Hardware Support for Hiding Cache Latency

University of Michigan Technical Report

Michael Golden and Trevor N. Mudge

Advanced Computer Architecture Lab
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract

As the decrease in processor cycle time continues to outpace the decrease in memory cycle time, even moderately sized on-chip caches may require several cycles of access time in the near future. This means that time is lost, even on a cache hit, if independent instructions cannot be scheduled after a read from memory. A novel hardware device is proposed that keeps track of the history of load instructions and predicts their targets before they are computed by the instruction pipeline. This allows the saving of several processor cycles. The storage required to implement such a device is quite large, but as the latency required to read from the first level cache grows, a moderate performance improvement is seen.

1.0 Introduction

As processor speeds increase to higher and higher levels, the need for a fast memory system becomes more pronounced. In the past, a small, fast first-level cache was adequate to match the memory speed to the processor cycle time[3]. These caches could be accessed in a single cycle to prevent memory from being a bottleneck, except in the case of a cache miss. Unfortunately, the moderately sized on-chip caches of current high-performance microprocessors have multiple-clock-cycle access times. The MIPS R4000 also has an on-board 8 KB data cache and three cycles of load latency[13]. If the delay slots of load instructions on this machine cannot be filled with instructions that do not depend on the load, cycles will be wasted as the CPU waits for the memory system, even on a cache hit.

For an instruction cache, this latency can be effectively hidden through architectural solutions such as a buffer into which future instructions can be prefetched. Much time has been invested in researching the various techniques of instruction prefetching[6]. Highly accurate branch prediction schemes, both static and dynamic, have been developed to make this process effective[7, 20, 12, 16].

A large latency in accessing the data cache presents a more difficult problem. Write buffers can eliminate the bottleneck in storing data to the memory system[6], but the loading of data cannot be effectively buffered in this way because the results are desired immediately. One method of hiding the memory latency of a load instruction is to pipeline the cache, thus allowing the issue of a memory instruction every cycle, and properly schedule the instructions to hide the load latencies. This method is used in polycyclic vector scheduling[19] for the inner loops of scientific code. Its use is explored for more general programs in[14].

Sohi and Hsu describe a hardware method of eliminating some of the memory latency by constructing an intermediate memory between the processor and the first level of the memory hierarchy[18]. This memory acts as a back-up register file. Data can be moved between the memory hierarchy and the intermediate memory and between the register file and the intermediate memory. These moves must be explicitly coded into the program being executed, and so this memory is dubbed a "programmable cache."

These techniques for hiding the load latency can be considered static methods, because they depend on the compiler's or the programmer's ability to properly schedule the code before it is executed and, unless some profiling techniques are used, they do not use run-time information in their decision making processes.

This paper proposes a dynamic technique for hiding the delays caused by a slow primary cache. It is a small cache memory loosely based on the concept of a branch target buffer, so it will be called a *load target buffer.*

A branch target buffer is a small cache that is accessed by the memory address of the current instruction being fetched[12]. The buffer contains the addresses of branch instructions, their predicted targets, and some kind of state machine that uses past results to predict which branches will be taken on the next execution. Because this buffer is

accessed during the instruction fetch stage, the predicted result of a branch instruction that lies in the buffer is available after a single cycle of delay, instead of being available further down the pipe when the branch is actually executed. This allows immediate fetching of the next instruction in the dynamic instruction stream.

A load target buffer performs the same function for instructions that read from memory. During the instruction fetch pipeline stage, the address of the instruction that is being fetched is used to access the LTB. Each entry of the LTB contains an address, which is the predicted target of that load instruction. A load command is immediately issued to the memory system. By the time the results of the load are needed, two or three stages down the pipe, the results of the speculative load have returned from the memory system. The predicted target address and the real target are compared, and if they match, the load latency has been successfully hidden.

This system works well for branch target buffers because, except for indirect branches, branch instructions all have a single target. Load instructions can have a single target but many of them use a register to allow indexed access to memory. This necessitates a more complicated scheme for determining the next target of a load. To allow better target prediction, the following information can be added to each entry in the load target buffer:

- The address of the previous load target.
- The "stride" of the load instruction, which is the difference between the two previous targets
- Some status bits to allow better prediction schemes.

The address of the previous load target allows the buffer to make a realistic prediction for the next load target, even when the current guess is incorrect. The stride field allows the buffer to make good predictions when the load instruction is used to scan an array in a regular fashion. The status bits allow extra information to be included in the buffer, such as putting an "inertia" onto the stride to lessen the effects of anomalous changes in the load target. This inertial effect is similar to the use of two or more bits in branch prediction in a branch target buffer[20,12].

For this scheme to effectively hide the load latency, it must be able to correctly predict a high percentage of the load targets. Alternatively, this could be combined with one of the static methods of solving the problem. Code scheduling will be able to hide slow memory part of the time, and those loads that cannot be effectively rescheduled will be marked for storage in the load target buffer.

At least two other papers have proposed methods that issue memory loads early in order to hide the latency. J.K. Illife describes a "forward looking" architecture that immediately issues a memory load whenever a potential address is formed instead of waiting for an actual load instruction to be encountered in the instruction stream[9]. A potential address is created through the normal machine instructions that have a destination register. In Illife's machine, registers are tagged. When a value is stored in an address register, a potential address is formed, and the machine issues a load to that address.

Sohi and Davidson describe the Structured Memory Access architecture, or SMA. This machine has an address processing unit that can accept a pattern in memory and issue loads to all addresses in the pattern before the values are actually used[17]. This feature works well to exploit the natural regularity of memory accesses to structures like vectors and multidimensional arrays.

2.0 Operation of the LTB

Like a branch target buffer, the load target buffer is accessed at the beginning of the instruction fetch cycle. The address of the desired entry is obtained by performing some hashing operation on the address of the instruction being fetched. In its simplest form, this will involve removing the higher order bits of the address.

Throughout this paper, the assumption will be made that the LTB is attached to the MIPS R2000/R3000 instruction pipeline[11]. This architecture was chosen because of its simpleness and generality. All of the descriptions and results can be generalized without too much difficulty. Figure 1 shows this pipeline and how data flows from it to the LTB and the memory system.

The LTB is indexed with the instruction address during the instruction fetch stage of the pipe. If a valid entry exists in the buffer under this address, the buffer immediately issues a data fetch to the memory system using the predicted target field of this entry. This data fetch should be initiated by the end of the instruction fetch cycle. Further down the pipeline the actual target is generated. In the MIPS pipeline, this takes place at the end of the Execute stage, two cycles later. This actual target is compared to the prediction. If the prediction is incorrect, the predicted load must be squashed and a new one issued, and time is neither saved nor lost. If the prediction is correct, the number of cycles between the instruction fetch stage and the stage that generates the actual target have been saved. In the case of the MIPS, this is two cycles. So a total of three cycles of latency can be tolerated without slowing down the pipe with load delay slots.

If the system has been implemented with an instruction prefetch buffer, the LTB can be exercised with addresses in the prefetch buffer before they ever reach the pipeline. Doing this allows predicted targets to be sent to the memory system at an even earlier time, enabling the system to tolerate even more cycles of load latency. Unfortunately, without perfect branch prediction, some of the load instructions in the prefetch buffer might not actually be executed. Because current branch prediction methods are so accurate, this paper will assume perfect branch prediction.

While these comparisons are being made, the entry in the load target buffer must be updated. The new stride is calculated by a subtraction:

```
new stride = actual target - previous target
```

If the inertial prediction scheme is used, the buffer is updated with the new stride only if the inertia bit is set. The new prediction is computed as follows:

```
new prediction = actual target + new stride
```

Finally, the `previous target` field is updated with the `actual target`.

This is explained in more detail in Section 3.0, where a layout of the fields of an LTB can be found.

If the above computations can be computed in one cycle, the buffer can be updated at the end of the MEM cycle. This allows a particular entry to be accessed every four cycles. This is the limit, in the MIPS architecture, of a practical loop. Certain pathological cases can reduce the loop size (See Table 1). The non-pathological loop scans through an array for the first non-zero element. Loops of this kind are especially common in the string handling libraries.

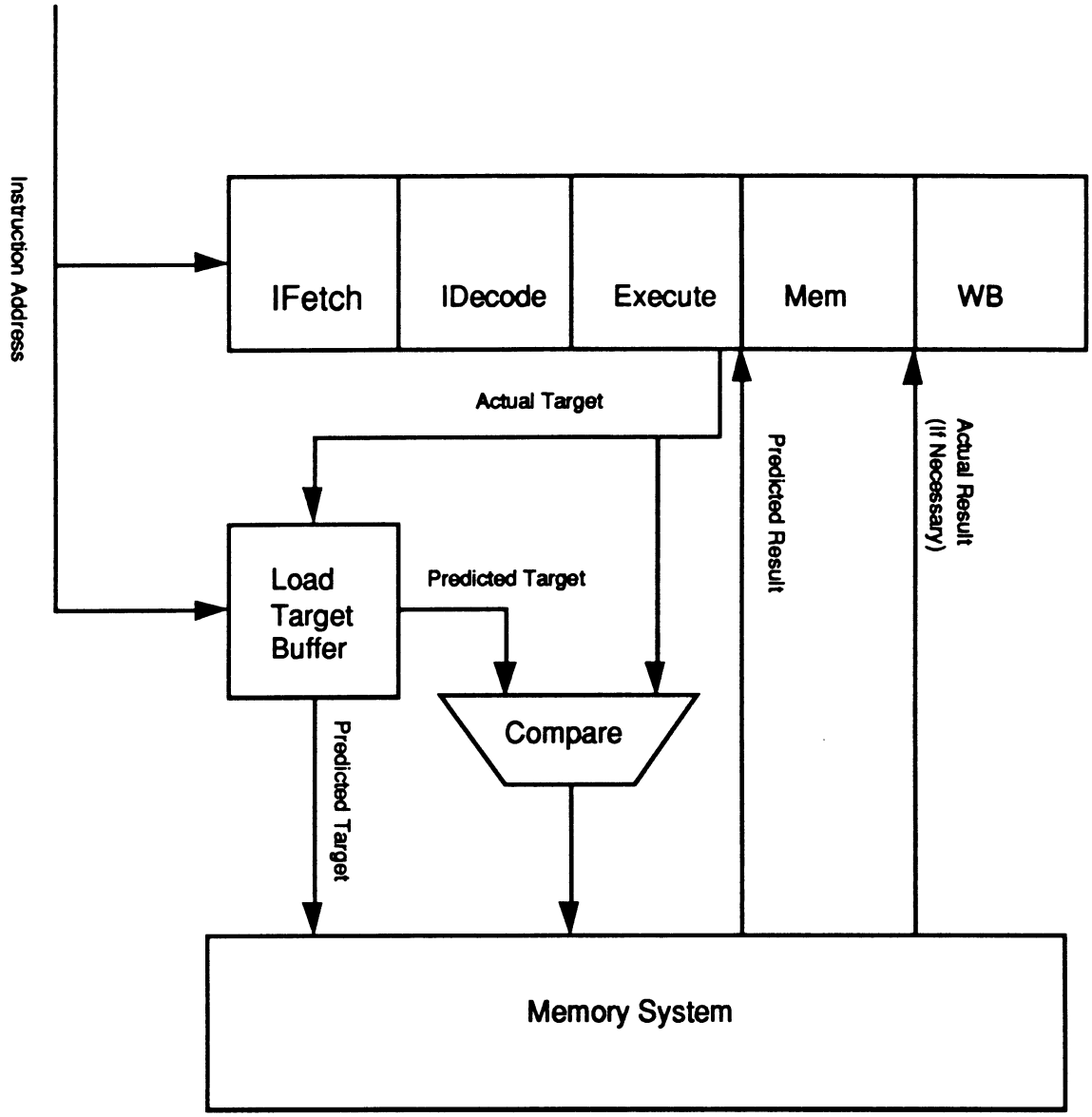
The pathological loop can cause a particular load instruction to be executed every two cycles. This particular sequence of instructions could either be forbidden by the compiler or could freeze the pipeline. The former solution seems quite practical as it involves no extra hardware and the loop is so pathological as to be unlikely in real code. In the benchmarks used in this paper, the tightest loops issued a load every four cycles. Section 4.2 shows that these four-cycle loops and other small loops occur quite rarely.

TABLE 1. Small Loops

Non-Pathological Small Loop			**Pathological Small Loop**		
label:	lb	t0,0(a0)	label:	bne	r3,zero,label
	nop			ld	r3,0(r3)
	bne	t0,zero,label			
	addi	a0,a0,1			

This functional description assumes a latency in the first level data cache of only three cycles and successfully hides the latency of that cache. If the latency is longer, the load target buffer could be accessed using some queue of prefetched instructions maintained by the processor. This would increase the cache latency that could be hidden, but would degrade the performance of the buffer by causing it to issue data fetches for load instructions that would not have been executed. This performance degradation varies inversely with instruction prefetch accuracy.

FIGURE 1. Data Flow in the Load Target Buffer



3.0 Prediction Strategies

As with branch target buffers, the choice of a prediction strategy has a great effect on the accuracy of the buffer. The more information that the buffer stores, the more likely a correct prediction becomes. Unfortunately, extra information makes the buffer larger and therefore slower. Some trade-off between speed and accuracy must be found.

This paper considers two different prediction algorithms. The first simply uses information provided by the previous and the current executions of the load instruction to predict the next target. The buffer computes a "stride", which is the difference between these two targets, and adds the stride to the current target to get the prediction.

This algorithm is similar to using a single bit of prediction information in a branch target buffer. If there is an anomalous stride, the buffer will predict the wrong target twice instead of once. One misprediction will be caused by the anomaly itself. A second will be caused by the incorrect stride that is used to make the following guess.

This situation can be tolerated with the addition of an "inertia" bit to the buffer. With the addition of this bit, the new algorithm requires the actual stride to change in two consecutive accesses for the stride field in the buffer to be reset. On the first stride change, the inertia bit is set to one, but the stride field remains the same. On the second stride change, the inertia bit is reset, and the stride field is changed to the new stride. The effects of this can be seen in Table 2 and Table 3.

TABLE 2. Non-Inertial Prediction

Actual Load Target	LTB Prediction	LTB Previous Target	LTB Stride	Miss
100	-	-	-	miss
104	100	100	0	miss
108	108	104	4	
10c	10c	108	4	
110	110	10c	4	
114	114	110	4	
100	118	114	4	miss
104	0f2	100	ffff fff2	miss
108	108	104	4	
10c	10c	108	4	
110	110	10c	4	
114	114	110	4	
100	118	114	4	miss
104	0f2	100	ffff fff2	miss
108	108	104	4	

As the tables demonstrate, the simple prediction strategy "warms up" more quickly. After only two misses, hits begin to take place. Unfortunately, if the loop which contains the load is re-executed, there are two misses in the load target buffer. The first miss is caused by the resetting of the target to the beginning of the targets to which the instruction points. The second miss occurs because the stride becomes erroneous when the large jump in the target takes place.

The inertial strategy takes longer to warm up. Two misses are required for the stride to change, so three misses are required for the buffer to correctly find the pattern of the load instruction. But when the loop is re-executed, the resistance to changes in stride decreases the number of misses to one. Although the miss that occurs when the target is reset to its original value is inevitable, the buffer avoids a second miss by keeping the stride a constant and setting the inertia bit. The figure shows that the loop must be executed at least three times for the inertial method to have any benefit.

TABLE 3. Inertial Prediction

Actual Load Target	LTB Prediction	LTB Previous Target	LTB Stride	Inertia	Miss
100	-	-	-	0	miss
104	100	100	0	0	miss
108	104	104	0	1	miss
10c	10c	108	4	0	
110	110	10c	4	0	
114	114	110	4	0	
100	118	114	4	0	miss
104	104	100	4	1	
108	108	104	4	0	
10c	10c	108	4	0	
110	110	10c	4	0	
114	114	110	4	0	
100	118	114	4	0	miss
104	104	100	4	1	
108	108	104	4	0	

4.0 Experiments

4.1 Prediction ratio

The first interesting statistic is the rate at which load target buffers of differing sizes and prediction strategies can correctly predict the target address of a load. Because no prediction can be made if information for a particular load is not in the buffer, a larger LTB increases this "prediction ratio" by decreasing the number of dimensional conflicts. Naturally, a better prediction strategy will give better performance.

Figure 2 and Figure 3 show the prediction ratios for load target buffers of varying sizes for several benchmarks in the SPEC suite. These buffers are all direct mapped. They were simulated using a software package called RCM that was written by Tom Conte at the University of Illinois[4]. This package uses the inclusion property to simulate all cache sizes and associativities for a given line size in a single pass. Since a load target buffer has a fixed line size of one entry, this works quite quickly. This package was modified to simulate the contents of the cache in addition to the normal simulation of the address stream.

The figures reveal some important facts. First, the prediction ratio is very good, close to 100 percent, for matrix 500, no matter which strategy is used. This result should not cause great surprise since this benchmark merely performs several elementary operations on two

large matrices. These matrix operations have very regular, and therefore very predictable, data access patterns. For spice2g6, on the other hand, the best approach is use the inertial prediction strategy which gets around 61 percent accuracy. Nroff, espresso, and doduc seem to give "typical" results of around 70 to 80 percent.

The figures also show that an inertial strategy performs better than a non-inertial one. For matrix 500, nroff, and spice, the inertial strategy makes a minimal improvement. The inertial strategy gives doduc and espresso five and ten percent boosts, respectively. This definitely argues for the use of an inertia bit. Experiments were run that set the inertia bit to one instead of zero at the beginning of the simulation. The reasoning behind this strategy was that this would allow the buffer to "warm up" to the correct stride more quickly. Some of the results got better, and some worse, but none by more than a tenth of a percent.

The "knee" of the graph is around 1K entries. A buffer of this size would have most of the prediction power of any larger buffer. This is too large. 1K entries translates into 9K bytes since each entry requires four bytes for each address that is stored, and about one byte for the stride and inertia. Figure 4 shows that increasing the set associativity has little effect on the prediction ratio for a given LTB size, thus ruling out increasing the set size as a method of decreasing the LTB size. This graph is for espresso, and all of the benchmarks revealed that the associativity of the LTB has little effect on its performance.

4.2 Small gaps

The second set of experiments involved the concern with the ability to update the buffer in time for the next issue of the load instruction currently being predicted. Table 4 gives the number of times a single load instruction is re-executed in the dynamic instruction stream after a very small number of cycles. The distance in the instruction trace between two executions of a single static instruction is called the "gap". In the discussion above, it is assumed that all of the necessary arithmetic operations can be performed in a single cycle. This allows the buffer to be exercised on a single load instruction every four cycles. This rate of update is acceptable for all of the benchmarks and, as was speculated above, will probably serve for all practical programs.

On the other hand, it may not be advisable to dedicate a fast adder, which could be quite large and expensive, to the load target buffer. Furthermore, if the instruction pipeline is long or the LTB is indexed in an instruction prefetch buffer, the actual target will be generated more than four cycles after the LTB produces a prediction. These situations will decrease the frequency at which a particular load target in the buffer can be updated. If there is a small gap between two executions of a single load instruction, the LTB may not be updated with the actual target of the first execution when it is called upon to make a prediction for the second execution.

This causes two problems. First, it affects the validity of the simulation, which assumes that the LTB can always be updated with the actual target before it needs to make the next prediction. Second, in a real system these small gaps need to be handled in some way and cannot be "assumed" away. The small gaps could be handled by assuming that the predicted target is correct, and updating the LTB with that value. If the predicted target turns

out to be incorrect, some cleanup would need to be done. Another option would be to stall the pipeline until the LTB can be updated with the actual target. Both of these degrade performance, and the second option has the added difficulty of adding pipeline stalls.

Table 4 also gives the number of times gaps of slightly larger values occur. As the minimum gap that can be handled increases, the utility of the buffer decreases, since less and less load targets can be put into the LTB and effectively updated. Fortunately, as the table reveals, these small gaps occur quite rarely, and so performance will not be affected too much.

TABLE 4. Small Gaps

Gap Size	matrix 500	doduc	espresso	nroff	spice
1-3	0	0	0	0	0
4	0	16	1883	16	1892
5	0	764	11,254	1193	11,064
6	124,390	1626	97,973	2873	1485
7	2	18	136,825	438,545	11,608
8	874,496	14	466,887	757,498	141,757
9	2509	5027	550,663	366	618,403
10	121,899	646,928	343,503	5263	21,802
11	0	3985	1,558,694	0	936,752
12	126,247	3385	1.029,232	2543	244,860
13	7	7671	881,640	190	2,116,011
14	446	6175	481,074	435,085	1,460,258
15	116	116	297,734	6740	226,626
16	1,126,130	26	709,516	741,696	7,753,396
All Gaps	**49,166,626**	**9,574,879**	**23,663,246**	**19,088,025**	**88,515,490**
Gaps <= 8	998,888	2438	714,882	1,200,125	167,806
(percent)	2.032%	0.025%	3.021%	6.287%	0.190%
Gaps <= 16	2,376,242	675,751	6,562,878	2,392,008	13,545,914
(percent)	4.833	7.058%	27.73%	12.53%	15.30%

FIGURE 2. Prediction Ratio with Non-Inertial Algorithm

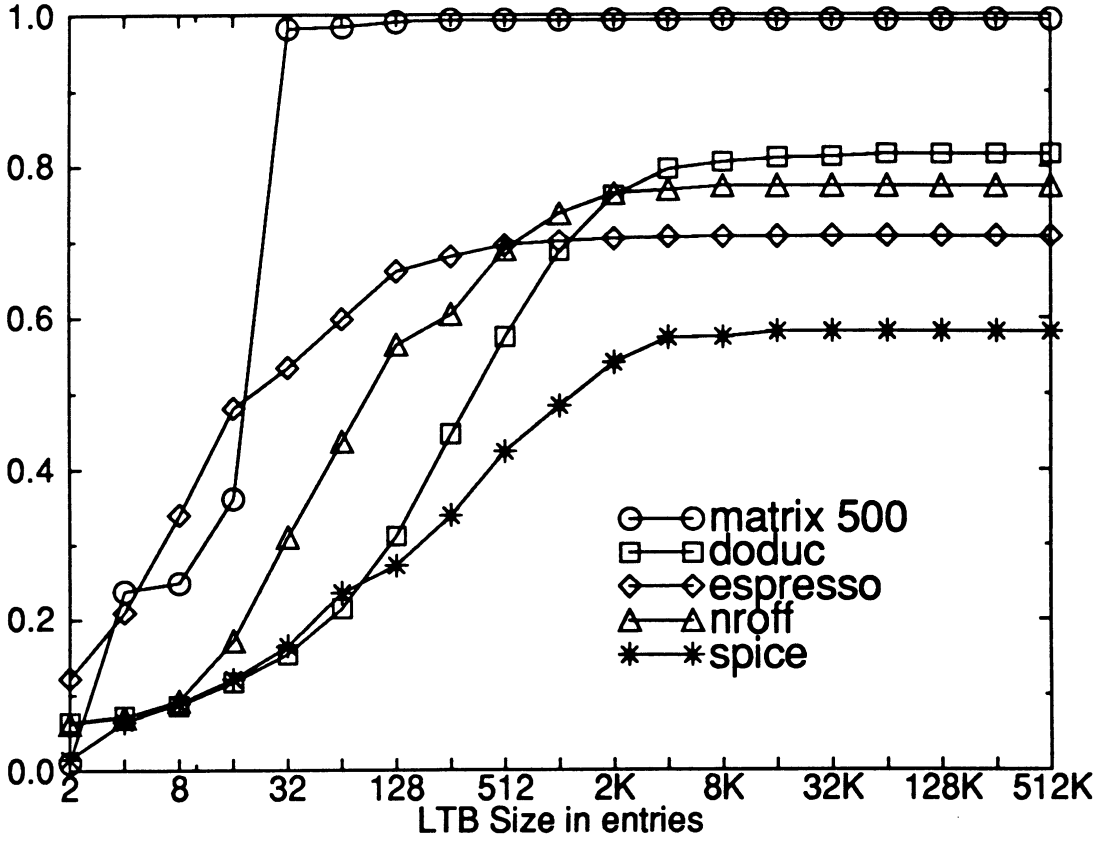


FIGURE 3. Prediction Ratio with Inertial Algorithm

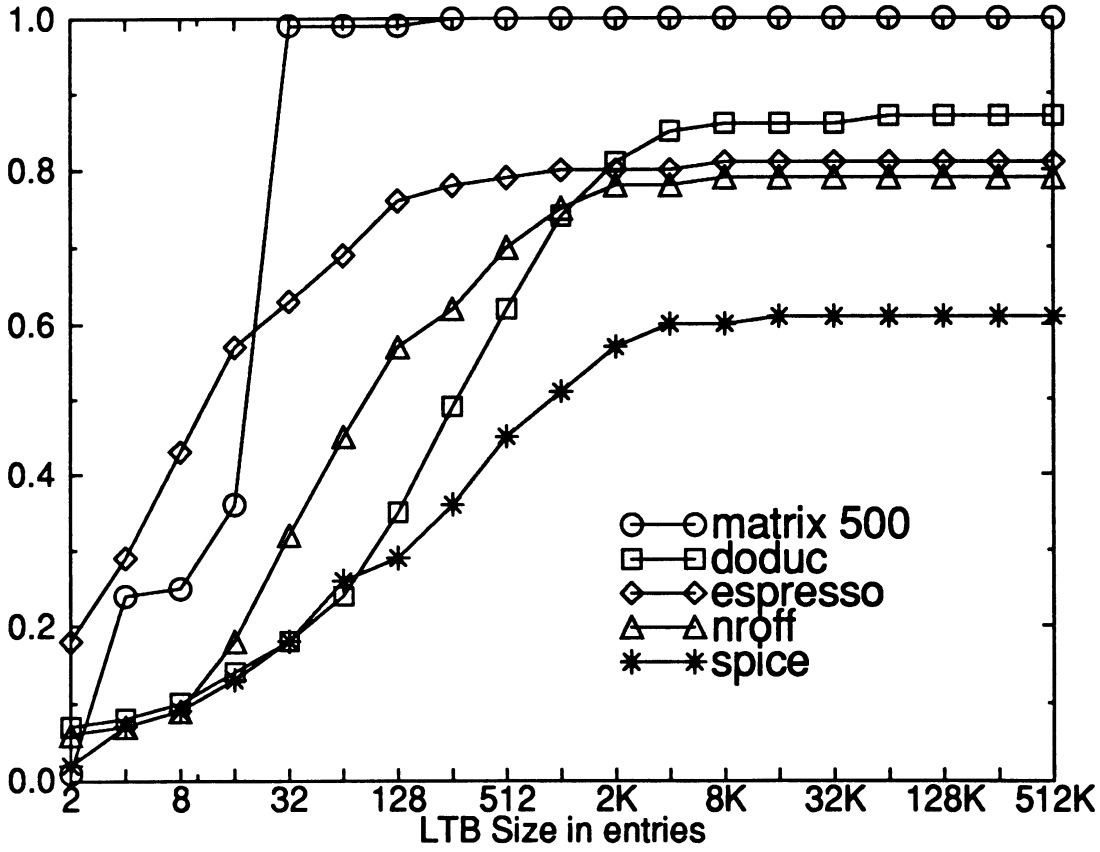
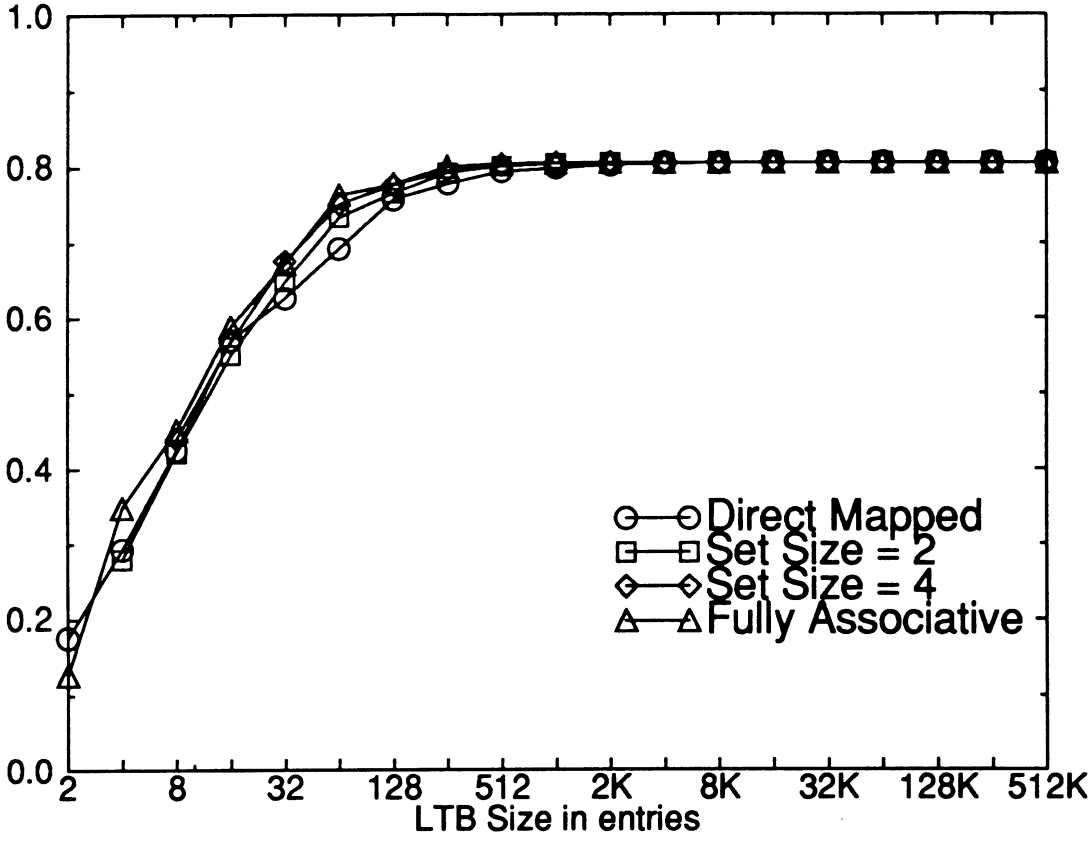


FIGURE 4. Prediction ratio vs LTB size for espresso for various set associativities.



4.3 Reducing the Buffer Size

4.3.1 The Problem

As mentioned in Section 4.1, a buffer size of 1K entries may be too large to get the short access time that is needed to quickly read and update the data in the LTB. Some strategy must be used to reduce the size of the buffer while still maintaining a high degree of predictive accuracy. One method might be to increase the set-associativity of the cache. In the experiments of the previous sections, a direct-mapped LTB was assumed. Unfortunately, the extra hardware required to give a set associative cache can slow down the access time. Because RCM uses a one-pass algorithm to simulate the LTB, it automatically gives results for all set-associativities. These results reveal a small shift in the location of the knee, and thus a small improvement in LTB size for the "reasonable" set sizes of 2 and 4 entries.

Another approach is follow the lead of Olukotun and Mudge in [14]and try to find those load instructions which can be rescheduled so that, if the cache is properly pipelined, their latencies are partially or completely hidden. Implementation of this approach would require two changes to the LTB system.

First, the compiler would have to be augmented to detect those loads which could have their latencies completely hidden through instruction rescheduling. These loads would be marked somehow to indicate that they will not have their targets stored in the LTB. This

could be done by having two types of load opcodes, bufferable and non-bufferable. Unfortunately, this would require a change in the instruction set architecture if it were to be implemented in an existing machine, thus introducing incompatibilities. Second, the LTB would have to be modified so that it ignores load instructions which are marked in this manner.

4.3.2 Simulation

Determining the effectiveness of this solution requires a more involved simulation. For different load latency times, different load instructions will have their latencies successfully hidden through instruction rescheduling. Furthermore, the penalty incurred by misses in the LTB will differ for loads that have their latencies only partially hidden in the rescheduling phase.

Because of these factors, the prediction ratio is no longer a good measure of the effectiveness of an LTB of a particular size. Instead, the effect of LTB size on the system CPI will be determined. The "knee" of the graph of CPI versus LTB size determines a good size for the LTB.

The simulation proceeds in two phases. In the first phase, the trace is scanned and all load instructions are examined. If a load instruction to a particular register is followed too closely by a use of that register, then a multi-cycle first level cache latency can cause a pipeline stall. To avoid this, the simulator percolates the load up the instruction stream until data dependencies prevent further motion. If the load can be moved far enough away from the use, the stall cycles will have been successfully eliminated through rescheduling. If not, the simulator records the address of the load instruction for the second phase of the simulation. This first phase corresponds to a compilation phase in which loads which cannot be well scheduled are marked by the compiler for insertion into the LTB. Only load instructions are moved; the simulation does not try to reschedule the instructions that cause the dependencies with the load instruction.

In the second phase, the simulator performs the same process of rescheduling load instructions and trying to hide their latencies through clever instruction scheduling. Whenever a load instruction is found which was marked by the first phase of the simulation, it is fed into the LTB. If the LTB successfully predicts the target of that load, the number of cycles that are saved is recorded.

Several assumptions are made in the rescheduling phase. They are that:

1. Load instructions can be moved through branch instructions.
2. There are no branch delay slots and branch prediction is perfect.
3. All instructions, except perhaps loads, are executed in a single cycle.
4. The window of instruction motion is 3n instructions, where n is the number of cycles of latency to the first level cache.
5. Once an instruction is moved, no other instruction can be moved above it in the instruction stream.

The first assumption is perhaps the most bold. It was made for two reasons. First, the MIPS is a uniprocessor machine with fairly simple scheduling for structures like loops, which receive a lot of attention in scientific machines. In a compiler for a scientific machine, techniques such as polycyclic vector scheduling and loop unrolling allow code motion which effectively moves instructions through branches. Second, in future machines, speculative execution of instructions may allow true rescheduling of code across branch instructions to enable compilation techniques such as trace and superblock scheduling[8].

The second assumption also has two justifications. It is desirable to isolate the effects of the load instructions without having the factors of branch prediction accuracy and pipeline depth in the design space. This required the elimination of all other variables besides load latency and LTB size. Furthermore, the state of the art in branch prediction allows almost perfect prediction, as was mentioned in the introduction[20]. Imperfect branch prediction could decrease performance because the instruction addresses used to index the LTB may never be executed if the resolution of a previous branch instruction squashes them. Extra predicted targets could then be sent to the memory system. Additionally, if the LTB is indexed from an instruction prefetch buffer, and the pipeline is refilled after a mispredicted branch without going through that buffer, either the LTB would not be indexed, decreasing performance, or an extra data path into the LTB would have to be built.

In a pipelined RISC machine, having each instruction require a single cycle of execution is a reasonable assumption. Even though the total latency of an instruction can be long, pipelining allows a result to be produced during every cycle.

Keeping the window of instruction motion to a limited number of instructions is done due to time considerations. To scan the entire trace for the earliest allowable time of execution of a load instruction would be impractical for the traces used, which were tens to hundreds of millions of instructions long. A window size of three times the load latency allowed successful motion of three dependent load instructions, as long as other data dependencies did not prevent motion. This seemed like a reasonable figure. No experiments were done to see the effect of changing the window size on the success of code motion. A more appropriate forum for that type of discussion would be a paper on the practical aspects of code rescheduling.

The last assumption is simply a heuristic. Occasionally, a load instruction will be moved and have all of its latency hidden through rescheduling. The simulator then tries to move a second load instruction. If it is allowed to take the place of the first load instruction, the simulator may decide that it has all of its latency hidden as well. A problem arises because the motion of the second load into the place occupied by the first load instruction would have forced the first instruction to move *downward* in the instruction stream. This downward motion may expose some of the latency that was previously hidden. To intelligently make this decision would be quite a programming chore, and is beyond the scope of this paper.

In[14], the authors determine how much load latency can be hidden through the introduction of pipelining in the first level cache. In doing so, they determine which load instruc-

tions can be rescheduled to hide this latency. They do this by examining each load instruction in turn, and deciding whether data dependencies prevent the code motion required to achieve this goal. They do not attempt to determine the interactions between the loads which are moved.

These interactions may be significant. First, loads may be forced downwards in the instruction stream, as was described above. If this effect is not considered, one could produce optimistic results about the success of code rescheduling. Also, if the compiler decides that a load cannot have its latency hidden through code motion, and then a second load instruction is moved into a delay slot of the first instruction, the first instruction will have one more cycle of its latency hidden. If a simulation does not consider this, it will give conservative results.

TABLE 5. CPI given by two different simulation methods.

Load Latency	Method	doduc	matrix 500	espresso	nroff	spice
2 cycles	Golden	1.022	1.0006	1.013	1.020	1.035
	Olukotun	1.023	1.0006	1.020	1.020	1.036
3 cycles	Golden	1.087	1.0025	1.064	1.049	1.106
	Olukotun	1.102	1.0033	1.078	1.051	1.128
5 cycles	Golden	1.264	1.0119	1.282	1.123	1.288
	Olukotun	1.310	1.0143	1.283	1.159	1.340
8 Cycles	Golden	1.624	1.387	1.756	1.308	1.722
	Olukotun	1.745	1.418	1.759	1.387	1.786

Table 5 shows that the method of Olukotun et.al. was very close to the simulation technique of this paper, which takes these interactions into account. In the cases where there is significant difference, Olukotun was slightly conservative. One should note that in Table 5, and all tables in this section, a CPI of one would indicate that each load instruction had an effective latency of one cycle.

In summary, the effect of the simulation is to produce an optimal schedule within the programming constraints listed above. The load instructions which have latencies that cannot be hidden through this scheduling process are placed in the LTB which tries to give an early prediction of their targets, thus hiding their latency in hardware.

4.3.3 Results

When examining the results given in this section, one should remember that a CPI of 1.0 indicates that all load latencies were successfully hidden, either through scheduling or the action of the LTB. Table 6 shows the actual CPI that is achieved for each program after rescheduling. A figure for the CPI before rescheduling makes little sense because even the most rudimentary compiler would try to fill in some of the delay slots with independent instructions that immediately follow the load

The table reveals that when there are only two cycles of latency to the first level cache, an LTB can, at the most, cause a 0.02 to 0.035 CPI improvement. Figure 5 reveals that this is indeed the case. For most of the benchmarks, the LTB provides around a 1 percent improvement in CPI. Because matrix 500 has most of its latency hidden through rescheduling, the LTB has almost no effect on it. Doduc shows increasingly better performance as the LTB size is increased, but even in this case 1024 entries are required for a meager 0.02 CPI gain in performance.

TABLE 6. Base CPI with rescheduling

load latency	doduc	matrix 500	espresso	nroff	spice
2	1.022	1.0006	1.013	1.020	1.035
3	1.087	1.0025	1.064	1.049	1.106
5	1.264	1.0119	1.282	1.123	1.288
8	1.624	1.387	1.756	1.308	1.722

When the cache latency is increased to 3 cycles, the benefits of an LTB become slightly more pronounced. Once again., matrix 500 requires nothing but rescheduling to achieve a CPI very close to 1. Espresso, nroff, and spice see an improvement of 0.015 to 0.045 CPI, and the curve is quite level as the LTB size increases. This seems to indicates that if an LTB was included in such a machine, it could be rather small, around 32 entries, and still get most of the available performance increase. Once again, doduc wants a rather large LTB size in order to capture most of its performance potential. With 1024 entries, almost all of the 0.087 CPI performance degradation due to load latency is recaptured. Unfortunately, this is probably too large of an LTB to be practical.

For a 5 cycle cache latency, the LTB becomes more of a practical solution. Figure 7 shows that an improvement in CPI of about 8 percent is available for an LTB with 64 to 128 entries for doduc, nroff, and spice. Espresso has a performance improvement of 0.15 to 0.18 CPI in this range. The "knee" of the graph for these programs, except for doduc, is around this size. Once again, doduc shows only a little sign that the slope is decreasing as the LTB size increases.

Figure 8 shows similarly shaped curves, but the improvement is larger, from 0.10 to 0.50 CPI. It is interesting to note that for 8 cycles of load latency, matrix 500 shows improvement through the use of an LTB, but requires only 32 LTB entries to capture almost all of this performance gain. This probably results from the tight loop structure of the benchmark. One the cache latency exceeds the number of cycles in the loop, an LTB is needed, but it need not be large since there is a very high degree of spatial and temporal locality in a tight loop.

5.0 Conclusion and Future Work

5.1 Conclusion

When is an LTB useful? When the first level cache latency is only two or three cycles, a good schedule combined with a pipelined cache should be able to hide most of the load latency. The inclusion of an LTB for a 2 to 3 percent CPI decrease is probably not worth it.

When the latency increases to 5 cycles, a moderately sized LTB can give a significant boost in performance, around 10 percent for most of the benchmarks. An even larger boost is seen when the latency increases to 8 cycles.

It is not clear that all of my assumptions hold when the load latency is as large as 8 cycles. The small gaps discussed in Section 4.2 have a minimal effect for smaller latencies, but when the latency increases to 8 cycles, nroff has 6 percent of the gaps smaller than the latency. Because the LTB needs to be updated with the correct target in the event of a miss, if a particular LTB location is exercised more rapidly than the amount of latency it hides, it could cause stalls while it updates itself or mispredicts load targets. Furthermore, as the LTB is exercised earlier in the pipeline, or even in the instruction prefetch buffer, in order to hide larger latencies, branch prediction has more of an effect on prediction accuracy.

Finally, when a machine has such a large latency to the first level cache, even more aggressive scheduling techniques might be used. These techniques could use register renaming to eliminate dependencies, thus increasing the amount of code motion that is allowed. Superblock scheduling is one example of such a method[8].

The use of a load target buffer seems to have a narrow window of opportunity. The latency to the first level of the memory hierarchy must be long enough to make scheduling difficult yet short enough to let a "quick fix" like the LTB hide it in many cases. In most machines, however, support for aggresive code rescheduling seems to be a more effective solution.

5.2 Future Work

While this research was being performed, an article by Ivan Sklenar appeared in *Computer Architecture News* that had some relevance to this work[15]. This article suggests using hardware similar to the LTB to perform prefetches into the data cache, thus enabling more sophisticated prefetching strategies than fetching sequential lines. This would be beneficial when a scalar processor accesses a vector data structure with a stride longer than the cache line length. Although Sklenar's paper presents a strategy, it does not give results.

Jean-Loup Baer and Tien-Fu Chen have published architectural studies which examine the use of a table to predict the strides of load instructions. This information can then be given to a more intelligent prefetch unit. They show that using this method of prefetching can drastically eliminate compulsory cache misses and cache pollution due to naive prefetch-

ing methods [1][2]. Fu, Patel, and Janssens at the University of Illinois have done similar work and achieved promising results [5].

All of these studies suggest that the stride information be used to prefetch data directly into the data cache. None of them try to use the cache conflict eliminating devices called stream buffers and victim caches which are proposed by Jouppi [10]. It would be interesting to see how a combination of these techniques would interact.

6.0 Acknowledgments

Thomas M. Conte provided the source code for the RCM simulator, which was modified to produce prediction ratios.

All of the students in the Advanced Computer Architecture Lab at the University of Michigan gave advice and created a dynamic working and learning environment. Special thanks to David Nagle for his help in negotiating administrative mazes.

FIGURE 5. CPI Improvement vs LTB Size (2 cycles of latency)

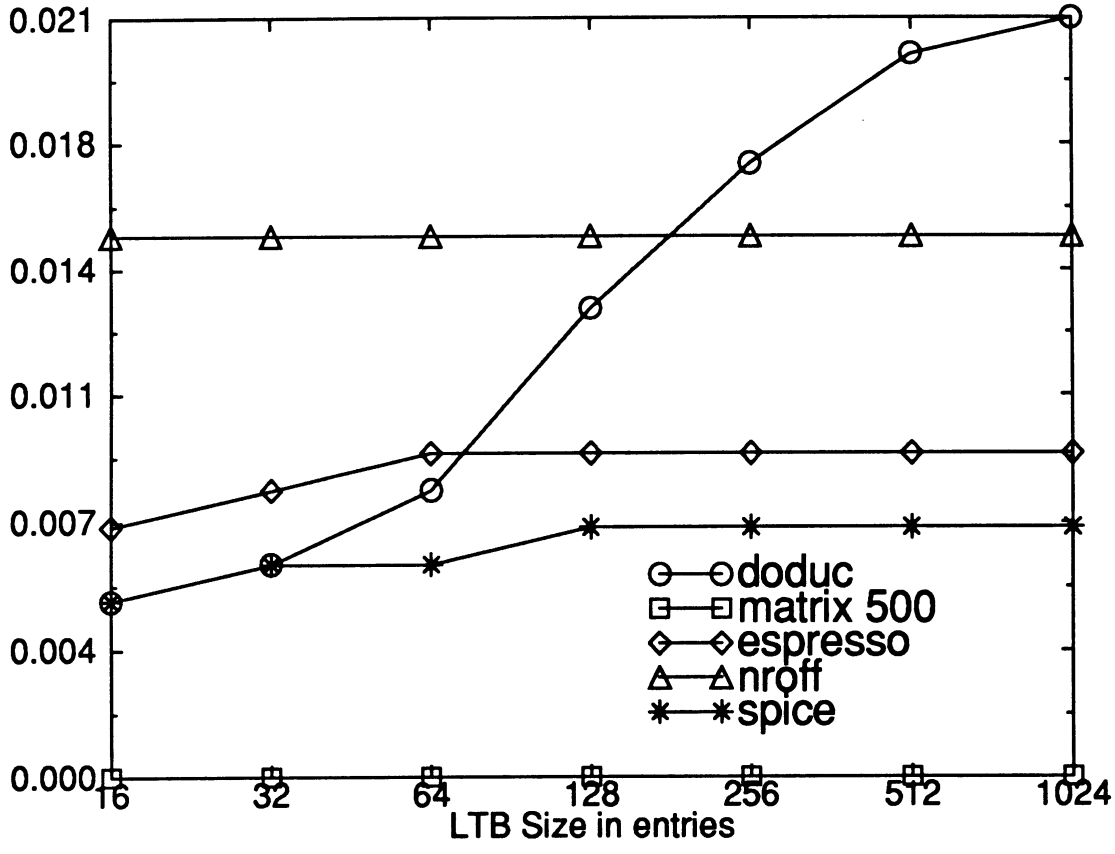


FIGURE 6. CPI Improvement vs LTB Size (3 cycles of latency)

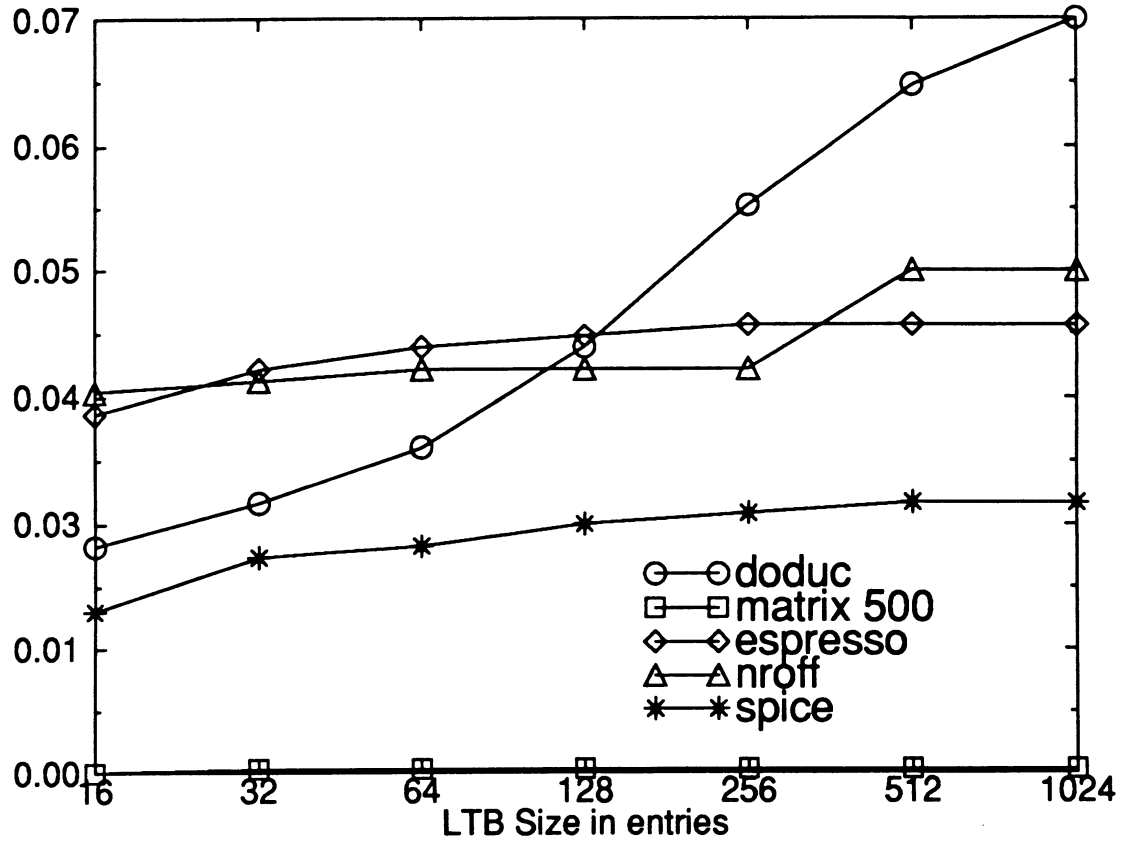


FIGURE 7. CPI Improvement vs LTB Size (5 cycles of latency)

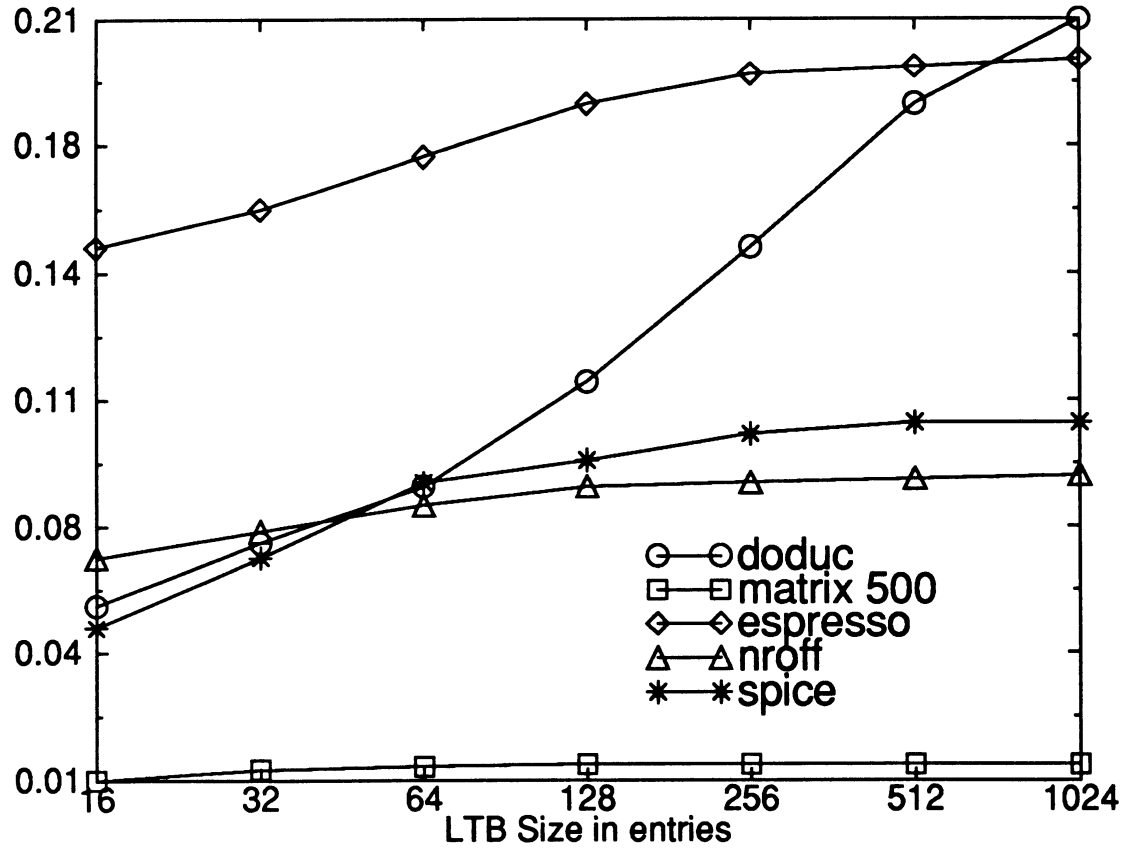
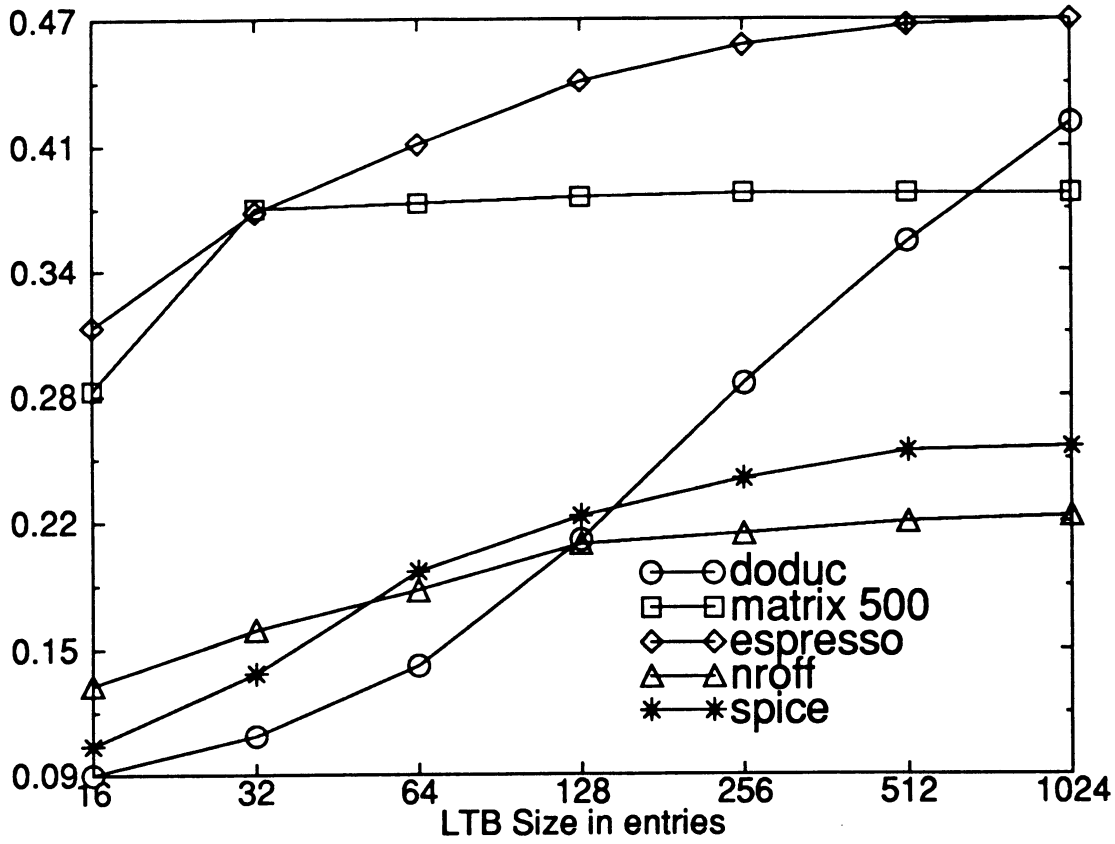


FIGURE 8. CPI Improvement vs LTB Size (8 cycles of latency)



BIBLIOGRAPHY

[1] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," *IEEE Computer Graphics and Applications*, vol. 12, pp. 176–186, March 1992.

[2] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," Technical Report 92-06-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, June 1992.

[3] C. J. Conti, D. H. Gibson, and S. H. Pitowsky, "Structural aspects of the System/360 model 85," *IBM Systems Journal*, vol. 7, no. 1, pp. 2–21, 1968.

[4] T. M. Conte. \newblock *RCM Users Guide*, 1991.

[5] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in *The 25th annual international symposium on microarchitecture : MICRO 25*, pp. 102–110, Portland, Oregon, December 1992.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[7] W.-M. Hwu and P. P. Chang, "Efficient instruction sequencing with inline target insertion," *IEEE Transactions on Computers*, 1992. Accepted for Publication.

[8] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Birmingham, R. G. Oullette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, 1993. To Appear.

[9] J. K. Iliffe, "A forward looking method of cache memory control," *Computer Architecture News*, vol. 15, no. 4, pp. 4–10, September 1987.

[10] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th annual international symposium on computer architecture*, pp. 364–373, Seattle, WA, May 1990.

[11] G. Kane and J. Heinrich, *MIPS RISC architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[12] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE*

Computer, January 1984.

[13] S. Mirapuri, M. Woodacre, and N. Vasseghi, "The MIPS R4000 processor," *Micro*, pp. 10–22, April 1992.

[14] K. Olukotun, T. Mudge, and R. Brown, "Performance optimization of pipelined memory caches," in *Proc. 19th Ann. Int'l Symp. Computer Architecture*, pp. 181–190, Gold Coast, Australia, May 1992, IEEE Computer Society Press.

[15] I. Sklenar, "Prefetch unit for vector operations on scalar computers," *Computer Architecture News*, vol. 20, no. 4, pp. 31–37, September 1992.

[16] J. E. Smith, "A study of branch predition strategies," in *Proc. 8th Ann. Int'l Symp. Computer Architecture*, pp. 135–148, June 1981.

[17] G. S. Sohi and E. S. Davidson, "Performance of the structured memory access SMA architecture," in *Proc. 1984 Int'l Conf. on Parallel Processing*, pp. 506–513, Bellaire, MI, August 1984.

[18] G. Sohi and W.-C. Hsu, "The use of intermediate memories for low-latency memory access in supercomputer scalar units," *The Journal of Supercomputing*, vol. 4, pp. 5–21, 1990.

[19] J.-H. Tang, E. S. Davidson, and J. Tong, "Polycyclic vector scheduling vs. chaining on 1-port vector supercomputers," in *Proc. Supercomputing '88*, Orlando, Florida, November 1988.

[20] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction," in *Proc. 19th Annual Intl. Symp. on Computer Architecture*, pp. 124–134, Gold Coast, Autralia, May 1992.