International Journal of Parallel Programming, Vol. 32, No. 3, June 2004 (© 2004)

Dynamic Memory Instruction Bypassing

Daniel Ortega,[1] Mateo Valero,[2] and Eduard Ayguadé[2]

Received December 1, 2003; revised January 12, 2004; accepted February 19, 2004

Reducing the latency of *load* instructions is among the most crucial aspects to achieve high performance for current and future microarchitectures. Deep pipelining impacts load-to-use latency even for loads that hit in cache. In this paper we present a dynamic mechanism which detects relations between *address producing instructions* and the *loads* that consume these addresses and uses this information to access data before the *load* is even fetched from the I-Cache. This mechanism is not intended to prefetch from outside the chip but to move data from L1 and L2 silently and ahead of time into the register file, allowing the bypassing of the *load* instruction (hence the name). An average performance improvement of 22.24% is achieved in the `SPECint95` benchmarks.

KEY WORDS: Prefetching; memory bypassing.

1. INTRODUCTION

Bridging the *Memory Wall*[1] is one of the most important goals in Supercomputing design and even in normal microarchitecture development. The increasing distance to main memory is becoming the dominant bottleneck for a huge amount of applications. This makes techniques to overcome this hurdle, such as caches and prefetching, a must in high performance computers.

Most current microprocessors follow the load/store principle, i.e. all operations work with registers but those who move data from registers to

[1]Barcelona Research Office, Hewlett Packard Laboratories, +34 935821300, Barcelona, Spain. E-mail: daniel.ortega@hp.com
[2]Depto. de Arquitectura de Computadores, Universidad Politécnica de Cataluña, +34 934017001, Barcelona, Spain. E-mail: {mateo,eduard}@ac.upc.es

0885-7458/04/0600-0199/0

memory and back, namely loads and stores.[3] This overall design principle allows for a simpler design, since instructions are more homogeneous in their operands, but on the other hand decreases knowledge of the computation, i.e. not knowing that a particular register contains a memory address inhibits the microprocessor to speculate on its use and prefetch it. Fortunately, part of this knowledge can still be gathered, specially by tracking memory operations and tracking the sources of the data.

Several prefetching techniques are oriented at detecting patterns and bringing data closer to the processor. In many cases this implies L1 or L2. In these scenarios memory operations must still exist to finish the trip between the on-chip caches and the register file. Our main objective with the technique discussed in this paper is the elimination of memory instructions from the critical path of the application. Achieving this would be equivalent to stating that every piece of data lives in registers, that no instruction shall wait for a piece of data coming from memory. This can be done in several ways depending on the different scenarios. Ortega *et al.*[2,3] present memory instruction bypassing techniques oriented at numerical applications where a combinational effort from the compiler and the hardware helped to shorten this gap. In this paper we are going to discuss a dynamic approach oriented for non-numerical applications.

We focus entirely on integer applications, since these are much harder for compilers to analyse. Aliasing effects, irregularities in the code and strange data access patterns complicate the potential benefits that compilers can extract from the code. As integer applications represent a very important part of computation in high performance systems, solutions tailored to their needs are necessary. These solutions normally come from the dynamic side alone, from the microarchitecture. We will try to illustrate this with examples.

On the left side of Fig. 1 we can see a small dependence graph of a synthetic kernel. The kernel itself consists on a *load* instruction which brings a piece of data from memory, two *ALU* operations which use this piece of data and operate on it with two constants, and another *ALU* operation, dependent on these two, which takes the output of both and computes a value that is stored in memory. This value is also fed back to the first *load* dependent on it.

On the right side of Fig. 1 we see a potential dynamic schedule that an out-of-order superscalar processor may derive for the code. Let us suppose that this kernel belongs to the critical path of our application, i.e. any shortening of its execution will shorten overall execution. Under

[3]IA32 family of microprocessors was one of the few not following this principle up until the core was redesigned to behave like a RISC core.

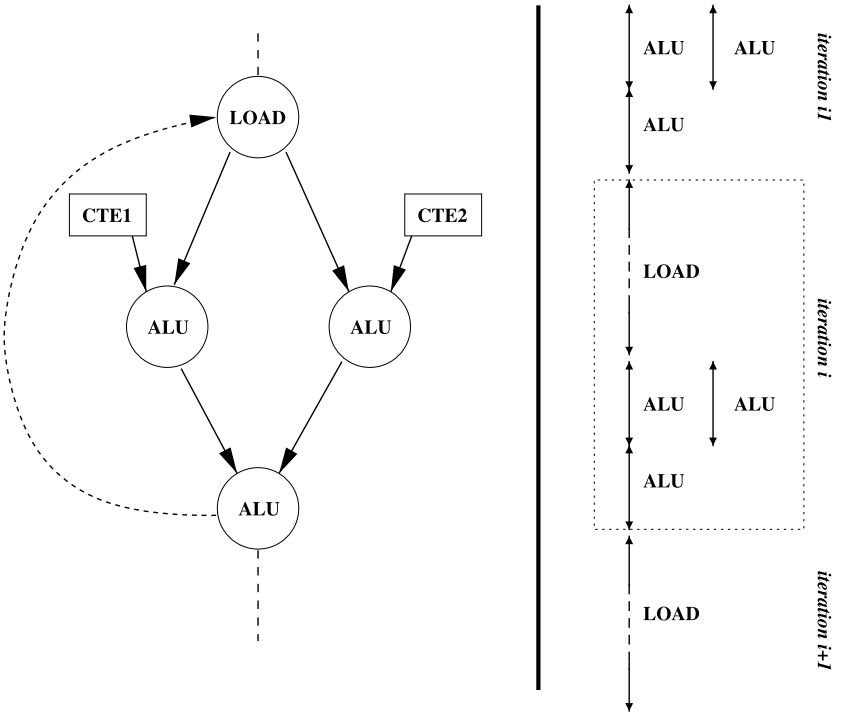

Fig. 1. Kernel example.

certain circumstances, the compiler can reduce the critical path by exposing more parallelism, perhaps by unrolling and software pipelining the loop. Nevertheless, the backwards dependence from the last *ALU* operation to the *load* disallows this kind of static optimisation. Another way of shortening the critical path is to shorten the latency of any operation involved in it.

Shortening the execution of a *load* operation usually means prefetch, i.e. bringing in advance the piece of data from memory to the firsts levels of cache so that the execution of the *load* hits in cache. Surely, if the piece of data is not in the cache memories, the *load* will be dominant in this critical path, making this critical path expand over a hundred cycles. In this scenario, prefetching from memory is surely the best technique. If we shorten these hundred cycles to a few tens (hitting in L2) or to a few cycles (hitting in L1) the kernel will execute much faster. But what if the piece of data is in the L1 cache? Can we still reduce this critical path?

The execution of the *load* in a current out-of-order superscalar processor implies fetching the instruction, decoding and renaming it, waiting

for its dependencies to resolve, issuing the instruction (finding a free port to L1) and waiting for the data to arrive. Due to higher clock frequencies, future microprocessors are expected to have L1 with higher latencies, from a couple of cycles to several more. During the execution of the *load* its dependent instructions may have already been fetched and may be waiting for the completion of the *load* to be issued. In such a scenario, the overall latency of the *load* (even if it hits in cache) is important and reducing it will improve overall performance.

The structure of the paper is as follows. In Section 2 we will explain the mechanism and the implications behind it. In Section 3 we will digress about our simulation environment. Later in Section 4 we will present the different experiments that were conducted to assess the characteristics of our mechanism. In Section 5 we will discuss the related work and we will conclude the paper in Section 6.

2. DYNAMIC MEMORY INSTRUCTION BYPASSING

Dynamic Memory Instruction Bypassing is a microarchitectural technique that *learns* from previous executions what are the producers of addresses for the *loads*. As soon as these addresses are known, the memory is accessed, hoping to bring the data to a register before any dependent instructions need it, thus allowing for the following bypassing of the *load*. Benefit is possible because in many cases the compiler separates the computation of the address from the issuing of the *load*. Good examples of when this happens are list traversals and stack management. When traversing a list, the compiler will issue a *load* instruction for the data of the node in the list and another for the next pointer in the list, which is indeed the computation of the address for the next node in the list. In the following dynamic basic block this address will be used to access another node. Our mechanism learns from this fact and allows the microarchitecture to issue these following *loads* as soon as their addresses are known. On the other side, the stack pointer is normally modified when the function is called, but used throughout the whole function for accessing local variables and parameters. This allows our mechanism enough time to bring data into registers.

In the next two subsections we will present the two parts in which our mechanism is divided, the learning scheme and the operational scheme.

2.1. Learning Scheme

Learning means establishing relations between instructions that produce addresses and *load* instructions that use them. For microarchitectural

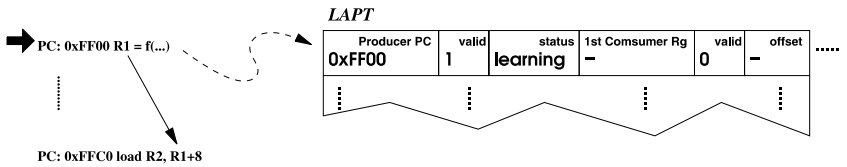

Fig. 2. Learning example: producer.

simplicity we have chosen to limit our mechanism to track only *BasePlusOffset loads*, those of the form `load rg,base_rg+offset`, where `base_rg` is a register containing a base address and `offset` is a constant field of the codeword. Most RISC microarchitectures have this kind of *load* instructions and make an extensive use of them, i.e. to access the stack, to access fields of structures, to access different elements of the same cache line, ... *Loads* with two source registers have intentionally been omitted since they are harder to group and associate to just one address producing instruction[4] and because they only represent a small amount of total dynamic *loads* in a programme.

Any instruction that defines an integer destination register[5] is potentially computing an address and is therefore of interest to our mechanism. In the decode stage, the register defining instructions access a 32 entry table called the Latest Address Producer Table (LAPT). Each entry keeps information about one address producing instruction and several *loads* that consume from it (see example in Fig. 2). When the instruction is decoded its logical destination register is used to access the LAPT. Its PC is checked against the value of the PC field to see if this entry holds information about this instruction. If this is not true, the old entry is taken away from the table and a clear new entry is associated to this instruction, by storing the correct PC and by setting the status field to *learning*.[6]

When a *BasePlusOffset load* arrives to the decode stage its base address register is used to index the LAPT. The entry selected holds information of the address producing instruction of this *load*.[7] Each *load* stores in this entry its destination register, its offset and sets the valid bit of the consumer sub-entry (see example in Fig. 3). We have made experiments (the results will be presented later) with a varying number of

[4]The importance of one producer per load will become clear when the microarchitectural part of the mechanism is presented.
[5]Notice that integer loads also define registers which may become base addresses for following instructions.
[6]The old entry is not discarded, as will be explained later.
[7]Hold in mind that a particular *load* may have statically more than one address producing instruction depending on the path taken to arrive to the *load*, but dynamically only one.

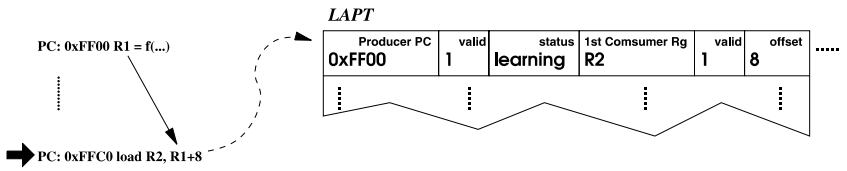

Fig. 3. Learning example: consumer.

consumer sub-entries per entry, ranging from 2 to an unbounded quantity, to explore the effect of this parameter. If the *load* finds an entry with such a combination of destination register and offset, it changes the status field from *learning* to *stable*.[8] If all the consumer sub-entries hold already valid information the *load* simply does nothing. More complex learning strategies were not considered to minimise the overall complexity of the implementation.

Due to a lack of logical registers, it is usually very common for the compiler to redefine a destination register as soon as no more instructions need to use it as source. Therefore, if the entries evicted from our LAPT were not stored somewhere, hardly any one would ever achieve a *stable* status and produce benefits. We have chosen to store these entries in a secondary table called Address Producers Table (APT). This table is tagged and indexed by the PC of the address producing instruction. This way, when these instructions are fetched, we can access this table and retrieve the information if it existed and insert it in the LAPT for its successive use.[9]

Notice that both the accesses of address producing instructions and *loads* to the LAPT share some conceptual parallelism with a normal renaming strategy found in any superscalar processor. In the next subsection we will learn what an important role plays renaming in the whole.

2.2. Operational Mechanism

In order to benefit from all this information, some further actions must be taken. If the status of an entry is *stable* when the address producing instruction is decoded, the execution of this instruction is microarchitecturally modified. As soon as the value of its destination register is computed (writeback stage) the operation mechanism will start the

[8]We have also made analysis on the amount of necessary offset bits, and approximately 5 bits suffice for all interesting offsets. This allows for smaller sizes of the tables involved.
[9]This secondary table needs only one read port since consecutive register defining instructions have consecutive PCs, and therefore we can group them in super-entries. Writes need a little more complexity but on the other hand, they can tolerate some latency.

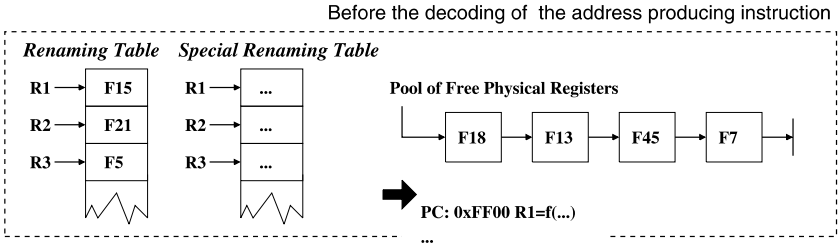

Fig. 4. Renaming example: prior state.

process of issuing special *loads* using the information in the table. For each valid entry, it will add the offset to the computed value to determine the address to be fetched and it will issue a memory request. This special *load* will have a microarchitectural representation in the processor state slightly different from that of a normal *load*. It will not occupy an entry in the reorder buffer since it will never need to get committed as an instruction. Nevertheless, it will get specially renamed (thus the need for the destination register in the LAPT) and it will occupy resources in a special *load-store* queue.

The renaming process of the previous example used is described in Figs. 4, 5 and 6. In them we can see how the address producing instruction renames the destination register of one of its consuming *loads*. This renaming is not stored in the normal renaming table but in a special one. Any intervening instruction between the address producing instruction and the *load* will still access the correct physical register. When the corresponding *load* is decoded, it will turn this renaming into active by moving the mapping from the special renaming table to the normal one. Once this is done, this *load* is converted into a `no-op`, since all its work has already been done. Turning the renaming into a correct one does not guarantee the data to have arrived into the register file. If it has, instructions

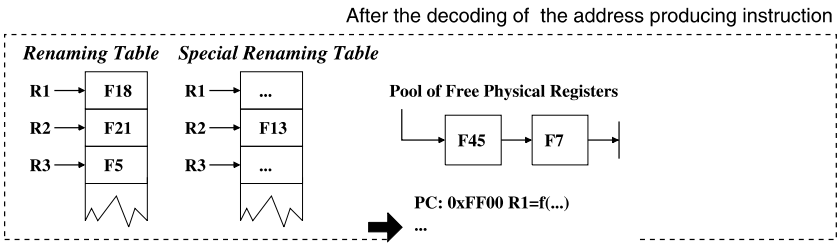

Fig. 5. Renaming example: after producer instruction.

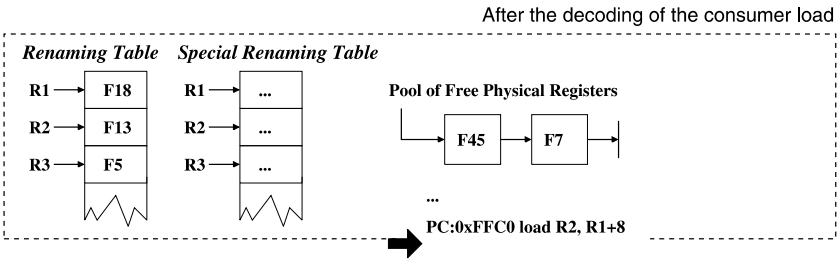

Fig. 6. Renaming example: after consumer instruction.

dependent on the *load* will be able to execute immediately and we can talk about a total bypassing of the *load*. If the value has not arrived yet, the normal scoreboarding of the microprocessor will wake dependent instructions when the value arrives. We call this scenario a partial bypassing of the *load*.

In order to assure memory consistency we must also track special *loads* issued before hand in a special *load-store* queue. Since these *loads* are being issued out of order, it may be possible that an intervening *store* writes to that same location before the real *load* thus making invalid the data brought by the special *load*. In this case some mechanism must be taken to re-issue the *load*[10] and re-execute any dependent instructions that may have been issued with incorrect data. Some current microprocessors implement an aggressive memory disambiguation technique like the one sketched here, since these mechanisms have been shown to be very exact and provide reasonable benefits. Our *Dynamic Memory Instruction Bypassing* needs no change to whatever disambiguation technique is implemented other than adding a small table to store the addresses of the special *loads* that are on the fly. Regarding our experiments, the size of this table is negligible (a four entry table is sufficient) since the amount of special *loads* on the fly is very small (take into account that once the real *load* is decoded the special *load* will not conflict with any *store* to come and it should be taken out of this table).

The mechanism presented is speculative in nature since we are issuing special *loads* before we really know if the real *loads* will ever come to execute. It may happen that after executing the address producing instruction (and therefore issuing and renaming all its consuming *loads*) the execution of the programme takes another path than the one taken during the learning part of our mechanism. In this case the *loads* may never be fetched and their values consumed (in average this happens to 24%

[10]Probably the simplest solution would be to take the value directly from this *store*.

of all the special *loads*). To detect this we track what renaming physical registers depend on the base address register. If this base address register is redefined by another instruction (which is easily detected in our mechanism with the LAPT) we free the physical registers associated with the address producing instruction and we clear the entries in the special *load-store* queue. Notice that this recovery mechanism is very simple and does not need to re-execute any instructions. We therefore consider this kind of speculation a win-win situation. The only negative effect that may affect execution is the increase in incorrect memory accesses. The total amount of incorrect memory accesses (24% in average) is not very big and does not negatively affect performance.

3. SIMULATION ENVIRONMENT

Our own simulator was used throughout all the experiments. Our simulator is cycle accurate, trace and event driven. For all the experiments we have designed a baseline microarchitecture named *LoadStore* which simulates a typical out-of-order superscalar processor. In the following paragraphs we will detail the assumptions and simplifications taken in the simulations.

The memory model is composed of three levels, L1, L2 and main memory. As our experiments do not need a highly accurate instruction fetch mechanism, we have assumed a perfect instruction L1 which can deliver up to `width` instructions per cycle, where `width` is the instruction fetch width (four in the present experiments). This is a nearly ideal fetch with only one constraint, taken branches stop the fetch. We consider this model to be enough for our purposes. The L1 can be parametrised by total size, associativity, replacement strategy in case associativity is not one, line size and latency. Normally, all simulations assume a direct mapped first level of cache with 32 bytes line size unless otherwise stated. Our experiments were conducted by modifying the latency of the cache and its total size.

The L2 cache can also be parametrised by the same characteristics. All our simulations have assumed a 4-way associative with LRU replacement policy and 64 bytes line size. The L2 latency and its total size was modified across experiments but normally 12 cycles was used for the latency and 512 Kb for the size. Finally main memory was simply parametrised by its latency, consisting of 60 cycles except where noted. Our memory model assumes that only one level is accessed at a time, so that if a *load* misses in L1, and hits in L2, its total latency will be L1 latency plus L2 latency. Some microarchitectures allow for the L2 cache to be probed in parallel with L1, thus making L2 total latency not dependent

on L1. We have chosen not to follow this mechanism. With respect to the bus utilisation we have assumed a perfect bus of latency mem_latency (normally 60 cycles) the bus between main memory and L2 cache. In many microarchitectural simulations, the behaviour of this bus is critical for performance, and better models are required to get consistent results. Nevertheless, our techniques are oriented at producing benefits when data is already on-chip, and this is why we have chosen to simplify this model so to increase simulation speed. On-chip buses are modelled using ports resources. A port resource must be accessed when the request is done (simulates sending the address for the loads) and when the data arrives back (simulates the coming of the data back from the cache module). In general we have one port between L1 and L2 and a variable number of ports (depending on the experiments) between L1 and the Load Store queue. TLB was not simulated at all (assumed perfect).

The width of the microprocessor has been set to the same as the issue width, four instructions per cycle. The reorder buffer has been programmed to have 64 entries. Three different windows for integer, floating point and the load store queue were assumed. As our main interest relied not on the scheduling capabilities of our proposals but on the potential performance of them, we chose to have issuing windows as big as possible so as to not affect performance at all. This was normally achieved with half the size of the reorder buffer or even sometimes the same size for all instruction windows. This is a little unrealistic, since having more instruction window slots than reorder buffer entries is kind of a waste, but as many scheduling policies achieve nearly perfect scheduling capabilities with more reasonable sizes, we have assumed that this problem is orthogonal to our proposals. The number of functional units and their latency is presented in Table I.

The branch prediction mechanism selected for all experiments assumes a 14 bit history gshare with perfect Return Address Stack prediction.

Table I. Quantity and Latency of Functional Units

	4way		8way	
	Quantity	latency	Quantity	latency
Integer arithmetic	2	1	4	2
Integer logical	2	1	2	1
Floating point arithmetic	2	4	4	6
Floating point logical	2	2	2	2

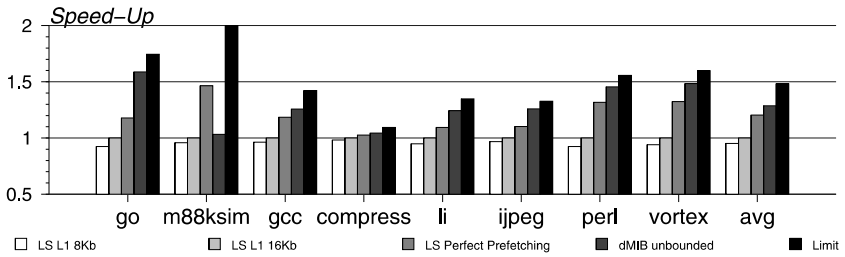

Fig. 7. Impact of memory behaviour on SPECint95.

We have chosen a set of benchmarks known as `SPECint95` . All integer traces were composed of a total of 50 million instructions per simulation.

4. EXPERIMENTS

In this section we will present the experiments done to assess the virtues and limitations of the dynamic mechanism presented in the previous sections. Our base architecture will be the one named *LoadStore* which we have discussed in the previous section. Our proposal will carry the label *dMIB* (which stands for *Dynamic Memory Instruction Bypassing*) in all the Figures.

4.1. Impact of Memory Behaviour on SPECint95

One of the reasons for choosing `SPECint95` for this set of experiments is the fact that their memory behaviour is not very intense. Mostly all of them fit in small L1–L2 caches. This can be seen analysing the first three bars of Fig. 7. These bars represent the performance results for different configurations of our baseline, the LoadStore machine, with different L1 memory sizes: a small 8 Kb as that of the Pentium IV, a 16 Kb, the size selected for our baseline, and a configuration with a perfect memory behaviour which has been tagged as perfect prefetching in the Figure. All bars have been normalised to the second one, the one with 16 Kb L1 cache, which will be used as our baseline for the rest of the section unless noted.

The `SPECint95` benchmark set has been traditionally thought of having a good memory behaviour. This is due to the results shown in the first three bars of Fig. 7. Having perfect memory only provides on average a 21% performance speed-up, which is not impressive and this fact has made a lot of people lose interest in memory performance techniques

for these applications. Nevertheless, as our interest resides in bringing data from L1 and L2 transparently, this suite of benchmarks and this scenario become ideal to establish the benefits of our mechanism. The last bar shows a Limit study which assumes that all data is present in registers at no cost (as opposed to assume it is always in L1 as in the perfect prefetching model). The fourth bar corresponds to a simulation of our mechanism with unbounded resources, i.e. unbounded entries in the producer-consumers table and unbounded consumers per entry. This simulation allows us to understand how all benchmarks are behaving when our proposal is simulated with no constraints. First of all, it must be noted that there still exists a gap between our dMIB unbounded bar and the Limit bar. This is due to the fact that our mechanism does not address all kind of *loads*, just those of the form `load reg,base_reg+offset`. Moreover, our mechanism uses a particular learning heuristic which does take some time to warm up and may not capture all possible *loads*. Partial bypassing is also responsible for not achieving the Limit bar. This partial bypassing may be due to two reasons, lack of enough distance from the producer of the address to the *load*, and data that is present in main memory which our mechanism can not bring early enough. Our first interesting result is that in average dMIB unbounded performs better than having perfect memory for the `SPECint95` benchmark suite. In this Figure our unbounded mechanism shows an average improvement of 30% over a baseline machine (with a peak of 60%).

One thing must be noted from this Figure is the fact that while most benchmarks behave consistently producing benefits, two of them, `124.m88ksim` and `129.compress` do not exhibit practically any performance benefits. Our analysis shows that `124.m88ksim` suffers from a problem of coverage while `129.compress` behaves poorly due to a timeliness problem. In Table II we show the average latency of *loads* in all the `SPECint95` benchmark suite. Correlating overall benefits with latencies in superscalar processors is a tricky thing, because in many cases it can deliver counterintuitive results. This has happened with our mechanism. We measured the average latency of all *loads* in the LoadStore baseline and in our dMIB model and found out an increase in this latency. This effect is explainable if we reason that in our dMIB model these special *loads* are issued much before, and although they may take longer in average, they are surely going to produce their results before the baseline would. In Table II we take this into account. The first column shows the average latency of all *loads* in the base architecture. The second column shows the average latency of *loads* running in our dMIB model but only those that our mechanism does not attack. Analysing the complement of the *loads* our mechanism speeds and the total will allow us to establish

Table II. Average Latency of loads

Benchmark	All loads	Non-captured loads
go	3.40	2.94
m88ksim	24.74	41.50
gcc	5.12	4.46
compress	3.03	1.21
li	3.59	2.27
ijpeg	3.92	2.21
perl	6.64	4.09
vortex	4.86	4.36
Average	6.91	7.88

All Loads means loads before any modification, as they behave in the LoadStore architecture. Non-captured Loads column shows the latency of loads not treated by our mechanism.

some facts about why `124.m88ksim` and `129.compress` behave like they do. In `124.m88ksim`, the average latency of all accesses to main memory is 24.74 cycles. This is a very big number compared to the other `SPECint95` applications where this average is under 10 cycles for our present configurations. Notice that `124.m88ksim` is the only application where the average latency of non-captured *loads* is even bigger than that of all *loads*. This increase explains that our mechanism is precisely bypassing instructions that do not miss in L1 nor in L2, leaving critical instructions without modifications. The critical path of `124.m88ksim` is ruled by these instructions, which our mechanism does not cover.

Average latency of non-captured *loads* is more appropriate to show if our mechanism is bypassing missing or successful *loads*, but it does not show how well our mechanism behaves on the *loads* it bypasses. Our mechanism may be learning from problematic *loads* and trying to bypass them earlier, but nevertheless, data still may be arriving too late for performance increments. In Table III we can see how many bypassed *loads* with respect to the total amount of *loads* do exist in every simulation. The last column of this Table shows us how many of the bypassed *loads* arrive before the *load* arrives at decode stage. A good thing about *Dynamic Memory Instruction Bypassing* is that it should not produce slowdowns normally. This is a particularity of prefetching techniques: even if prefetching does not bring data in time, any decrease in the total stall time produces some benefit, although sometimes this benefit is negligible. Prefetching only produces slowdowns if it prefetches incorrect data which displaces correct data from the cache or when these prefetch instructions interfere with other instructions in the fight for resources. Our technique

Table III. Relation of Bypassed Loads

Benchmark	Bypassed/Loads %	Bypassed/Issued %	Total B./Bypassed %
go	53.01	78.19	49.62
m88ksim	43.41	82.65	29.79
gcc	38.48	56.21	38.35
compress	85.71	100	0.03
li	78.97	70.39	29.66
ijpeg	69.78	90.70	70.53
perl	51.80	64.52	43.50
vortex	56.36	65.89	48.75
Average	59.69	76.06	38.77

The first column is the % of all loads that our mechanism captures. The second column shows how many loads are bypassed with respect to the total memory accesses that our mechanism issues. The third column represents the % of total bypassing from the total number of bypassed loads. All numbers represent dynamic instances of loads.

brings data located in the higher levels of cache directly to free physical registers, so the displacement problem does not affect us. Besides, in Table III we can see that our learning strategy shows very good results. In average 76.06% of everything brought is finally bypassed. As this number is already very good, we have delayed for future research the analysis of other learning (or unlearning) strategies that could improve it. With respect to fighting for resources, our special *loads* have less urge for execution, thus better tolerating a couple of cycles of waiting for a particular resource since they have been issued beforehand.

From the results of Table III we derive that `129.compress` is suffering from a timeliness problem. In `129.compress` we do cover problematic *loads*, as can be shown by noting that the average latency of non-captured *loads* is decreased from 3.03 to 1.21 cycles, but hardly any of these arrive early enough to produce substantial benefit.

4.2. Bounding Resources

In order to analyse the effect of the number of entries in the APT[11] and the number of consumers per entry in both tables, we run a series of experiments in which we constrained either of these two parameters, leaving the other unbounded. In Figs. 8 and 9 we present the performance results of these experiments. Notice that the last two bars on both Figures correspond to the limits explained in the previous subsection.

[11]The amount of entries in the LAPT is determined by the amount of logical registers.

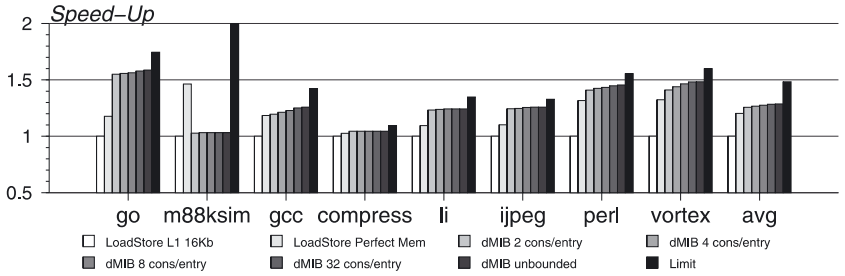

Fig. 8. Performance speed-ups with unbounded entries an varying quantity of consumers per entry.

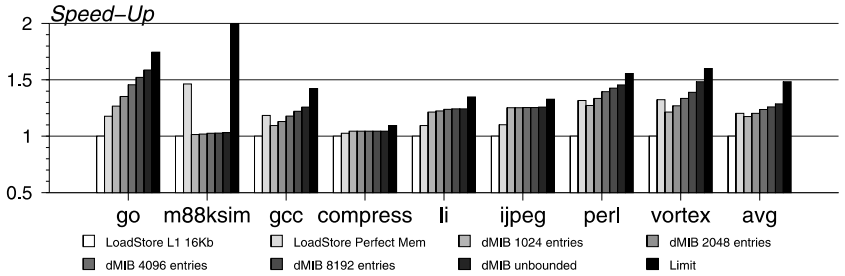

Fig. 9. Performance speed-ups with unbounded consumers and varying quantity of entries in the APT.

As results show, the number of consumers per entry increases performance, but not dramatically. In the rest of the experiments we have decided to fix this number to two consumers per entry, since this greatly decreases the cost while maintaining the overall performance. The other parameter has more impact on performance. As can be seen in Fig. 9 the more entries the better. Where possible, we will present results for three of these configurations, 2048, 4096 and 8192 entries. In Figures where space is an issue, we will summarise the benefits of our technique with the results from 4096 entries.

Figure 10 presents the first bounded simulation results. When we constrain both the number of consumers (which have been selected to only 2 in this Figure) and limit the number of total entries, we see a decrease of the potential performance presented by the unbounded simulations. Fortunately, this decrease is minimal. Our performance results range from 20% with 2 K entries to up to 24.5% with 8 K entries, with peaks of nearly 50% in programmes like `099.go`. These results are very promising indeed,

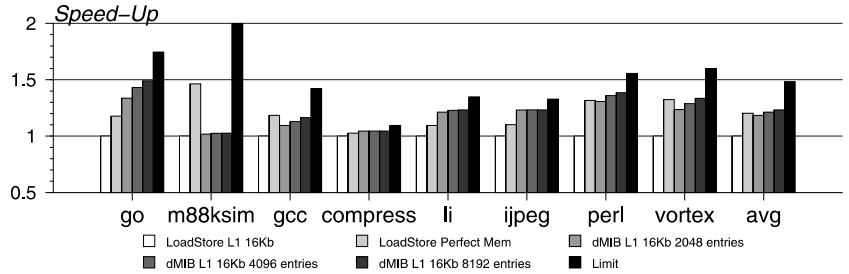

Fig. 10. Main results (two consumers per entry) and varying number of entries in the APT.

for they show that a good percentage of the total performance shown by the unbounded experiments can be achieved under realistic assumptions.

4.3. Renaming Registers

Our technique relies on free physical registers in order to increase performance by issuing *loads* before they are fetched. Although there exist different renaming techniques such as Gonzalez *et al.*[4] that delay the renaming until the data is really needed, we have not investigated any of these in combination with *Dynamic Memory Instruction Bypassing*. Instead we are more concerned in analysing the impact that a reduced number of registers may have in our technique.

Some processors such as the Alpha 21164 have been designed with as many physical registers as the processor may ever need, so that they never stall due to lack of free physical registers.[12] As every instruction can at most define an extra renaming register, having an equal amount of registers as live instructions in the processor is equivalent to an unbounded number of renaming registers. In our case, as the Reorder Buffer has 64 entries, 64 extra renaming registers are enough to satisfy this point.

In *Dynamic Memory Instruction Bypassing* , an address producing instruction may issue *loads* before these *loads* arrive in the window, i.e. it does not only use a renaming register for its destination register (the address) but it also consumes registers for consuming *loads*, whose lifetimes will expand since the address is produced until their *load* instructions appear in the processor or until another instruction redefines the address register on which they are based. Assuming that a considerable amount of these renamings are successful this implies that our technique

[12]This is also done because implementing a pipeline stall due to lack of registers is costly, and in many cases having this amount of registers is feasible. Under other assumptions it may be preferable to stall the pipeline when registers are unavailable.

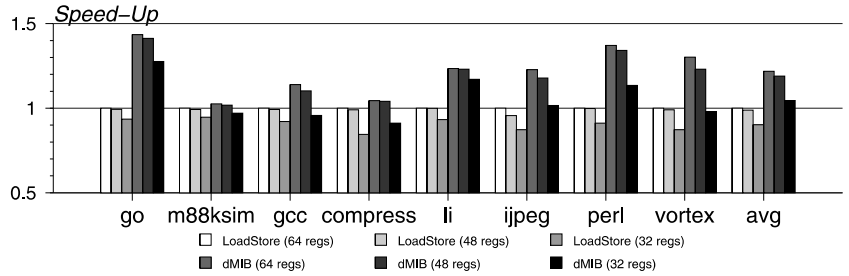

Fig. 11. Effect of the amount of renaming registers.

should be more dependent on the total number of renaming registers. In Fig. 11 we can see the impact of having 32, 48 or 64 renaming registers both in the LoadStore baseline and in our mechanism. When we reduce the number of renaming registers, the relative improvement of our mechanism over the LoadStore architecture with the same number of registers decreases: our 22.24% of average improvement gets reduced to a 20.75% with 48 registers and with only 32 renaming registers our mechanism only outperforms LoadStore machine with 32 renaming register by a 16.3%.

Notice that the LoadStore baseline benefits very little from having 64 renaming registers as compared to having only 48. This is because the probability of having a reorder buffer completely full of register defining instructions is very small. Our mechanism takes more benefit of these *extra* registers. On the other side of the spectrum, with only 32 renaming registers, our mechanism continues to outperform the base architecture, even a base machine with 64 renaming registers. This is very promising indeed, for it shows that our mechanism uses registers more intelligently. In our machine, when an address generating instruction arrives at the decode stage, it will consume renaming registers for its consuming *loads* even if this implies stopping the pipeline due to lack of registers shortly after. This can be seen as an increase in the priority of getting renaming registers for *loads*, which are usually considered critical instructions.

4.4. Tolerating Cache Latencies

One of the first aims of our technique is to better tolerate the future increasing latency of on chip caches. In this subsection we will explain the experiments conducted to analyse how much benefit comes from bypassing from L1, how much from bypassing from L2 and the effect of increasing latencies to both levels of cache. To analyse this we have run experiments

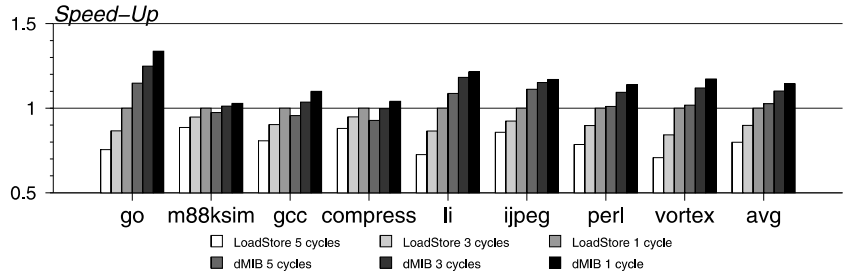

Fig. 12. Effect of L1 latency supposing perfect memory (dMIB assumes 4096 entry APT with 2 consumers per entry.

where L1 is of infinite capacity and produces no misses (no capacity nor cold misses). With this scenario we are effectively isolating the impact of memory latency (or L1 cache since we are considering that everything hits in L1 cache) and thus we can measure how our technique behaves under different latencies. In Fig. 12 we can see six bars per programme. The first three represent the execution of a microarchitecture without our mechanism and with latencies to perfect L1 of 1, 3 and 5 cycles. The last three bars incorporate our bypassing mechanism (all bars are normalised to the LoadStore architecture of 1 cycle [third bar]). The first thing that these results show us is that with perfect memory, our average speed-up decreases from a 22% as seen in Fig. 10 to a 15%. This number allows us to derive that on average, 75% of our performance improvement comes from the bypassing of data present in L1 cache. Nevertheless, we can not forget where the other 25% improvement comes from, bypassing data present in the L2 cache as will be shown with the results from Fig. 13. Another outstanding fact is that the microarchitecture enhanced with *Dynamic Memory Instruction Bypassing* executes faster in average with a 5 cycle L1 cache than a LoadStore architecture with just 1 cycle latency to L1. This shows that our mechanism is a good choice if designers decide to user larger caches at the expense of access time.

In Fig. 13 we have conducted another series of experiments assuming perfect L2. In this case we have analysed different latency pairs for both L1 and L2. The simplest case is 1 cycle hit in L1 plus 12 cycle hit in L2 (L2 is always accessed after the miss, never in parallel, what augments L2 hit latency to 13 cycles in this case) which has been used as baseline for all our experiments. The second pair assumes a 3 cycle hit latency to L1 and another 15 cycles for L2. The biggest latency pair assumes 5 cycles to L1 plus another 20 to hit in L2. In this Figure we can see that the improvement obtained in the 1–12 case grows up to a 22%, just

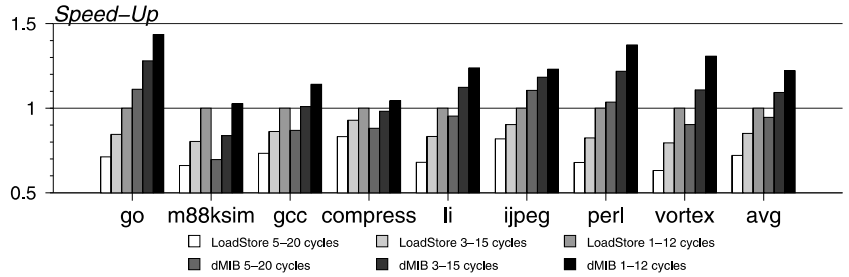

Fig. 13. Effect of L1–L2 latency pairs supposing perfect L2.

what happens in the general case with normal memory. This validates our previous assessment that in average 25% of our performance improvement comes from bypassing from L2 cache. This potential improvement allows our mechanism to obtain bigger relative performance improvements when increasing L1–L2 latency pairs, as can be seen in Fig. 13 where the average relative performance grows to 28.65% with 3–15 L1–L2 latency pairs and up to 31.5% with 5–20 L1–L2 latency pairs.

With the experiments of Fig. 14 we wanted to know how our mechanism would behave under certain L1 latency-size pairs. The general trend in microprocessor design is making L1 size smaller not because lack of transistors, but in order to increase its speed. We have analysed three different configurations of size and latency, the bigger the size the larger the latency. We believe these configurations represent the microarchitectural trend due to increased microprocessor pipelining[5–7]. The simpler configuration has a L1 cache of 1 Kb accessible in 1 cycle. The second configuration has a L1 cache with 4 Kb with a latency of 3 cycles. Finally the last configuration has a L1 cache of 16 Kb but at a distance of 5 cycles. All three configurations have a 4 Mb L2 cache at a distance of 20 cycles. As can be seen from the data on Fig. 14 all configurations with *Dynamic Memory Instruction Bypassing* outperform in average those without this technique, even the 1 Kb L1 cache with dMIB outperforms a LoadStore machine with 16 Kb.

4.5. Incremental Prefetching

As our technique is oriented to data present in on-chip cache, it will benefit from a good memory behaviour. This is because when our mechanism encounters a particular instruction that misses, the overall reduction of the critical path is minimal with respect to the total memory latency. What we tried to do at this stage of our research is to augment

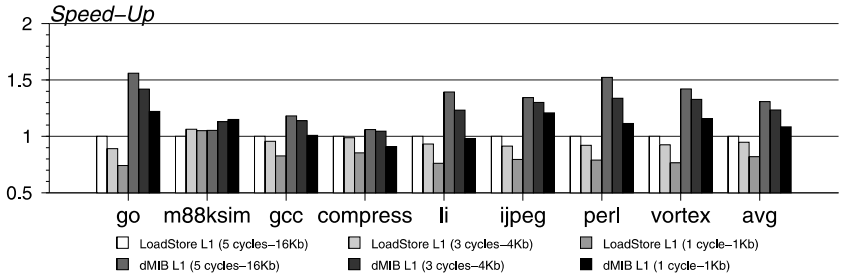

Fig. 14. Effect of different L1 latency+size pairs.

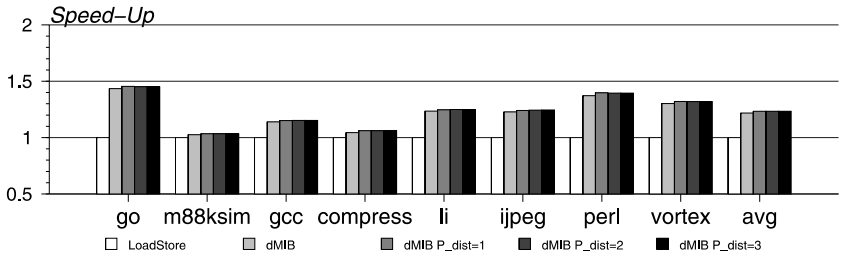

Fig. 15. Effect of incremental strided prefetching (dMIB assumes 4096 entry APT with four consumers per entry).

our dynamic bypassing mechanism with a linear prefetching mechanism (similar to the one presented in Ortega *et al.*[3]) that could overcome the hurdle imposed by bad memory performance. This way we could be effectively exploiting more opportunities in a simple and cost effective way. Unfortunately and contrary to what happened for numerical applications in Ortega *et al.*[3] this incremental prefetching has not been successful at all.

The prefetching mechanism proposed over our bypassing mechanism should only target the *load* instructions that our bypassing mechanism targets, but should focus on subsequent executions of the same instruction by learning their behaviour from past instances.

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 just one field (plus the tag), the *Last Effective Address*.

When a *load* instruction is executed by our mechanism, the prefetching mechanism is activated. If there is a tag match in the CLPT, the current stride between the current effective address and the tables *last effective*

Table IV. Dynamic Percentage Type of Producer Instructions

Benchmark	Arithmetic	Loads	Movs
go	39.9	58.5	1.55
m88ksim	47.5	48.62	3.9
gcc	38.77	52.3	8.9
compress	90.9	9.1	0.0
li	31.5	57.6	10.8
ijpeg	26.6	71.5	1.8
perl	26.2	66.9	7.0
vortex	27.4	62.75	9.8
Average	41.09	53.40	5.46

address is computed. This stride is multiplied by the prefetching distance (a parameter in our study) and is added to the current effective address to compute the potential to-be next instance address. If there is no match in the CLPT the prefetching mechanism simply allocates a new entry by using LRU replacement policy.

The results of the simulations for the proposed mechanism can be observed in Fig. 15. This Figure presents five bars per application. The first bar represents our LoadStore baseline, and the rest of them represent different versions of our technique, the last three are the ones with different prefetching strategies. We decided to implement static prefetching distances, ranging from the next dynamic instance up to the three next dynamic instances. P_dist stands for prefetching distance, thus the third bar represents our technique with an prefetcher associated at distance 1, and so on. These flat results show that incremental prefetching is not at all interesting for this dynamic mechanism.

The reasons that explain this behaviour are mainly two. First of all, this dynamic mechanism is oriented for non-numerical applications of whom their good memory behaviour is well known, as we have stated before. This makes it less interesting to include an incremental prefetcher. Secondly, the quantity and possibilities of strided incremental prefetching are less than in the static mechanism. This can be noted by analysing Table IV where we show the dynamic percentage type of producer instructions. All our producer instructions fall into one of these opcodes: `addiu`, `addu`, `and`, `ld`, `lw`, `lhu`, `lh`, `lbu`, `move`, `movn` and `movz`.[13] We have classified all opcodes into three sets: arithmetic, composed of `addiu`,

[13]All these opcodes belong to the MIPS R10K ISA.

addu and and; loads, composed of ld, lw, lhu, lh and lbu; movs, composed of move, movn and movz. In this Table it can be seen that in average less than half or all dynamic producers of valuable information fall into the arithmetic cathegory. This cathegory is the only one capable of producing strided patterns so the possibilities of prefetching are very small indeed.

5. RELATED WORK

Moudgill *et al.*[8] presents a speculative renaming that is done in parallel with the fetch of the instruction. Its global aim is to reduce the path of all the instructions in the pipeline. We share some conceptual relation with this paper, since our techniques renames certain instructions before they ever arrive to the Rename stage. Another piece of work that relates to ours with respect to the renaming is Balasubramonian *et al.*[9] This paper proposes a double strategy for renaming register assignment. The first strategy is the normal one while the second strategy gets fired when all but a subset of all the renaming registers have already been assigned. This strategy renames instructions but frees their registers after a certain amount of time. This way if the dependent instructions have already read the value, the processing will continue as normal, otherwise, the dependent instructions will receive incorrect data. This technique is based in the fact that the average live cycles of a register is usually very small. This is specially true of registers devoted to address generation and branch computation. Although all the instructions executed in this way must get re-executed when the amount of free registers increases, this speculative execution mode allows for the warming of branch prediction tables and caches, accelerating the subsequent execution. One of the benefits of this technique, prefetching to L1, is only achieved for linear data structures. Linked structures can not be prefetched by this technique since the address for the next piece of data must be brought from memory. If only one of the nodes does not hit in cache, the specific load instruction will be a long latency instruction and the mechanism will free its destination register to assign it to another instruction. This way the dependence prefetching chain is broken.

Besides its renaming component, this paper also deals with a concept or field which we have referred to as *Memory Instruction Bypassing*, while others refer to it as Memory Renaming or as Memory Bypassing. In some cases this term is understood as mechanisms which communicate memory operations allowing some of them to be bypassed by using registers. Other times, bypassing is understood as the shortcutting of the data from the producer to the consumer, and it is therefore nearer to the concept of

memory renaming than to our concept of instruction bypassing. Nevertheless, these two definitions mix frequently in the literature.

One of the first works in pure memory renaming is Tyson and Austin.[10] This paper proposes to predict dependencies among memory instructions using value prediction and then rename the memory instructions accordingly to execute them faster. By using value prediction, their renaming is speculative, and may need recovery actions in case of mis-speculations. A similar approach in concept is the one presented in Moshovos *et al.*[11] The authors propose also to detect dependencies, what they call memory cloaking, and convert these dependencies into def-use chains that can be subsequently used to shortcut the data. They also propose a Transient Value Cache similar in concept to a victim cache specially designed for data that is likely to be bypassed. As the first one, the mechanism is inherently speculative.

One year later appeared the following work[12] developed at Intel. This work is highly focused to the IA32 architecture, where the lack of registers is very important. Their goal is to better use the register file. To achieve this they analise when two instructions are going to produce the same result, and when this is so, they get assigned the same physical register. This makes renaming more complex, since a count of defs of each register must be maintained, but on the other hand it allows instructions to execute faster because once two instructions are assured to produce the same result, the dependents on the second one do not need to wait for it to finish, thus proceeding faster. The authors also present the concept of unification, which tries to detect when two instructions are going to use the same memory location so that they can eliminate the need for the second access.

In Reinman *et al.*[13] the authors propose to identify potential dependencies among stores and loads by software, leaving the hardware the responsability of the bypassing only. Moshovos *et al.*[14] is another piece of work (as Jourdan *et al.*[12]) that analyses the communication among different loads that access the same memory location.

Austin and Sohi[15] is one of the first pieces of work that we definitely put under the umbrella of *Memory Instruction Bypassing*, leaving memory renaming aside. This paper tries to accelerate the execution of the *loads*, to achieve the total bypassing of the instruction. The mechanism predecodes these type of instructions, it caches previous base registers and introduces a quick computation of the memory address of the data that allows the *load* to proceed two cycles before to the cache. Assuming a 1 cycle access time to the L1 cache these *loads* execute in 0 cycles.

Black *et al.*[16] is another piece of work oriented at speeding up the pipeline of *loads*. This work proposes to predict *load* addresses so that

the cache access can be started before. Something similar, although more extended is presented in Yoaz *et al.*[17]. This paper presents three solutions related to the scheduling of memory instructions. The solutions proposed are the prediction of dependencies among memory instructions and the prediction of which bank the data is going to be located in. A cache way detector is also presented in Chung *et al.*[18]. The authors also propose to learn from previous executions how the computation of base addresses is done. Once the *load* arrives to decode stage, this information is used to speculate the *load*, generating a predicted *load* which will proceed to the cache. As the learning is not good enough, it is enhanced with prediction, making the mechanism speculative in nature. If the speculation fails, the predicted *load* is squashed and the normal *load* proceeds to the cache.

Bekerman *et al.*[19] presents a very good analysis of the problems that the different classes of *load* instructions have with respect to its potential bypassing. The authors analise how many references belong to the stack, how many can be detected in the fetch stage, how many have an absolute value ... They claim they can resolve the addresses of over 65% of all the *BasePlusOffset loads* in the fetch stage.[14] Although this piece of work is very interesting at first, they concentrate more on possibilities and talk less about specific implementations of the different mechanisms presented.

Roth *et al.*[20] is another piece of work that is highly related with the mechanisms presented in this document. It combines prefetch with memory renaming. In this work the authors propose to track *loads* that produce addresses and *loads* that consume those addresses. These *loads* are responsible of traversing pointer chasing structures such as trees or lists. The communication of the data between *loads* is very complex.

In Ortega *et al.*[2] the authors of this document experimented with a combined hardware-software approach to this problem in the context of numerical applications. The compiler annotates *pref* instructions allowing the hardware to bypass the execution of other *load* instructions to the same line of cache. In Ortega *et al.*[3] the work was extended by combining both bypassing and prefetching.

6. CONCLUSIONS

In this paper we have thoroughly analysed a dynamic mechanism aimed at detecting relations between address producing instructions and the *loads* that consume these addresses. This knowledge allows us to implement a *Memory Instruction Bypassing* mechanism that issues the *loads* before they are fetched by the processor. By issuing *loads* before,

[14]All these values take into account a CISC machine as the I32 architecture.

our mechanism allows the microarchitecture to *see* further in the future. The mechanism brings data from L2 and L1 directly into the register file, diminishing *load-to-use* latency. This will become more important in the near future, where pipeline depths and cache access time will affect negatively memory operations. Besides, the execution of the *load* instruction ahead of time allows the processor to better utilize the resources involved in the execution such as the ports to L1 and the renaming registers. As *loads* are executed when their addresses are known, our mechanism behaves as if it were assigning a higher priority to these kind of instructions, allowing them to use renaming registers ahead of other less critical instructions. While the bringing of data to the register file is done speculatively, the use of this register is not decided until the *load* arrives. Thus, the mechanism does not need a special recovery mechanism for incorrectly issued *loads*, what greatly simplifies the design of the pipeline.

The overall performance improvement produced by the mechanism is 22.24% in average with 4096 entries and two consumers per entry and grows up to nearly a 50% in applications like `099.go`. In average, our *Dynamic Memory Instruction Bypassing* with 32 renaming registers, outperforms a LoadStore base architecture with 64 renaming registers. We have also shown the relation between our mechanism and the sizes and latencies of the memory hierarchy, and that our mechanism can bridge the gap caused by larger caches at longer distances.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01, and by Hewlett Packard Laboratories.

REFERENCES

1. W. A. Wulf and S. McKee, Hitting the memory wall: Implications of the obvious, *Computer Architecture News*, Vol. 23, ACM Press, pp. 20–24 (1995).
2. D. Ortega, M. Valero, and E. Ayguadé, A novel renaming mechanism that boosts software prefetching, *Proceedings of the 15th Annual International Conference on Supercomputing*, ACM Press, pp. 501–510 (2001).
3. D. Ortega, J.-L. Baer, E. Ayguadé, and M. Valero, Cost-effective compiler directed memory prefetching and bypassing, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society Press, pp. 189–198 (2002).
4. A. González, J. González, and M. Valero, Virtual-Physical Registers, *Proceedings of the Annual International Symposium on High-Performance Computer Architecture* (February 1998).
5. M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar, The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays, *Proceedings of*

the 29th Annual International Symposium on Computer Architecture, IEEE Computer Society Press, pp. 14–24 (2002).
6. A. Hartstein and T. Puzak, The optimum pipeline depth for a microprocessor, *Proceedings of the 29th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, pp. 7–13 (2002).
7. E. Sprangle and D. Carmean, Increasing processor performance by implementing deeper pipelines, *Proceedings of the 29th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, pp. 25–34 (2002).
8. M. Moudgill, K. Pingali, and S. Vassiliadis, Register renaming and dynamic speculation: An alternative approach, *Proceedings of the 26th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 202–213 (1993).
9. R. Balasubramonian, S. Dwarkadas, and D. Albonesi, Dynamically allocating processor resources between nearby and distant ILP, *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ACM Press, pp. 26–37 (2001).
10. G. Tyson and T. Austin, Improving the accuracy and performance of memory communication through renaming, *Proceedings of the 30th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 218–227 (1997).
11. A. Moshovos and G. Sohi, Streamlining inter-operation memory communication via data dependence prediction, *Proceedings of the 30th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 235–245 (1997).
12. S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, A novel renaming scheme to exploit value temporal locality through physical register reuse and unification, *Proceedings of the 31st Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 216–225 (1998).
13. G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin, Classifying load and store instructions for memory renaming, *Proceedings of the 13th Annual International Conference on Supercomputing*, ACM Press, pp. 399–407 (1999).
14. A. Moshovos and G. Sohi, Read-after-read memory dependence prediction, *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 177–185 (1999).
15. T. Austin and G. Sohi, Zero-cycle loads: Microarchitecture support for reducing load latency, *Proceedings of the 28th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 82–92 (1995).
16. B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. Shen, Load execution latency reduction, *Proceedings of the 12th Annual International Conference on Supercomputing*, ACM Press, pp. 29–36 (1998).
17. A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, Speculation techniques for improving load related instruction scheduling, *Proceedings of the 26th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, pp. 42–53 (1999).
18. B.-K. Chung, J. Zhang, J.-K. Peir, S.-C. Lai, and K. Lai, Direct load: Dependence-linked dataflow resolution of load address and cache coordinate, *Proceedings of the 34th Annual International Symposium on Microarchitecture*, IEEE Computer Society Press, pp. 76–87 (2001).
19. M. Bekerman, A. Yoaz, F. Gabbay, S. Jourdan, M. Kalaev, and R. Ronen, Early load address resolution via register tracking, *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ACM Press, pp. 306–315 (2000).
20. A. Roth, A. Moshovos, and G. Sohi, Dependence based prefetching for linked data structures, *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM Press, pp. 115–126 (1998).