Even if the ISA extension is possible, old dusty-deck codes might require a recompilation step to benefit from these extra registers, and this recompilation might not always be possible.

In this paper we present a general technique to dynamically eliminate the excess traffic caused directly or indirectly by the limited number of vector registers available. Our goal is to present a solution that is fully binary compatible with existing vector instruction sets. Thus, our technique is fully transparent to the end user and does not require recompilation nor the addition of special opcodes to the ISA. The proposal is based on the concept of a victim cache [9] next to the vector register file.

As it will be seen, the technique is not only useful for reducing spill code traffic, but it can also detect and eliminate normal, non-spill traffic. Some vectorizable codes do have temporal locality between stores and loads that are close enough (maybe just one iteration away from each other) to allow hits in the victim cache.

Program	Suite	Mem Ops	Vect Ops	Scalar Insns	% Vect	avg. VL
swm256	Spec	4499	5036	4	99.9	127
hydro2d	Spec	1785	2203	23	99.0	101
arc2d	Perf.	1959	2157	30	98.5	95
flo52	Perf.	706	551	18	97.1	54
nasa7	Spec	1795	2190	64	96.2	58
su2cor	Spec	1561	1862	66	95.7	125
tomcatv	Spec	527	403	92	87.9	127
bdna	Perf.	795	889	128	86.9	81
trfd	Perf.	826	438	156	75.7	22
dyfesm	Perf.	502	298	108	74.7	21

Table 1: Basic operation counts for the Perfect Club and Specfp92 programs (Columns 3–5 are in millions).

Program	Total Traffic	Load Spill %	Store Spill %	Total Spill %	Avg. reuse
swm256	4498.5	7.0	7.0	14.0	1.0
hydro2d	1770.7	1.2	1.2	2.4	1.0
arc2d	1929.9	6.3	4.5	10.8	1.4
flo52	691.4	5.9	5.9	11.9	1.0
nasa7	1721.6	1.2	1.2	2.4	1.0
su2cor	1494.8	13.5	6.9	20.3	2.0
tomcatv	513.5	20.3	20.3	40.5	1.0
bdna	700.6	38.0	31.5	69.5	1.2
trfd	657.3	0.0	0.0	0.0	–
dyfesm	397.9	0.1	0.1	0.2	1.0

Table 2: percentage of spill code in vector memory operations. Column 2 is in millions of operations.

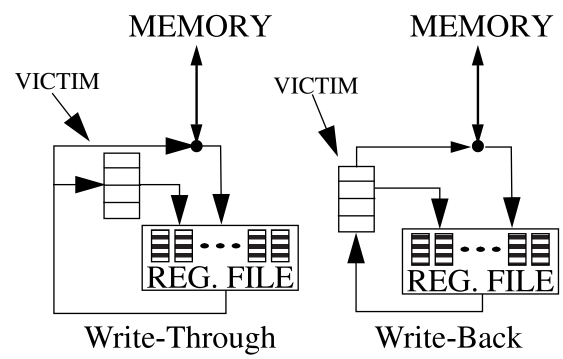

Figure 1: Write-Through and Write-back architectures.

2 Spill Code in Vector Programs

2.1 Benchmarks and Platform

The platform chosen for this study is a Convex C3400 having 8 vector registers, each holding 128 64-bit elements [2]. The Convex C3400 consists of a scalar unit and an independent vector unit. The scalar unit issues a maximum of one instruction per cycle. The vector unit consists of two computation units and one memory accessing unit. All units are fully pipelined. The eight vector registers are connected to the functional units through a restricted crossbar. Pairs of vector registers are grouped in a register bank and share two read ports and one write port that links them to the functional units. The compiler is responsible for scheduling vector instructions and allocating vector registers so that no port conflicts arise. The reference machine implements vector chaining from functional units to other functional units and to the store unit. It does not chain memory loads to functional units, however.

Regarding our benchmarks, we have selected a vectorizable subset of the Perfect Club and Specfp92 programs. We compiled all programs from these suites on the C3400 and using Dixie [3] a detailed trace was collected. From these traces, the percentage of vectorization was computed and we selected the ten programs that had the highest degree of vectorization (over 70%). Table 1 presents some statistics for the selected Perfect Club and Specfp92 programs.

2.2 Spill Code Traffic

To estimate the performance obtainable through dynamic memory traffic elimination, we measured the amount of spill code present in our benchmarks. Table 2 presents the results. As it can be seen, there is a disparity in the amount of spill code found in each program, ranging from almost none in `dyfesm` and `trfd`, up to a 70% in `bdna`. Nonetheless, the potential improvements in total traffic that could be achieved by the victim cache are quite high.

We have split the percentages of spill code belonging to loads and stores (columns 2 and 3) and also give the combined percentage (column 4). Column 1, total vector and scalar memory traffic is in millions of operations. Finally, we have also computed the average reuse of each vector spilled to memory. By dividing the amount of load spills by the amount of store spills, we get the average number of times that a vector spilled to the stack is later reloaded. The result is presented in column 5. For almost all programs the reuse is almost non-existent, that is, a vector is spilled and reloaded only once.

3 Victim Cache for Registers

Our proposal for eliminating excess traffic due to spill is based on the traditional concept of adding a small cache inside the processor that holds vector data (see figure 1). This small cache *only* holds store traffic and this store data is kept in the cache for as long as possible. Vector loads check if the data they are loading is in the cache. If so, they read it from there instead of accessing main memory.

It is very important to remark that load data is never kept in the cache. Each slot in the cache will be able to hold a full 128-element vector and, hence, the total size in slots of the cache will necessarily be small (on the order of 4-16 slots). If load data were to be kept in the cache it would rapidly overflow its contents and negate any of the spill code elimination effects that we are seeking. This is the main difference of this proposal with respect to previous cache proposal for vector processors, such as [10, 13, 8, 7].

We term this cache as a victim cache because it only keeps data that has been "evicted" from the register file through a store instruction. In this sense, our cache follows the same spirit as the victim caches proposed and studied in [9]. The main differences with other approaches is that our victim cache is right next to the register level, instead of next to a first-level cache.

Depending on the write policy of the cache, either write-back or write-through, we have slightly different connectivity patterns between the register file, victim cache and main memory. In the write-through case, the path for writing into the victim is shared with the path for writing into main memory. A new path for reading from the victim into the vector register file has been added. Hence, both a read from the cache and a read from memory can proceed simultaneously, or, alternatively, a write into the victim (and main memory simultaneously) and a read from the victim can proceed in parallel.

For the write-back cache, the peak parallelism is higher. The paths for writing into the victim and the path for writing into main memory are split, so we could have up to three memory operations proceeding in parallel: a store that does not imply an eviction from the victim cache, a load from the victim (without conflicts with the store) and a load from main memory (ditto). In the much more frequent case that a store operation forces the eviction of some slot of the cache, the parallelism drops to 2: the store (plus the eviction) and a load from the victim cache.

Each slot in the victim cache holds a full vector. Therefore, the tag associated with each slot in the cache can not be a simple memory address. Instead, we use the concept of a *region of addresses* to describe the contents of each slot. Each tag is a 6-tuple: $\langle @_1, @_2, \mathtt{vl}, \mathtt{vs}, \mathtt{sz}, \mathtt{v} \rangle$. Virtual addresses $@_1$ and $@_2$ define a consecutive region of bytes in memory and vl, vs, and sz are the vector length, vector stride and access granularity used when the tag was created; v is a validity bit. The address $@_2$ is computed as $@_1 + (vl - 1) \times vs + sz$.

Regarding scalar data, the cache works as follows. In the write-through case, scalar loads simply ignore the cache, while in the write-back case, if a scalar load has a conflict with a slot in the cache, the slot is sent to memory prior to performing the load. Scalar stores are disambiguated against all tags present in the cache. If an overlap between the region of addresses defined by the scalar store and any of the region of addresses found in the cache is detected, then the slots that overlap must be updated: in the write-through case, we simply invalidate them. In the write-back case, the slots must be sent to memory prior to performing the store.

4 Examples of Traffic Elimination

We start with two examples of load and store traffic elimination, based on a particular version of the victim cache. Later sections will look in more detail into other alternatives.

Vector Load Elimination

To see how spill code traffic can be effectively killed, consider the two basic blocks from program Su2cor shown in figure 2. For each basic block we present a schematic view of all spill-loads, spill-stores and stores (normal loads have been removed for clarity). For each spill-load there is a back arrow connecting it to the store from where it is reading its data. We have labeled each arrow with the number of intervening stores between the spill-store saving a register and the spill-load reading it. For example, in basic block B, between spill-store-1 and spill-load-1 there are a total of 8 stores (spill-store-1 itself, spills 2, 3, 4, 6 and stores 5, 7, 8).

Now, assume that we add a small FIFO cache that is able to hold 4 vector registers. Each time a vector store is performed, a copy of the data is kept in this small cache. If the cache is full, the first store is discarded and in the slot being freed the new store is copied. In our example, after executing store-3 we have three full slots in the cache. When load-2 is performed, it checks in the cache. Since it finds the data it was looking for, it uses the special data-path to copy the data from the FIFO cache into the destination register. The memory port is not used at all and is available for a subsequent load instruction.

By merging and ordering all the labels on each arc found in the two basic blocks of figure 2 we obtain the following

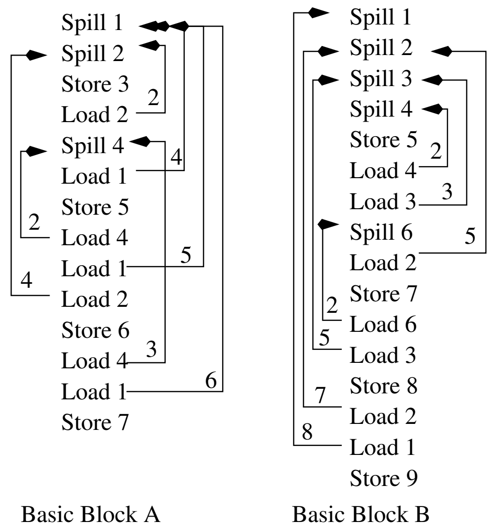

Figure 2: Schematic view of two basic blocks from `su2cor`. Included are only spill-loads, spill-stores, and normal stores. The label on each arc is the number of intervening stores between a spill-store and its matching spill-load.

sequence: 2,2,2,2,3,3,4,4,5,5,5,6,7,8. From this ordering we can statically determine that with a FIFO cache-like structure of 4 slots we should be able to catch 8 spill-loads and enlarging the cache up to 8 slots would catch all spill-loads for these two blocks.

Vector Store Elimination

The possibility of store elimination can also be considered. Instead of having a write-through policy for this FIFO cache just described, we could use a write-back policy. When a store is executed, it is kept in the FIFO cache and does not update memory. Only when the store is evicted from the cache, memory is actually updated.

Every vector store performed must be checked against all stores already present in the cache. If there is overlap between the younger store and one or more of the older stores, these must be sent to memory and their slots invalidated. If there is an exact match between the young store and one of the older stores, this older store can be invalidated without sending it to memory, since the most up-to-date data for that particular memory region belongs to the younger store.

The problem with dynamically eliminating a vector store is that main memory is not in a precise state if an exception is raised after some store has been discarded. This may or may not be a problem depending on the requirements of the baseline architecture.

Going back to our example, we can statically determine the size of the victim cache needed to eliminate store spill traffic. Consider spill-store-1 of basic block A. In iteration i this store data is inserted in the victim cache, and main memory is *not* updated. If we reach iteration $i+1$ and spill-store-1 is still in the victim cache, when we re-execute the same store instruction it will have an exact hit against the old spill-store-1. This old store will be removed from the cache without generating any extra traffic and the new store will be inserted in the victim cache. For this situation to happen, the victim cache needs to be able to hold 7 different vectors. With 7 slots and a FIFO policy, all three spill stores present in loop A would be eliminated.

5 Victim Cache Parameters

The victim cache is a small fully associative cache. We have studied caches having 4, 8 and 16 slots. In all cases, the cache behaves as write-allocate: that is, a vector store always requires a slot in the victim cache.

Regarding the replacement policy we have looked at three variations. The first policy under study is an LRU replacement policy, where the least recently referenced slot is the one being replaced. In case one or more slots are empty or invalid, then one of these slots is preferred for replacement rather than using whatever slot the LRU algorithm indicates. The second policy is a pure FIFO replacement, ignoring empty and valid bits. Although the FIFO scheme sounds odd for a cache, it is well suited for spill code. Looking at the two basic blocks presented in figure 2 we can see that there is a certain degree of locality of references to nearby stores. The third policy, termed FIFO+MRR, is a potential enhancement to the simple FIFO. Looking back to column labeled "avg. reuse" in table 2 we can see that most programs spill and re-load each register only once. To take advantage of this behavior, we add a reference bit to each slot. When a slot is read by a load, its reference bit is set. When looking for a slot to be replaced referenced slots are first selected. If none is available, then the FIFO ordering dictates which slot must be replaced.

6 Performance Results

6.1 Simulation Environment

To assess the performance benefits of the victim cache we use a trace driven simulator. The simulator models a C3400, and executes cycle-by-cycle the binaries compiled using the Convex FC V8.0 compiler at optimization -O2 (vectorization). The simulator closely models all aspects of the vector and scalar parts of the C3400. For the memory system, on the other hand, we assume a simple model where each vector access takes L+VL cycles to complete, where L is the memory latency in cycles and VL is the vector length used. In all following simulations, L is set to 50.

6.2 Write-Through Victim Cache

We will consider the performance of the write-through victim cache under two different models. In the first model, only the vector store traffic properly identified as spill code is cached in the victim slots (model SPILL-ONLY). In the second model, all store traffic, both normal and spill stores, are cached in the victim slots (model ALL-STORE).

How do we identify and differentiate stores that are spill code from normal stores at run time ? Since the compiler always stores spilled registers in the stack and all accesses to the stack are made relative to `a7`, it is straightforward to distinguish normal and spill stores. This scheme has the drawback of relying on the compiler being used. Note, however, that we present this SPILL-ONLY model mostly for evaluation purposes. We believe the ALL-STORE model would be the implementation of choice in most real cases. The rationale for studying both models is to be able to show the fraction of normal store traffic captured by our proposal.

In the following sections we will look at the performance of the victim cache from two related points of view: percentage of traffic served from the victim cache and speedup of each program with the victim cache relative to the reference machine.

Traffic Reduction

Figures 3 and 4 present, for each program, the percentage of data elements that were retrieved from the victim cache and that did not access main memory. This percentage is over total vector and scalar memory traffic. Figure 3 shows the traffic served from the victim cache under the SPILL-ONLY model while figure 4 presents the same data under the ALL-STORE model. For each program we show three different bars, one for each of the replacement policies described above. Also, for each policy we present data for three caches having 4, 8 and 16 slots respectively.

When the victim cache is used to hold spill only data, the amount of load traffic served from the cache is very high in relative terms. The absolute traffic reductions vary between 0% and 4% for four programs, between 5% and 10% in three programs (`swm256`, `arc2d` and `flo52`) and between 13% and 20% for `su2cor`, `tomcatv` and `bdna`. Yet, comparing the percentages shown in figure 3 to the third column of table 2 (percentage of spill-loads) we see that just 4 slots seem to be able to capture almost all available spill load traffic. For example, `swm256` has a 7% of spill load traffic and the victim cache captures a 7% of load traffic. In only two cases, `tomcatv` and `bdna`, the cache falls short of its full potential. For `tomcatv` it only captures a 10% of all load traffic while the spill load traffic represents a 20%. For `bdna`, the cache captures a 9.5% of traffic while it could theoretically be able to capture 38% of all load traffic. To be able to capture the remaining load spill one needs to enlarge the cache up to 16 slots.

As for the effectiveness of each replacement policy, we can see that the differences are very small when the cache has 4 or 8 slots. Only at 16 slots there is a peak favoring the FIFO policy for program `bdna`. Since the cache has very few slots the cost of one policy versus another is very small. Since the FIFO policy is simple and seems to yield slightly better performance it would be our choice.

Comparing figure 4 to figure 3 we can see that there is a fair amount of "hidden" spill code in our set of benchmarks. Except for `swm256` and `hydro2d`, the other 8 programs clearly increase the amount of traffic served from the victim cache when *all* stores are routed through the cache. In the case of `trfd` and `dyfesm`, the rise is very spectacular, reaching 30% of all load traffic. In the case of `bdna`, though, the fact that normal, non-spill, traffic is going through the cache decreases the percentage of traffic served for the LRU and FIFO+MRR cases.

Our analysis of `tomcatv` and `dyfesm` show that all this traffic suddenly captured by the victim cache comes from (a) recurrences and (b) unsolved intra-iteration dependences that the compiler could not rearrange. As an example, we analyze the `trfd` case. Program `trfd` spends most of its time in a vectorized DO-LOOP from routine OLDA. This routine is called from the main program 7 times, and each time the parameter NUM takes a value that goes from 10 to 40 in steps of 5. By manually propagating variables from the main program into OLDA, we eventually collapsed the original code into the code shown in figure 5. We only show the load/stores relevant to our discussion. Note that the notation XRSIQ(MI,MQ) indicates a vector access of length NUM into column MQ of matrix XRSIQ. Analyzing this loop, one can see that there will be an exact match between store (2) and load (3) whenever MP equals MQ. Moreover, when $1 <= MQ < MP-1$ we will have a match between store (4) of iteration MQ and load (3) of iteration $MQ+1$. There is also a potential match between store (2) of iteration $\langle MP, MQ\rangle = \langle j-1, i\rangle$ and load (1) of iteration $\langle MP, MQ\rangle =$

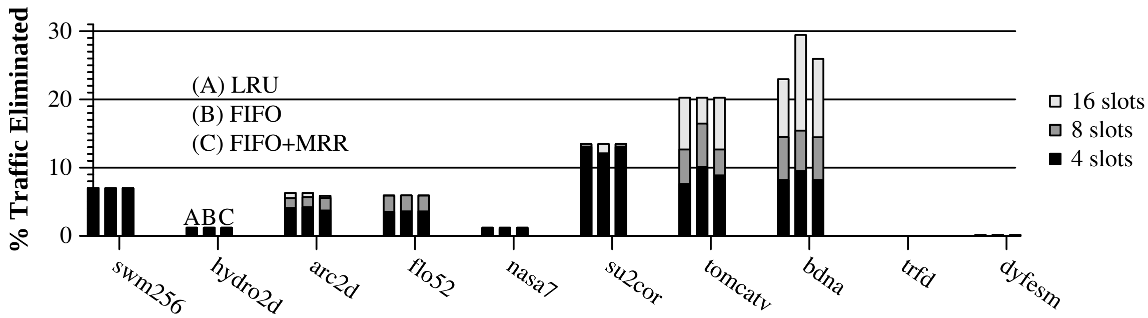

Figure 3: Percentage of total traffic eliminated by the Write-through victim cache under the SPILL-ONLY model.

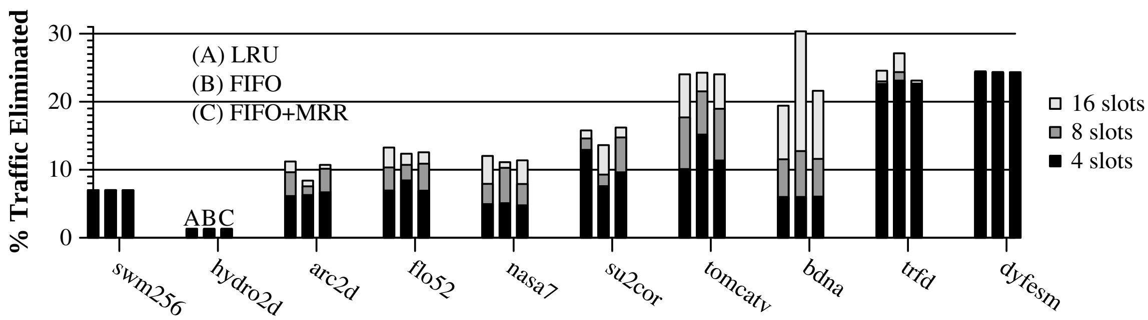

Figure 4: Percentage of total traffic eliminated by the Write-through victim cache under the ALL-STORE model.

```
DO 100 MRS=1,(NUM*(NUM+1))/2
 init code...
 DO 40 MP=1,NUM
  DO 30 MQ=1,MP
    (1) Vector-load  XRSIQ(MI,MQ)
    (2) Vector-store XRSIQ(MI,MQ)
    (3) Vector-load  XRSIQ(MI,MP)
    (4) Vector-store XRSIQ(MI,MP)
  30 CONTINUE
 40 CONTINUE
 update code deleted....
100 CONTINUE
```

Figure 5: Main loop from `trfd`, vectorized and with variables propagated.

$\langle j, i \rangle$. For this latter match to be captured by the cache, we would need at least $(MP-1) \times 2$ slots in order to hold all the stores generated in between.

This example shows how the victim cache can sometimes also be useful as a caching mechanism for normal program variables and not only spill code.

Speedup

Figures 6 and 7 present the associated speedups achieved by the victim cache under the same assumptions described in the previous section.

For the SPILL-ONLY model, typical speedups at 4 slots range between 1.02 and 1.05 with only `tomcatv` and `bdna` slightly exceeding 1.05. On the other hand, under the ALL-STORE model typical speedups range between 1.05 and 1.10, with a couple of programs reaching speedups over 1.15 (`tomcatv` and `bdna` at 16 slots).

The main question is why all the traffic reduction seen in the previous section does not translate into higher speedups.

Program	FIFO		
	4	8	16
swm256	1.3	1.4	1.4
hydro2d	0.0	0.1	0.2
arc2d	1.9	2.0	2.2
flo52	1.3	1.5	2.1
nasa7	1.8	3.8	3.9
su2cor	2.0	2.4	3.6
tomcatv	8.4	11.8	12.2
bdna	2.7	4.4	10.2
trfd	10.2	10.8	12.1
dyfesm	8.7	8.7	8.7

Table 3: Percentage of total execution cycles where the memory port and the victim cache port were both simultaneously busy.

When a vector load is serviced from the victim cache the performance benefits are threefold: first, the latency delay is only 1 cycle (in contrast with 50 cycles for main memory). Second, chaining is allowed between the load that hits in the cache and a subsequent operation using the register being loaded. Third, and most important, the memory port is idle and could and should be used to immediately start another load operation.

The problem is that this third advantage is difficult to achieve. Whenever a load that reads from the victim is dispatched, there is a high probability that the decoder will stall within two or three instructions before seeing another load that could use the main memory bus. This behavior is due to the in-order simplicity of the underlying vector processor. The decode unit only fetches one instruction per cycle and does not do any instruction look-ahead. To quantify this effect, table 3 presents the percentage of cycles that the main memory port and the victim cache port were simultaneously busy. As it can be seen, in most programs the fraction of cycles where the full bandwidth is used is very low. Only for three programs is close to 10%. When the cache size is increased the percentage also increases, but still never exceeds 12%.

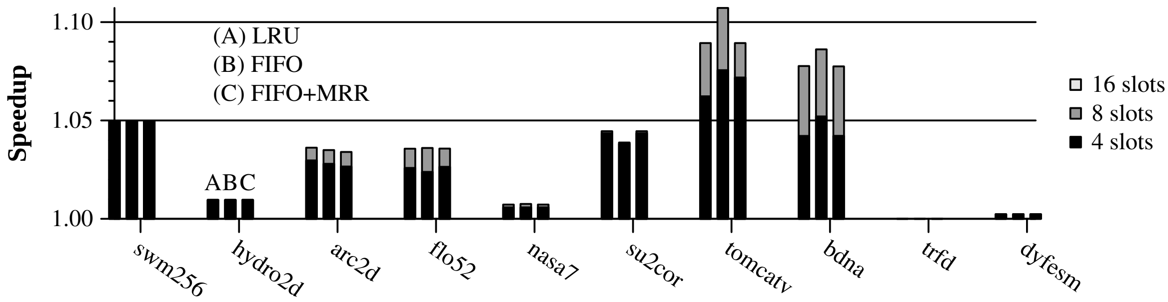

Figure 6: Speedup due to the Write-through victim cache under the SPILL-ONLY model.

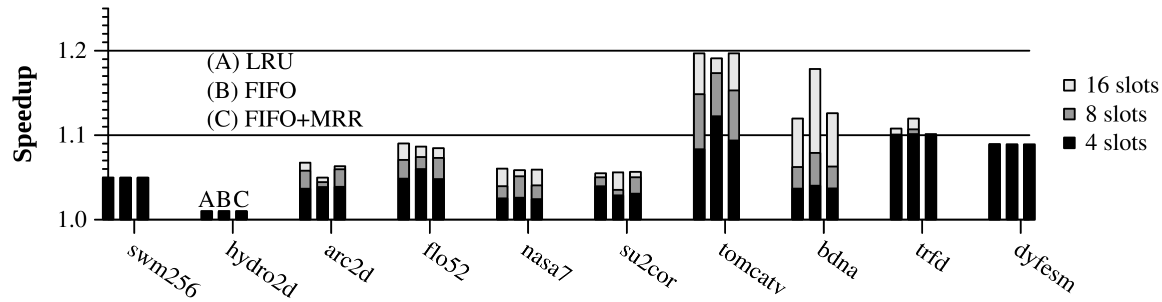

Figure 7: Speedup due to the Write-through victim cache under the ALL-STORE model.

The conclusion from this data is that the victim cache is acting mostly as a latency elimination technique, without delivering an effective bandwidth improvement to the processor. Although the victim cache adds a second input port to the register file, the data shows that the main memory port and the victim port are seldom used together. If the processor had some out-of-order techniques, such as the ones described in [5], the chance for using both ports simultaneously would increase.

6.3 Write-Back Victim Cache

In the write-back victim cache all stores must go through the victim cache. Therefore, the ALL-STORE models will be the only under study in the following sections.

Traffic

Figure 8 presents the fraction of traffic eliminated when using a write back cache, following the same conventions as described in section 6.2 for figure 4.

In almost all programs, the fraction of traffic eliminated is increased with respect to the traffic for the write-through ALL-STORE model presented in figure 4. Programs `trfd` and `dyfesm` have an almost twofold increase in traffic eliminated, due to their special characteristics already mentioned.

Comparing the results to column "Total" of table 2, we see that, except for `tomcatv` and `bdna`, the victim cache is capturing more traffic than the spill present in the programs. This is especially notorious in `nasa7`, `trfd` and `dyfesm`, where the traffic being eliminated is not pure spill code, but is the result of a caching effect.

Speedup

Figure 9 presents the speedups over the reference machine for the write-back victim cache. Surprisingly enough, although the amount of traffic eliminated by the write-back victim is larger than the traffic eliminated by the write-through victim, five programs present worse speedups in

Program	LRU			
	4		16	
	%	cost	%	cost
swm256	n.a.	n.a.	n.a.	n.a.
hydro2d	0.3	85.9	0.8	86.8
arc2d	12.8	95.1	13.1	94.7
flo52	13.3	24.3	14.4	26.6
nasa7	8.4	71.5	10.9	61.2
su2cor	11.7	123.3	19.5	125.1
tomcatv	0.0	0.0	0.0	0.0
bdna	5.2	30.9	7.4	50.2
trfd	0.7	13.9	1.0	16.5
dyfesm	4.4	15.4	7.4	12.5

Table 4: Invalidations in the write back cache: column '%' indicates percentage of all loads that caused an invalidation in the victim. Column 'cost' indicates the average number of data elements sent to memory per invalidation.

the write-back case than in the write-through case (`arc2d`, `flo52`, `nasa7`, `su2cor` and `bdna`). Program `arc2d` presents an even worse situation, since at 4 slots, takes longer to execute with the write-back cache than with no cache at all. The other four programs (`swm256`, `tomcatv`, `trfd` and `dyfesm`), behave as expected and their speedups are parallel to their increase in traffic elimination.

Why is the write-back cache incurring such a performance penalty ? Our analysis reveal that these lower speedups are the result of invalidations caused by load operations. Load instructions must be disambiguated for consistency with all stores present in the victim. If a load is found to overlap with one or more of the stores in the cache, the associated slots must be sent to memory prior to performing the load. All the time needed to send the victim slots to memory adds up to the effective latency seen by the processor for that particular load. Since loads are almost always in the critical path of our programs, the total execution time is likely increased by the number of cycles taken to put memory in an up-to-date state.

We measured for each program two indicators of this situation: (a) the percentage of loads over all vector loads that caused an invalidation and (b) the average cost (in cycles)

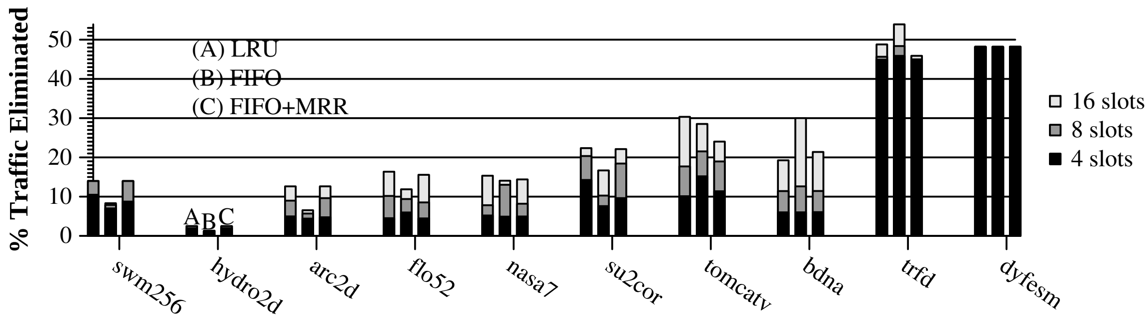

Figure 8: Percentage of total traffic eliminated by the Write-back victim cache (ALL-STORE model).

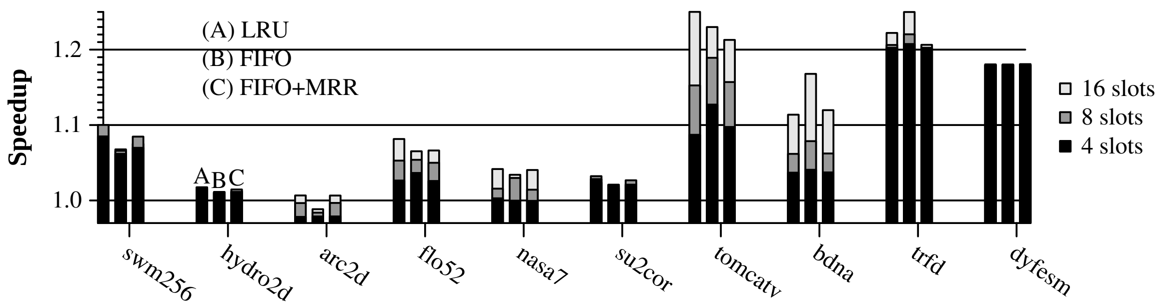

Figure 9: Speedup due to the Write-back victim cache (ALL-STORE model).

of each invalidation. The cost of an invalidation is taken as the number of cycles needed to evict all the slots that have a conflict with a load. The results are presented in table 4. We present the data only for the LRU case with 4 and 16 slots due to lack of space. The data for all other cases is very similar.

Table 4 shows that in all five programs where the unexpected decrease in speedup occurred there is a large percentage of loads that cause an invalidation ($> 10\%$) and that the average cost of an invalidation is also large. The general conclusion is that the write back cache provides better traffic elimination than the write through but that, due to invalidation costs, it actually performs worse. We are currently studying an hybrid approach where spill stores are write-back while normal stores are write-through.

7 Effects of Main Memory Latency

We finally present a study of the effects that different memory latencies have on the victim cache proposal. For the sake of brevity, we will present data referring only to our best performing configuration, that is, the write-through, ALL-STORE, FIFO victim cache. Other configurations behave in a very similar way.

Figure 10 presents the total execution time of each program when main memory latency is varied from 1 to 50 to 100 cycles. We present the execution time for the reference machine (curve labeled "REF") and for the write-through, ALL-STORE victim cache for three different cache sizes: 4, 8 and 16 slots (labeled "WT 4", "'WT 8" and "WT 16" resp.).

The figure shows that even being a vector processor, all programs are severely influenced by memory latency. There are two important points to note: first, at latency 1, the victim cache provides an execution time advantage in those programs that were able to use the victim port and the memory port simultaneously (programs, `tomcatv`, `bdna`, `trfd` and `dyfesm`, see table 3). For the rest of programs, loading a vector from the victim or from main memory is almost the same, since in both cases latency is 1 cycle.

Second, the most important effect of the victim cache is changing the slope of the execution time curves, thereby making the processor more latency tolerant. See how, for example, the slope of the REF and WT curves for program `bdna` are clearly different. This effect would allow to trade off the cost of the memory system against the cost of the victim cache.

8 Related Work

This paper has presented a dynamic spill-code elimination technique based on the concept of caching. To the best of our knowledge, this is the first general technique presented to deal with this specific problem in vector processors.

Previous work has focused on general caching techniques for all load/store traffic. For example, Fu and Patel [7] studied caching for the Perfect Club programs and report that results are very much dependent on stride patterns. Gee and Smith [8] also studied the general problem of caching using traces from a Cray X-MP and the Ardent Titan, but focused on caching all types of references. Following the general caching technique, Yang [13] presented the prime-mapped cache for vector machines, which improved upon previous designs.

Partial solutions specifically targeted at the spill problem where presented in two previous papers. In [4] the same idea of spill code elimination was implemented using the load/store data queues present in a decoupled vector architecture. The technique was able to catch spill loads but was not effective for spill stores. Also, in [5] a technique aimed at reducing spill code was introduced which was based on register renaming and register tagging. This second technique was as powerful as the one presented in this paper but has the drawback of requiring register renaming, which is seldom present in vector architectures.

In this work we start from a traditional, in-order, vector

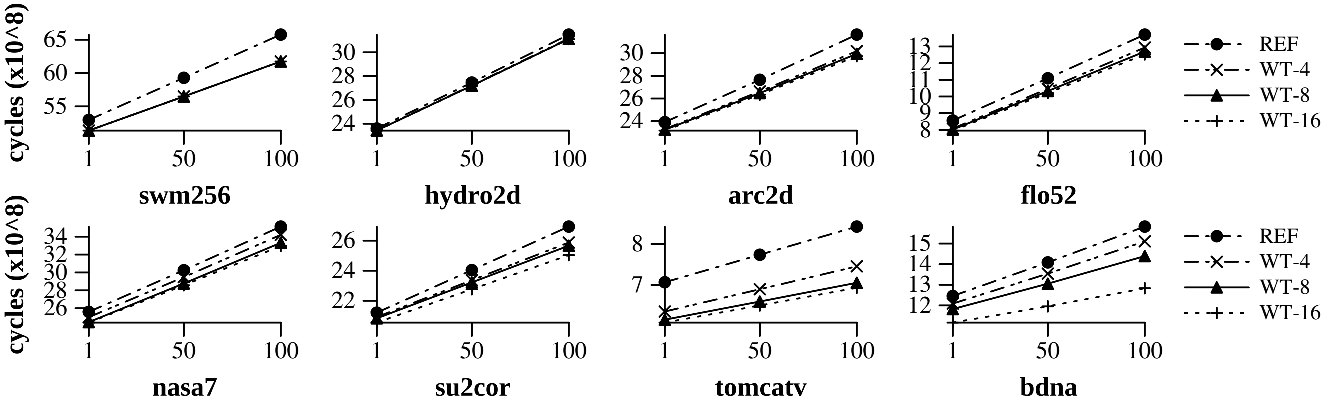

Figure 10: Execution time comparison of reference architecture ('REF') and three configurations of the write-through, ALL-STORE victim cache, under different memory latencies.

architecture and add a victim cache to it without modifying its basic execution paradigm.

9 Summary

This paper has considered the usefulness of adding a victim cache next to the register level of a vector processor. We have seen through measurement of numerical vectorizable codes that the low number of available vector registers causes the introduction of a large quantity of spill code. We have shown that the victim cache is able to filter this spill code traffic and serve most of the data requested by the processor without accessing main memory.

We have studied several configurations of a write-through victim cache that was able to eliminate most of the load spill traffic. With just 4 slots, simulations have shown that between a 5 and a 20% of load traffic can be eliminated.

Although the study was initially targeted at eliminating spill code traffic, when all store traffic is kept in the victim cache, we have seen that some extra traffic elimination is achievable. In particular, hidden spill and reused vectors between iterations of a loop can be captured when all the store traffic is routed through the victim cache.

The speedups associated with these traffic reductions go anywhere from 1.05 up to 1.20, with the speedups associated with the ALL-STORE model being larger than the SPILL-ONLY model.

In order to also eliminate spill store traffic, we have studied a write-back version of the victim cache. The goal was to keep spill stores of a given loop long enough in the victim cache so that at each successive iteration of the loop a spill store would replace in the cache its own copy from the previous iteration of the loop. The traffic reductions associated with this write-back victim were much larger than those of the write-through cache, typically in the 10–50% range. The write back cache, though, has two drawbacks. First, it does not allow for precise exceptions if stores are eliminated without updating memory. Second, the speedups achieved can be worse than in the write-through case due to invalidations caused by load operations.

Finally, we have looked at the effect of the victim cache when main memory latency is varied from 1 to 100 cycles. The simulations show that the victim cache can improve execution time even with a perfect memory system having a 1 cycle latency provided that the instruction level parallelism in the program allows to use the main memory port and the victim cache port simultaneously. For larger latencies, the victim cache hides some of the main memory latency and allows the processor to be more latency tolerant. At latency 100 cycles, the simulations have shown speedups as high as 1.24.

References

[1] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 53–65, White Plains, NY, June 1990.

[2] Convex Press, Richardson, Texas, U.S.A. *CONVEX Architecture Reference Manual (C Series)*, sixth edition, April 1992.

[3] R. Espasa and X. Martorell. Dixie: a trace generation system for the C3480. Technical Report CEPBA-RR-94-08, Universitat Politècnica de Catalunya, 1994.

[4] R. Espasa and M. Valero. Decoupled vector architectures. In *HPCA-2*, pages 281–290. IEEE Computer Society Press, Feb 1996.

[5] R. Espasa, M. Valero, and J. E. Smith. Out-of-order Vector Architectures. Technical Report UPC-DAC-1996-52, Univ. Politècnica de Catalunya–Barcelona, November 1996.

[6] R. A. Fatoohi. Vector performance analysis of three supercomputers: Cray 2, cray Y-MP, and ETA 10-Q. In *Proceedings of the Supercomputing 89*, pages 779–788, Reno, NV USA, 1989. ACM Press , New York, NY , USA.

[7] J. W. C. Fu and J. H. Patel. Data prefetching in multiprocessor vector cache memories. *International Symposium on Computer Architecture*, pages 54–63, 1991.

[8] J. Gee and A. J. Smith. The performance impact of vector processor caches. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume 1, pages 437–449, January 1992.

[9] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[10] L. Kontothanassis, R. A. Sugumar, G. J. Faanes, J. E. Smith, and M. L. Scott. Cache performance in vector supercomputers. In *Proceedings of Supercomputing'94*, Washington D.C., November 1994. IEEE Computer Society Press.

[11] R. Perron and C. Mundie. The architecture of the Alliant FX/8 computer. In *Digest of Papers: Compcon 86*, pages 390–394, Washington, D.C., March 1986.

[12] R. M. Russell. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[13] Q. Yang. Introducing a new cache design into vector computers. *IEEE Transactions on Computers*, 52(12):1411–1424, December 1993.