A Novel Renaming Mechanism that boosts Software Prefetching

Daniel Ortega, Mateo Valero and Eduard Ayguadé
Departamento de Arquitectura de Computadores,
Universidad Politécnica de Cataluña – Barcelona, Spain
{dortega,mateo,eduard}@ac.upc.es

Abstract

The detection and correct handling of data and control dependencies constitutes one of the biggest issues to expose ILP in current architectures. The ever increasing memory latencies and working space of programmes are making prefetching techniques crucial for the attainment of a high sustained performance. Software prefetching allows the compiler to use the information discovered at compile–time about the programme to effectively bring the data the programme will need beforehand, thus hiding all or part of the latency from main memory.

On the other hand, renaming is a technique that allows the hardware to break register name dependencies, thus exposing more parallelism to the hardware. In this paper we will present a new compiler–directed renaming mechanism focused on prefetch instructions. The compiler informs the hardware on the association of prefetch and load instructions, thus making it possible for the hardware to convert non–binding prefetch into binding prefetch, without any of the compile–time limitations this other kind of prefetching may have.

The mechanism can be implemented at a very low cost in terms of area and we believe it will not impact on cycle time. The research presented in this paper is at a first stage, nevertheless, the results achieved for numerical applications is quite impressive, ranging from a 5% to a 22% of speed–up for programmes where prefetching improves performance, without being detrimental to the rest.

1. Introduction

Most current high performance microprocessors rely heavily on the superscalar out of order paradigm in order to exploit instruction level parallelism (ILP). The basic idea behind this paradigm is the execution of several instructions per cycle in parallel, while maintaining the expected semantic order of the programme. To assure this, microprocessors must only execute instructions from the instruction stream when they are free of dependencies.

Instruction dependencies can be classified into three categories [10]: control dependencies, data dependencies and name dependencies. Control dependencies appear due to the presence of conditional branches in the code. In order to exploit parallelism among more instructions, the outcome of the branch, i.e. the direction it is going to take, must be predicted [26]. The nature of speculation implies
taking recovery actions when the execution of the branch produces a different output than the predicted one.

Data dependencies appear when an instruction consumes the value produced by another instruction and are inherent to the algorithm represented by the code. Some research has stated the predictability of the output of certain instructions [16] and have proposed breaking data dependencies by speculating the outcome of predictable instructions based on their previous executions.

Name dependencies are not implied by the algorithm as the two types before, but by the way the algorithm is implemented, being therefore also called false dependencies. They appear due to the reuse of storage locations, namely registers or memory locations. The fact that the same storage location is used to pass information between different instructions, makes it impossible for them to execute in parallel. Renaming techniques [29] assign different physical storage locations dynamically to the same logical identifier, thus breaking the dependency. Register renaming is closely related with several other microarchitectural techniques, such as the implementation of precise interrupts [27] or control speculation. New ways of implementing register renaming are still being produced, such as those present in [19, 5, 13] to cite only a few.

Memory renaming presents a tougher problem than that of register renaming. Although it is impossible for the processor to implement a deterministic memory renaming technique, various techniques have been proposed that implement speculative memory renaming through registers to increase performance [18, 30].

Renaming is basically a dynamic mechanism, implemented in the hardware to overcome limitations on how the code is presented to it. Very little work has been done on how the compiler can help the hardware in this process, such as in [28, 22]. We have the strong belief that a higher communication, in terms of semantics, is useful between the compiler and the hardware. In this paper we exploit this principle by making the compiler a vital part in the process of renaming. The mechanism presented in this paper is not directly related with name dependencies, but with the way in which these dependencies are solved by the hardware, in other words, it is related with the mechanism of register renaming.

Dependencies are not the only problem a processor must stand up to. Memory access latencies represent a major problem too. Memory speed has not increased at the rate the processor has and therefore, accesses to main memory represent higher and higher costs. Different techniques have been proposed to tolerate or hide these latencies, such as caches [25] or prefetching. Non-blocking caches [15] capture the locality of accesses and by their non-blocking nature, allow a certain degree of both latency tolerance and concurrence. Prefetching consists on the bringing of a piece of data from memory, before it is requested by the processor, thus hiding all or part of its latency. Prefetching can be classified to be hardware or software, depending on who is responsible for its accomplishment. Hardware prefetching is started by the processor (or by the memory hierarchy) in a speculative way. Its aim is to detect patterns of access to the memory and bring the potential requests to a certain cache level. Several mechanisms have been proposed, differing one from another in what is brought to the L1 cache and how. The literature on this topic is very vast [12, 4, 1, 11].

In software prefetching, it is the programmer or compiler responsible of deciding when and what is going to be brought to the cache or to a register. Most research on software prefetching has been devoted to regular access patterns as those found in numerical applications [7, 3, 14, 8, 20, 21], but lately there has also been research that tries to detect and prefetch recursive data structures [17, 23] which appear in non-numerical applications. Software prefetching can be classified to be binding or non-binding. Binding prefetch advances the load instruction so that it executes earlier bringing a piece of data from
memory to a register, thus the binding name. The load instruction is inserted preferably at a distance that allows the data brought by it to be in the register file by the moment it is needed. If the compiler knows that more than one element from the cache line is going to be needed, it will benefit from this locality by advancing one load, instead of all of them. This load will bring the whole line to the cache and the requested piece of data to the register file. The rest of the loads will find their pieces of data in the cache, and will not have to pay the whole latency to main memory (we are supposing now that the line of cache will not be thrown away in the meantime).

Non-binding prefetch uses a different approach, while still being a software directed one. Non-binding prefetch needs a new type of instruction, the prefetch instruction, that tells the hardware what pieces of data it must fetch. The prefetch instruction is placed in the code to be executed before the multiple possible loads that will consume the line of cache brought by it. Its non-binding nature comes from the fact that this instruction only brings data to the cache, not to the register file. As prefetch instructions do not modify the state of the machine, an easy implementation of them (that of course does not benefit from prefetching) may be to change them to no-ops at runtime.

Binding and non-binding prefetch, although being similar mechanisms, have subtle differences. The main one is the introduction of the new instruction in the flow. Non-binding prefetch will need the execution of one new instruction called pref for each line of cache that is going to be prefetched, while binding prefetch does this advancing one load. The advancement of this load stretches the life-time of the logical register that this load defines, from where the binding prefetch load is inserted until the use of the data. Having such a long life-time in a register puts more pressure in the register allocator part of the compiler. There exists no exact rule about when to apply binding or non-binding prefetch. Binding prefetch uses less instructions but may imply spilling registers due to the increased register pressure. Compilers use different heuristics to decide which technique use, or how to mix them conveniently.

One of the things researchers have done in computer architecture throughout the years is sharing components of different mechanisms to attack known problems in a fresh and interesting way, as can be read in the papers on value prediction for prefetching [6] or branch speculation [9], when register renaming techniques have been applied to disambiguate loads and stores [18] or even when prefetching techniques have been applied to TLB preloading [24]. Our mechanism is also based in this sharing principle. We benefit from data prefetching by combining this mechanism with register renaming. Our goal is to convert all non-binding prefetches into binding, but without any of the limitations that binding prefetch imposes to the compiler. We allow non-binding prefetches to allocate physical registers and bring data into them, boosting software prefetching mechanisms, without increasing the pressure on the register allocator nor having to spill code in the process. Several enhancements can be applied to this basic idea and will be further analysed.

The structure of the paper is as follows. In section 2 we present our renaming scheme for boosting software prefetching. Then in section 3 we discuss certain implementation issues of the prefetching mechanism. In sections 4 and 5 we present the simulation environment, the experiments carried on and what can be inferred from them. Finally in section 6 we will conclude the paper.

2. Novel Renaming Mechanism

In this section we are going to present our novel renaming mechanism in an incremental way. We will do this with the help of a simple example that will eventually cover all the complexity of the mechanism. We will start showing the motivation behind the new renaming scheme and its relation with software
prefetching.

2.1. Motivation for Renaming Prefetchs

In many algorithms the compiler can effectively deduce the pattern of access to a specific data structure (mainly matrices or vectors), and therefore it can introduce prefetch instructions (either loads or prefs, depending on the binding or not binding nature of the mechanism selected) that fetch a particular piece of data prior to its use. This kind of prefetching is extensively used in conjunction with software pipelining.

Binding prefetch does not need the use of the extra prefetch instruction, and therefore it should seem a better solution than non-binding prefetch. Nevertheless, binding prefetch defines a register. Since the load instruction inserted by the binding prefetch is normally scheduled at a miss distance from the consumer of the data, the life-time of the register that allocates the data grows relative to this miss latency, thus increasing the pressure in logical registers (with respect to the compiler) and maybe causing data to be spilled to the stack. An intelligent combination of both types of software prefetching is very important to achieve high performance, specially in software pipelined loops.

![Figure 1. Time-line of a load instruction and the use of the data brought](image1)

The main idea behind software prefetching is the decoupling of the access to data. When the compiler needs to use a piece of data from memory, it must insert a load instruction that fetches it, as shown in figure 1. The load instruction must be fetched and decoded taking this a fixed number of cycles to conclude. Then the address of the data must be computed and subsequently, it must be disambiguated with the rest of the pending memory stores, to assure correctness. These two latter phases take a non-deterministic amount of time, denoted in the figure by a dashed line. The next phase consists of sending the request for the piece of data to the memory and waiting for the data to arrive to the processor. Finally, the data is written back to the register file, becoming available (sometimes it may be available to active instructions in the window at the same moment of writing it to the register file due to bypasses, as has been assumed in figure 1).

![Figure 2. Time-line of a pref instruction, the load and the use of the data brought](image2)

In average, the biggest non-deterministic part of the load instruction is the fetching of the data [2]. Since memory latency of a fetch depends on the exact position of the data in the memory hierarchy, this phase may vary from one or a couple of cycles, hitting in L1 cache, to tens or even hundreds of cycles if...
the data must be brought from main memory. Non–binding prefetch addresses this problem by inserting a prefetch instruction that executes part of the work done by the load, namely it brings the piece of data from where it is located in the memory hierarchy to the L1. This can be seen in figure 2. We can see in this figure that the decoupling of the access to memory between the pref and the load instruction brings out some redundancy, for both must compute their addresses, wait for the availability of resources, etc. Nevertheless, the redundancy introduced pays its cost if the load instruction hits in cache. Notice that the dotted line in the load instruction is left as having non–deterministic duration. Unfortunately, nothing assures us that the data prefetched is not going to be thrown out of the cache due to cache pollution before it is requested by the load instruction or even that it will have already arrived by the moment we request it.

![Renamed prefetch-use time line](image)

**Figure 3. Time–line of our renaming scheme for a pair pref–load instructions**

Our mechanism tries to exploit the redundancy in the pair pref–load. The movement of the data from L1 to the register file is advanced to the execution of the prefetch instruction (figure 3), making the load instruction useless in terms of execution, and therefore we nullify it dynamically. Therefore the non–binding instruction acts as a binding one, bringing its piece of data to the register file, without putting pressure in the compiler over the logical register set. We will explain this with the example shown in figure 4.

![Code Example](image)

**Figure 4. Code Example**

Let us suppose that we have a predictable stride access, denoted by a [i] in the example in figure 4, inside a loop and that the compiler introduces a pref instruction ahead of the load instruction. For the purpose of our example we can assume that the distance between them is enough to assure that the data prefetched will be in L1 by the moment the load executes. The new method implies a change in the semantics of the pref instruction. Instead of only bringing the piece of data to the L1 cache, it will forward it to the register file, to an non–allocated register. Notice that when speaking of a register we mean a physical register, not a logical one. Afterwards, when the load instruction is decoded, its only commission would be to modify the renaming table so that any instruction that needs its logical destination register, uses the register allocated by the pref instruction.

5
To do all this, we need a new `pref` instruction that can communicate with its respective load counterpart, establishing an association between both instructions. The new `pref` instruction will be very similar to a `load` instruction in the sense that it will also define a destination register. The ISA should allow a new opcode with similar format to existing memory operations. By specifying the same destination register, we communicate the relationship among them to the hardware, as shown in the example in figure 5, where we can see how the renaming mechanism works. Suppose that there exists a valid definition for r1 in position (1), as can be seen in the right–hand side of figure 5, that associates this logical register with physical register f14.

<table>
<thead>
<tr>
<th></th>
<th>r1 &lt;- ...</th>
<th>r1 def f14</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>pref r1, a[i]</td>
<td>r1* def f15</td>
</tr>
<tr>
<td>3</td>
<td>r1 &lt;- r1 op r3</td>
<td>r1 def f16 r1 use f14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r3 &lt;- r1 op r4</td>
<td>r1 use f16</td>
</tr>
<tr>
<td>5</td>
<td>load r1, a[i]</td>
<td>r1* def f15  r1 def f15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r5 &lt;- r1 op 4</td>
<td>r1 use f15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loop branch</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Code Example and Renaming Mapping for r1

When the `pref` instruction (2) is decoded, it will introduce a special new mapping for r1, denoted as r1*, selecting a free physical register, lets say f15. The new `pref` instruction will not only bring the cache line to L1, but it will also load the data into register f15. This new entry does not invalidate the previous mapping for r1, as a normal register definition would do in conventional renaming schemes. Then in position (3), logical register r1 is at the same time used and redefined. A new mapping would be associated with it, lets say f16. When the instruction (4) in the example reaches the decoding phase, it will try to detect what mapping exists for r1, and it will see that f16 is the valid mapping, just as it would do in a conventional renaming scheme. Afterwards, when the `load` instruction (5) is decoded, its only task would be to convert the special mapping of r1, previously defined by the `pref` instruction to be f15, to be the normal association for r1. Any instruction after this load that uses r1 will be using physical register f15. Since nothing more is expected of the `load`, it is dynamically converted into a NOP instruction.

This scheme allows the data defined by the `load` to be accessible (if the memory hierarchy allows it to arrive) at the decoding phase of the `load`, i.e. removing the `load` from the critical path (see figure 3). It is as if the non–binding prefetch achieved certain characteristics of a binding one, such as the binding of the logical register, without imposing for this any restrictions to the register allocator of the compiler, i.e. without augmenting the life–time of the logical register, nevertheless we do increase the life–time of the physical allocated register.

### 2.2. Multi-element prefetching

With this simple explanation of the scheme, we can expect benefits from the fact of not having to execute the `load` and from having the data before (if the memory hierarchy allows it). Nevertheless, this technique does not finish here. As can be noted, the compiler does not introduce a prefetch instruction for every `load` instruction it wants to hit in cache, but for all those that fit in the same line of cache.
Assuming, without loss in generality, a line size of 32 bytes, i.e. 4 double words, we can assure that if the algorithm is going to access all four elements of a line, the compiler will only introduce one prefetch instruction to assert the 4 subsequent hits by the loads. The technique presented so far would only be good for one of all the loads, leaving room for improvement.

To continue with the example presented before, suppose that the pattern of access of the algorithm only touches positions 0, 1 and 3 of each 4 positions in the cache line. Current compilers would generate the kind of code shown in figure 6 (we will omit the instructions that are not useful for this explanation).

![Figure 6. Code Example for Multi-Element Prefetching](image)

Our method adapts to this new situation increasing the association between the pref and the set of loads that consume from it. The pref instruction must now have more information, i.e. which positions in the line are needed and some information about the loads that access them. The first piece of information is easy to provide. We introduce in the instruction a map of bits (as many as positions in the line, 4 in this example) that we have called line position bits or LPB. The information about which load instructions are going to be fed by this pref has been solved in an inexpensive way, without need to increase the fields of the instruction.

As we can not augment the instruction with all the possible logical registers defined by the load instructions due to size problems, we have chosen to implicitly pass this information. As before, we will only include one destination register, the rest of them will be defined implicitly in numeric order. The example presented before will be modified to appear like in figure 7.

![Figure 7. Code Example for Multi-Element Prefetching with LPB](image)

When the pref instruction executes, it will create three special mappings in the renaming table. One associated with logical register r1, in which the instruction will deposit the first element of the line;
another associated with logical register r2, in which it will deposit the second element of the line, and a
third one associated with logical register r3 in which it will deposit the fourth element of the line, since
the third one is not required to be loaded into the register file. If it happens that the third element is
not always used and therefore only loaded under certain conditions, it will be loaded by a completely
different load instruction, not interfering at all with the binding mechanism, but nonetheless, achieving
benefit from the prefetch, as it would do in an environment without this mechanism. When the selected
load instructions appear they will convert the special mappings to normal ones. Notice that the order of
the load instructions in the instruction stream is not relevant. We have chosen to disorder the loads on
purpose to stick out this point.

With this implicit method we have assured an inexpensive way of passing the information needed
to the hardware. Nonetheless, this is not the only way this can be done. Other possibilities exist and a
particular one should be selected with the final architecture in mind. This implicit association can always
be worked out, since starting with a particular register mapping, there always exists a permutation of the
register numbers that leaves us the desired mapping. Of course, certain permutations may not be selected
due to the semantics of registers in particular architectures, but this only adds a certain complexity to the
colouring phase of the scheduling. Note also that this implicit association can be modified to fulfil other
needs without loss of generality.

2.3. Prefetching Distance

One last thing remains to be said about this method before we analyse the binding case. Up to now
we have considered that between the dynamic execution of a prefetch instruction and its consuming
loads no other load to the same logical registers may appear. This is so because this other load would
confuse the renaming mechanism. In the simple examples presented so far, this was easy to achieve.
Any intervening loads should be made to define other registers not used by the loads involved in the
prefetch. However, this imposes a very hard restriction.

The problem with it comes from the fact that many compilers generate instruction patterns as the
example shown above, but with one important difference. Due to the latencies to main memory, and to
the lack of work to be done (sometimes unrolling must be limited to improve performance) the compiler
can not leave as many static instructions as it would like between the prefetch instruction and the loads.
A solution for this would be to leave a gap of dynamic instructions between the prefetch and the loads.
This means that in the first iteration of the loop, the pref instruction prefetches the data for the loads that
are going to be executed in the second iteration as can be seen in figure 8. When the loads of the first
iteration execute, they will receive the prefetches introduced by prefetch instructions in the header of the
loop. Then the second iteration executes: the pref instruction brings the data for the third iteration loads
and the loads consume the data brought by the previous iteration. This effect can not only occur at a
distance of one iteration, but of more.

This forces us to introduce another piece of information in the prefetch instruction, one that indicates
the distance to the consumer of the data. We will design this field as iteration distance bits (IDB). We
have seen that 3 bits suffice nearly all applications analysed, therefore we have chosen this field to be
this size, although the mechanism could be easily adjusted to use more or less bits.
2.4. Instruction formats

Up to here, the only instruction modified has been the prefetch instruction. We have designed a new prefetch instruction, similar to a load instruction, but with two more fields, counting for a total of a seven bit increase in our definition. We consider that such a new instruction is feasible of being introduced in existing ISAs, or in yet to design ones, but we have not analysed any particular ISA to show how this could be done. Just a hint: considering that the prefetch instruction may have one register source, a displacement field and a register destination, as some loads do in certain architectures, we may decrease the size of this displacement field in order to allocate the IDB and the LPB. Notice that the LPB acts as addressing mode, so less bits are necessary for the displacement. Having no direct restrictions in our research due to the implementation of our mechanism in a real architecture, we have developed no experiments with reductions in these fields. Our opinion is that these seven bits are feasible, although future research will be conducted to determine the effect of reducing them.

2.5. Applying the same idea to binding prefetch

The mechanism presented so far has limited its range of application for codes that make use of non–binding prefetch. Without the information inserted in the new prefetch instruction, we can not know beforehand which data to associate with which load. Non–binding prefetch is not used as extensively as we would want. Many times compilers use binding prefetch, i.e. they advance one of the loads do in certain architectures, we may decrease the size of this displacement field in order to allocate the IDB and the LPB. Notice that the LPB acts as addressing mode, so less bits are necessary for the displacement. Having no direct restrictions in our research due to the implementation of our mechanism in a real architecture, we have developed no experiments with reductions in these fields. Our opinion is that these seven bits are feasible, although future research will be conducted to determine the effect of reducing them.

The mechanism presented so far has limited its range of application for codes that make use of non–binding prefetch. Without the information inserted in the new prefetch instruction, we can not know beforehand which data to associate with which load. Non–binding prefetch is not used as extensively as we would want. Many times compilers use binding prefetch, i.e. they advance one of the loads do in certain architectures, we may decrease the size of this displacement field in order to allocate the IDB and the LPB. Notice that the LPB acts as addressing mode, so less bits are necessary for the displacement. Having no direct restrictions in our research due to the implementation of our mechanism in a real architecture, we have developed no experiments with reductions in these fields. Our opinion is that these seven bits are feasible, although future research will be conducted to determine the effect of reducing them.
To benefit also from binding prefetch, we have adapted the proposed mechanism. We have designed a new instruction called \textit{load+pref}, that would act the same way as a \textit{load} would do, but with the improvements exhibited by the \textit{pref} instruction. This new instruction substitutes the binding \textit{load} instruction and it also has the bit fields IDB and LPB. Its behaviour would be very similar to the \textit{pref} instruction, only that it would only generate special mappings for its successive registers according to the LPB, and a normal renaming for its destination register. The problems of fitting both the IDB and the LPB in this new instruction may be more acute than that of the new \textit{pref} instruction, bringing out more restrictions depending on the final architecture.

2.6. Binary Compatibility

One of the goals when designing this new renaming scheme was to make it as transparent as possible. We are capable of proving that this scheme will exhibit a good performance increment with a little cost associated in modern superscalar architectures, but we can not sustain this assertion for future configurations. Several techniques, such as the delay slot, considered a very intelligent solution to the problem they addressed, have resulted a dead weight for today designers. We did not want our technique to constrain by any means future designers, and that is why we have emphasised so much the design and semantics of the new instructions. Once introduced these two new instructions in a particular ISA, two possible classes of architectures can be implemented. The first class of architecture respects the new semantics introduced in this paper, while the other class, simply ignores the new semantics. In this one, the \textit{pref} instruction would be considered a normal \textit{pref} instruction, the \textit{load+pref} instruction a normal \textit{load} instruction, and all \textit{loads} will be executed as normally. This feature not only allows future designers to forget this mechanism if they want, but it also allows various implementations of the same ISA, ranging from the most complex to the simplest, maintaining binary compatibility.

3. Implementation issues

Many current techniques in microarchitecture propose mechanisms that comply with the following scheme: a \textit{learning} part of the mechanism collects information from the computation of the application, and another part, strongly speculative, tries to take benefit from it, accelerating certain computations that belong to the critical path, thus achieving an improvement in the performance of the processor. The improvement of these mechanisms is usually dependent on the size of the \textit{learning} tables used, and therefore, the cost in size and power consumption if often not negligible to the architect. Only well known techniques such as branch prediction are capable of standing implementation costs.

The proposed technique tries to benefit from compiler knowledge of the program structure, thus eliminating the need for dynamically \textit{learning}, and therefore the enormous tables associated with these techniques. The cost of implementing this technique is linear with the decoding width. It only needs a secondary renaming table where the special mappings created by \textit{pref} or \textit{load+pref} instructions reside. The organisation of these auxiliary table could be as is shown in figure 9.

Notice that the secondary register mapping table exhibits a width that the normal one does not have. This width in mappings is necessary for maintaining the several mapping due to the iteration distance. We have chosen to mark with one bit the entry point, i.e. where the next definition should be introduced by the next \textit{pref} or \textit{load+pref}, and with another bit the extraction point, i.e. the first position to be consumed by the \textit{loads}. This organisation presented here was chosen to simplify the understanding of
the process. Several other possibilities seem feasible of simplifying the whole structure.

Notice that the insertion in these tables can be done in the background, after the decoding has taken place. Nevertheless, the extraction from it should be done in the decoding phase, to exploit all the potential improvements.

In the next sections we will also show that the presence of prefetch instructions at a time in the code is not at all excessive but quite scarce. Therefore, another simplification of this table could be the limitation of the mechanism to allow just a subset of logical register to appear as destination registers in `pref` instructions, thus reducing the total size of this table.

Orthogonal to this fact, we can state that the secondary renaming table will never be part of the critical path of the renaming phase. The complexity of the renaming grows quadratically with the instruction fetch width, for the hardware must check the dependencies among the instructions fetched. The secondary renaming table only works with prefetch instructions that define mappings and specific `load` instructions that read them, so its working width should be smaller than the fetch one. Besides this point, the probability of having a dependency between a `pref` and a `load` in the same fetch block is nearly zero, the compiler should never introduce a `pref` instruction and following it, a consuming `load` (the `load` will not be able to benefit from it). Assuming this, the checking of these dependencies may be brought out of the critical path (we can also rely on hardware traps such as those present in Alpha to detect if this happens and re–execute the `load`). With these arguments, it is clear that there exists a simpler implementation (in terms of critical path time) of the secondary renaming table than that of the primary one.

4. Simulation environment

The technique presented does not only influence the overall performance that a certain architecture could have, but it also influences compilation techniques in order to achieve the best out of it. To measure all the possible effects that this technique could have, we should introduce it in a efficient compiler and measure it under different architectural configurations. Unfortunately for our research purposes, we do not have at our disposal such an environment, the academic compilers available to us (i.e. the only ones which we can modify) are not equipped with the prefetching technology necessary to produce as efficient code as certain product compilers can. As a first step in our research, we have decided to study the effect that our technique could have under current product compilers. In a near future we would like to introduce the mechanism in the decision maker of a versatile academic compiler, and evaluate other
aspects, such as more aggressive prefetching strategies due to the reduced logical register pressure.

The compiler framework selected for our experiments is the MIPSpro compiler (Version 7.30). We have compiled a subset of the SPECfp95 benchmarks using the Fortran 77 front end with different compilation parameters, and then we have analysed and annotated the produced code to incorporate the changes that the mechanism proposes. In any way we have re-scheduled the code, nor changed any other instruction than what the mechanism states.

Of course, this compiler framework assumes that the underneath hardware has certain characteristics, and therefore the code produced is focused for such a hardware. This does not only mean that the compiler will not take advantage of our mechanism as much as it could, but that the target product will be focused for an architecture very similar to that present in the SGI, a MIPS R10K. This limits us in the architectural configurations chosen for the simulation. Simulating these programmes for architectures with dissimilar configuration parameters, although possible under our simulation environment, would yield unrealistic performance benefits.

For these reasons we have chosen to limit the configuration parameters to be very similar to those present in a MIPS R10K. The simulated architecture fetches, issues and retires a maximum of 4 instructions per cycle. The execution model is a superscalar out of order processor with an instruction window of 64 entries. The branch speculation model uses a 2K positions BTB with 2 bit saturation counters. The fetch engine can fetch up until the first taken branch or maximum fetch width. The latencies and quantity of all the functional units are very similar to those present in the target architecture. The sizes and configuration of the caches are very similar to those present in our machines, namely 32 Kbytes direct mapped L1 cache with lines of 32 bytes and a 2 Mbytes, 4 way set associative L2 cache with lines of 64 bytes.

The only deviations in the experiments lie in the memory latencies. We have chosen to experiment three sets of latencies, a first one considered normal, such as 1 cycle to L1, 11 cycles to L2 and 51 cycles to main memory; a second one considered perfect, 1 cycle to main memory; and a third one considered future, 3 cycles to L1, 23 cycles to L2 and 123 cycles to main memory. The first latency model is very similar to the current one in our architecture, the second one allows us to analyse different effects under a perfect memory configuration, and the third one allows us to see how our mechanism could work in the future, when we expect latencies to grow as shown. Of course, if latencies grow, the compiler should know it and could do something to skirt round its negative effects. Moreover, if the compiler knew about this mechanism, it could produce better code than the one we are using to evaluate it. Therefore, these results should be seen as a lower bound of the potentials of the new renaming technique.

All the applications selected for the study were chosen from the SPECfp95 suite. Three different executables were compiled for each application and run under two different architectures. The first architecture models current architectures, i.e. without the mechanism proposed, which we will call L1 binding architecture. The "L1 binding" name comes from the fact that the pieces of information are brought to the L1 cache (of course, all except for the presence of the primary data in binding loads). The second architecture models our mechanism and we have called it register binding architecture, for it is in this model in which all pieces of data are brought directly to the register file. The three different executables differ one from each other in the parameters passed to the compiler: the first one was compiled with normal optimisations (-O2 in our environment); the second one, using fairly common heuristics for the insertion of prefetching (-O3 -prefetch=1); the last one was compiled using the most aggressive heuristics (-O3 -prefetch=2). We must notice that the use of -O3 in our architecture may imply a slowdown in the performance. We will see that some benchmarks exhibit this nature, which can
not be only attributed to the insertion of prefetching, but to other aggressive optimisations.

5. Results

In this section we will present the results from the experiments and we will infer certain characteristics from the programmes analysed. The applications selected for the study are the following: applu, apsi, hydro2d, swim and tomcatv. All of them were selected due to their amenability to simulation. The other SPECfp95 programmes have resulted in very long latency simulations and thus were discarded from the experiment. If accepted, the final version will include results for more programmes.

One of the things the novel renaming mechanism does is decouple the renaming of a physical register in the cases of pref and load pairs or load+pref and load pair. Our mechanism extends the use of physical registers from the prefetch operations until the loads (of course the physical registers continue to live after the decoding of the load, but this happens also in a normal renaming scheme). One of the things we are more interested in is the increase in the number of live registers that our mechanism may cause.

Several architectures are built with enough physical registers so as not to stall the pipeline due to a lack of them. The maximum number of physical registers needed is equal to the number of logical registers plus one register for each entry in the instruction window. Thus in the architecture analysed with 64 entries in the window, we would need 96 registers in order not to stall the pipeline at all. This maximum number of registers would only be needed in the case of having every entry of the window define a register (not all instructions do define a register) and having the window full. Several papers have stated that the number of physical registers live is smaller than this maximum. In fact, one of the motivations of our mechanism initially was to exploit these extra registers.

In our mechanism, a maximum number of registers can also be computed. As every instruction in the window can define up to 4 registers (assuming that all of them are pref instructions bringing four pieces of data), the maximum would be four times the size of the window plus the number of logical registers. In our analysed architecture this would be $64 \times 4 + 32$, a total of 288. This number is by far too big to be considered to implement. Besides, we know that they would never be used all at a time, for the distribution of prefetch instruction in the code is not too dense. Therefore, our intention at this point was to measure the increase in the number of physical registers that our mechanism is going to have. We have done this on a programme per programme basis, analysing the average maximum number of extra registers needed by each of them. The results are the following: applu 8 registers, apsi 27, hydro2d 11, swim 8 and tomcatv 15.

Notice that some of the programmes use a very small number of extra registers. This is so because the compiler has either not inserted too much prefetching or the distance between pref and load is very small. In others, such as apsi, we see a considerable increase in the number of physical registers, due to the opposite effect. In the rest of the experiments we have considered the existence of this maximum quantity of registers for each benchmark. In future research, when we dispose of a compiler environment in which we will have introduced our mechanism, we will analyse different heuristics that will insert more or less prefetching depending on the number of physical registers needed for the execution of the different parts of the programmes. This way we will be able of more efficiently exploiting the physical register file.

Every programme has been simulated five times, as can be seen in figure 10 with different combinations of compilation and hardware techniques. All the bars represent relative speed-ups with respect to the first bar, which represents the compiler optimised code without any prefetching, running in the L1
binding architecture. The second bar represents the speed–up obtained when the compiler introduces normal software prefetching techniques but still simulated in the L1 binding machine. The third bar represents the speed–up of this code with prefetching, simulated in the register binding architecture. The last two bars represent the same as the second and third, but with a code with aggressive prefetching.

As it can be noted from figure 10, under normal memory latency constraints, we can see that not all the programmes exhibit the same nature with respect to prefetching. Some programmes achieve slow–downs, as can be seen in apsi and swim, when prefetching is applied to them. We must also notice that prefetching is inserted under -O3 compiler flag in the MIPSPro Compiler suite, and that this mode of optimisation may introduce certain optimisations (not necessarily prefetching) that although considered beneficial may in some cases hurt performance. Other programmes, such as applu and tomcatv exhibit a considerable amount of speed–up ranging from a 10% to a 20% approximately. Our last programme analysed, hydro2d exhibits an intermediate nature, achieving an insignificant speed–up.

In figure 10 we can see the effects that our mechanism provides to code with prefetching are very surprising. Not only does our mechanism almost always improves considerably the performance of a L1 binding machine with respect to the same programme (the relative speed–ups range from 0.3% in the case of swim to up to 22.3% in apsi). We have analysed this relative speed–up in apsi, which surpasses the approx. 7% of both hydro and applu, and we may give hints to its cause. The overhead imposed by the prefetching in apsi does not pay its cost in a L1 binding machine, while in our machine, due to the bypassing effect of the register binding mechanism, we can improve substantially the performance, even surpassing the base line. Notice also that our mechanism does not always improve performance. In swim we can see that prefetching does not really affect at all performance, and our mechanism can not improve at all. This application deserves future analysis to understand fully its behaviour.

The first thing that we can conclude from figure 10 is that our mechanism produces approximately the same relative speed–up with respect to simple and aggressive compiler heuristics. This is due to the fact that the prefetching we are looking for is usually inserted using the simplest prefetching heuristics. Of course, in -O3 with -prefetch=2 additional prefetching is inserted in the code, but it usually refers to more bizarre structures, such as speculative prefetching or others, from which our mechanism does not benefit.

An interesting effect to analyse from our mechanism is its relation with memory latencies. Assuming it is a somehow prefetching related mechanism, it should not benefit at all in a perfect memory environment. This is not at all true as can be seen in figure 11. In fact, it behaves completely the opposite, when the memory is perfect (which can be understood as an approximation to perfect prefetching) the
speed–ups obtained grow from those seen in figure 10. This effect can be easily explained. Our mechanism does not solely benefit from prefetching the data, the L1 binding architecture also does that. What it really benefits from is from the bypassing of the data in the execution of the loads, something similar to changing all non–binding prefetch instructions into binding ones. This happens after every prefetch, but is only beneficial if the prefetch has arrived, which always happens in a perfect memory environment. In a normal environment, the data may not always arrive on time, and therefore the bypassing sees diminished its accelerating effect.

This explanation does not mean at all that our mechanism will not produce benefit in high latency memory hierarchies, as the one for future machines presented in figure 12. Of course the beneficial effects that the register binding has, appear diminished, but in a very small relation to that of normal memory configurations. This fact asserts that the effective load of the data into the register files is part of the critical path, and will still be in future memory configurations. We must point out again that the results for the future memory latencies have been gathered using the same compiled programmes as the other two. Future research should refine this analysis with different compilations for different architectures. Nevertheless, this results appear as a lower bound to the potential benefits.

Gathering the information presented in the three graphs, one can see that our mechanism is loosely related with the latency of the memories. Nevertheless, we can state that the better the prefetching becomes, the better performance increase we should expect from the register binding mechanism, what encourages us to further investigate into this mechanism.

We have also analysed other potential sources of the benefit, not only the bypassing effect. In figure 13 we see the memory hit ratio with respect to the total amount of memory accesses. The five bars have been given the following names: L1B02, which stands for L1 binding with 02; L1BP1, L1 binding with...
prefetch 1; RegBP1, register binding with prefetch 1; L1BP2 L1 binding with prefetch 2; and RegBP2 register binding with prefetch 2. We can see that these programmes exhibit a good behaviour in L1 cache, but what appears more astonishing at first sight is the decline in the hit ratios in our register binding architectures. This may seem contrary to logic, that a decrease in hit ratio may produce better performance, but we have to take into account that the register binding architecture nullifies a lot of loads that would have become hit accesses (as they are part of a prefetch mechanism) because their data is already in the register file or in its way to it.

### Table 1. Distribution of load memory operations of the memory hierarchy for normal memory latencies (1-11-51)

<table>
<thead>
<tr>
<th></th>
<th>applu L1BP1 &amp; RegBP1</th>
<th>tomcatv L1BP1 &amp; RegBP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>82.45% 71.20%</td>
<td>91.32% 75.16%</td>
</tr>
<tr>
<td>Prefs</td>
<td>17.55% 17.55%</td>
<td>8.68%  8.68%</td>
</tr>
<tr>
<td>Null Loads by Prefs</td>
<td>6.70%</td>
<td>6.70%  0.02%</td>
</tr>
<tr>
<td>Null Loads by Load+Prefs</td>
<td>2.72%</td>
<td>2.72%  9.39%</td>
</tr>
</tbody>
</table>

This nullifying effect is explained in table 5. The table must be understood as follows. We have divided the memory access instructions (stores do not count for this experiment) into different classes: loads, prefs and load+prefs, each represented by one of the rows. The numbers that appear are all relative to the total dynamic execution counts. The last two rows contain memory instructions that were nullified, i.e. that never accessed the memory hierarchy. Notice that in the L1 binding architecture, no loads are considered Load+Pref, and no Load instructions are nullified.

We have chosen two of the applications of our set of benchmarks, applu and tomcatv, as representatives of the whole set of benchmarks. The first one, applu shows a considerable percentage, nearly 7% of load instructions that are nullified by pref instructions, while the second one, tomcatv presents a negligible amount of nullified instructions of this class. These numbers can be correlated with the performance improvement seen in the programmes: the insertion of prefetch instructions in applu provides a considerable amount of improvement (see the difference between the first two bar in figure 10), while in tomcatv this is not so great. On the other side, tomcatv has a great percentage of
load instructions that have been changed into load+pref, producing a considerable amount, over 9%, of load instructions nullified by them. On total, both applications show more or less the same reduction in memory accesses, which can be correlated with the relative improvement (approx 5-6%) shown in figure 10.

6. Conclusions and Future Work

In this paper we have presented a novel renaming mechanism, a combined compiler–hardware approach, that converts non–binding prefetches into binding ones. Current compilers are limited in the amount of binding prefetch they can insert in codes due to the restricted available logical registers. Nevertheless, when compilers know what data is going to be accessed, they try to bring it as near as possible to the register file, usually to the L1 cache. They do so either using prefetch instructions or exploiting the fact that caches implicitly prefetch all the elements of a line when any of them is accessed.

The proposed mechanism passes all this information to a specialised renaming hardware, allowing it to convert non–binding accesses to binding. Subsequent load instructions are nullified, and the data already binded in the register file is bypassed to its consumers as soon as they need them. The benefits do not only come from this bypass, but from a reduction in memory accesses and in the use of certain functional units, two characteristics that also imply a reduction in power consumption, a critical issue in current architectures.

The overall performance improvement produced by the mechanism has been shown to correlate with prefetching technology, showing up to 22% relative improvement in numerical programmes. Even with future memory latencies, we have shown a considerable performance benefit, and we have established the fact that loading data from L1 cache is a critical issue. The criticality of this fact will presumably increase with the depths of current and future pipelines. Two important properties of our mechanism are its low cost in hardware, and its transparency with respect to a potential future disposal of the whole mechanism. We are also quite confident that the mechanism will not impose a reduction in the frequency of the processor, as its complexity is lower than conventional renaming schemes.

This paper constitutes the first stage in what we hope will be a fruitful research. This paper attempts to present a novel and interesting idea, and therefore opens new paths which we expect to cover with future work. Conceptually we have divided the future work in hardware and compiler analysis. With respect to the hardware we are interested in developing new experiments that will measure new characteristics of the mechanism, its limitations and restrictions with a broader range of applications. The aim of this future experiments would be not only to determine the characteristics of the mechanism but primarily to understand more deeply common applications (and specifically their bottlenecks).

With respect to the compiler, our intention is to port our mechanism to a more versatile compiler environment and measure the potential benefits it may produce. Notice that our present results imply a lower bound in the performance benefits. If the compiler becomes aware of our renaming mechanism, more opportunities to improve performance appear.

In a near future we expect to analyse the effect that our mechanism has in non–numerical applications. As our mechanism relies heavily on prefetching technology, we expect that its benefits will increase with new advances in prefetching. Nevertheless, we are eager to analyse the mechanism in non–numerical applications with current published prefetching mechanisms such as those presented in [17, 23].
References


