Reducing Memory Latency via Non-blocking and Prefetching Caches

Tien-Fu Chen and Jean-Loup Baer

Technical Report 92-06-03

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

June 1992

Reducing Memory Latency via Non-blocking and Prefetching Caches [1]

Tien-Fu Chen and Jean-Loup Baer

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

tchen@cs.washington.edu baer@cs.washington.edu

July 2, 1992

Abstract

Non-blocking caches and prefetching caches are two techniques for hiding memory latency by exploiting the overlap of processor computations with data accesses. A non-blocking cache allows execution to proceed concurrently with cache misses as long as dependency constraints are observed, thus exploiting *post*-miss operations. A prefetching cache generates prefetch requests to bring data in the cache before it is actually needed, thus allowing overlap with *pre*-miss computations.

In this paper, we evaluate the effectiveness of these two hardware-based schemes. We propose a hybrid design based on the combination of these approaches. We also consider compiler-based optimizations to enhance the effectiveness of non-blocking caches. Results from instruction level simulations on the SPEC benchmarks show that the hardware prefetching caches generally outperform non-blocking caches. Also, the relative effectiveness of non-blocking caches is more adversely affected by an increase in memory latency than that of prefetching caches. However, the performance of non-blocking caches can be improved substantially by compiler optimizations such as instruction scheduling and register renaming. The hybrid design can be very effective in reducing the memory latency penalty for many applications.

[1] This work was supported in part by the National Science Foundation under Grants No. CCR-9101541 and CCR-8702915, and by Apple Computer.

1 Introduction

As the gap between processor cycle time and memory latency increases, the cache miss penalty becomes more severe and thus results in lower processor utilization. Several enhancements to cache designs have been proposed to reduce the miss penalty. Multi-level cache hierarchies [2] provide a cost-effective way to reduce the average memory access times. Hit ratios can be improved by complementing caches with small buffers or specialized caching structures [3, 12]. Another possibility is to hide the memory latency of a thread or of a process by providing fast context-switching [21]. Still another approach is to exploit the overlap of processor computations with data accesses within one process by using write buffers, non-blocking caches, and prefetching caches. The focus of this paper is on this latter approach.

Usually, a processor must stall on a cache miss until the miss is resolved. In the case of write misses, this can be avoided by the use of a write buffer. The basic idea in non-blocking and prefetching caches is to hide the latency of (read and write) data misses by the overlap of data accesses and computations to the extent allowed by the data dependencies and consistency requirements. A *non-blocking* (or *lockup-free*) cache [15, 20] allows execution to proceed concurrently with one (or more) cache misses until an instruction that actually needs a value to be returned is reached. Such caches exploit the overlap of memory access time with *post*-miss computations. *Prefetching* [1, 8, 13, 17, 18] hardware and/or software techniques can eliminate the cache miss penalty by generating prefetch requests to the memory system to bring the data into the cache before the data value is actually used. These techniques exploit the overlap of computations *prior to* an actual cache miss.

In this paper, we evaluate the effectiveness of these hardware-based techniques on reducing the memory latency. We consider ways to improve the approaches by compiler-based optimizations (e.g., code rescheduling, software register renaming). We also propose a hybrid design which is a combination of these techniques. Our results confirm previous studies [5] indicating that buffering writes can remove most of the write miss penalty when reads are allowed to bypass writes. Our experiments show that hardware prefetching caches, which require extra hardware complexity, generally outperform non-blocking caches and that the prefetching caches are less sensitive than non-blocking caches to the increase in memory latency. However, the compiler optimizations that we propose can significantly improve the effectiveness of non-blocking caches.

The rest of the paper is organized as follows: Section 2 gives some background information on non-blocking and prefetching caches. Section 3 describes the processor and memory architectures under study as well as the evaluation methodology. Simulation results are presented in Section 4. Section 5 describes the compiler optimization algorithms and discusses the results. In Section 6, we propose and evaluate a hybrid design. Finally, we conclude in Section 7.

2 Background and Performance Issues

We start this section with a brief description of non-blocking caches, write buffers, and prefetching caches. We then discuss performance issues and the extra processor requirements that are needed to support the features found in these caches.

2.1 Non-blocking Caches

Lockup-free caches were originally proposed by Kroft [15]. In his design, the following three features are included:

1. Load operations are non-blocking.
2. Write operations are non-blocking.
3. The cache is capable of servicing multiple cache miss requests.

In order to allow non-blocking operations and multiple misses, Kroft introduced Miss Information/Status Holding Registers (MSHRs) that are used to record the information pertaining to the outstanding requests. Each MSHR entry includes (1) the data block address, (2) the cache frame for the block, (3) the word in the block which caused the miss, and (4) the function unit or register to which the data is to be routed. Subsequently, lockup-free caches have been mentioned often in the literature [5] but there is some confusion on what part of the processor-cache-memory interface should support a given feature. Our view is that non-blocking loads are features specified in the processor, non-blocking writes are supported by buffering writes, whereas whether the cache allows multiple pending accesses or not depends not only on the presence of MSHRs, but also on the available cache bandwidth as defined by the interface between caches and memory modules. In the remainder of this paper, we will call non-blocking cache a cache supporting non-blocking reads and non-blocking writes, and possibly servicing multiple requests.

Non-blocking loads require extra support in the execution unit of the processor in addition to the MSHRs associated with a non-blocking cache. If static instruction scheduling in pipelines is used in the processor, some form of register interlock (like a full/empty bit for each register) is needed for preserving correct data dependencies. In the case of dynamic instruction scheduling, introducing out-of-order execution, some scoreboarding mechanism is required. Both instruction scheduling strategies need interrupt handling routines that can deal with interrupts generated by the non-blocking operations [9].

Write buffers are used to eliminate stalls on write operations. They permit the processor to continue executing even though there may be outstanding writes. Write buffers in conjunction with write-through caches are especially useful in reducing the write penalty. For write-back caches (with write-allocate), write buffers are used to temporarily store the written value until the data line is returned. Some implementations [3] allow multiple writes on the same line to be combined, thus reducing the total number of writes to the next level of the memory hierarchy. Another example of write buffers is the write-back buffer used for temporary storage of replaced dirty blocks in a write-back cache.

A consistency problem can arise when the processor allows non-blocking writes since a later (in program order) read may be needed before a previous buffered write is performed. If these two operations are on the same data block, an associative check in the write buffer or the MSHRs must be done to provide the correct value to the following read. When the processor is part of a shared-memory multiprocessor, the problem becomes more complex and the solution depends on the model of memory consistency that is adopted.

2.2 Prefetching Caches

Prefetching hides, or at least reduces, memory latency by bringing data in advance rather than on demand in a level of the memory hierarchy which is closer to the processor. Prefetching can be either hardware-based [1, 12] or software-directed [8, 13, 17, 18], or a combination of both. The main advantages of the hardware-based approach are that prefetches are handled dynamically without compiler intervention and that code compatibility is preserved. However, extra hardware resources are required and unwanted data could be prefetched. In contrast, software-directed approaches rely on data access patterns detected by static program analysis and allow the prefetching to be done selectively and effectively. The drawbacks are that some useful prefetching cannot be uncovered and that there is some execution overhead due to the extra prefetch instructions.

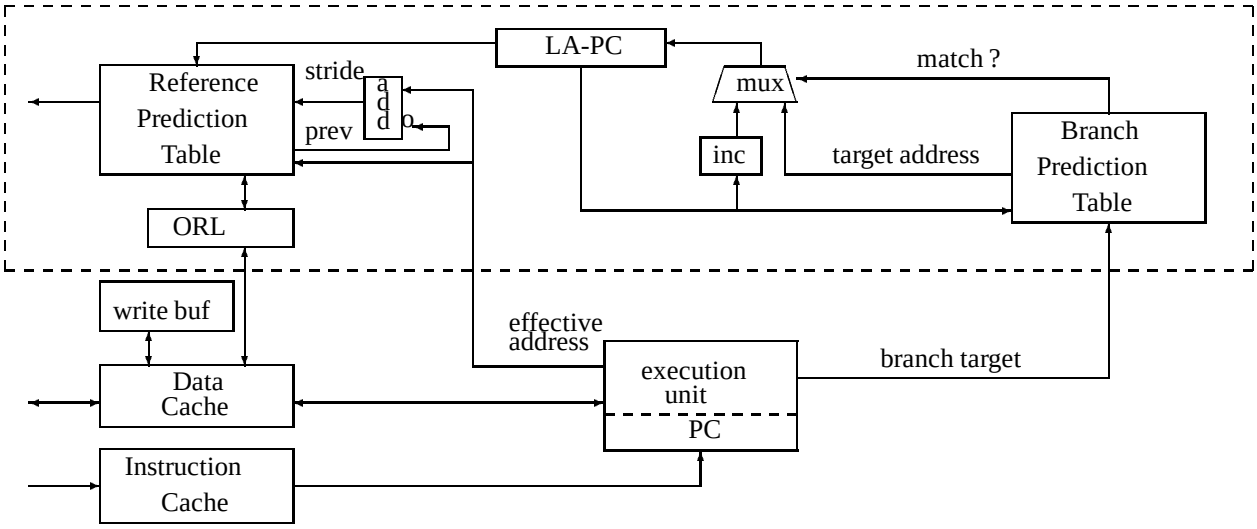


Figure 1: Overall structure of data prefetching

The prefetching scheme used in this paper is derived from the hardware-based approach proposed in [1]. It consists of a support unit (cf. Figure 1) for a conventional data cache whose design is based on the prediction of the execution of the instruction stream and associated operand references in load/store instructions. The latter, and their referencing patterns, are kept in a reference prediction table (RPT) which is organized as a regular cache. An entry in the RPT consists of the instruction address (used as the cache tag), the effective address of the operand generated at the last access, the stride (updated at each access), and two state bits for the encoding of a finite state machine to record the access patterns and to decide whether subsequent prefetches should be activated or prevented. The RPT will be accessed ahead of the regular program counter (PC) by a look-ahead program counter (LA-PC). The LA-PC is incremented and maintained in the same fashion as the PC with the help of a dynamic branch prediction mechanism. The LA-PC/RPT combination is used to detect *regular* data accesses in the RPT and to generate prefetching requests. The prefetched data blocks will be put in the data cache. Data pollution resulting from erroneous prefetches can be almost totally avoided by the fine tuning of the RPT finite state machines. The supporting unit is not on the critical path. Its presence should not increase the cycle time or data access latency except for an increase in bus traffic.

The key to hiding memory latency in this scheme is to keep enough distance between PC and LA-PC (possibly equal to memory latency) so that the prefetched data arrives "just before" it is needed. The LA-PC, which initially points to the instruction following the current PC, moves ahead of the PC when the execution stalls on real misses. Incorrect branch predictions and other mechanisms limit the distance from growing too large. Note that if the (prefetched) data is not resident in the cache in time when it is accessed by the PC, or if it has not been predicted, then the processor will always wait until the cache miss is completed. In other words, cache miss operations are blocking in this prefetching scheme.

2.3 Performances Issues

As we mentioned previously, the non-blocking operations exploit the *post*-miss overlap of computation and memory access while prefetching exploits the *pre*-miss overlap. We give now a brief qualitative view of the expected benefits for both types of overlap.

Non-blocking loads delay processor stalls until the necessary data dependence is encountered. Non-blocking loads will become necessary for processors, such as superscalar processors, capable of issuing multiple instructions per cycle [20]. However, the *non-blocking distance*, which is the number of instructions that can be overlapped with the memory access (e.g., instructions between the reference and the first dependent instruction), is likely to be small in the case of static scheduling. It can be increased when compilers produce code optimized for this potential overlap (see Section 5). Dynamic instruction scheduling (out of order execution) obtained at a significant increased cost in hardware complexity, can provide a larger non-blocking distance. However, the effectiveness is still subject to data dependence effects, branch prediction, and the size of the lookahead window provided by the architecture [6].

By comparison, non-blocking writes can be more advantageous in reducing the write miss penalty because the non-blocking distance is usually equal to the memory access time[2]. Moreover, the write buffer, a FIFO queue buffering pending writes, does not need a supporting unit in the processor. The other side of the coin is that even without a write buffer, the write miss penalty may not be a large fraction of the total data access penalty. In this paper, we consider write buffers both with read *bypass* (i.e., read misses have priority over writes) and with *no-bypass*.

In contrast with the non-blocking distance, the *lookahead distance*, i.e., the number of cycles which a prefetch request is generated ahead of the reference instruction, can be tuned by the designer and be as large as a small multiple of the memory latency. In our hardware-based scheme, its magnitude is constrained by effects such as the capacity of the RPT, the amount of regular data access patterns, and the success of branch prediction techniques. The implementation costs of prefetching caches, additional on-chip support units and more hardware complexity, are substantially higher than those of non-blocking caches. Software-directed approaches need an intelligent compiler to identify those memory accesses that can be prefetched and to determine the right position in the code to issue the prefetch orders.

A final point to mention is that in the case of non-blocking loads, the binding of a register with a certain value starts at the moment the non-blocking load is initiated. In contrast, the prefetch request is *non-binding*; it is only a *hint* to bring a data line close to the processor without involving any register binding.

[2] In other words, the processor does not stall on most write misses. A stall would occur only if a write miss is followed by a read miss on a different word in the same block. In that case, the stall is attributed to the read miss.

3 Architectural Models and Evaluation Methodology

In this section, we first describe the architectural models on which we will base our evaluation of non-blocking and prefetching caches. We then present our simulation methodology and the benchmarks used in the evaluation.

3.1 Processor-cache Models

Our baseline system consists of a CPU with a load/store architecture similar to the MIPS R3000 and an ideal instruction cache (thus no I-cache misses). The CPU has an instruction decoding unit, a fixed point unit (FXU), a floating point unit (FPU), and a cache interface. The decoding unit issues an instruction per clock cycle and the FXU can execute an integer operation in one cycle (perfect pipelining).

The FPU, which behaves like a co-processor, can accept one floating-point operation at every cycle until a data dependency on an unfinished instruction occurs. In this case, the dependent instruction needs to wait until the conflicting operation terminates. The FPU will handle FP operations in a multicycle pipeline with the execution times shown in the following table:

FP operations	# of cycles
Fadd	2
Fsub	2
Fmultiply	4
Fdiv	12
Fcvt	2

The cache interface can handle one data access at each cycle and, in case of a hit, the load latency is one cycle (i.e., delayed load with one delay slot). In the case of a write hit, an extra cycle is required to modify the data block in the cache. The refilling of a prefetched line will be delayed when it competes with real data accesses for the cache. Also, real cache misses could conflict with prefetch or outstanding write requests in the cache interface. We will assume, conservatively, that a fetch in progress cannot be aborted. However, a real read miss will be given priority over buffered prefetch requests or writes.

All caches used in this study are direct-mapped with 32K bytes and a block size of 16 bytes, unless otherwise specified. These caches use a write-back, write-allocate policy. In the baseline architecture a cache operation is always blocking.

When studying non-blocking loads, we assume a static scheduling of the pipeline. A status bit is associated with each register. On a cache miss, the target register status bit is reset and the outstanding information is recorded. When the miss is resolved, the register status bit is set. An instruction needing the value from a register with its status bit reset will cause the processor to stall until the value is returned from memory. If a cache miss occurs when one (or more) request is in progress, the cache controller will check to ascertain that the same block is not requested twice.

When studying non-blocking writes, we assume an 8-entry write buffer. Each entry consists of a data block and associated "valid" bitmap. A write miss will allocate an entry in the write buffer, update the word in the entry, set the corresponding valid bit, and then initiate a data access to memory whenever

the memory interface is available. Subsequent accesses check the write buffer. A read miss finding a matched valid word in the write buffer is treated as a cache hit. A subsequent write on a matched entry in the write buffer can be merged by writing the data in the buffer and setting the corresponding valid bit. When the block is returned from memory, those words with valid bits set in the buffer entry replace the corresponding ones returned from memory before the entire block is written into the cache.

A 256-entry reference prediction table is used to record and generate data prediction streams in the case of prefetching caches. This RPT and its associated complexity require roughly as much real estate on the chip as a 2 Kbyte data cache[1]. Branch predictions are performed by the Branch Target Buffer with two-bit state transition design [16]. Like the baseline caches, the prefetching caches will cause the processor to stall on each cache miss.

We experimented with various architectural choices summarized in Table 1. Since the above techniques for reducing the data access penalty are not mutually exclusive, each choice of architecture is based on the combination of components described earlier. This allows us to study the effect and contributed performance gain of various techniques, including prefetching caches (PREFETCH), write buffer (WB), prefetching caches coupled with write buffer (PREFETCH/WB), non-blocking caches (NBC), and whether allowing bypassing of writes by reads or not.

Table 1: Architectural Choices

			Non-blocking		Ordering	
	Cache	Prefetch	write	read	no bypass	bypass
BASELINE	X					
PREFETCH	X	X				
WB	X		X		X	
PREFETCH/WB	X	X	X		X	
NBC	X		X	X	X	
WB	X		X			X
PREFETCH/WB	X	X	X			X
NBC	X		X	X		X
HYBRID	X	X	X	X		X

3.2 Memory Models

Data bandwidth is an important consideration in the design of an architecture that allows overlap of computation and data access since several data requests can be present simultaneously. Assuming sufficient memory bandwidth or simply measuring hit ratios without taking into account the bandwidth can lead to erroneous conclusions. In this study, we present three memory interfaces with increasing capabilities of concurrency. Since several requests can be present, either in process or waiting to be processed, we associate an Outstanding Request List (ORL) with the prefetching and non-blocking caches (in this latter case, the ORL and MSHRs are similar). A requirement for this list is that it can be searched associatively.

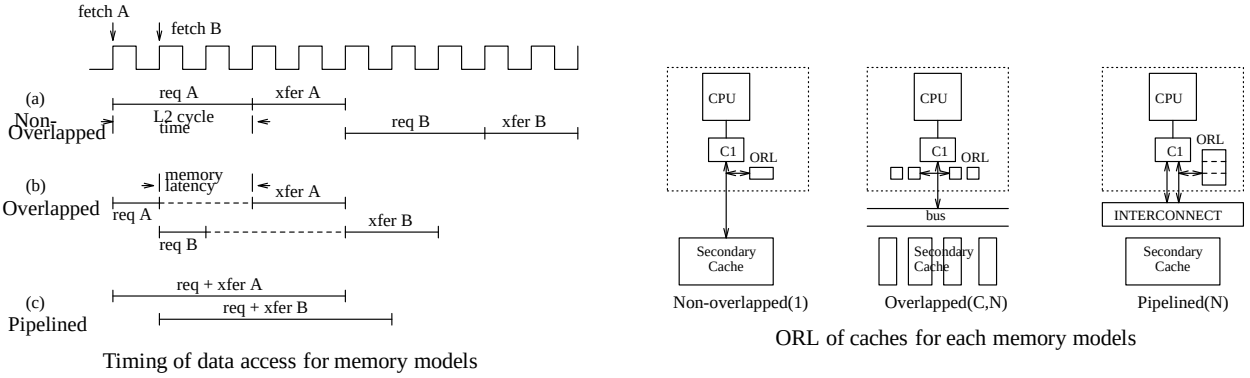


Figure 2: Three memory models

The three memory interfaces are as follows (cf. Figure 2 for timing charts and block diagrams).

- **Non-overlapped(1)**: As soon as a request is sent to the next level, no other request can be initiated until the (sole) request in progress is completed. This model is typical of an on-chip cache backed up by a second level cache.

  This interface supports only one cache request at a time.

- **Overlapped(C,N)** : The access time for a memory request can be decomposed into three parts: request issue cycle, memory latency, and transfer cycles. We assume that during the period of memory latency other data requests can be in their request issue or transfer phases. However, no more than one request issue or transfer can take place at the same time.

  This model represents split busses and a bank of C interleaved memory modules or secondary caches. An ORL with N entries is associated with each module.

- **Pipelined(N)** : A request can be issued at every cycle. This model is representative of processor-cache pairs being linked to memory modules through a pipelined packet-switched interconnection network. We assume a *load through* mechanism[19], i.e., the desired word is available as soon as the first data response arrives. An N-entry ORL is associated with the cache.

The memory latency used is denoted by δ, which usually is equal to 30. The configurations of the ORLs used in our experiment are **Non-overlapped(1)**, **Overlapped(8,2)**, and **Pipelined(8)** respectively.

Note that a non-blocking cache with a *Non-overlapped* interface model can only service one request a time and the *Overlapped* and *Pipelined* models are capable of handling multiple misses.

3.3 Simulation Method

We evaluated our proposed architectures using cycle-by-cycle trace generation combined with on-the-fly simulation. Benchmark programs were instrumented on a DECstation 5000 (R3000 MIPS CPU) using the *pixie* facility. To avoid the overhead of re-running the trace generation for every configuration, we simulated several configurations simultaneously. The simulator, written in C++, first reads in a variety of configuration descriptions and creates simulation objects (which models most aspects of the

CPU and the memory system) initialized with various configurations and parameters (such as cache size and latency time). To simulate the interlock mechanism for non-blocking reads, the simulator also reads the object code of the benchmark program and decodes instructions so that it is aware of which registers are involved in each instruction as well as boundary information on basic blocks. This knowledge is vital to simulate the detail behavior of non-blocking operations. Then the simulator runs the *pixified* benchmark programs, which will generate address traces at the same time. The simulator reads the trace through a *pipe* facility and feeds the trace records to each simulation object. The experiment results are collected at the clock cycle level from the individual configurations. This on-the-fly trace simulation methodology is advantageous in the sense that large traces do not need to be saved and furthermore it provides the flexibility of simulating a rescheduled code.

The traces captured at the beginning of the execution phase of the benchmarks were discarded because they are traces of initial routines that generate the test data for the benchmarks. No statistical data was recorded while the system simulated the first 500,000 data accesses. However, these references were used to fill up the cache, the Branch Prediction Table, and the Reference Prediction Table in order to simulate a warm start. After the initialization phase and the warm-start period, simulations results are collected for the first 100 million instructions or for the entire execution, whichever is first.

3.4 Benchmarks and Metrics

We use the SPEC[3] Benchmarks (see Table 2), which are compiled by the MIPS C compiler and the MIPS F77 compiler, both with default options. Table 2 shows some reference characteristics for the applications. The data is collected based on the simulation of our baseline cache. The number of writes is smaller than the number of reads and the proportion of write misses is considerably smaller than the proportion of read misses.

In the figures of the following sections, we will present the results of our experiments by using the CPI due to the data access penalty as the main metric. This contribution is defined as:

$$CPI_{data\ access} = \frac{\text{total data access penalty}}{\text{number of instructions executed}}$$

In the discussions of the following sections, when an average reduction of $CPI_{data\ access}$ is summarized, a geometric mean is used to average the percentages of the penalty reduction for the benchmarks [10]. In the figures, we present a breakdown of the data access penalty as follows: the bottom section (light grey area) of each bar represents the stalls for reads, the black section shows the write miss penalty, and the section on top of that (grey area) represents the stalls due to the memory interface being busy or waiting due to the ORL or the write buffer being full.

4 Simulation Results

In this section, we present a comparative analysis of the performance achieved by the various architectural choices and show the impact of the memory models.

[3] SPEC is a trademark of the Standard Performance Evaluation Corporation.

Table 2: Statistics of benchmarks (for first 100 million instructions on 32K baseline cache)

Name	ratio over total instructions			miss ratio	proportion of cache miss (%)	
	data refs	reads	writes		read miss	write miss
Matrix	0.461	0.307	0.154	0.087	99.1	0.9
Tomcatv	0.418	0.326	0.092	0.063	82.4	17.6
Spice	0.258	0.209	0.049	0.116	98.7	1.3
Espresso	0.182	0.167	0.015	0.184	99.5	0.5
Doduc	0.301	0.223	0.078	0.017	58.7	41.3
Nasa	0.303	0.152	0.151	0.281	84.9	15.1
Fpppp	0.567	0.449	0.118	0.004	62.2	37.8
Gcc	0.338	0.223	0.115	0.018	65.3	34.7
Xlisp	0.467	0.315	0.151	0.014	65.5	34.5
Eqntott	0.299	0.265	0.035	0.033	79.2	20.8

4.1 Effect of Architectural Variations

Figure 3 shows the results for the benchmarks when simulated on the various architectures. In this first set of experiments, we used the *Pipelined* memory model so that we could temporarily ignore bandwidth limitations. Thus, there is no busy time penalty. We examine the data of Figure 3 according to three groups of architectures: (i) processors always stalling on a miss (blocking caches: BASE and PREFETCH), (ii) architectures with non-blocking writes and no bypass, and (iii) architectures with non-blocking writes and bypass of writes by reads. In the last two categories, we consider a baseline cache with non-blocking writes (WB), a prefetching cache with non-blocking writes (PREFETCH/WB), and a non-blocking cache (NBC) (cf. table 1).

A comparison between the baseline and the prefetching cache (the first two bars in the figures) shows a moderate to very significant reduction in the penalty for data access when the prefetching facility is added to the baseline cache. The access penalty is reduced by 96% for Matrix, 95% for Tomcatv, 19% for Spice, 27% for Espresso, 12% for Doduc, 36% for Nasa, 4% for Fpppp and Gcc, 41% for Xlisp, and 19% for Eqntott (geometric mean is 21%). The prefetching scheme can achieve reasonable gains at the cost of the RPT and the additional logic. The effectiveness of the prefetching technique relies mostly on the presence of regular data access patterns, hence the large gains in Matrix and Tomcatv. Also since the prediction occurs for both reads and writes, the prefetching has a proportion of write miss similar to that of the baseline case.

The effect of non-blocking writes on the baseline architecture is shown by the difference between the first, third (for no-bypassing), and sixth (bypassing) bars in the figures. Recall that in the baseline architecture, the processor stalls on a write miss until the write completes. In the non-blocking writes (WB) implementations, the write is put in the write buffer and the processor will stall at the next read operation in the case of no-bypass. In the case of bypass, the only case where stalls occur is on a read

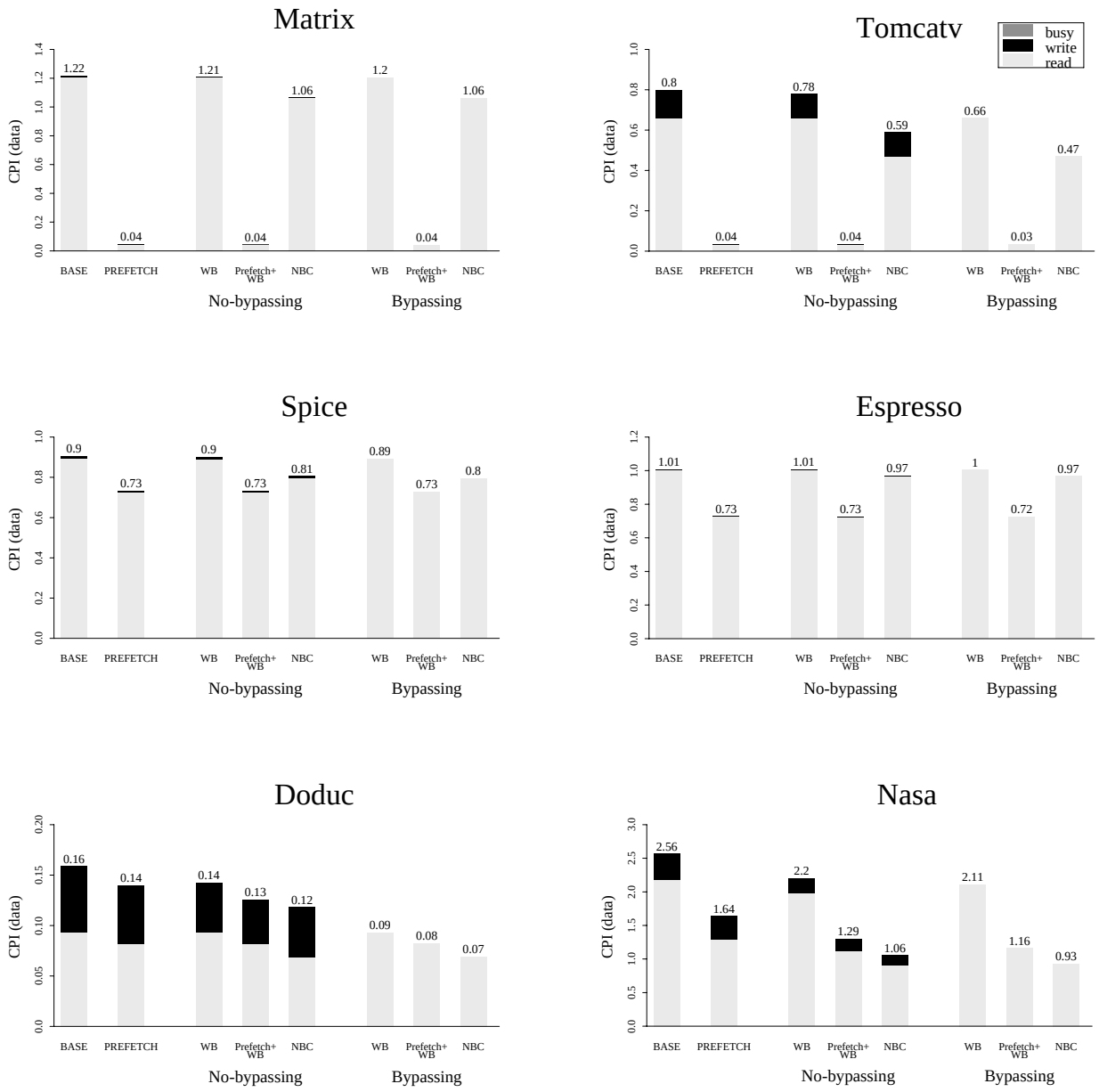


Figure 3: (a) Simulation Results for $\delta = 30$ (*Pipelined*)

miss – and the read will have priority over buffered writes – or if the write buffer is full. As can be seen, the WB with no-bypass has almost no effect on the write penalty (Nasa has a small gain). This is because the writes are most often followed by a read within a very small number of instructions. When the restriction of stalling on a subsequent read is lifted, i.e., WB with bypass, the penalty due to write misses is in essence totally avoided (cf. Tomcatv, Doduc, and Nasa). However, such a reduction by the WB may not contribute to a significant overall performance improvement over the total penalty when the fraction of write miss is very small (cf. Table 2). A surprising but subtle result is that a write buffer may even reduce slightly the read miss penalty (e.g., 12% reduction for Nasa). This reduction is

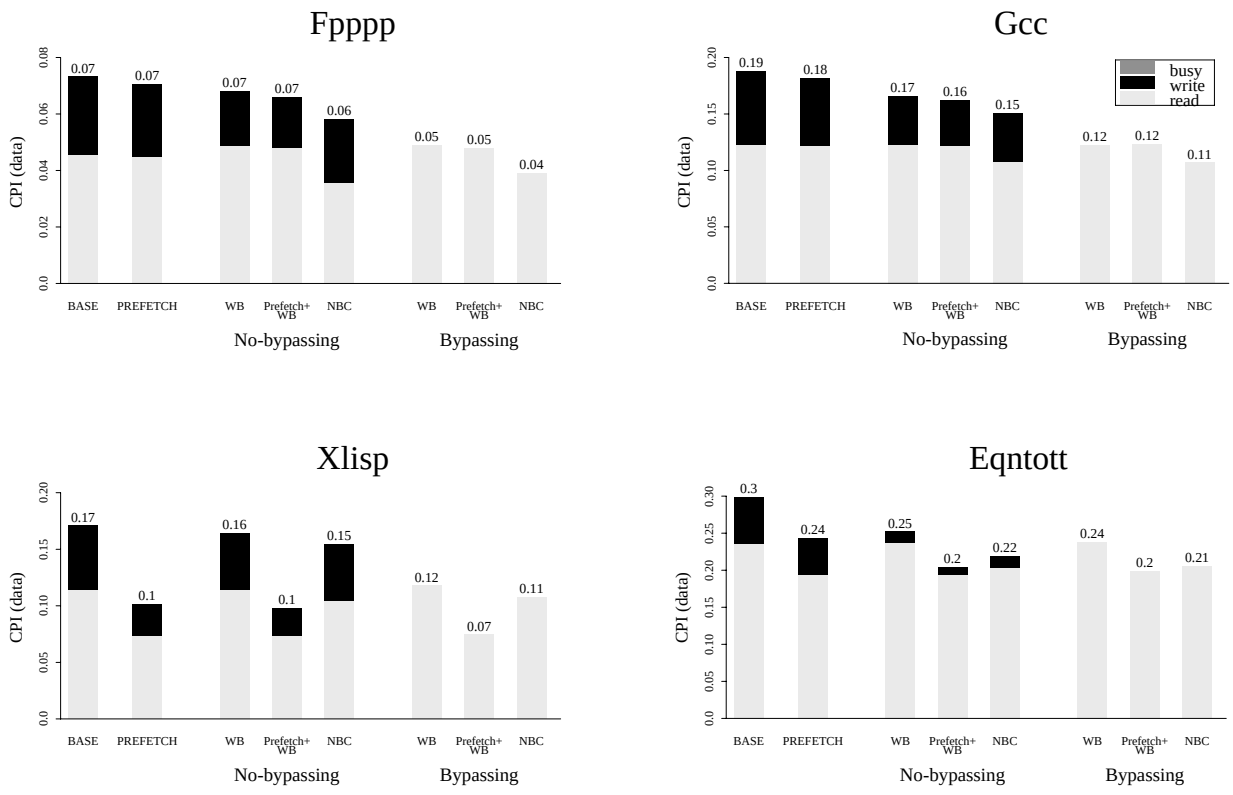


Figure 3: (b) Simulation Results for $\delta = 30$ (*Pipelined*)

a consequence of forwarding data from a write to subsequent reads.[4]

We now look at the performance of WB in conjunction with prefetching (PREFETCH/WB, fourth and seventh bars) and non-blocking caches (NBC, fifth and eight bars). The purpose of showing PREFETCH/WB is to give a fair base of comparison to contrast the effect of the read penalty reduction between the *pre*-miss overlap and the *post*-miss overlap. We focus only on WB with bypass. The results for no-bypass are qualitatively similar. The relative performances of NBC and PREFETCH/WB can be divided into three groups: i) PREFETCH/WB performs extremely well, ii) PREFETCH/WB has moderate improvement and also outperforms NBC, and iii) the performance of NBC is better than that of PREFETCH/WB.

The Matrix and Tomcatv benchmarks belong to the first group. These programs have very good reference predictability. Although a non-blocking cache contributes to some penalty reduction (12% for Matrix and 28% for Tomcatv), prefetching still significantly outperforms NBC since almost all the data access penalty has disappeared.

Spice, Espresso, Xlisp and Eqntott are the benchmarks in the second group. The effectiveness of NBC is even less than that in the first group (reductions of 10%, 3%, 8% and 12% respectively) but also is PREFETCH/WB's. As we shall see in Table 3 (Section 5), the average size of the basic blocks in

[4] There is also a small chance of increasing the read miss penalty since the penalty of a write miss is observed by a following read on the same line, but on a different word.

these two programs is smaller than that of basic blocks in the other programs. The small size usually restricts the prediction of references for PREFETCH/WB (due to the impact of branch prediction) and also implies a limited non-blocking distance. Therefore, for Spice and Espresso, PREFETCH/WB has some moderate gains over the baseline WB, and NBC is only slightly better than WB. Also, WB does not help much since the fraction of write misses is fairly low.

Doduc and Nasa form the third group where NBC becomes more attractive than PREFETCH/WB. NBC is quite efficient for the two programs and the reductions are larger than those attained through PREFETCH/WB. The weak performance of PREFETCH/WB in Doduc can be related to the size (or associativity) of the reference prediction table. Doubling the size of the table, or making it of larger associativity, would remove the large number of conflicts (35% with a 256-entry direct-mapped RPT). In the case of Nasa, both schemes lead to a fair amount of performance gain, with NBC showing an advantage.

PREFETCH/WB and NBC have little improvement over WB on the performance of Fpppp and Gcc. Since Fpppp has already a low miss ratio and a large loop size (a 256-entry RPT is too small), the improvement due to a prefetching scheme is very marginal. Gcc is a big program with many conditional branches. Since its predictability is very poor and the basic block size is small, prefetching will not be often done.

4.2 Effect of Large Latency

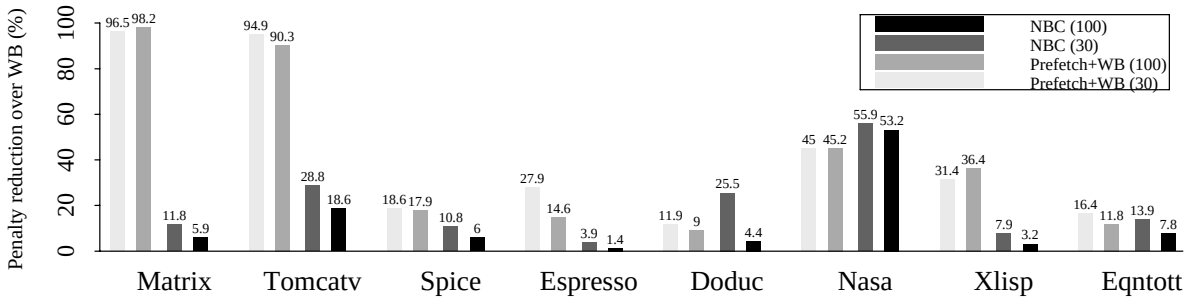
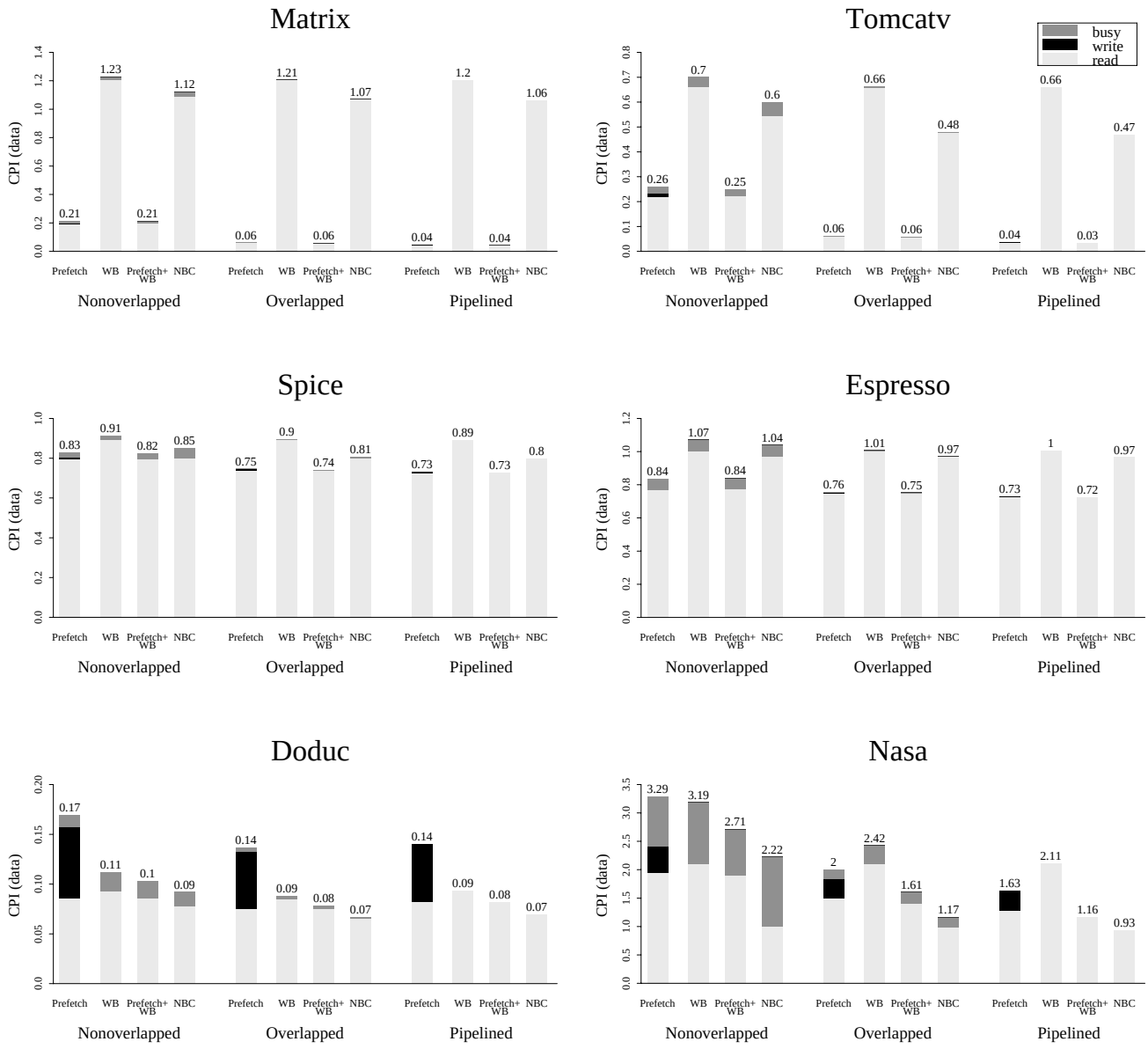


Figure 4: Effect of a larger latency (for $\delta = 30$ vs $\delta = 100$ *Pipelined*)

Figure 4 shows the experiment results of eight benchmarks[5] when the memory latency is larger. The figure plots percentages of reduction in data access penalty of PREFETCH/WB and NBC over WB with bypassing when the memory latency δ is either 30 (left bars) or 100 cycles (right bars). In general, the effectiveness of PREFETCH/WB is fairly insensitive to the (large) memory latency (except for Espresso, see below) and is more stable than NBC's. This is because the lookahead distance of the prefetching can be *dynamically* as large as the memory latency so that data may be prefetched early enough to hide the latency. In contrast, the non-blocking distance, which is *statically* determined by the programs, becomes relatively small when the latency increases. Thus NBC's relative effectiveness is reduced significantly (almost a factor of 6 in Doduc). Note, however, that the predictability will

[5] Fpppp and Gcc are not shown because comparisons of the marginal improvement could be misleading.



PREFETCH is blocking ; WB is with *bypassing*
In the *Overlapped* model, the request takes 2 cycles,
the memory latency is 20, and the transfer takes 8 cycles.

Figure 5: (a) Effect of memory models for $\delta = 30$

decrease as the latency increases mostly because branch prediction becomes less reliable. The program Espresso, where the average size of basic blocks is 5.6 instructions, is an example with poor reference predictability . As can be seen, the PREFETCH/WB's effectiveness is cut in half when δ increases from 30 to 100.

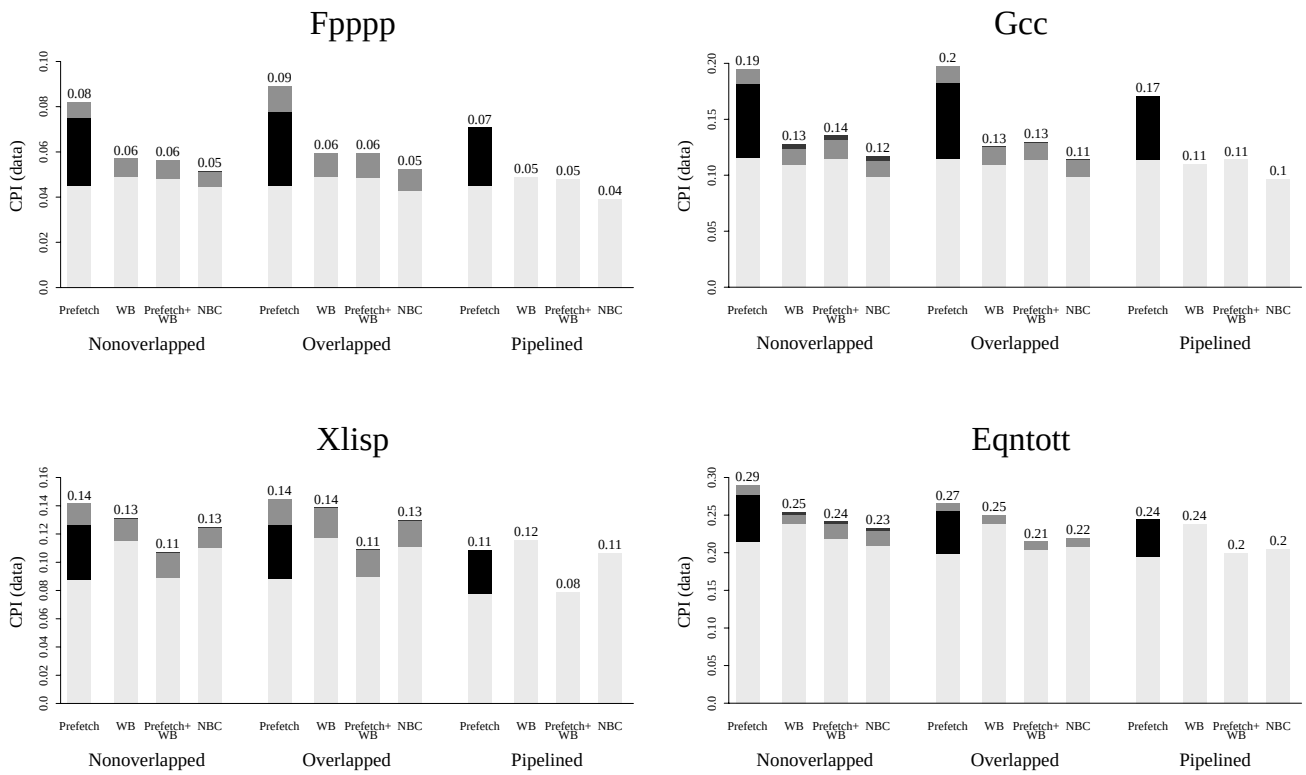


Figure 5: (b) Effect of memory models for $\delta = 30$

4.3 Impact of Memory Models

Figure 5 presents the data access penalties of some of the architectural choices with respect to the three memory models. An additional component of the access penalty, namely *busy time*, is now present in the *Overlapped* and *Non-overlapped* models since the interface cannot always forward a request at the next cycle as in the *Pipelined* case. As could be expected, the stall penalty increases when memory bandwidth is restricted. This is especially noticeable in three of the benchmarks: Espresso, Doduc, and Nasa. It indicates that an adequate interface is necessary to meet the memory bandwidth demand of the prefetching and non-blocking techniques that exploit the overlap of computations with multiple misses. In particular, for Nasa, a large portion of busy time is eliminated when passing from the *Non-overlapped* model to the *Overlapped* and then the *Pipelined* models. The primary reason is that the average interval of time between (data) cache misses is 12 cycles. Clearly, the bandwidth of an interface like the *Non-overlapped* model with $\delta = 30$ is insufficient.

One interesting observation regarding the comparison between PREFETCH and NBC is that when moving to a larger bandwidth, both the read miss and busy time portions of the penalty in Prefetch are reduced, whereas, in NBC, only the busy time portion diminishes. The reason is that when the bandwidth is sufficient, the prefetching will bring some blocks in the cache sufficiently ahead of time and thus avoid some of the misses that would be generated otherwise.

5 Compiler Assistance for Non-blocking Loads

In this section, we consider code generation optimizations for non-blocking loads. We examine two kinds of optimization: instruction scheduling for exploiting a possibly large non-blocking distance within a basic block and register renaming for removing false dependencies before the instruction scheduling is applied.

5.1 Instruction Scheduling

Several instruction scheduling techniques based on the architecture of the target machine have been proposed in the literature [14]. The instruction scheduling that we study here, based on the scheme given by Gibbons and Muchnick [7], is performed after register allocation. The goal of the scheduling algorithm is to create as much distance as possible between a load and the first instruction dependent on that load. At the same time, we want to intersperse the loads so that the lack of memory bandwidth does not become too much of a constraint. The algorithm schedules instructions only within basic blocks. Instructions within the block are the nodes of a weighted directed acyclic graph (DAG). Edges represent dependencies and are labeled with latencies. The latency of an edge between two dependent nodes is one except when the first instruction is a load. To achieve a maximum possible non-blocking distance and to avoid the clustering of loads at the beginning of a schedule, we estimate the latency of a load edge as the minimum of either the size (in number of instructions) of the basic block size divided by the number of loads in the basic block, or the actual memory latency. Once the latencies of the edges have been determined, we can assign weights to the nodes of the DAG, with the weight of a node being the number of child nodes plus the maximum (over its children) of the sum of the weight of a child and of the weight of the edge leading to the child. After the weighted DAG is built, we apply a list scheduling algorithm to derive the final schedule (see Appendix A).

The way to estimate equal latencies of loads within a basic block could be improved if there were more compiler information. For instance, a load of an array element with a large stride is likely to be a cache miss while accesses to the stack area will most often result in cache hits. An intelligent compiler could take this into account when assigning edge latencies.

Register renaming at compile time has been used in conjunction with software pipelining [11]. The purpose of register renaming, similar to that of dynamic instruction scheduling, is to remove write-after-read (WAR) and write-after-write (WAW) dependencies, thus allowing greater freedom in moving instructions around. The algorithm we use first identifies the live ranges (from a new definition to the last use before the next definition) for each register to be renamed (local registers). Then, for the live ranges entirely falling within the basic block, except the last live range, the destination register used in a load operation is replaced by a new register. This renaming is carried on for those instructions using the same register within the live range. Since the scheduling is performed after register allocation, we assume that there is a set of “spare” registers available. This is in order to keep the algorithm simple. Otherwise, we would have to identify temporary registers and unused registers in the basic block and our algorithm would become global rather than being restricted to the basic block level. After the register renaming process, we apply the instruction scheduling described above on the new DAG from which some false dependence edges have been removed.

A potential criticism of our study is that we adversely increase the register pressure in a basic block. A compensating factor is that WB may help the extra spilling store/load instructions that could be generated by buffering writes. Our point is that we give priority for register use to a load operation with a large latency, even at the cost of adding spill code. Although the results of our register renaming procedure are optimistic since we do not limit the number of registers, the approach is still feasible if the compiler identifies the unused registers or performs a priority-based register allocation [4] by taking into account the cost of data access penalty.

5.2 Effect of Instruction Scheduling

Table 3 shows the effect of the reordering of data accesses. All the data is in "weighted average" form where the weight is the execution frequency of the individual basic blocks. The data in the columns "non-blocking distance" are the average numbers of instructions between a read access and the subsequent instruction dependent on the value being read, regardless of whether the read is a cache miss or not. In general, the non-blocking distance based on the original code is fairly small.[6] A comparison between the second and third columns of the table shows that the instruction scheduling algorithm is very effective for increasing the non-blocking distance for Matrix and Tomcatv. This indicates that in these two benchmarks there are several data loading phases followed by computations on that data. The scheduling algorithm reorganizes the instructions to allow more overlap between the independent loading phases. For the other benchmarks[7] that do not have this characteristic, the distance is moderately increased.

Table 3: Average of basic block size, non-blocking distance, and "spare" registers needed (weighted by frequency of execution)

Name	Basic block size	Original code	scheduling only	Renaming + scheduling	
		non-blocking distance	non-blocking distance	non-blocking distance	"Spare" regs needed
Matrix	39.15	2.23	8.33	15.14	8.65
Tomcatv	54.47	3.38	10.03	11.91	12.13
Spice	7.79	2.46	3.27	3.83	0.05
Espresso	5.63	1.47	2.25	2.93	0.11
Doduc	10.01	3.44	4.88	8.14	0.77
Nasa	27.49	2.09	4.89	8.17	0.28

When register renaming is added to instruction scheduling, the compiler has more flexibility to optimize the code reordering. As shown in Table 3, a significant increase in non-blocking distance is achieved

[6] Note that the original MIPS compiler should strive to yield a non-blocking distance greater than 1 since the R3000 has a delayed load of one cycle.

[7] In the following discussion, we omit the results of the other benchmarks because the effects of Xlisp (4.97) and Eqntott (3.84) with small average block sizes are similar to those of Espresso and Spice.

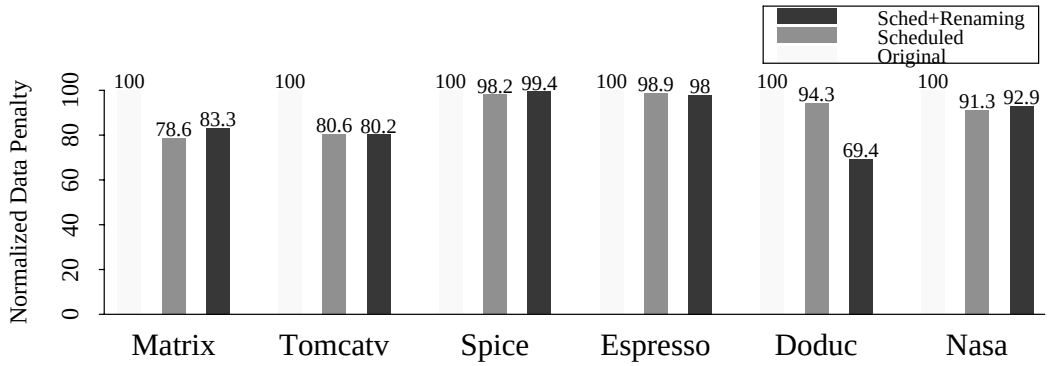


Figure 6: Effect of instruction scheduling on NBC for $\delta = 30$ (*Non-overlapped*)

in Doduc and Nasa with the use of a *small* number of extra registers (less than one per block on the average). On the other hand Matrix and Tomcatv need more registers with not much improvement for the latter. Note that the number of registers required for renaming is overestimated, since two live ranges, which are originally far apart, are less likely to be live at the same time after renaming because of other dependence chaining between them. This was not taken into account in our algorithm but could be checked out by the compiler.

Figure 6 and Figure 7 show the relative performance of the optimizations for the NBC architecture. The data access penalty for the two code optimization algorithms is normalized to the penalty of the original code. The results of the *Non-overlapped* model in Figure 6 show that only those programs with low miss ratios (Matrix, Tomcatv, and Doduc) can benefit from instruction scheduling. This is not surprising because the *Non-overlapped* model does not provide sufficient bandwidth to fully exploit the advantage of the overlap. Also, register renaming does not contribute much performance gain to the NBC in the *Non-overlapped* model, and it might even degrade the performance slightly (cf. Matrix). Instruction scheduling tends to increase the clustering of read accesses. Scheduling instructions which have no false dependencies by applying register renaming causes the read accesses to be more clustered.

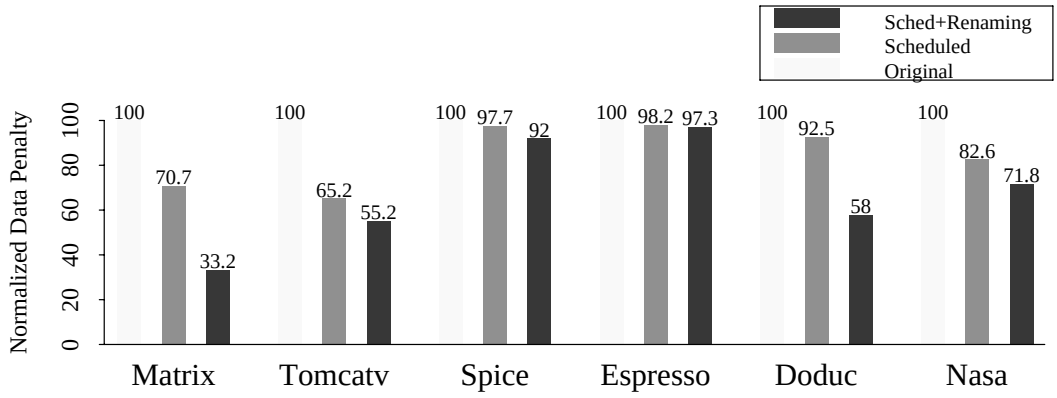


Figure 7: Effect of instruction scheduling on NBC for $\delta = 30$ (*Pipelined*)

When the *Pipelined* model is assumed (shown in Figure 7), the clustering of reads becomes an ad-

vantage that can be exploited by the NBC. In all cases except Espresso and Spice, the experiments show significant gains from instruction scheduling (improvement varies from 2% for Espresso to 35% for Tomcatv). Even better results are achieved when register renaming is applied before instruction scheduling (improvement varies from 3% for Espresso to 67% for Matrix but recall that the results are optimistic). The geometric mean of penalty reduction by instruction scheduling for those benchmarks is 9.5% over the original code and when register renaming is added, this geometric mean is up to 24% over the original code. This illustrates that instruction scheduling and register renaming provide an inexpensive solution to help hide the large memory latency for non-blocking loads in processors whose design is based on static instruction scheduling. However, techniques of instruction scheduling across basic block boundaries should be further investigated.

6 A Hybrid Design

Since prefetching and non-blocking caches are not mutually exclusive, a further enhancement would be to combine the two schemes: a prefetch hint is provided prior to the load instruction and the binding of a loaded value with a register is delayed until the value is actually used. This hybrid design is attractive since the combined scheme can tolerate the drawbacks of poor predictability and of short non-blocking distance. However, the cost to be paid is that of an RPT and associated logic for prefetching, an ORL (or MSHRs) that can be searched associatively, and register interlocks for the non-blocking caches.

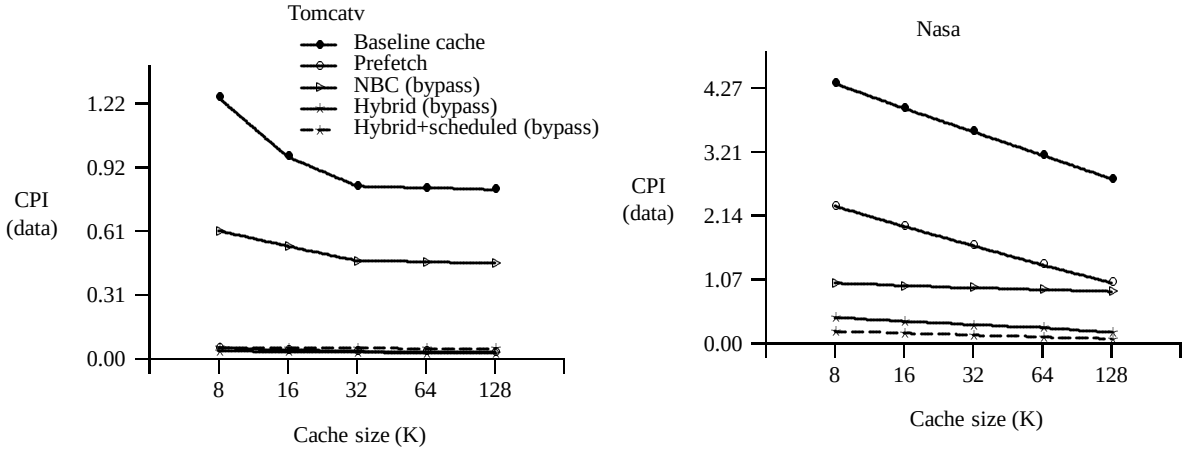


Figure 8: Hybrid design on varying cache size $\delta = 30$ (*Pipelined*)

In figure 8, we present the results of the simulation of such an hybrid scheme with and without instruction rescheduling when compared to the baseline cache, a prefetching only scheme, and a non-blocking cache with bypass. We vary the cache size from 8K bytes to 128K bytes and show only two benchmarks: Tomcatv where prefetching was performing much better than NBC, and Nasa where the converse was true. In Tomcatv, the prefetch scheme already had reduced the data access penalty to only a few hundredths of a cycle. The hybrid design has now nearly ideal performance. The performance of the hybrid scheme has more dramatic effects in Nasa. The data access penalty that was far from being negligible if either prefetching or NBC was applied alone becomes small even at the smallest cache size. Code optimization helps the hybrid combination further so that only 4% of the initial penalty incurred with a

baseline cache remains. These results indicate that the length of the overlap from *pre*-miss to *post*-miss can be large enough to cover the memory latency to a great extent. The additional cost paid for the hybrid design is justified by the significant performance improvement.

7 Conclusion

In this paper, we have compared the effectiveness of prefetching caches, write buffers, and non-blocking caches in exploiting the overlap of data accesses with computation. These comparisons were made using the SPEC benchmarks and simulations were performed on a cycle by cycle basis. Three models of memory interface were used with each model showing an increasing possibility of concurrency of access to the next level of the memory hierarchy. The results show that a prefetching cache can eliminate significantly and, in some cases, almost entirely the data access penalty. On the average, the percent reduction was 35% over a baseline cache. We confirmed previous studies showing that buffering writes while allowing bypass of reads can eliminate entirely the write miss penalty. When the non-blocking write with bypass is used as a basis, the average percentage of read penalty reduction by prefetching caches was 35%, whereas the average percentage of read penalty reduction by non-blocking caches was 16%. Also, the effectiveness of prefetching caches is less sensitive to a large memory latency than that of non-blocking caches.

Code optimization via instruction scheduling can reduce prominently the data access penalty in the case of non-blocking caches. We have presented a local (at the basic block level) algorithm that, on the average, reduced the penalty by 9.5%. With the addition of an (optimistic) renaming scheme, this reduction went up to 24%. These results illustrate that a non-blocking cache assisted by a good code optimizer and associated with a statically scheduled processor can achieve remarkable gains at a cost of less complicated hardware complexity than what is needed for a dynamically scheduled processor.

Finally, we have proposed a hybrid design incorporating features from both prefetching and non-blocking caches. We have showed that the combination of pre-miss overlap and post-miss overlap present in such a scheme can be very effective in hiding large memory latencies.

This study should be expanded to cover for example (1) cache hierarchies – which levels should be prefetching, which levels should be non-blocking, and what is the influence of the block size are questions which come immediately to mind, (2) superscalar processors where the pressure on first level caches will be increased – how non-blocking caches are to be incorporated should be investigated, and (3) shared-memory multiprocessors – what is the impact of the non-blocking or the hybrid approach on memory consistency is one of the many issues to be considered.

References

[1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91*, pages 176–186, 1991. Also TR 91-03-07, Department of Computer Science and Engineering, University of Washington.

[2] J.-L. Baer and W.-H. Wang. Multi-level cache hierarchies: Organizations, protocols and performance. *Journal of Parallel and Distributed computing*, 6(3):451–476, 1989.

[3] B. K. Bray and M. J. Flynn. Writes caches as an alternative to write buffers. Technical Report CSL-TR-91-470, Stanford University, April 1991.

[4] F. C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[5] K. Gharachorloo, A. Gupta, and H. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Proc. ASPLOS-IV*, pages 245–259, 1991.

[6] K. Gharachorloo, A. Gupta, and H. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proc. of the 19th Annual Int. Symp. on Computer Architecture*, 1992.

[7] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. of SIGPLAN Symp. on Compiler Construction*, July 1986.

[8] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessor with memory hierarchies. In *Proc. 1990 Int. Conf. on Supercomputing*, pages 354–368, 1990.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[10] R. Jain. *The Art of Computer System Performance Anaylsis*. John Wiley & Sons, Inc., 1991.

[11] S. Jain. Circular scheduling: a new technique to perform software pipelining. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 219–228, 1991.

[12] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of the 17th Annual Int. Symp. on Computer Architecture*, pages 364–373, May 1990.

[13] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proc. of the 18th Annual Int. Symp. on Computer Architecture*, pages 43–53, 1991.

[14] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[15] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of the 8th Annual Int. Symp. on Computer Architecture*, pages 81–87, 1981.

[16] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, pages 6–22, January 1984.

[17] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessor. *Journal of Parallel and Distributed computing*, 12(2):87–106, June 1991.

[18] A. K. Porterfield. Software methods for improvement of cache performance on supercomputer application. Technical Report COMP TR 89-93, Rice University, May 1989.

[19] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[20] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processor. In *Proc. ASPLOS-IV*, pages 53–62, April 1991.

[21] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc. 1989 Int. Conf. on Supercomputing*, pages 273–280, 1989.

A Instruction Scheduling Algorithm

1. Build DAG $G(V, E)$ for a basic block:

   Each instruction is a vertex $v_i \in V$; an edge $e(v_i, v_j) \in E$ if v_j depends on v_i.
   $l(v_i, v_j)$ is the estimated latency between nodes v_i and v_j:

$$l(v_i, v_j) = \begin{cases} \frac{\text{basic block size}}{\text{\# of loads}} & v_i & v_j & e(v_i, v_j) \\ & load & any\ other^a & \text{true dependency} \\ 1 & any\ other & any & \text{true dependency} \\ 1 & any & any & \text{false dependency} \\ 1 & leaf & branch & \text{control dependency} \end{cases}$$

   [a] Any instruction node other than load

2. Define $weight(v_i)$:

$$weight(v_i) = \begin{cases} 0 & \text{if } v_i \text{ is a leaf node} \\ n - 1 + \underset{1 \le k \le n}{MAX}\{l(v_i, v_{j_k}) + weight(v_{j_k})\} & \\ \quad \text{where } v_i \text{ has } n \text{ child nodes } v_{j_1}, \ldots, v_{j_n} & \end{cases}$$

3. Schedule the instructions based on DAG

   The scheduling algorithm is a variation on list scheduling. Several sets of nodes are maintained: S_{ready} (a set of vertices that have all their predecessors already scheduled) and $S_{slot[i]}$ where i varies from 1 to the largest estimated latency. Whenever a node v_i from S_{ready} is scheduled, if it was the only unscheduled predecessor of its child node v_j, the latter is included in the set $S_{slot[l(v_i, v_j)]}$. When an instruction is scheduled, all of $S_{slot[i]}$ sets are shifted "left" by one slot with $S_{slot[1]}$ joining S_{ready}. When there is no instruction available in S_{ready}, we do not insert a NOP, but simply keep moving $S_{slot[i]}$ until S_{empty} is not empty.

```
procedure reorder (G)
    initialize S_slot[i] with empty pointer
    S_ready = {v_i | v_i has no parent node in DAG}
    new_order = 1
    while new_order ≤ length of BB
        while S_ready is empty
            S_ready ⇐ S_slot[1] ⇐ ... ⇐ S_slot[n]

        choose a node v_i in S_ready, where weight(v_i) is largest.
        order(v_i) = new_order^++
        for each child v_j of v_i (with an edge latency l)
            if v_j has no other unscheduled parent then
                S_slot[l] = S_slot[l] ⋃ {v_j}

        S_ready = S_ready ⋃ S_slot[1]
        S_slot[1] ⇐ ... ⇐ S_slot[n]
    end
end
```