Performance Optimization of Pipelined Primary Caches *

Kunle Olukotun
Computer Systems Laboratory
Stanford University, CA 94305

Trevor Mudge and Richard Brown
Dept. Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109

Abstract

The CPU cycle time of a high-performance processor is usually determined by the the access time of the primary cache. As processors speeds increase, designers will have to increase the number of pipeline stages used to fetch data from the cache in order to reduce the dependence of CPU cycle time on cache access time. This paper studies the performance advantages of a pipelined cache for a GaAs implementation of the MIPS based architecture using a design methodology that includes long traces of multiprogrammed applications and detailed timing analysis. The study evaluates instruction and data caches with various pipeline depths, cache sizes, block sizes, and refill penalties. The impact on CPU cycle time of these alternatives is also factored into the evaluation. Hardware-based and software-based strategies are considered for hiding the branch and load delays which may be required to avoid pipeline hazards. The results show that software-based methods for mitigating the penalty of branch delays can be as successful as the hardware-based *branch-target buffer* approach, despite the code-expansion inherent in the software methods. The situation is similar for load delays; while hardware-based dynamic methods hide more delay cycles than do static approaches, they may give up the advantage by extending the cycle time. Because these methods are quite successful at hiding small numbers of branch and load delays, and because processors with pipelined caches also have shorter CPU cycle times and larger caches, a significant performance advantage is gained by using two to three pipeline stages to fetch data from the cache.

1 Introduction

The organization of the cache has a significant impact on CPU performance for two reasons. First, the access time of the cache usually sets the CPU cycle time. Second, the miss rate of the cache determines how much of the time the CPU is stalled waiting for data to be fetched from main memory. The access time and miss rate of a cache are determined by its size and its set-associativity. Making the cache smaller or less set-associative will reduce the cache access time and increase the cache miss rate.

As the speeds of CPUs increase relative to the speed of DRAM-based main memory, the cost of missing the cache increases. This presents computer designers with the problem of building a cache that has both a low miss rate and a short access time. A low miss rate ensures that the high cost of cache misses do not dominate execution time and a short access time ensures that the cache does not slow down the rest of the CPU. A partial solution to this cache design problem is to provide more than one level of cache memory. In a two-level cache hierarchy, the level one (primary) cache is made small and fast to match the CPU speed and the level two (secondary) cache is made slower and much larger to keep the overall cache miss rate low. Although this is better than having a single large cache, the primary cache will still limit the CPU cycle time. The only way to reduce the effect of cache access time on processor cycle time is to increase the number of cache pipeline stages. This spreads the fixed delay of the cache access time over more CPU cycles making it possible for the CPU cycle time to be reduced. Implementing a pipelined cache involves splitting the access of the cache into two or more stages and placing latches between each stage. Some of the pipelining may be implemented inside the SRAM itself [CCS+91]. An example of a processor that uses this technique is the MIPS R4000 which takes two pipeline stages to access an on-chip cache [KH92]. It is likely that as microprocessor speeds increase we will see wider use of pipelined caches.

While cache performance evaluation has received considerable attention [Smi78, Hil87, Prz90], the effect of pipelining cache access has received far less attention. This paper presents a methodology for accurately evaluating and optimizing the performance of pipelined primary caches and a study that shows the performance improvement that pipelined caches can provide.

This study was done during the design of microprocessor based on the MIPS instruction set architecture (ISA) [KH92] that is to implemented in GaAs direct-coupled FET logic with multichip module packaging [MBB+91]. A general block diagram of the processor, annotated with the ranges of parameters considered in this study, is shown in Figure 1. The two-level cache organization is necessary to hide the large disparity in speeds between the processor and main memory. The level one (L1) cache is split in to instruction (L1-I) and data (L1-D) halves to provide an instruction or data access every cycle. The justification of these design choices is discussed in detail in [Olu91].

Increasing the number of cache pipeline stages can reduce the CPU cycle time, but it will increase the number of load and branch delay cycles due to pipeline hazards. If no useful instructions are executed during these delay cycles they will increase the number of

*This work was supported by Defense Advanced Research Projects Agency under DARPA/ARO Contract No. DAAL03-90-C-0028

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-509-7/92/0005/0181 $1.50

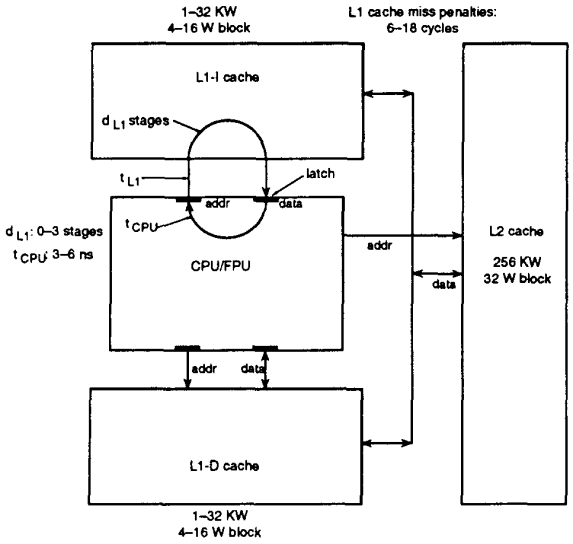

Figure 1: The basic system architecture and the range of parameters that are considered in this study. The access paths of the pipelined primary instruction cache are shown.

cycles a program takes to execute, and potentially overwhelm the performance benefits of pipelining. Our experiments show that as many as three load or branch delay cycles can be hidden by static compile-time instruction scheduling or by dynamic hardware-based methods. Static schemes are more effective at hiding branch delay cycles, but dynamic methods are more effective at filling load delay slots. When these results are combined with results from cache simulation and timing analysis the conclusions are that the caches with two to three pipeline stages have higher performance than caches with less pipeline stages.

The organization of the rest of this paper is as follows. The next section of the paper defines the pipelined cache optimization problem and describes the design optimization methods used in this study. Sections 3 shows how the number of clocks per instruction (CPI) are affected by the primary instruction and data cache organizations. In particular, the impact of pipelining is examined. Section 4 shows how the cycle time, t_{CPU}, are affected by the size of the primary cache. The results of Sections 3 and 4 are combined to yield the final performance evaluation of pipelined primary caches in Section 5. Concluding remarks are given in Section 6.

2 The Primary Cache Optimization Problem

A widely recognized metric for architectural performance comparison is the time it takes to execute a realistic set of benchmarks [HP90]. Given a specific ISA and compiler, a performance metric that is directly proportional to execution time is *time per instruction* (TPI). The equation for TPI can be written as,

$$\text{TPI} = \text{CPI} \times t_{CPU} \tag{1}$$

In this paper we use TPI as the performance metric.

The objective of primary (L1) cache optimization is to select a cache organization that minimizes TPI subject to implementation technology constraints. To solve this optimization problem, we must determine the effect cache organization, including the number of cache pipeline stages, has on t_{CPU} and CPI.

If the L1 cache access is on the critical timing path, t_{CPU} depends on the access time of L1, t_{L1}, and its pipeline depth, d_{L1}. The value of t_{L1} is defined as the elapsed time, measured in nanoseconds, between sending an address to the cache and receiving the data from that address. The value of t_{CPU} is related to t_{L1} and d_{L1}, as follows,

$$t_{CPU} = \frac{t_{L1}}{d_{L1}} \tag{2}$$

Although the details of the timing are more complicated, for the purposes of this discussion, the access path to the L1 cache can be viewed as part of a circular pipeline with $(d_{L1} + 1)$ stages clocked at intervals of t_{CPU}, see Figure 1. The address generation takes one cycle and the cache access takes d_{L1}. From (2), it follows that fixing d_{L1} causes t_{CPU} to get longer as t_{L1} increases. In particular, larger or more set-associative caches which have longer access times result in longer CPU cycle times. Increasing the value of d_{L1} by placing more latches along the L1 access path to increase pipelining makes it possible to reduce t_{CPU}. The amount of reduction depends upon the placement of the latches, the latch delay overhead, and the clocking scheme [Olu91].

Given a constant time L1 miss penalty, then CPI depends on t_{CPU}, d_{L1}, and the miss rate of the cache, m_{L1}. As the cache is made larger or more set-associative m_{L1} gets smaller and so does CPI. With a constant time L1 miss penalty, CPI decreases with increasing t_{CPU} because fewer CPU cycles are required to handle a miss. The dependence of CPI on d_{L1} is more subtle. In general, CPI is an increasing function of d_{L1} because increasing d_{L1} will increase the number of load and branch delay cycles that cannot be hidden. Furthermore, software techniques used to hide branch delays often replicate code, resulting in larger static code sizes which in turn may lead to higher L1 instruction (L1-I) cache miss ratios.

As we have seen, CPI and t_{CPU} depend on both the organization which specifies the pipeline depth, size and associativity of the cache and the hardware implementation technology which determines the access time of the cache and the delays of the rest of the CPU. It is evident that organization and technology are interdependent because CPI is dependent on t_{CPU} and both are dependent on d_{L1}. The further dependence of CPI on the workload of the CPU necessitates that t_{CPU} and CPI be determined by using a simulation-based multilevel design methodology that considers the organization and technology aspects of cache design together.

The multilevel optimization design methodology that we use in this study combines the optimization of the traditionally separate organization and implementation phases of system design [OBL+91]. This results in a system design that is optimized for performance in a specific technology. The process of multilevel optimization uses optimizing compilers, trace-driven simulation, delay macro-modeling, and timing analysis to accurately predict the TPI of an architecture [Olu91]. Delay macro-modeling and timing analysis are used to predict t_{CPU} while trace-driven simulation of optimized code is used to predict CPI. The timing analyzer used in this study, $minT_C$, is capable of finding the minimum clock cycle time of a synchronous digital circuit [SMO90]. The compiler used in this study is one that was developed by MIPS for the R2000 instruction set architecture [MIP88]. The trace-driven simulator used in this study, `cacheUM`, is capable of evaluating a variety of two-level cache organizations and techniques for reducing the effect of load and branch delays [Olu91].

The process of multilevel optimization systematically improves the performance of a base architecture. A base architecture is defined by a register transfer level (RTL) organization and implementation technology. The main requirement of a suitable base architecture is that it be as simple as possible in order to accurately assess the performance impact of adding to it. Taking the base architecture as a starting point in the design space, and using the performance analysis tools, multilevel optimization proceeds as

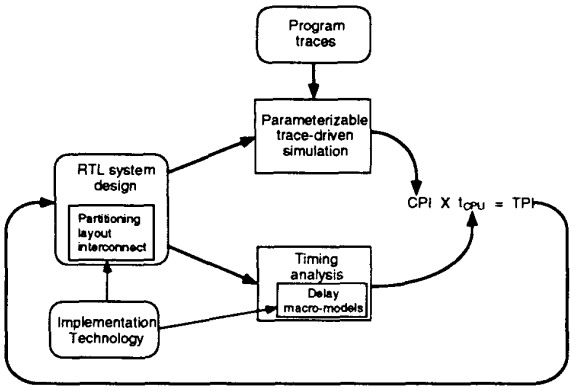

Figure 2: Multilevel design optimization.

follows. The base architecture is simulated to establish a base level of performance. To explore the effect of changes in the base architecture a set of candidate designs is encoded into the trace-driven simulator and changes to the RTL model are made to reflect the new organization. The compiler is also changed to produce code that is optimized for the new RTL organizations. The performance of the set of designs is simulated using trace-driven simulation to reveal the general nature of the CPI-t_{CPU} tradeoffs. Once these tradeoffs has been established, delay modeling, timing analysis and floor planning provide the technology constraints. These technology constraints make the CPI-t_{CPU} tradeoffs specific and allow the selection of the highest performing design alternative. The RTL specification represented by this design choice becomes the new base architecture. This process of generation and evaluation of different organizations continues until the design meets the performance specifications or the costs of further changes in organization are not justified by the performance improvements they provide. The multilevel design methodology is diagrammed in Figure 2.

In the remainder of this paper the multilevel optimization design approach is used to study the performance of pipelined primary caches. The trace-driven simulations used in this study use multiprogramming traces created from the benchmark programs listed in Table 1. The CPI values presented in this paper represent the weighted harmonic mean of all of the benchmarks in this table. The weights for each benchmark correspond its fraction of total execution time. The instruction count used to calculate CPI is that of optimized MIPS R2000 code for for an architecture with no load or branch delay cycles.

3 CPI Measurements

3.1 The effect of branch delays

As long as the CPU is executing instructions sequentially, the pipeline depth of the L1-I cache access has no effect on CPI. However, when a control transfer instruction (CTI) is executed, the next instruction to be fetched must be delayed until its address is known. Assuming that the branch condition of a conditional branch CTI can be evaluated in one cycle the pipeline depth of the L1-I cache, d_{L1-I}, is the number of cycles by which the next instruction is delayed. These cycles are called *branch delay cycles*. Since most applications dynamically execute control transfer instructions 10–20 % of the time (Table 1), a unit increase in d_{L1-I} will result in a 10–20 % increase in CPI. A number of hardware and software schemes have been proposed to limit this effect on CPI [Smi81, LS84, MH86, Lil88, HCC89, KT91]. Two representative schemes are evaluated here: a hardware-based branch-target buffer, and a software-based delayed branches with optional squashing.

A branch-target buffer (BTB) is a cache whose entries contain address tags and target addresses of branch instructions. Every instruction address is checked against the tags in the BTB. Those addresses that hit are predicted to be branches (CTI). The BTB simulated here uses the 2 b (bit) prediction scheme described in [LS84]. If the BTB predicts a branch taken, the target address is used as the next instruction address. If the BTB can be accessed in a single cycle and its prediction is correct, it can completely hide the branch delay. However, when the BTB miss predicts a branch, we assume a one cycle stall is required to fill the BTB with the correct information in addition to the stall cycles necessary for branch delay. This penalty is the minimum that can be expected. The size of the BTB that will be evaluated in this study is 256 entries. With two 32 b addresses plus 2 b of prediction per entry, the BTB requires approximately 2 KB of SRAM storage. This size represents the upper limit of an SRAM having single cycle access if we assume that the ALU adder sets the lower bound on cycle time [Olu91].

In the other approach evaluated, delayed branches with optional squashing, each branch instruction is followed by a number of instructions called *branch delay slots* equal to the number of branch delay cycles in the architecture. The compiler must rearrange the code so that these delay slots contain useful instructions. For each branch there are three potential sources of useful instructions: (1) before the branch instruction, (2) after the branch instruction, or (3) from the branch target. Rearrangements that move instructions from before the branch are always correct no matter what direction the branch takes. In contrast, instructions from after the branch must be squashed if the branch is taken and instructions from the branch target must be squashed if the branch is not taken in order to preserve program correctness. Squashing dynamically converts delay slots instructions to `noop` instructions. In addition, there is a code expansion penalty associated with using instructions from the branch target because these instructions must be replicated. We will consider a delay slot to be "hidden" if it is filled with useful instructions that do not require squashing.

To evaluate the performance of these schemes for reducing the impact of branch delay cycles we need to know the number of delay cycles, the number of these that are hidden, the effect on CPI, and the effect of the expanded static code size associated with delayed branches. Because we did not have access to a retargetable compiler, a post-processor for MIPS object code was developed to collect this information. This post-processor can schedule code for architectures having differing numbers of branch delay slots. It creates a translation file that contains a mapping of instruction addresses from the MIPS code, which assumes one branch delay slot without squashing, to those of an architecture that may have a different number of delay slots with optional squashing.

A translation file for an architecture with zero delay cycles is used in the BTB experiments. Such a file is produced by removing all `noop` instructions that appear after CTIs in the original MIPS object code and then recording the mapping between the basic block entry points of the original object code and the transformed code.

The object code for a one cycle branch delay architecture is the same as that produced by the MIPS compiler. However, the MIPS ISA does not include optional squashing and so it has less freedom to move instructions from after the branch and from the branch target into the branch delay slot. Cases where the compiler was unable to move any instructions into the branch delay slot have a `noop` instruction following CTI. The procedure that is followed in such cases is covered in the general (two or more) delay slot insertion scheme described below.

In the description of the general delay slot insertion procedure that follows, b is the number of branch delay cycles (slots) in the

Benchmark	Description	Inst. (M)	Loads (% inst.)	Stores (% inst.)	Branches (% inst.)	Syscalls
5diff	File comparison (I)	218.3	15.3	3.4	20.7	305
awk	String matching and processing (I)	209.5	19.0	12.6	14.3	101
doducd	Monte Carlo simulation (D)	96.3	31.0	10.0	8.7	427
espresso	Logic minimization (I)	238.0	19.9	5.6	16.2	17
gcc	C compiler (I)	235.7	23.3	13.8	20.1	487
integral	Numerical integration (D)	110.5	37.0	10.4	7.6	12
linpackd	Linear equation solver (D)	4.0	37.4	19.7	5.4	10
loops	First 12 Livermore kernels (D)	275.5	29.3	10.9	5.3	3
matrix500	500 x 500 matrix operations (S)	202.2	24.3	3.5	3.5	10
nroff	Text formatting (I)	15.7	22.4	10.8	24.6	1701
small	Stanford small benchmarks (I/S)	16.7	19.9	8.8	19.6	0
spice2g6	Circuit simulator (S)	297.3	29.8	8.6	8.0	395
tex	Typesetting (I)	133.8	30.2	14.2	11.7	697
wolf33	Simulated annealing placement (I)	115.4	30.0	7.5	14.8	407
xlswins	X-windows application (I)	52.2	22.5	17.7	17.1	65294
yacc	Parser generator (I)	193.9	19.6	2.4	25.2	49
Total		2414.9	24.7	8.7	13	69915

Table 1: Benchmarks that were used to create the multiprogramming traces. Integer benchmarks are denoted by (I), single precision floating point benchmarks by (S), and double precision floating point by (D). The heading "Branches" represents all control transfer instructions.

Delay slots	% code increase
1	6
2	14
3	23

Table 2: Static code size increase versus the number of branch delay slots.

architecture that is being simulated. The term r is the number of branch delay slots that can be filled from before the branch and s is the number of delay slots that must be filled from the sequential path, from the branch-target path, or, in the case of register-indirect jumps, with noop instructions ($b = r + s$). The steps in the procedure are:

1. Check to see if the MIPS compiler inserted a noop instruction after the CTI. If it did, the CTI cannot be moved any higher in the basic block, Therefore, set $r = 0, s = b$ and go to step 3.
2. Move the CTI up in the basic block as far as the data dependencies of the preceding instructions allow. No attempt is made to rearrange the ordering of any other instructions in the basic block besides the CTI. The number of instructions that the CTI can be moved is r and, then, $s = b - r$.
3. Determine the prediction of the CTI. Backward branches and unconditional jumps are predicted to be *taken*. Forward branches are predicted to be *not taken*.
4. Insert s noop instructions after CTI that are predicted to be taken in order to simulate the instructions that are replicated from the target path.

When this procedure is completed, each CTI will have associated with it a value for s, the number of instructions in the delay slots that may need to be squashed, and a flag that indicates whether the CTI is predicted to be taken or not-taken. In the case of register-indirect jumps, set $s = 0$ because the target of these jumps is not computable at compile time and noop instructions will have to be fetched. The translation file contains values for s, the branch prediction flags, and the basic-block entry-point mappings.

Table 2 shows, for 1 to 3 branch delay slots, the average static code size increase for the benchmarks of this study compared to that of an architecture that has no delay slots. The code size increase comes from filling the delay slots of the 60 % of CTIs that are statically predicted to be taken. Measurements taken from our benchmarks indicate that the MIPS compiler is able to fill 54 % of all first branch delay slots with instructions from before the CTI. For CTIs predicted to be taken, this number decreases to 52 %. The rest of the delay slots for these CTIs must be filled with instructions from the target. The delay slots of register-indirect jumps cannot be filled with target instructions and so are filled with noop instructions. Register-indirect jumps make up roughly 10 % of all CTIs in our benchmarks.

Translation files enable us to use a trace from a computer with an ISA that has one branch delay slot and no squashing to simulate the instruction referencing behavior of a computer with b branch delay slots and optional squashing. Thus the translation files can be used to simulate the instruction cache behavior as follows. Each basic block entry-point instruction address that comes from an instrumented benchmark is translated to a new address using the data in the translation file. This address is used to simulate l sequential instruction references, where l is the length of the new basic-block.

For accurate cache simulation, we need not be concerned with the specific mechanism used to squash instructions in the delay slots, we need only ensure that the correct instruction reference stream is produced. To get the correct reference stream, it is necessary to check the prediction at each CTI. If the prediction at the CTI is that it will be taken and the prediction is correct, then the value of s associated with the CTI is added to the instruction address of the target basic block thus leaving $l - s$ instructions to be executed in the target basic block. This assumes that s instructions of the target basic block have been executed in the delay slots of the CTI. If there are fewer than s instructions in the target basic block, then the delay slots are assumed to be padded with noop instructions. If the prediction is that the CTI will be taken and the prediction is incorrect, then the instruction address of the sequential basic block is left unchanged. This simulates the effect of the extra delay slot instruction references. If the prediction is that the CTI will not be taken and the prediction is correct, then no action is taken because the sequential instructions are in the delay slots of the CTI. However, if the CTI is predicted not to be taken and the prediction is incorrect, then s extra instruction references are made in the sequential basic block before control is transferred to the target basic block.

Our experiments show that CPI increases as the number of branch delay slots is increased. This increase comes from two sources: (1) increases in the number of L1-I cache misses due to the larger code size and (2) the extra useless instruction references that are executed when the static branch prediction is incorrect. To measure the effect of the extra L1-I cache misses, various L1 cache configurations were simulated as the number of branch delay slots was varied from 0 to 3. Apart from delay slots, the other major parameters of

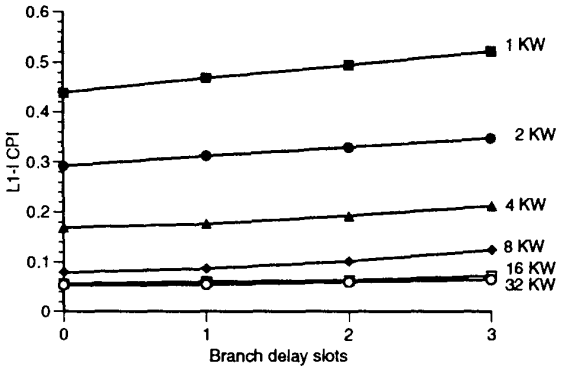

Figure 3: Effect of cache misses due to branch delay slots on L1-I performance: $B_{L1} = 4$ W, $P_{L1} = 10$ cycles.

the cache organization that were varied in these simulations were the L1-I cache size, S_{L1}, the line or block size, B_{L1}, and the L1 miss penalty, P_{L1}.

Cache experiments were performed for L1-I cache sizes that varied from 1 KW to 32 KW; for block sizes of 4 W, 8 W, and 16 W; and for L1 miss penalties of 6, 10, and 18 cycles. The miss penalties correspond to refill rates of 4, 2 and 1 word per cycle plus a 2 cycle startup. For each value of miss penalty the block size was selected to achieve the lowest CPI. The results of the experiment for the case $B_{L1} = 4$, $P_{L1} = 10$ are shown in Figure 3. For smaller miss penalties the slope and intercept of the lines is reduced and for larger miss penalties the slope and the intercept of the lines is increased. This reveals that the number of delay slots has a measurable impact on the performance of the L1-I cache. This impact diminishes as the cache size is increased and the miss penalty is decreased. For the smallest L1-I cache size of 1 KW (4 KB) the rate of increase in CPI is roughly 0.06 per delay slot for a miss penalty of 18 cycles. When the miss penalty is reduced to 6 cycles the rate of CPI increase drops to 0.015 per delay slot. For the largest L1-I cache size of 32 KW, the rate of increase in CPI is quite small. The rate of increase varies from 0.014 to 0.004 CPI per delay slot as the miss penalty is varied form 18 to 6 cycles.

Table 3 lists, for 1, 2 and 3 delay slots, the static branch prediction statistics, cycles per branch, and additional CPI due to extra instruction reference cycles. The data in this table support two conclusions. First, static branch prediction with optional squashing is an effective scheme for mitigating the branch-delay penalty. For example, since 13 % of instructions executed are CTIs, three branch delay slots could increase CPI by 39 %; in fact, the increase is only 8.7 %, due to good branch prediction. Secondly, the effect of instruction cache misses should not be ignored when considering the performance of aggressive static branch prediction schemes. Though this is less important for large caches and small miss penalties, it is noteworthy that the increase in CPI due to increased cache misses with three delay slots is 9 % for a 1 K-W L1-I cache with a 10-cycle miss penalty (see Figure 3).

Experiments with the hardware approach for hiding branch delay cycles show that the BTB achieves a hit rate of over 91 %; however, incorrect predictions reduce this hit rate to a branch prediction accuracy of 86%. This is still better than the static prediction accuracy of 70%. However, the overall effectiveness of the BTB method is reduced because an extra cycle is required to update the BTB with the correct information every time there is a BTB miss or an incorrect prediction. When these cycles are included in the branch penalty, the performance of the BTB is reduced to that shown in Table 4.

Delay cycles	Cycles per CTI	Extra CPI
1	1.44	0.057
2	1.65	0.082
3	1.85	0.110

Table 4: BTB prediction performance.

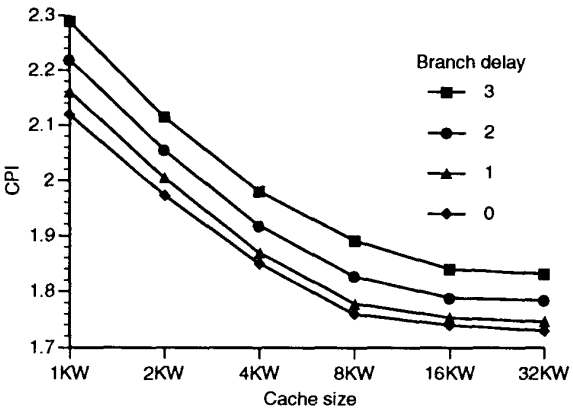

Figure 4: Branch delay slots versus L1-I cache size: $B_{L1} = 4$ W, $P_{L1} = 10$ cycles.

A comparison of Tables 3 and 4 shows that the static scheme performs better. This is because static rearrangement allows 0.5 to 0.8 of the delay slots to be filled with instructions from before the CTI, so that fewer cycles are wasted even if the CTI prediction is incorrect. The BTB scheme loses one cycle per delay slot every time a CTI misses the BTB or the CTI prediction is incorrect. One could argue that the relatively small size of the BTB compromises its performance; recall, though, that the BTB was restricted to 256 entries to ensure single cycle access, which is necessary to make this scheme worthwhile. Though they did not investigate the effects of code expansion, other researchers have also shown that static branch prediction techniques using sophisticated program profiling and fetch strategies are competitive with much larger BTBs [HCC89, KT91]. Of course, for static prediction, the additional CPI due to increased L1-I misses (see Figure 3) must be considered; for small cache sizes and large miss penalties, this would give the performance edge to the BTB approach. Nevertheless, because its performance is roughly comparable and its hardware cost is lower, the static prediction scheme is used in the remainder of the L1 cache experiments.

Figure 4 plots the total CPI for the same values of L1-I cache size, numbers of delay slots, and penalty that was used in Figure 3 This figure shows that for L1-I cache sizes of 1–16 KW, it is always possible to decrease the CPI of the system by doubling the cache size and increasing the number of delay slots by one. The reason for this is that in this region of L1-I cache size the relative increase in CPI from increasing the number of delay slots (0.03–0.15) is less than the decrease in CPI from doubling the cache size (0.05–0.2).

Finally, an example of the dependence of CPI on t_{CPU} is illustrated in Figure 5 which plots CPI versus t_{CPU} for various cache sizes in a system having two branch delay slots and a miss penalty of 10 cycles. Smaller caches are affected more by the code size increase that comes with additional delay slots. Likewise, the higher miss ratios of smaller caches means that they experience more performance loss as the miss penalty is increased. Figure 5 shows

Delay slots	CTIs Predicted Taken % of total	% correct	CTIs Predicted Not-Taken % of total	% correct	Cycles per CTI	Additional CPI
1	47	93	53	49	1.092	0.012
2	47	93	53	49	1.339	0.044
3	47	93	53	49	1.670	0.087

Table 3: Performance of branch prediction versus number of branch delay slots. The numbers of predict-taken and predict-not-taken CTIs are expressed as percentages of the total number of executed CTIs. CTIs make up 13 % of all executed instructions.

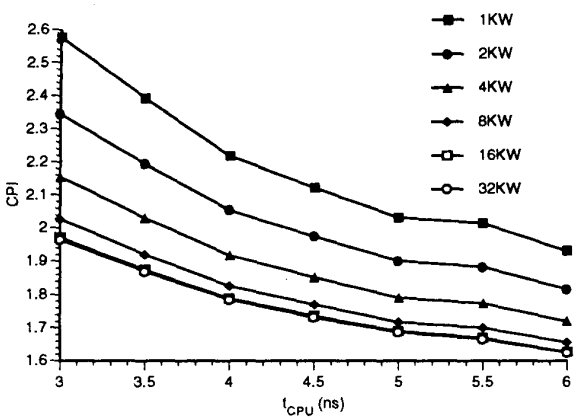

Figure 5: t_{CPU} versus L1-I cache size: $b = 2$, $B_{L1} = 4$ W, $P_{L1} = 10$ cycles. The curves are not smooth because of the need to make the L1 miss penalty an integer number of cycles.

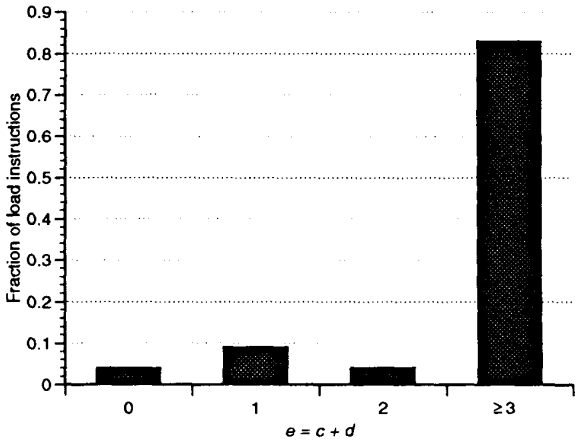

Figure 6: Histogram of ϵ values for benchmark suite used in this study.

that CPI decreases as t_{CPU} increases because the miss penalty (in cycles) decreases with increasing t_{CPU}.

3.2 The effect of load delays

The L1-D cache supplies data to the CPU when *load* instructions are executed. The number of CPU cycles between the execution of a load instruction and the time at which the data arrives at the CPU is determined by the pipeline depth of the L1-D cache, d_{L1-D}. These cycles are called load delay slots.

The MIPS ISA has only one memory addressing mode for all load instructions. This mode, called *register plus displacement*, uses a 16 b signed displacement from a 32 b general purpose register. To aid our discussion of load delay cycles, the following fragment of code, including an `lw` (load word) instruction is given as an example:

```
subu r5, r5, r4
  .
  .          c
  .
lw r3, 100(r5)
  .
  .          d
  .
addu r4, r3, r2
```

The `lw` instruction loads a word from memory location 100 plus the contents of address register `r5` into `r3`. It is preceded by an instruction that subtracts the contents of `r4` from `r5` and places the contents in `r5`. It is followed by instruction that adds the contents of `r3` to `r2` and places their sum into `r4`.

We define the following terms: 1) c is the number of instructions between the last instruction to modify the address register and the load instruction; 2) d is the number of instructions between the load instruction and the first instruction that uses the result of the load; 3) ϵ is the sum of c and d; and 4) l is the number of load delay cycles in the architecture. In this example, it is possible to execute the `lw` directly after the `subu` instruction, implying $c \geq 0$, but the `add` instruction can not be executed until l cycles after the `lw` instruction.

If nothing is done to hide the lost cycles due to load delays, the CPI will increase by the fraction of instructions that are loads, times the number of delay cycles per load. In the benchmark suite used in our experiments this fraction is 0.25. To reduce the effect of load delay cycles, load instructions can be rescheduled so that load delays do not result in stall cycles. The rescheduling may be done statically at compile time or dynamically at execution time. The restriction that a load instruction must execute after the instruction that modifies its address register limits the the number of delay cycles that can be hidden in this manner to ϵ.

The number of delay cycles that can be hidden is determined from the dynamic distribution of values for ϵ, given in Figure 6. The large fraction (over 80 %) of loads that have ϵ values of three or more indicates a great opportunity for moving load instructions away from where their results are used. The reason for this high percentage of loads having $\epsilon \geq 3$ is that many variable references are to global static variables or to local automatic variables whose address registers do not change often. Program measurements reveal that over 90% of array and structure references are to global variables and over 80% of scalar references are to local variables [PS82]. The MIPS compiler allocates most global static variables from a 64 KB area of memory called the *gp area*. Variables in this area are addressed using a single dedicated register that is set once at the beginning of the program. Local automatic variables

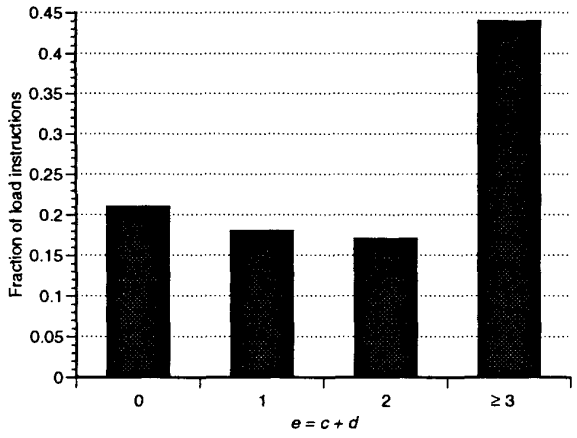

Figure 7: Histogram of ϵ values with with restrictions due to basic-block boundaries.

Delay slots	Static Delay cycles per load	Static CPI	Dynamic Delay cycles per load	Dynamic CPI
1	0.21	0.05	0.04	0.01
2	0.62	0.18	0.19	0.05
3	1.21	0.29	0.39	0.08

Table 5: The increase in CPI due to load delay cycles.

are addressed using the stack pointer register, which only changes at procedure call entry points. Furthermore, the MIPS compiler attempts to address as many memory locations as possible from the same address register [CCH+87].

Though Figure 6 would indicate that most load delay cycles can be hidden dynamically for an architecture with $l \leq 3$, the presence of control transfer instructions reduces the number of opportunities for hiding load delay cycles at compile time. Delayed loads assume l delay slot instructions before an instruction that uses the result of the load. A compiler attempts to fill these delay slots with useful instructions from within the basic block that appear before the load instruction. These instructions must be independent of the load. If enough independent instructions cannot be found to fill all the delay slots of a load the excess delay slots are filled with `noop` instructions. Not being able to look beyond a basic block greatly restricts the ability of the compiler to statically hide delay cycles compared to a completely dynamic technique. Figure 7 illustrates this by presenting the data of Figure 6 with the restrictions introduced by basic block boundaries. These boundaries prevent a load instruction from being moved away from the instructions that uses its result, causing the value of c to be reduced. A comparison of Figures 6 and 7 shows clearly that these boundaries change the distribution of ϵ, so that far fewer delay slots can be hidden in an architecture having $0 < l \leq 3$. However, even Figure 6 is optimistic in that it represents the best static scheduling for load instructions within a basic block. This scheduling assumes that only true dependencies are preserved and includes perfect memory address disambiguation which allows a load to be moved before a store if their addresses do not reference the same word.

Table 5 shows the CPI increase that results from 1, 2 and 3 load delay cycles. The data in this table is calculated using a dynamic frequency for load instructions of 0.25 from Table 1, and the data in Figures 6 and 7. Even though the data in Table 5 shows that dynamic

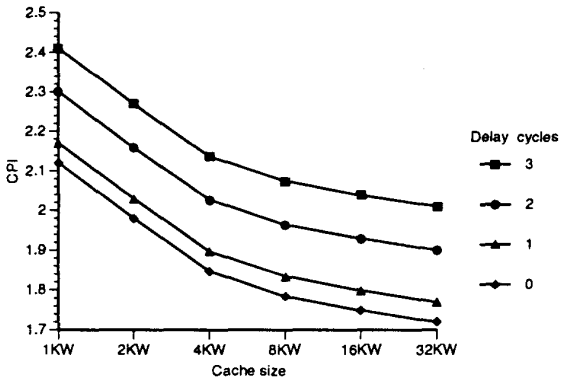

Figure 8: CPI versus L1-D cache size for different load delay cycles: $B_{L1} = 4$ W and $P_{L1} = 10$ cycles.

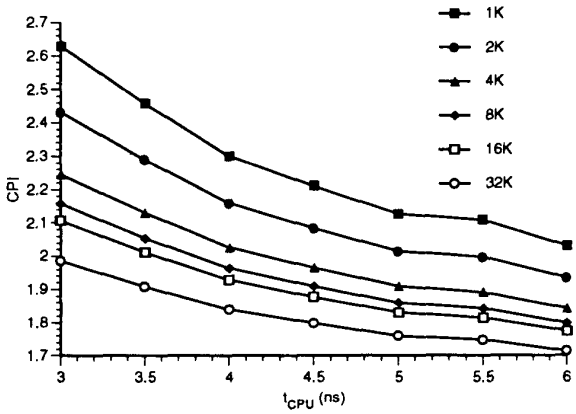

Figure 9: t_{CPU} versus L1-D cache size: $l = 2$, $B_{L1} = 4$ W, $P_{L1} = 10$ cycles.

load delay slot hiding could potentially be much better at hiding load delay slots than static instruction scheduling, dynamic schemes would require out-of-order instruction execution, extra register-file ports, and a separate load address adder. This extra hardware will increase the cycle time. Rather than trying to estimate the change in t_{CPU}, we assume static instruction scheduling in the remainder of our analysis, and refer to Table 5 to estimate the performance of dynamic scheduling or to estimate how much the t_{CPU} could be increased in a dynamic scheme before it has less performance than static instruction scheduling.

Figure 8 shows CPI versus L1-D cache size for 0 through 3 delay cycles. As in the L1-I experiments, the block sizes of the caches have been optimized for the refill latency and miss penalty. This figure also shows the effect that load delay cycles have on CPI. We have assumed that load instructions are interlocked. This avoids the code expansion associated with the need to insert `noop` instructions into load delay slots which cannot be filled with useful instructions. The figure shows that the code expansion effect becomes more significant for delay values greater than one cycle. In fact, in order to decrease CPI after increasing the number of load delay cycles from one to two requires at least a four-fold increase in cache size.

Figure 9 combines the effects of varying t_{CPU} with the variation of L1-D size. The parameters used in this figure are two load delay cycles and a miss penalty of 10 cycles. This plot shows CPI for L1-

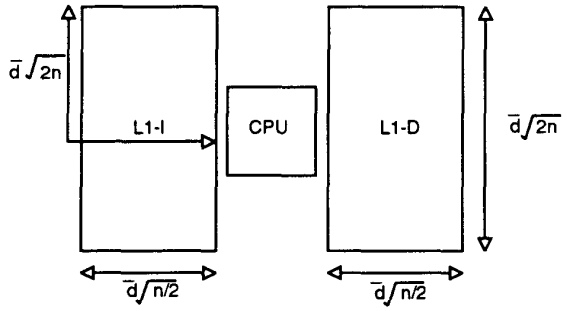

Figure 10: The minimum delay arrangement of $2n$ L1 cache SRAM chips.

D caches with 2 delay cycles and perfect compile time instruction scheduling to hide load delays. The curves in this figure will be shifted up or down by the same distances as the curves in Figure 8 for different numbers of delay cycles.

4 Cache Access Time

In this section we develop a macro-model, or simple expression, to estimate the access time of a direct-mapped cache that is implemented from GaAs SRAM chips mounted as bare die directly on an multichip module (MCM). It will be seen from the macro-model that t_{L1} increases linearly with number of chips. In the next section the results of Sections 3 and this section are combined to find an optimum primary cache organization.

The access time of an MCM-based L1 cache t_{L1} can be divided in two parts: the on-chip access time of the SRAM array, t_{SRAM}, and the round trip delay from the CPU to the cache and back, t_{MCM}. The equation for t_{L1} is given by,

$$t_{L1} = t_{SRAM} + 2t_{MCM} \tag{3}$$

In general, t_{MCM} is dependent upon the electrical characteristics of the MCM interconnect (R, L and C) and the longest distance from the CPU to any cache SRAM. Given n, the number of SRAM chips in the L1-I or L1-D (L1 data) cache, t_{MCM} can be approximated by the following linear equation

$$t_{MCM} = k_0 + k_1 n \tag{4}$$

where k_0 is a constant term associated with the delay of the off-chip drivers and receivers and k_1 is a linear coefficient that represents the additional delay per chip.

If n is the number of SRAM chips in the L1-I or L1-D cache, then to minimize interconnection delay these chips should be arranged as closely as possible to a $\sqrt{n/2} \times \sqrt{2n}$ rectangle as shown in Figure 10. If the CPU is placed in the middle of the long side of this rectangle, the maximum length l of a wire from the CPU to any chip is $\bar{d}\sqrt{2n}$ where $\bar{d}$ is the average chip pitch. More specifically, the term $\bar{d}$ is defined as the average of the horizontal and vertical pitches of the chip, including the width of adjacent wiring channels.

The value of the linear coefficient can be expressed as

$$k_1 = Z_0 C_{bond} + 2\bar{d}^2 R_{MCM} C_{MCM} \tag{5}$$

where Z_0 is the characteristic impedance of the MCM interconnect and R_{MCM} and C_{MCM} are the resistance and capacitance per unit length of interconnect. This equation is a modified form of an equation for the packaging delay of interconnect presented in

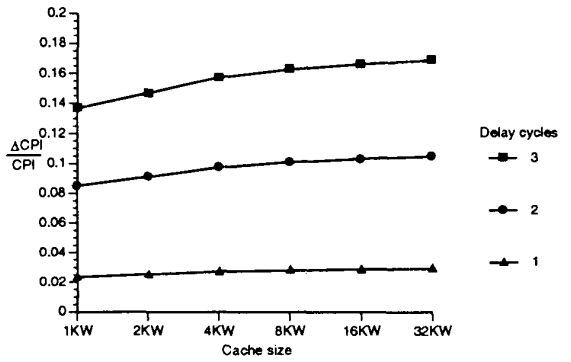

Figure 11: $\frac{\Delta CPI}{CPI}$ for L1-D caches.

[Bak90]. The first term of (5) is the delay due to parasitic capacitance (C_{bond}) of the bonding method and the pad that connects the chip to the MCM. The second term is the distributed RC delay of the MCM interconnection lines and is proportional to the square of the length of the MCM interconnect that is being driven. However, this length is proportional to the square root of the number of chips in the cache n, making the second term proportional to n. Equation (5) assumes the interconnect is quite lossy and so neglects any delay from transmission line behavior. The value of k_1 calculated using (5) is within 1 % of the value calculated using SPICE circuit analysis of real layouts [Olu91].

Equations (3)-(5) can be combined to produce the following expression for t_{L1}

$$t_{L1} = t_{SRAM} + 2k_0 + 2n\left[Z_0 C_{bond} + 2\bar{d}^2 R_{MCM} C_{MCM}\right] \tag{6}$$

Equation (6) defines t_{L1} for an MCM based direct-mapped L1 cache as a function of its size in SRAM chips. This equation can be combined with the pipeline depth of the L1 cache, d_{L1}, to determine the value of t_{CPU}.

The macro-model for t_{L1} defined by (6) is, of course, specific to particular SRAM chips and MCM technology. In general, the L1 cache may be on a PCB or, more likely, in the future, on the same chip as the CPU. Whatever the case, the important point from the multilevel optimization viewpoint is to develop a macro-model for t_{L1} as a function of cache organization.

5 Calculating TPI

In Figures 5 and 9, it was shown, among other things, that CPI decreases with increase in t_{CPU}. Equation (2) shows that t_{CPU} in turn is simply the cache access time scaled by the pipeline depth, $\frac{t_{L1}}{d_{L1}}$. Thus CPI will decrease with increase in t_{L1}. Increasing d_{L1}, or pipelining cache access, has the potential to decrease t_{CPU} without decreasing t_{L1}, however, as discovered in Section 3, increasing the pipeline depth also increases CPI (Figures 4 and 8).

If we consider incremental changes to the quantities in (1), the following equation can be obtained,

$$\frac{\Delta TPI}{TPI} = \frac{\Delta CPI}{CPI} + \frac{\Delta t_{CPU}}{t_{CPU}} \tag{7}$$

Thus, although pipelining cache access has the potential to decrease t_{CPU}, in order to increase system performance, the relative decrease in t_{CPU}, $\frac{\Delta t_{CPU}}{t_{CPU}}$, must be greater than the relative increase in CPI, $\frac{\Delta CPI}{CPI}$. In Figure 11 $\frac{\Delta CPI}{CPI}$ is plotted for L1-D caches in order to measure the t_{CPU} change required to improve performance. The

L1 size	L1-I(0)	L1-I(1)	L1-I(2)	L1-I(3)	L1-D(0)	L1-D(1)	L1-D(2)	L1-D(3)
1KW	10.0	5.0	3.5	3.5	9.2	4.6	3.5	3.5
2KW	10.1	5.1	3.5	3.5	9.4	4.7	3.5	3.5
4KW	10.3	5.2	3.5	3.5	9.6	4.8	3.5	3.5
8KW	10.9	5.4	3.6	3.5	10.1	5.1	3.5	3.5
16KW	12.0	6.0	4.0	3.5	11.2	5.6	3.7	3.5
32KW	14.2	7.1	4.7	3.5	13.4	6.7	4.5	3.5

Table 6: Optimal cycle times for L1 caches for $B_{L1} = 4$. All times are in nanoseconds. The numbers in parentheses specify the pipeline depth of the caches.

relative CPI is plotted against cache size for an architecture with no load delay cycles. This figure shows that as the number of delay cycles is increased, the relative decrease in t_{CPU} required to improve performance grows larger. For 2 delay cycles, the decrease is less than 10%. This indicates that performance will be improved if t_{CPU} can be reduced by more than 10%. The figure also shows that as the cache size grows, the required improvement in t_{CPU} grows larger, suggesting that pipelining the cache access path is less effective for large caches. In general, Figure 11 shows that if the CPI of a system is low, deep pipelining will require a greater reduction in t_{CPU} before the performance of the system increases.

To determine exactly how t_{CPU} varies with cache size and pipeline depth we used the timing analyzer developed in [SMO90] to estimate t_{CPU} for L1 cache sizes of 1 to 32 KW and L1 access pipeline depth values of 0 to 3. The value of t_{L1} is estimated using the delay model for MCM-based caches that was developed in the Section 4. We have assumed that the SRAM chips have both address and data registers. The overhead delay of these latches was included in all timing analyses. In each case, the timing analyzer was used to optimize the timing of the circuit using a multiphase clocking scheme. Optimized clocking produces a t_{CPU} which increases by a factor of $\frac{1}{d_{L1}+1}$ for each unit increase in the access time, t_{L1}, of the L1 cache. The factor $d_{L1} + 1$ is the number of pipeline stages, including the ALU, in the the address calculation path and the cache access path (see Figure 1). This means that there is a smaller dependence of t_{CPU} on cache access time in deeper cache pipelines.

The results of the timing analysis are tabulated in Table 6. The minimum cycle time (3.5 ns) shown in the table is set by the time required to add two integer operands in the ALU (2.1 ns) and feed the result back to the ALU (1.4 ns). The pipeline depth of this path is one.

The data in Table 6 shows that for a pipeline depth of 0 the L1-I and L1-D caches limit t_{CPU} to more than 10 ns. This demonstrates that requiring the L1 cache to be accessed in the same cycle as the execution unit will lead to excessively long cycle times compared to the ALU add time which is 2.1 ns. When the pipeline depth of the L1-I and L1-D caches are increased to 3, the feedback loop around the ALU is critical for all cache sizes, and the cycle time is limited to 3.5 ns.

The L1 cache experiments have been presented above in terms of L1-I or L1-D. In order to combine a particular L1-I organization and an L1-D organization we take the maximum t_{CPU} of each, as the new system cycle time t_{CPU}. These are combined with the results of Figures 4 and 8 to give Figure 12.

A number of conclusions can be drawn from Figure 12. It shows that when the L1 cache is divided equally between L1-I and L1-D, performance is maximized when the number of branch delay slots is equal to the number of load delay slots, *i.e.*, $b = l$. The reason for this is that pipelining the different sides of the L1 cache to different depths causes the t_{CPU} set by one side to be shorter than that of the other. Since, the side with the longest cycle time will set the system cycle time, the extra pipelining on the other side will be wasted, *i.e.*, there will be extra CPI without the benefit of reducing t_{CPU}. Figure 12 also shows that for every combination of load

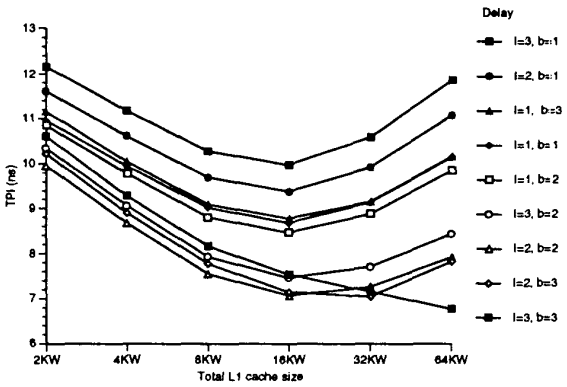

Figure 12: TPI versus cache size for various instruction and data cache delay-slot combinations; $B_{L1} = 4$ W and $P_{L1} = 10$ cycles.

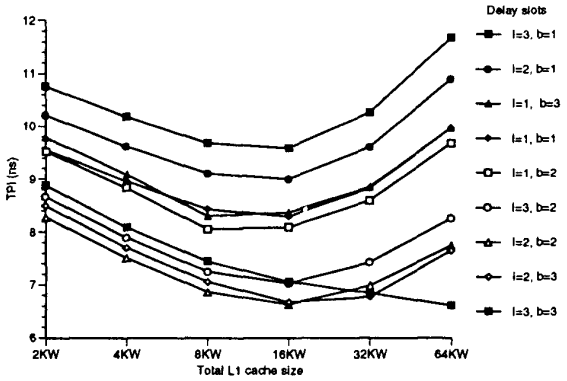

Figure 13: TPI versus cache size for a number of delay-slot combinations, with $B_{L1} = 4$ W and $P_{L1} = 6$ cycles.

and branch delay slots, there is an L1 cache size that maximizes performance. Maximum performance is reached for medium size caches at TPI = 6.8ns, when $b = 3$, $l = 3$, $S = 64$ KW, and $t_{CPU} = 3.5$ ns. The final conclusion is that increasing the number of branch and load delay slots is able to increase performance because doing so reduces the dependence of t_{CPU} on the size of the cache. This allows larger caches to be accessed without increasing cycle time.

If dynamic out-of-order load execution were used instead of static load instruction scheduling, a new maximum performance of TPI = 6.2ns could be reached when the number of branch and load delay slots are both equal to three ($l = 3, b = 3$) and the combined L1 cache size is 64 KW. The value of TPI is strongly dependent on the cycle time. We calculate that if the implementation of out-of-order load execution required more than a 10% increase in t_{CPU}, the performance of the dynamic scheme would be worse than the performance of the best static load delay scheduling organization.

Different L1 penalties change the performance and location of the optimal design points: higher penalties increase both the L1 cache size and pipeline depth. Lower penalties have the opposite effect, as shown in Figure 13, where maximum performance is reached at TPI = 6.61ns ($b = 2$, $l = 2$, $S_{L1} = 16$ KW, and $t_{CPU} = 3.5$ ns). Smaller refill penalties also make it possible to take advantage of the fact that increasing the number of branch delay slots increases CPI less than a comparable increase in load delay slots. This can be done by using a larger size L1-I cache

than L1-D cache and pipelining the access of the L1-I cache more deeply. When this is done, the maximum performance is reached at TPI = 6.5ns (S_{L1-I} = 32 KW and S_{L1-D} = 8 KW).

6 Conclusions

The performance of pipelined primary caches of various configurations has been evaluated using a multilevel design optimization procedure. The objective is to reduce the cycle time by pipelining cache, while minimizing the penalty from the branch and load delay cycles. For the instruction cache, we found that delayed branching with optional squashing results in a lower CPI than does a branch-target buffer that is small enough to allow single cycle access. However, for small cache sizes and high miss penalties, the extra cache misses caused by the larger code size of the delayed branching scheme make the performance of these two schemes comparable.

Tradeoffs among cache-access pipeline depth, primary cache size, and t_{CPU}, were investigated using timing analysis. We found that in systems having no delay slots for cache access, the value of t_{CPU} can be up to five times the integer-addition delay. However, even for the largest caches that were considered, pipelining the cache with 2 or 3 stages ensured that the ALU, rather than cache, was the critical path.

This study has shown significant performance benefits from pipelining the primary cache of a CPU which has a peak execution rate of one instruction per cycle. The extra cycles added through pipelining can be hidden often enough that overall performance improves. Furthermore, pipeline depth is better tolerated on the instruction side than on the data side, where basic-block boundaries make static instruction scheduling less effective at hiding load-delay slots. This suggests that for maximum performance, the instruction cache should be larger and more deeply pipelined than the data cache.

Finally, we have shown that the benefits of pipelining are twofold. First, it reduces the t_{CPU} and the dependence of t_{CPU} on cache access time. Second, it allows larger caches to be accessed without increasing t_{CPU}, thus lowering CPI. These results suggest that the cache size versus set-associativity tradeoff may need to be re-examined. If t_{CPU} is less dependent on the access time of pipelined L1 caches, then increasing the associativity of the cache to lower the miss ratio will have a larger performance benefit for pipelined caches.

References

[Bak90] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

[CCH+87] F. Chow, S. Correll, M. Himestei, E. Killian, and L. Weber, "How many addressing modes are enough?," in *Proc. 2nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 117–121, Oct. 1987.

[CCS+91] T. I. Chappell, B. A. Chappell, S. E. Schuster, J. W. Allan, S. P. Klepner, R. V. Joshi, and R. L. Franch, "A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture," *IEEE Jour. of Solid-State Circuits*, vol. 26, pp. 1577–1585, Nov. 1991.

[HCC89] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc.16th Annual Int. Symp. Computer Architecture*, pp. 224–233, June 1989.

[Hil87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, 1987.

[HP90] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*. San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[KH92] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.

[KT91] M. Katevenis and N. Tzartzanis, "Reducing the branch penalty by rearranging instructions in a double-width memory," in *Proc. 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 15–27, Apr. 1991.

[Lil88] D. J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer Magazine*, vol. 21, pp. 47–55, July 1988.

[LS84] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer Magazine*, vol. 17, pp. 6–22, Jan. 1984.

[MBB+91] T. N. Mudge, R. B. Brown, W. P. Birmingham, J. A. Dykstra, A. I. Kayssi, R. J. Lomax, O. A. Olukotun, K. A. Sakallah, and R. Millano, "The design of a microsupercomputer," *IEEE Computer Magazine*, vol. 24, Jan. 1991.

[MH86] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proc.13th Annual Int. Symp. Computer Architecture*, pp. 396–403, june 1986.

[MIP88] MIPS Computer Systems, Inc, *MIPS RISCompiler Languages Programmer's Guide*, Dec. 1988.

[OBL+91] O. A. Olukotun, R. B. Brown, R. J. Lomax, T. N. Mudge, and K. A. Sakallah, "Multilevel optimization in the design of a high-performance GaAs microcomputer," *IEEE J. Solid-State Circuits*, vol. 26, May 1991.

[Olu91] O. A. Olukotun, *Technology-Organization Tradeoffs in the Architecture of a High Performance Processor*. PhD thesis, The University of Michigan, Ann Arbor, 1991.

[Prz90] S. A. Przybylski, *Cache and Memory Hierarchy Design*. San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[PS82] D. Patterson and C. Séquin, "A VLSI RISC," *IEEE Computer Magazine*, vol. 15, pp. 8–21, Sept. 1982.

[Smi78] A. J. Smith, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Trans. Software Engineering*, vol. SE-4, pp. 121–130, Mar. 1978.

[Smi81] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annual Int. Symp. Computer Architecture*, pp. 135–147, July 1981.

[SMO90] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "*checkT$_c$* and *mint$_c$* : Timing verification and optimal clocking of synchronous digital circuits," in *Proc. IEEE Conf. Computer-Aided Design*, (Santa Clara, California), Nov. 1990.