Cost-Effective Compiler Directed Memory Prefetching and Bypassing

Daniel Ortega[†], Eduard Ayguadé[†], Jean-Loup Baer[‡] and Mateo Valero[†]

[†]Departamento de Arquitectura de Computadores,
Universidad Politécnica de Cataluña – Barcelona, Spain
{dortega,eduard,mateo}@ac.upc.es

[‡]Department of Computer Science and Engineering,
University of Washington – Seattle, WA
baer@cs.washington.edu

Abstract

Ever increasing memory latencies and deeper pipelines push memory farther from the processor. Prefetching techniques aim is to bridge these two gaps by fetching data in advance to both the L1 cache and the register file. Our main contribution in this paper is a hybrid approach to the prefetching problem that combines both software and hardware prefetching in a cost-effective way by needing very little hardware support and impacting minimally the design of the processor pipeline. The prefetcher is built on-top of a static memory instruction bypassing, which is in charge of bringing prefetched values in the register file. In this paper we also present a thorough analysis of the limits of both prefetching and memory instruction bypassing. We also compare our prefetching technique with a prior speculative proposal that attacked the same problem, and we show that at much lower cost, our hybrid solution is better than a realistic implementation of speculative prefetching and bypassing. In average, our hybrid implementation achieves a 13% speed-up improvement over a version with software prefetching in a subset of numerical applications and an average of 43% over a version with no software prefetching (achieving up to a 102% for specific benchmarks).

1 Introduction and Related Work

Memory operations represent over 20% of all dynamic instructions [3] and while other types of instructions have fixed latencies, *loads* have latencies that depend on the placement of the data in the memory hierarchy. When the data is not in the cache closest to the processor, stalls may occur. In scientific applications this stalling effect has been measured [7] and represents a significant amount of total application time.

Technological trends point towards relatively longer latencies at all levels of the memory hierarchy, therefore techniques to tolerate them better become increasingly important. In particular, larger and more efficient cache hierarchies and more precise prefetching techniques have become necessary to sustain high performance processors.

Prefetching is generally defined as the action of bringing a data item closer to the processor before it is accessed by a *load* instruction, thus reducing the *load*'s overall latency. Prefetching relies on address prediction, i.e., the prediction of the effective address of the operand of a (predicted to be executed) subsequent *load* instruction, thus making prefetching speculative. If prefetching is limited to bringing data in the cache hierarchy, processor state is not affected and there is no need for microarchitectural recovery mechanisms.

Prefetching can be extended by considering the physical registers to be part of the memory hierarchy. That is, even if the data is present in the lower level cache (L1), more can be gained by bringing it closer to the processor. This technique is referred to as binding prefetching when it is performed in a non-speculative way [7]. There exist several ways to perform this binding speculatively [10], including value prediction.

Initiation of the prefetching can be done by either the software, the hardware, or a hybrid combination of both. In software-based prefetching, the compiler [7] can insert a *pref* instruction which, when executed, will bring the referenced data into the L1 cache. This explicit prefetching has no semantic implication and can be ignored by the hardware, e.g. for binary compatibility. The compiler can also advance a *load* ahead in the instruction stream and in this case, the (implicit) prefetching becomes binding since data is brought into a register. If the *load* instruction has been advanced across basic block boundaries, it can become speculative and a recovery mechanism is required [6]. This concept, akin to forwarding mechanisms [3] used to alleviate the latency of dependent instructions, can be extended to the case of non-speculative *pref* instructions. In [8], a novel mechanism was introduced that transforms non–binding prefetches into binding, thus bypassing the execution of the *load* instructions that effectively bring data into the register file. This bypassing, defined as *memory instruction bypassing*, extracts

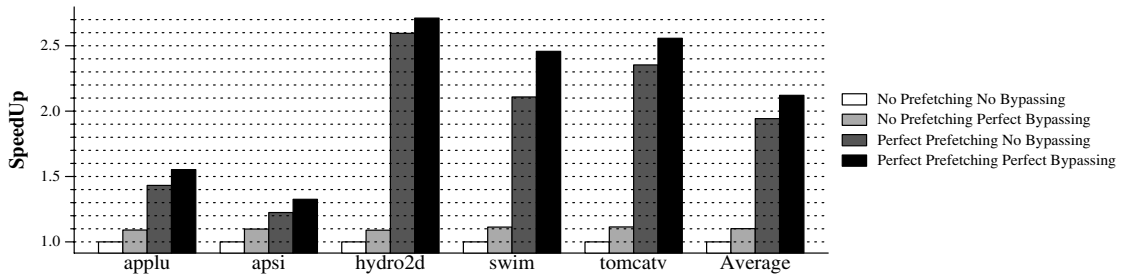

Figure 1. Analysis of perfect prefetching and perfect bypassing in a 4-way machine

the execution of the *load* instructions from the critical path. On the other hand, hardware-based prefetchers rely on dynamic learning mechanisms to detect memory access patterns and predict future memory references [4]. In scientific applications, the learning mechanism is used to detect strided streams, and state-machines [1] or confidence counters [2] are used to filter out potentially non-useful prefetches. Data can be brought in L1 or in registers with the same implications as in software-based prefetchers. Recently hybrid prefetchers have been proposed, mostly in the context of prefetching for recursive data structures [5]. The basic idea is to have slices of programmes be executed by a hardware prefetcher under compiler control. [12] also uses slices to help in the precomputation of addresses and branch speculation.

The main contribution of our paper is a novel hardware prefetching mechanism, oriented at a subset of all memory operations, just those involved in our compiler directed *memory instruction bypassing*. By focusing just on a few amount of instructions, our mechanism can be simple in terms of hardware, effective in terms of accuracy as the compiler already knows the behaviour of these instructions and concious on the type of prefetching to be accomplished (in terms of distance or other requirements). We also analyse the relation between prefetching and *memory instruction bypassing*. We will show that the latter becomes more important when better prefetching is applied. Finally, in section 4 we will compare our mechanism with a highly speculative mechanism known as APDP (Address Prediction for Data Prefetching) [2], which benefits from the same sources as our proposal. Conclusions and future work can be found in section 5.

2 Theoretical Limits of Prefetching and Bypassing

Prefetching's main objective is to have all data that will be accessed as close to the processor as possible. Perfect prefetching can be thought of as a technique that achieves the scenario where all *loads* find their data in the L1 cache. The characteristics of this prefetching technique would be perfect accuracy (no misprediction), total coverage (all data is prefetched), and timeliness (data is prefetched early enough so that it is in cache when accessed).

The aim of *memory instruction bypassing* is to hide the execution of memory instructions. Therefore, under this definition, perfect *memory instruction bypassing* would achieve zero latency for all *loads*, hiding perfect prefetching benefits. For the sake of our research, we will separate the benefits of perfect prefetching and perfect *memory instruction bypassing*. We will consider perfect *memory instruction bypassing* just bringing data from L1 (where prefetching stops) to the register file in zero cycles (ocurring at the decode stage, thus allowing a perfect bypassing of the memory instruction). This way we will be able to distinguish between the potential benefits of moving data in the memory hierarchy from those of bringing data from the L1 cache to the register file.

Figure 1 presents the results obtained with perfect bypassing, perfect prefetching, and their combination in a 4-way machine very similar to a MIPS R10K (the configuration of this processor will be thouroughly explained in section 3). The improvement of performance produced by perfect prefetching is, as expected, bigger than that of perfect *memory instruction bypassing*. Perfect prefetching shortens a critical path that can be very big, specially with current memory hierarchies where bringing cache lines from main memory may take a hundred processor cycles or more. Perfect *memory instruction bypassing* is shortening a much smaller path. Nevertheless, from the figures, we can see that the effect produced by perfect *memory instruction bypassing* is bigger when combined with perfect prefetching. This can be explained using Amdhal's Law. As we improve the overall time consumed in bringing data from main memory to the L1 cache, the rest of the cost of the memory operation (the bypassing part) becomes relatively more important. This result demonstrates the synergetic effect between prefetching and bypassing.

We also run experiments with deeper pipelines in

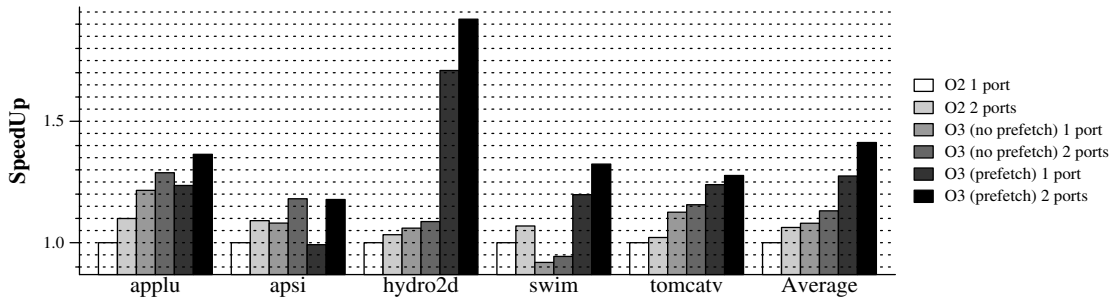

Figure 2. Effect of compiler optimisations and the number of ports in a 4-way machine

which both the L1 latency and the memory stage were increased from 1 to 2 cycles. We found out that the relative improvement of *bypassing* with respect to prefetching increases, which shows that *bypassing* will become more important with deeper pipelines.

3 Prefetching and memory instruction bypassing implementation techniques

3.1 Software Prefetching

Initiating prefetching can be done automatically by the hardware or can be driven by the insertion or placement of memory operations in the code, i.e. software prefetching, which itself can be decided by the compiler alone or may require the aid of the programmer.

Software prefetching can be classified as being binding or non-binding. Binding prefetch brings data from the memory to the register file before it is needed. In order to achieve this, the *load* instruction must be advanced far enough in the instruction stream so that the contents of the register are updated before its consumers start executing. This advancement can be achieved, for example, by executing the *load* instruction a few iterations ahead of where it is needed.

Sometimes hiding the total latency of the *load* is not possible and in many architectures *load* instructions cannot be executed speculatively because they may access memory locations that trigger exceptions, modifying the normal execution of the application (specially if it should have never been executed). In certain codes the moment in which we know that a particular *load* is going to be executed is too near to the use of the data to completely hide its latency. In other situations it is the lack of resources that the *load* needs (such as logical registers) that inhibits us from expanding the lifetime of the *load*. In these two latter scenarios though, the benefits produced from hiding partially the *load* latency are far better than no prefetching at all.

Non-binding software prefetch tries to overcome the limitations of the binding case. A new instruction, namely *pref* instruction, is used to inform the hardware which data will potentially be accessed in the near future, which is consequently brought to the L1 cache (thus allocating no registers). In most architectures, this *pref* instruction is speculative, thus raising no exceptions at all. This *pref* instruction (and the non-binding prefetch it represents) does not have the drawbacks of the binding *load*, but on the other hand has other drawbacks. As it leaves the data in the L1 cache, a conventional *load* instruction must still be present in the code to bring the data to the register file. This implies the overhead of two memory operations to do the work of one. Moreover, the conventional *load* instruction is usually found near its dependent use, and therefore, the cost of bringing data from the L1 to the register file (what we have called *memory instruction bypassing*) is not hidden at all.

The choice of when to aply either is not simple. Normally the compiler will combine both techniques trying to achieve the best performance. In all our simulations, we have used code produced by the MIPSpro compiler (Version 7.30), which is considered to be a very good compiler in terms of prefetching and which combines both binding and non-binding prefetch. We have analysed two base machines, roughly based on the MIPS R10K processor. The first one assumes a 4-way out-of-order core and the second, more aggressive, assumes an 8-way out-of-order core. The number of functional units and the rest of the resources have been leveraged to the issue width. The base memory hierarchy analysed consists of a 32K direct mapped L1 cache accessible in 1 cycle, a 4-way 2Mb L2 cache with hit in 13 cycles and main memory with an 88 cycles latency.

In figures 2 and 3 we can see the effects that software prefetching has on five numerical applications running on our base machines. All five applications were chosen from the SPECfp95 suite due to their amenability to simulation.

The compiler produced three different versions of each application. In the compiler framework used for our experiments, prefetching can only be inserted in -O3 mode. This is why we have chosen to produce

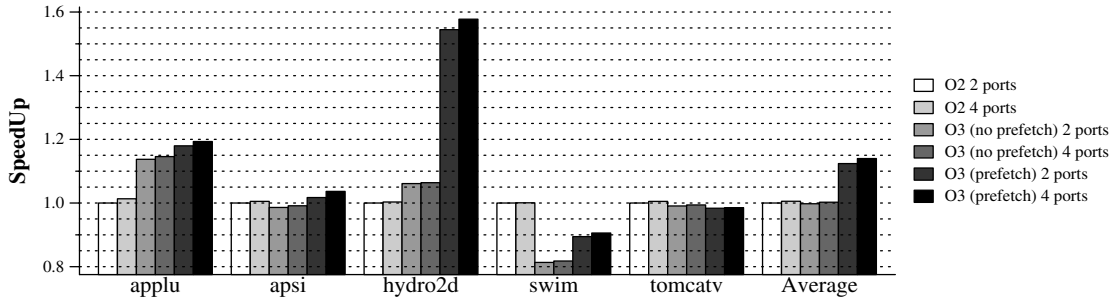

Figure 3. Effect of compiler optimisations and the number of ports in a 8-way machine

results with the following optimisations. The first version uses fairly common safe optimisations (-O2 in our environment). The second version has aggressive optimisations turned on (-O3) but no software prefetching at all. The third version has both the aggressive optimisations and the prefetching turned on. The use of -O3 may introduce optimisations that in some cases degrade performance, as we will see for certain configurations of the experiments conducted. In this experiments we have also analysed the effect of the number of ports when accessing the L1 cache.

From these figures we have observed that in general, -O3 (no prefetch) results in a definite improvement over -O2 in a 4-way machine. `swim` is the exception. For this reason, in forthcoming graphs we will be using -O3 (no prefetch) as our baseline. We note, however, that the compiler impact diminishes when the machine becomes more powerful (cf. figure 3). We have also noted that prefetching with a single port in a 4-way machine can be detrimental (`apsi`) or provide little benefit (`applu`). We will therefore use two ports and four ports for the 4-way and 8-way machine although this factor is less important for the latter.

3.2 Compiler Directed Memory Instruction Bypassing

A little more must be said about binding and non-binding prefetch before we describe this technique. The former is better as it leaves data directly into registers, while the latter is less restrictive in terms of resources, specially logical register names. A binding prefetch in the form of an advanced *load* instruction not only brings data in the register file, but also brings a full cache line to the L1 cache. If other elements of the cache line are needed in the near future, this *load* is acting upon them as a non-binding prefetch. The compiler normally can deduce this reuse, and therefore, will not need to issue prefetches for these other elements in the line; instead it will consider them already prefetched (in a non-binding manner). Similarly a *pref* instruction will act as a non-binding prefetch for all the elements in the cache line that will be accessed in the future. This allows the compiler to economise the number of extra *pref* instructions needed to prefetch elements that live in the same line of cache.

The goal of the mechanism proposed in [8] is to convert non-binding to binding prefetches with the combined benefits of both types of prefetching. This is accomplished by the use of two new instructions: $pref^{\ddagger}$ and $load^{\ddagger}$. We will explain the first one in detail and afterwards point out how the second one behaves.

The main difference between a $pref^{\ddagger}$ and a normal *pref* instruction is the extra information that the $pref^{\ddagger}$ carries. This new instruction has a destination register and a bit field indicating which elements in the cache line will be accessed (more information about how this information can fit into the instruction encoding can be found in [8]).

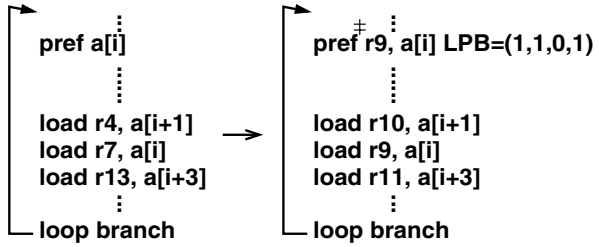

Figure 4. Pref case

Figure 4 shows an example of the use of $pref^{\ddagger}$. On the left hand side is the code that a usual compiler would generate and on the right the code using $pref^{\ddagger}$. This new instruction is annotated with the destination register of the subsequent *load* instruction that will read the first element of the prefetched line. The resf of *loads* that read elements from the same line are remapped to use consecutive numbers in their destination registers, thus allowing the hardware to detect which *loads* will consume the data prefetched by this $pref^{\ddagger}$. At runtime, the hardware not only issues the prefetch but it also brings the required elements of the line into the register file. These values are assigned free physical registers, and special mappings are created in

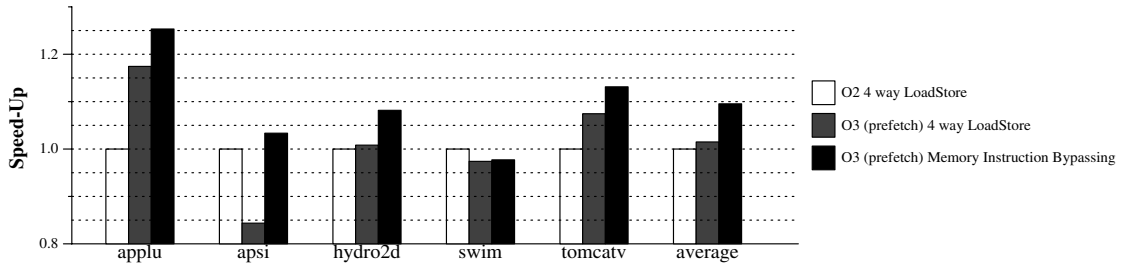

Figure 5. Improvement due to compiler-directed memory instruction bypassing

a separate table. When the *loads* reach the decoding stage, the hardware recognises them and instead of issuing and executing them in a load-store unit, it moves the special mappings from the separate table to the normal renaming table. This way, any instruction before the *loads* will see those logical registers as having their normal value, and after the *loads* they will be directed to the values brought by the $pref^{\ddagger}$ instruction. Notice that these values may not have reached the register file at the decoding of the *loads*, in which case the dependent instructions on the values produced by the *loads* should wait just as if they were waiting for the *loads* to complete (i.e. some form of scoreboarding is still necessary). In the case that the values have arrived to the register file, the dependent instructions may consume them inmediately, thus allowing for the *memory instruction bypassing* to occur.

Although the number of logical registers is fixed in any given ISA (possibly preventing the compiler of inserting more binding prefetches), the number of physical registers may vary with each implementation of the ISA. This mechanism dynamically makes use of free physical registers when the compiler lacks logical ones. The mechanism behaves very similarly with the new $load^{\ddagger}$ instruction. The only difference with respect to what we have just explained is that the primary data brought by the $load^{\ddagger}$ is renamed normally, as it would have been if it were a normal *load* instruction.

In figure 5 we can see the results observed when this particular *memory bypassing mechanism* is implemented in a 4 way out-of-order processor. This *memory instruction bypassing* mechanism based on renaming shows speed–ups ranging from 5% to 22%. As was expected, *memory instruction bypassing* yields very good benefits when combined with software prefetching alone. We will show that this relation becomes stronger in the presence of hardware prefetching. The results of this figure have been extracted from [8].

3.3 Our proposal: A decoupled prefetcher

As was noted in section 2, *memory instruction bypassing* produces better benefits when prefetching is present. The mechanism presented in [8] relies on software prefetching, which in many cases is not enough to allow all the potential benefits that *memory instruction bypassing* may offer. To boost the prefetching capabilities of the software prefetcher, we propose a compiler-controlled hardware prefetcher that is simple and thus cost-effective. This hardware prefetcher will assist the compiler instructions $pref^{\ddagger}$ and $load^{\ddagger}$ when appropriate. We call this prefetching scheme decoupled because the actual prefetching of the data is shared among different resources. The compiler inserts software prefetching instructions, responsible for bringing data closer to the processor. The compiler also instructs the special prefetching hardware as to which instructions should be prefetched with greater distances and without extra software intervention. These supplementary prefetches will bring data into the L1 cache. The *memory instruction bypassing* mechanism (also compiler directed) is responsible for bringing the data from the L1 cache to the register file in a non-speculative way.

A documented drawback of hardware prefetchers is trying to prefetch for too many *load* instructions unless some (complex) filtering is implemented [9]. Our mechanism issues as few useless prefetches as possible. This is accomplished by issuing only one prefetch per cache line and in a conservative manner since it is controlled by relying on the new type of $pref^{\ddagger}$ and $load^{\ddagger}$ instructions. Each time one of these instructions arrives in its execution phase, it is executed normally (which includes the binding of all the requested elements of the line), and a special prefetching hardware assist starts the process of prefetching successive elements. The purpose of this hardware is to improve the timeline of the prefetch by increasing the chance that future executions of this same instruction will hit in cache, making the most out of the bypassing effect.

The nature of our prefetching mechanism allows the work of prefetching and *memory instruction bypassing* to be separated among the hardware and the software. Our hardware prefetcher is in charge of non-binding prefetching into the L1 cache, while the software con-

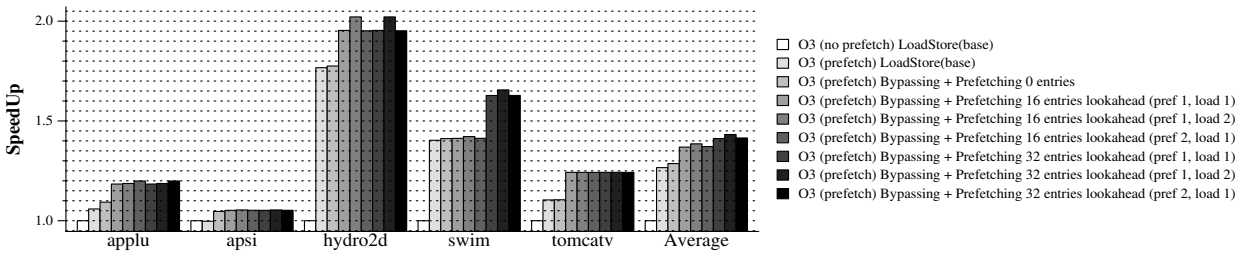

Figure 6. Influence of the number of entries and lookahead policies (4-way)

trolled $pref^{\ddagger}$ and $load^{\ddagger}$ instructions bypass the data into registers. This leaves the compiler with more freedom, as it does not need to insert binding prefetch too far away from its use, thus reducing logical register pressure. Moreover, as our prefetcher leaves data in the L1 cache, our hardware mechanism is non-speculative and does not require recovery mechanisms.

Our hardware consists of a small PC tagged table called Cache-Line Prefetching Table (CLPT). In our experiments we have chosen to make it very small (from 8 to 32 entries) and fully associative with least recently used (LRU) replacement policy. Each entry in this table has two fields (besides the tag): (1) *Last Effective Address*, which contains the last address produced by this instruction, and (2) *Type Bit* which allows us to differentiate among $pref^{\ddagger}$ and $loads^{\ddagger}$. This bit is strictly necessary as we could keep different instructions in different tables. We have kept them together to simplify the understanding of the mechanism.

When a $pref^{\ddagger}$ or a $load^{\ddagger}$ is decoded, the hardware mechanism is activated. If there is a tag match in the CLPT table, the current stride (difference between effective address and last effective address) is computed. The address of the next line to prefetch is computed adding $N \times stride$ to the effective address and a prefetch is initiated. N, the depth of prefetching, is a parameter and we have performed experiments with $N = 1$ and $N = 2$, and with either value for $pref^{\ddagger}$ and $load^{\ddagger}$ instructions (thus the need for the Type Bit). In this paper we kept N fixed for the whole application. We are planning future experiments in which the depth may vary dynamically or even be compiler directed on an instruction per instruction basis.

In the second case, if the instruction has no allocated entry in the CLPT, the hardware assists simply allocates one (using LRU replacement policy). In both cases, the effective address in the CLPT is updated. Once our mechanism decides to prefetch a cache line, the request is maintained until it effectively finds a free port to the memory hierarchy and can be executed. Some prefetching mechanisms filter prefetches when the ports are busy, not to exhaust the memory hierarchy with requests. Since our mechanism is driven by a subset of all memory operations we do not have this problem and do not have to introduce any hardware filtering mechanism. Moreover, the *renaming bypassing mechanism* on which the hardware prefetching relies has been shown to minimise the need of ports by utilising them more intelligently. We also note that the process of looking in the table and prefetching can be divided into several stages, therefore not impacting cycle time. As prefetched data is not used immediately, we can be more lax in choosing the time to issue the prefetch without impacting severely the performance benefits.

The cost of our decoupled prefetching and *memory instruction bypassing* mechanism consists of the number of entries in the CLPT and the necessary circuitry to handle prefetches, plus the hardware associated with the *memory instruction bypassing* mechanism. The *memory instruction bypassing* mechanism relies on a secondary renaming mapping table and on an increase in the number of physical registers. The cost in size and impact on cycle time has been shown to be minimal[8].

In figures 6 and 7 we can see the performance results obtained for our decoupled prefetching and *memory instruction bypassing* mechanism. The first two bars represent the improvement due to software prefetching alone. The next bars represent the different configurations for our decoupled prefetching mechanism on top of the *memory instruction bypassing* mechanims. The first bar (0 entries) is the *memory instruction bypassing* mechanism alone. The other bars represent the results obtained by our prefetching and *memory instruction bypassing* mechanism with varying number of entries (16 and 32) and different lookahead depths (1 and 2). We can see that all applications except `apsi` benefit from our hybrid approach averaging a 43% speed-up over an execution without software prefetching and a 15% speed-up over software prefetching in the case of a 4-way processor. With respect to the varying depth parameters, in average, the best combination of depths is 1 for $pref^{\ddagger}$ and 2 for $load^{\ddagger}$. This observation agrees with the fact that our compiler framework usually in-

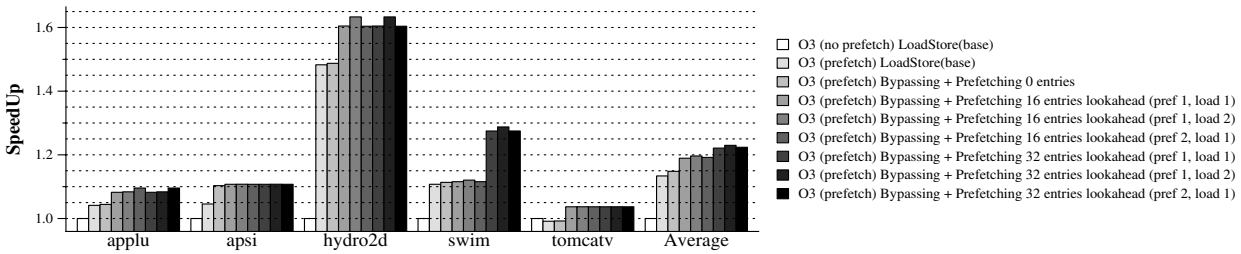

Figure 7. Influence of the number of entries and lookahead policies (8-way)

serts *prefs* with iteration distances bigger than one, while normally binding *loads* are inserted very near their consuming *loads*. This way, having a depth of 2 for $load^{\ddagger}$ is compatible with the software prefetching to allow data to be in L1 at bypassing time while for $pref^{\ddagger}$ software prefeching already has the correct timeliness.

As can be seen, 16 entries are sufficient in all cases except `swim` where performance improves with an increasing number of entries. We have performed a sensitivity study on the number of required entries. In figures 8 and 9 we present results for an increasing number of entries, with the last bar being a limit analysis in which an infinite number of entries was simulated. As can be seen, all programmes saturate their performance improvement at some point, namely 16 entries for all but for `swim`.

We feel that better performance could be obtained with other configurations in the tables (we are now using fully associative and LRU replacement) in the case of a small number of entries. $pref^{\ddagger}$ and $load^{\ddagger}$ instructions usually access the CLPT in a round robin fashion using LRU replacement which is known not to be good in such situations. We intend to implement CLPT with other hardware configurations to establish this point.

4 Comparison

Although we believe this is the first paper in which the concept of *memory instruction bypassing* is defined and analysed in depth, we are aware that several microarchitectural techniques have already been published that partially exploit the same benefits of *memory instruction bypassing*, in many cases without separating them from other microarchitectural benefits. A speculative technique that uses both prefetching and *memory instruction bypassing* is Address Prediction for Data Prefetching (APDP) [2]. We have found that this technique, whose conceptual roots are based on Value Prediction, benefits from very similar concepts as those explained previously, and is a perfect match for comparison purposes.

APDP predicts the effective addresses of memory operations (both *loads* and *stores*) and using this prediction, prefetches data which is allocated in a special table called Memory Prefetching Table (MPT). The execution of *load* instructions varies from that of a normal pipeline. When a *load* instruction arrives at the decode stage, its PC address is used to access the non-tagged MPT. Each entry of the MPT has the following fields: (1) *Last Effective Address* of the instruction, (2) *Stride*, (3) *Stride History Bits*, which consist on two bit saturating counters that assigns confidence to the prediction, (4) *Value*, which contains the loaded value from memory, and (5) *Valid Value Bit*, which indicates if the value has arrived from memory.

If the *Stride History Bits* show confidence in the prediction and if the *Valid Bit* is set, the predicted value found in the *Value* field can be sent, in the decode stage, to the register file for dependent instructions to use it (this is both address and value prediction). If there is confidence in the prediction but the *Valid Bit* is not set, i.e. the data has not yet arrived, the *load* is executed with this predicted address, shortcutting the computation of the address. In both cases, the speculation is resolved by computing the effective address of the *load*. If it differs from the predicted one, a recovery action must be taken.

Regardless of whether bypassing has occurred or not, each *load* instruction accesses the MPT at the commit phase to update its entry. The fields *Last Effective Address* and *Stride History Bits* are updated according to their semantics. In order to prevent too many modifications to the stride (specially at the end and start of loops) the *Stride* field is only updated when the prediction is not confident. Prefetches are only issued in the case of confident predictions and free ports being available. In the original paper [2] the authors proposed (but not implemented) to buffer prefetch request to issue them when a free port becomes available. We have compared these two options, and they both seem to be very similar in performance when the number of ports is sufficient. With few ports, issuing only when a port is free is the best alternative.

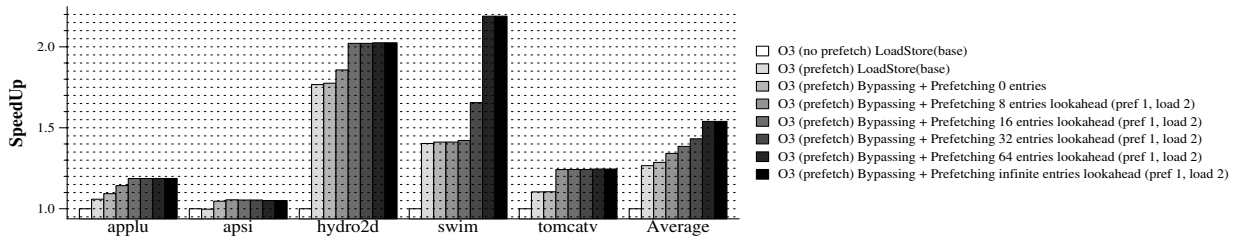

Figure 8. Influence of increasing the number of entries in our decoupled prefetcher (4-way)

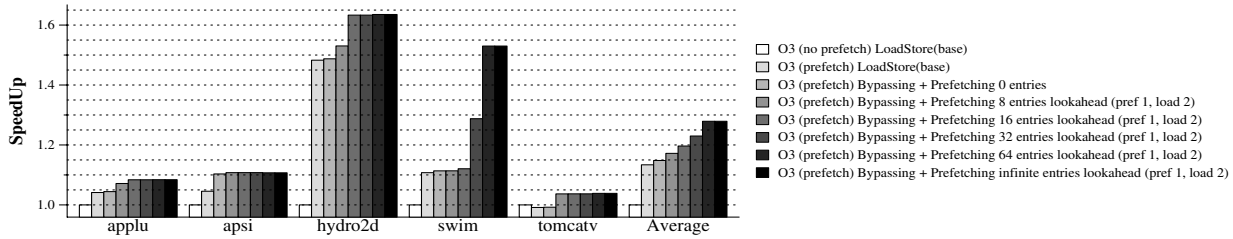

Figure 9. Influence of increasing the number of entries in our decoupled prefetcher (8-way)

The last issue regarding the implementation of APDP is the recovery mechanism. A good explanation of recovery mechanisms can be found in [11] where recovery schemes are divided into: (1) squashing recovery, similar in spirit to branch prediction recovery whereby all instructions following the mispredicted *load* are reexecuted; and (2) selective recovery where only those instructions dependent on the mispredicted *load* are reexecuted. While neither of these two mechanisms is easy to implement, selective recovery is much more complex for an out-of-order processor. To our knowledge, no such cost-effective implementation has been proposed. We will show that APDP is very sensitive to the choice of recovery mechanisms (only selective recovery was analysed in [2]).

The original paper on APDP did not quantify separately the performance benefits due to prefetching to those due to *memory instruction bypassing*. In figure 10 we present results for various simulations with respect to our 4-way base machine described previously. The configuration parameters for the APDP were taken from [2]: in particular the MPT table has 2048 entries. These results (cf. figure 10) show the relative contributions of prefetching and *memory instruction bypassing* in the APDP scheme. We did simulations with the executables highly optimised (-O3) with and without software prefetching. With these two executables we run simulations for the base machine (LoadStore) and for three versions of the APDP, namely: (1) Prefetching only; (2) Prefetching and *memory instruction bypassing* with squash recovery; and (3) Prefetching and *memory instruction bypassing* with selective recovery.

As can be seen in the figure prefetching only to L1 can cover up to 75% of the total performance benefits of APDP with selective recovery. It can be noted that APDP with squash recovery may be worse (up to a 5% in `applu`) than just bringing data to the L1 cache, specially in applications with software prefetching. In these applications squash recovery mechanisms seems to produce a very small benefit (in avg. a 2%) over prefetching only to L1. A possible explanation for this behaviour is that when the compiler introduces software prefetching, it usually unrolls loops. This unrolling converts one memory operation into a series of memory operations besides cutting down the total iterations of the loop. When the loop finishes and is reexecuted, in the first iteration, not only one, but all memory operations will fail their speculation, resulting in a series of squashes, which, due to the shorter number of iterations of the loop, represent a bigger handicap for APDP. What is gained through value prediction and speculation is almost totally counterbalanced by the cost of the series of squashes.

In figure 11 we compare relative performance results of a subset of the configurations presented so far. In average, it can be seen that having a decoupled prefetching and *memory instruction bypassing* mechanism as introduced in section 3 behaves midway APDP with squash recovery and APDP with selective recovery mechanism.

In order to fully understand these results and why we consider that our mechanism is a cost-effective solution, we must take into consideration the costs of both mechanisms. Cost cannot be measured in a simple way with just one number. Cost implies sizes of tables, power consumption, potential increase in cycle

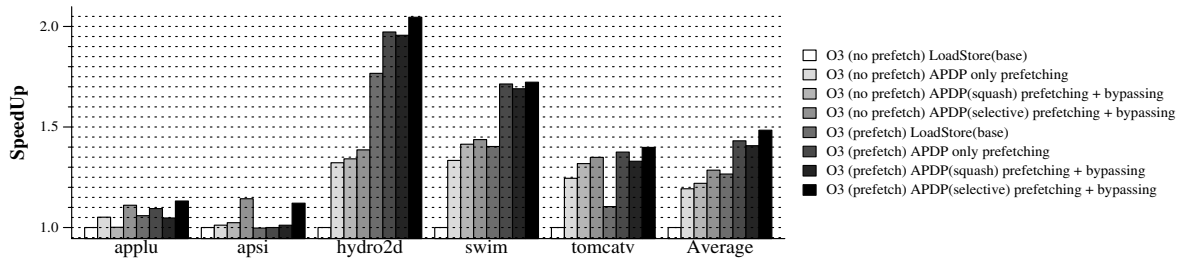

Figure 10. Effect of only prefetching in APDP (4-way and 2 ports)

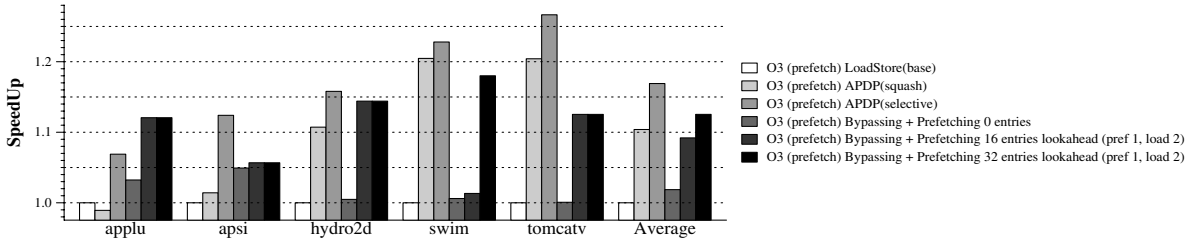

Figure 11. Comparison of APDP and our decoupled prefetcher (4-way)

time, etc. The size of tables can be easily established for both our mechanism and APDP. In [8] it was shown that the cost in bytes of the renaming table associated with the *memory instruction bypassing* mechanism was less than 500 bytes. In addition, the maximum amount of extra registers to achieve all the potential benefits of this mechanism is 25. The cost of the CLPT depends on the number of entries but never exceeds 150 bytes. Thus the cost of the hardware to implement our decoupled prefetching mechanism on top of the *memory instruction bypassing* mechanism of [8] is minimal, approximately 700 bytes. The whole mechanism is non-speculative and therefore does not need any kind of recovery mechanism, contrary to APDP. In the APDP mechanism, just the MPT table has 2048 entries, each one around 131 bits of width, what adds up to a total of nearly 33 Kbytes. This would imply about 44 times more memory than what we propose.

	`applu`	`apsi`	`hydro2d`	`swim`	`tomcatv`
LoadStore	1.00	1.00	1.00	1.00	1.00
APDP sel.	1.14	1.14	1.24	1.21	1.21
APDP sq.	1.14	1.15	1.24	1.21	1.22
Bypassing	0.90	0.74	0.82	0.72	0.89
Proposal	0.98	0.85	0.95	0.83	0.94

Table 1. Relative memory traffic in a 4-way machine with memory 2 ports

Besides this size constraints, each mechanism relies on a circuitry that in some cases may affect processor frequencies, degrading overall performance. In our proposal, the circuitry of the renaming has been shown not to affect cycle time [8], and the circuitry for the decoupled prefetching does not seem to be too complex, noted that the calculation and issue of these prefetches may be delayed a couple of cycles. Something similar happens with the lookup of the tables in APDP and the following prefetch. Nevertheless, APDP relies on a recovery mechanism with significant complexity [11], that could potentially affect cycle time.

In table 1 we show a relative measure of memory traffic incurred by each of the different mechanisms. All numbers are relative to those of the LoadStore base machine. One interesting thing to notice is the important reduction in memory traffic that our *memory instruction bypassing* mechanism gets, up to a 27.4% in `swim`. This reduction is due to the fact that the *memory instruction bypassing* mechanism *groups* memory accesses if they belong to the same cache line, thus reducing the need for requesting several times data on the same cache line. The number of data items brought from L1 cache is the same, although the number of requests is minimised. Adding our decoupled prefetcher on top of this increases the overall memory traffic, but always leaving it under that of the normal base architecture. On the contrary, APDP needs a lot more memory traffic to support prefetching and speculation. In average, our decoupled mechanism produces 35% less memory traffic than APDP. Knowing the relative cost in power of memory operations, these numbers show that our mechanism is also more power aware than APDP.

All these cost results make our proposal cost effective. Taking this into consideration, our proposal appears to be a cost-effective solution to perform de-

coupled prefetching on top of *memory instruction bypassing*. At much lower cost our proposal is better than a realistic APDP (with squash recovery) and close to a highly complex APDP (with selective recovery). Our decoupled prefetching mechanism would get better with better compiler technology and we are confident it would close the gap with respect to the complex implementation of APDP.

5 Conclusions

In this paper we have presented a cost effective decoupled prefetcher driven by the compiler insertion of special prefetching instructions. We have achieved very good performance results while keeping the cost low because the hardware mechanism is limited by what the compiler can predict. For a subset of SPECfp benchmarks we have achieved a 13% speed-up in average over a version with software prefetching and a 43% speed-up over a version without software prefetching.

The insertion of new *pref/load* instructions allows the hardware to bypass the execution of successive *load* instructions, allowing subsequent instructions to have their source operands ready earlier. As this bypassing phase is not speculative, it needs no recovery action. This property in conjunction with the prefetching to L1 done by a compiler-controlled simple stride based hardware prefetcher, makes our mechanism very easy to implement.

Besides this main contribution, we have also discussed in detail the concept of *memory instruction bypassing* and its relation with prefetching. We have shown that *memory instruction bypassing* combined with both software and hardware prefetching, can effectively increase performance, with the advantage that it implies no recovery action since it is non-speculative.

We have also analysed a previously proposed mechanism, APDP [2], with respect to its prefetching and *memory instruction bypassing* implementations. APDPs motivation for improving performance came from a different angle, namely value prediction and speculation. We have analysed APDP sources of improvement and compared them with those in our mechanism. Our mechanism can achieve better performance than a realistic implementation of APDP with squash recovery, and is close to a much more complex APDP. We have also compared our mechanism in terms of cost: sizes of tables, complexity of the hardware involved, memory traffic, etc. Our mechanism utilises less than 40 times the amount of real estate than the tables that APDP use, has a minimal impact in the design of the processors pipeline and has in average a 35% less memory traffic than a realistic APDP. Moreover, as our mechanism relies on compiler technology, we expect that compiler improvements will eventually increase the potential improvements of our decoupled prefetching and *memory instruction bypassing* mechanism, reaching those of APDP.

6 Acknowledgements

This work has been supported by the Spanish Ministry of Science and Technology and the European Union FEDER contract TIC2001-0995-C02-01, by the NSF Grant CCR-0072948 and the CEPBA.

References

[1] T. Chen and J. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, May 1995.

[2] J. González and A. González. Speculative execution via address prediction and data prefetching. *1997 International Conference on Supercomputing*, July 1997.

[3] J. Hennessy and D. Patterson. *Computer Architecture. A Quantitative Approach. Second Edition.* Morgan Kaufmann Publishers, San Francisco, 1996.

[4] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *17th Annual Symposium on Computer Architecture*, May 1990.

[5] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[6] S. Mahlke, W. Chen, W. Hwu, B. Rau, and M. Schlanker. Sentinel scheduling for vliw and superscalar processors. *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[7] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[8] D. Ortega, M. Valero, and E. Ayguadé. A novel renaming mechanism that boosts software prefetching. *2001 International Conference on Supercomputing*, June 2001.

[9] S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, 1996.

[10] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Classifying load and store instructions for memory renaming. *1999 International Conference on Supercomputing*, June 1999.

[11] H. Zhou, C. Fu, E. Rotenberg, and T. Conte. A study of value speculative execution and misspeculation recovery in superscalar microprocessors. Research report, North Carolina State University, January 2001.

[12] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. *International Symposium on Computer Architecture*, June 2001.