

Citation: O. Olukotun, T. Mudge, R. Brown. Performance optimization of pipelined caches. *IEEE Trans. Computers*, to appear.

Multilevel Optimization of Pipelined Caches[†]

O.A. Olukotun[‡]

Department of Electrical Engineering
Computer Systems Laboratory
Stanford

and

T.N. Mudge and R.B. Brown

Dept. Electrical Engineering and Computer Science
University of Michigan, Ann Arbor

Abstract

This paper formulates and shows how to solve the problem of selecting the cache size and depth of cache pipelining that maximizes the performance of a given instruction-set architecture. The solution combines trace-driven architectural simulations and the timing analysis of the physical implementation of the cache. Increasing cache size tends to improve performance but this improvement is limited because cache access time increases with its size. This trade-off results in an optimization problem we referred to as multilevel optimization, because it requires the simultaneous consideration of two levels of machine abstraction: the architectural level and the physical implementation level. The introduction of pipelining permits the use of larger caches without increasing their apparent access time, however the bubbles caused by load and branch delays limit this technique. In this paper we also show how multilevel optimization can be applied to pipelined systems if software- and hardware-based strategies are considered for hiding the branch and load delays.

The multilevel optimization technique is illustrated with the design of a pipelined cache for a high clock rate MIPS-based architecture. The results of this design exercise show that, because processors with pipelined caches can have shorter CPU cycle times and larger caches, a significant performance advantage is gained by using two or three pipeline stages to fetch data from the cache. Of course, the results are only optimal for the implementation technologies chosen for the design exercise; other choices could result in quite different optimal designs. The exercise is primarily to illustrate the steps in the design of pipelined caches using multilevel optimization, however, it does exemplify the importance of pipelined caches if high clock rate processors are to achieve high performance.

Index terms—optimizing cache design, trace-driven simulation, multichip modules, pipelining, caches, cache access times, macromodels of delay.

[†] This work was supported by the Advanced Research Projects Agency under DARPA/ARO Contract No. DAAL03-90-C-0028.

[‡] Corresponding author: Department of Electrical Engineering
Computer Systems Laboratory, CIS 209
Stanford, CA 94305-4070
kunle@ogun.stanford.edu
415-725-3713

1 Introduction

The performance evaluation of cache-based systems has received considerable attention [1][2][3]. These studies have considered the impact of architectural-level issues like cache size, associativity, line length, write policies, etc. However, different cache organizations, in particular size, change a cache's access time, and thus also affect performance. Increasing cache size tends to improve performance but this improvement reaches a point of diminishing returns because cache access time increases with its size. In this paper we show how to account for this by simultaneously considering two levels of machine abstraction that are normally dealt with separately in the design process: 1) the architectural level, where a range of cache sizes and different pipe depths are considered; and 2) the logic gate delay level which determines the feasible cycle time of these different combinations of cache size and pipe depth. Architectural trade-offs are characterized with simulations driven by long traces of multiprogrammed application codes. These simulations also account for hardware and software techniques for hiding pipeline delays, so that the full effect of cache pipelining can be assessed. The feasible cycle times of the different architectures are determined from detailed timing analyses of critical paths. We refer to the simultaneous consideration of two level of machine abstraction as multilevel optimization [4][5].

The methodology is demonstrated on the design of a pipelined cache for a high-performance microprocessor, which is based on the MIPS instruction set architecture (ISA) [6] and was planned to be implemented in GaAs direct-coupled FET logic with multichip module (MCM) packaging [7]. The use of GaAs supports fast logic but at relatively low integration densities. This suggests a simple high clock rate pipeline implementation with MCM packaging to reduce inter-chip delays, which became the starting point of the design exercise.

A block diagram of the processor is shown in Figure 1. The cache is split into instruction and data halves (i-cache and d-cache) to provide an instruction or data access every cycle. Implementing a pipelined cache involves splitting the access of the cache into two or more stages and placing latches between each stage [8]. The complete architecture includes a floating point unit, a memory management unit, and a second level of cache. Our experiments are performed for i- and d-cache sizes that varied from 1K to 32K words (W) of 32-bits, with

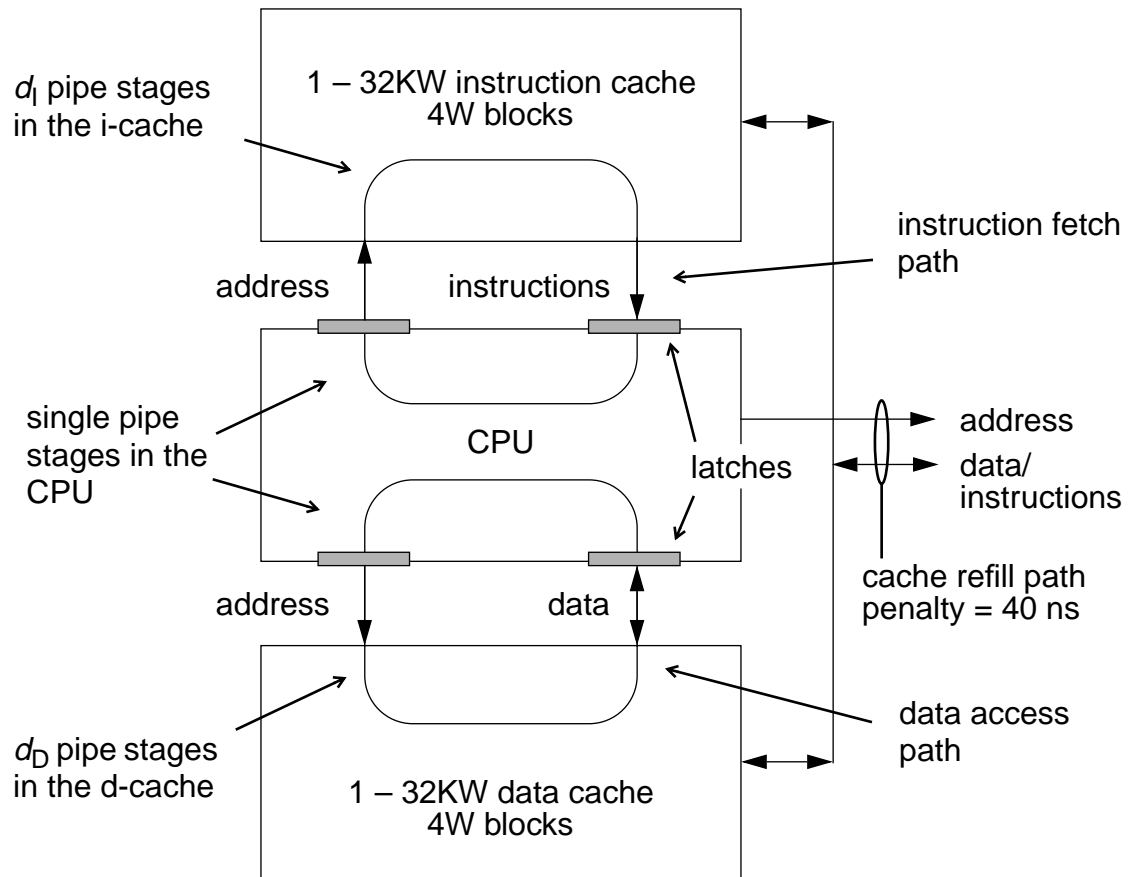


Figure 1: The processor architecture

Shows the range of parameters that are considered in this study and the circular access paths of the pipelined primary i- and d-caches.

block sizes of 4W and miss penalties of 40 nanoseconds (ns). Further details, and the more complex problem that results from varying the miss penalty by changing the line size, transfer rates, or introducing a second cache level, can be found in [9].

Our experiments in through show that as many as three branch delay or load 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 fewer pipeline stages. Of course, the results are only optimal for the implementation technologies chosen for the design exercise; other choices could result in quite different optimal designs. The exercise is primarily to illustrate the steps in the design of pipelined caches using multilevel optimization, however, it does exemplify the importance of pipelined caches if high clock rate processors are to achieve high performance.

The organization of this paper is as follows. The next section of the paper, using time-per-instruction as the performance metric, formulates the cache optimization problem when cache access can be pipelined. It also describes the multilevel optimization method used in this study. In Section 3 simulation is used to show how the number of clocks per instruction (CPI) is affected by the primary instruction and data cache organizations. The resulting data characterizes CPI as a function of the cache size, pipeline depth, and processor cycle time t_{CPU} . In Section 4 we use a simple analytical delay model, or macromodel, for MCM-based caches to show how the cycle time is affected by the size of the cache and its pipeline depth, developing a functional dependence between t_{CPU} and cache size and pipe depth. 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 Cache Optimization Problem

2.1 The performance metric

A widely recognized metric for architectural performance comparison is the time it takes to execute a realistic set of benchmarks [10]. Given a specific ISA and compiler, a performance metric that is directly proportional to execution time is average time per instruction (TPI). The equation for TPI can be written as,

$$\text{TPI} = \text{CPI} \times t_{CPU} \quad (1)$$

where CPI is the familiar clocks-per-instruction and t_{CPU} is the cycle time of the CPU. In the following discussion we use TPI as the performance metric. The optimization problem seeks to minimize TPI subject to the constraints of architectural organization and implementation

that are implicit in the two terms on the right hand side of (1).

The CPI term in (1) is a function of cache size, S , that decreases when S increases, because larger caches reduce the miss rate, m , and with it the total penalty due to cache misses. CPI is also a function of the depth of the pipeline to the instruction cache, d_I , because this pipeline can result in branch delays that cannot be filled with useful instructions. These hazards in the pipelined processing of instructions occur more frequently as d_I increases, leading to an increase in CPI. Similarly, CPI is also an increasing function of the depth of the pipeline to the data cache, d_D . However, the exact relationship between pipeline depth and CPI is more complex because, as we showed in [9], a significant number of load or branch delay cycles can be hidden by static compile-time instruction scheduling or by dynamic hardware-based methods. This tends to slow the increase in CPI with d^\dagger . To further complicate matters, in the case of software techniques, i-cache miss ratios tend to increase because these techniques increase code size. It is necessary to rely on trace-driven experiments to determine the net result of these conflicting effects when characterizing the exact dependence that CPI has on d .

Finally, CPI is a function of t_{CPU} because we are considering the case where the cache miss penalty, P , is a fixed time penalty. The contribution to CPI due to cache misses is thus mP/t_{CPU} , making CPI a decreasing function of t_{CPU} . We can make explicit the foregoing dependencies by writing, CPI as $\text{CPI}(S, d, t_{CPU})$.

As we have noted earlier, the t_{CPU} term in (1) is a function of cache size, increasing as S increases for a given pipeline depth, d . It is also a function of d , because increasing the depth of the pipeline allows the clock cycle to be shortened. These dependencies can be made explicit by writing, t_{CPU} as $t_{CPU}(S, d)$, and then (1) becomes,

$$\text{TPI} = \text{CPI}(S, d, t_{CPU}) \times t_{CPU}(S, d) \quad (2)$$

2.2 Multilevel optimization

Equation (2) clearly identifies the two levels of machine abstraction across which the optimization must be performed. CPI is an architectural figure of merit and t_{CPU} is a logic or circuit implementation figure of merit. Architectural level simulations are performed to obtain

[†] We will use d without using a subscript when context makes it clear, or when d could refer to either the i- or d-cache.

an empirical definition of the function $CPI(S, d, t_{CPU})$ for a range of values of cache size (S), pipeline depth (d) and cycle time (t_{CPU}). Only a subset of the values for CPI defined on the points in the volume $\langle S, d, t_{CPU} \rangle$ are feasible. These are defined by the values of t_{CPU} that can be implemented given values for S and d . Within this feasible range of CPI the minimum value of TPI is selected as the optimum.

The architectural simulations were performed using a trace driven simulator, `cacheUM`, originally developed to study two-level cache organizations [11]. The traces were created using the MIPS program analysis tool `pixie` [12] from load modules of the benchmark programs listed in Table 1. A record of system calls was also made during the normal execution of the set of benchmarks. Each benchmark was used to represent a single process. To model the effect of multiprogramming, a system call file and a process configuration file controlled context switches between benchmarks when system calls were encountered or when a predefined time quantum had expired. More details can be found in [9]. The benchmarks included selections from the SPEC suite, the Livermore loops, and an X-windows application that generated a large number of context switches. The benchmark load modules were created using the optimizing compiler developed for the MIPS ISA [13]. In addition, we developed a post-processor to modify the traces produced by `pixie` to simulate the hazards that result from software or hardware optimization techniques for hiding pipeline delays. The CPI values presented in this paper represent the weighted harmonic mean of all of the benchmarks in this table. The weight for each benchmark correspond to its fraction of total execution time. The instruction count used to calculate CPI is that of optimized MIPS R2000 code for an architecture with no load or branch delay cycles.

Delay macromodeling and timing analysis are used to determine $t_{CPU}(S, d)$ for specific values of S and d . The timing analyzer used in this study, `minTC`, is capable of finding the minimum clock cycle time of a synchronous digital circuit [14], given accurate macromodels of gate and interconnect delay [15]. It thus sets a lower bound on $t_{CPU}(S, d)$ for specific values of S and d , and for a particular technology. Values greater than the lower bound for t_{CPU} are obviously feasible and may be desirable because they may cause $CPI(S, d, t_{CPU})$ to decrease due to its inverse dependence on t_{CPU} .

Benchmark	Description of benchmark	Instructions (millions)	Loads (as a% of instructions)	Stores (as a% of instructions)	Control transfer instructions (as a% of instructions)	Number of 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
doduced	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

The above set of benchmarks 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 "Control transfer instructions" (CTIs) includes both branches and unconditional jump instructions.

Figure 2 illustrates further the steps in multilevel design optimization. A set of candidate designs obtained by varying S and d , is encoded into the trace-driven simulator and the post-processor is also changed to reflect code optimizations and hardware techniques for reducing hazards for each value of d . The set of designs is simulated and the values for $CPI(S, d)$ are

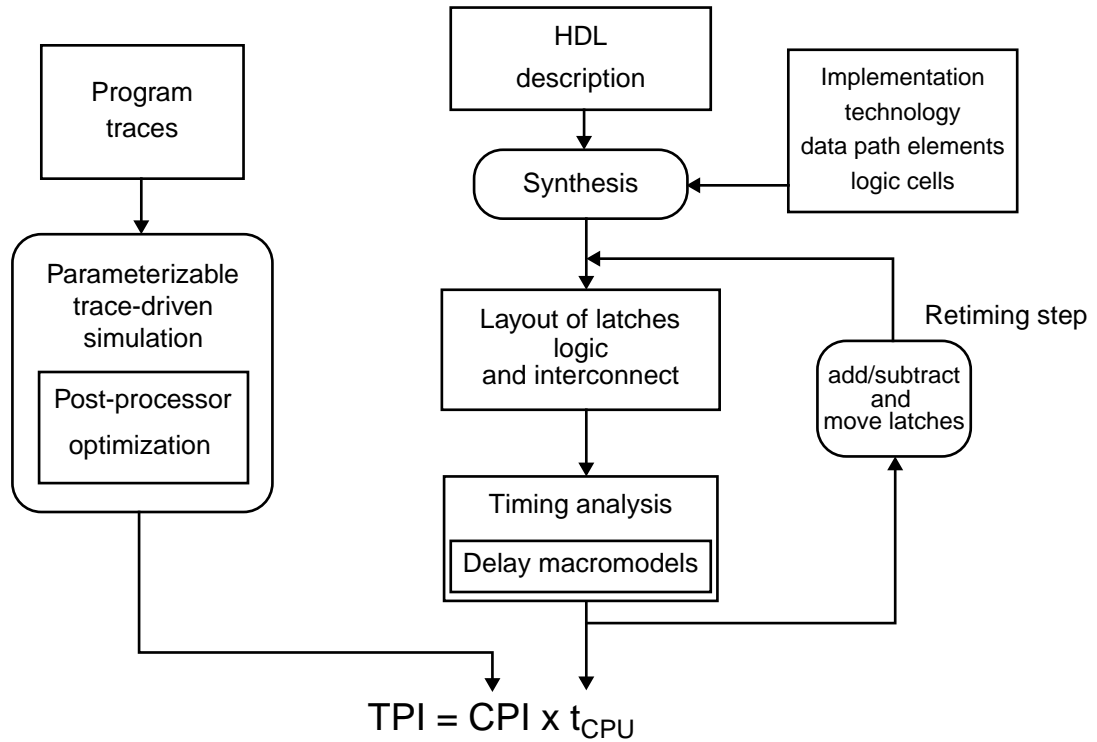


Figure 2: Multilevel design optimization

The process starts with an architecture defined in an HDL that is then synthesized. The design is successively refined until the design specifications are satisfied.

recorded. The simulator needs the cache miss penalty in cycles to calculate CPI. It is a straightforward matter to recalculate the effect of varying t_{CPU} on CPI, provided the relationship between the miss penalty in time and cycles is known. In our example a fixed time penalty of 40 ns is used to represent memory access time. The implementation of the cache refill path requires an 8 ns set-up and has a $2W$ wide bus running at 16 ns per transfer. Thus the penalty in cycles is given by,

$$\left[\frac{8 + \frac{\text{block-size}}{\text{refill-path}} \times 16}{t_{CPU}} \right] \quad (3)$$

where, $\text{block-size} = 4W$ and $\text{refill-path} = 2W$. The realizable values of t_{CPU} corresponding to the values of S and d used to determine $\text{CPI}(S, d, t_{CPU})$ are obtained. from implementations of the CPU. The implementation starts with a register-transfer level (RTL) description in an

Delay slots	CTIs Predicted Taken		CTIs Predicted Not-Taken		Cycles per CTI	Additional CPI
	% of total	%correct	% of total	% correct		
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 2: Branch Prediction Performance

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. Average prediction accuracy = $(93 + 49)/2$, i.e., about 70%.

HDL (Verilog in our case) which is then synthesized to a physical layout [16]. The timing analyzer, `minTc`, is applied to the resulting layout and the minimum value of t_{CPU} is obtained. Changing the value of d does not require a complete resynthesis step; it can be accomplished through a retiming analysis again using `minTc` (see Figure 2).

3 CPI Measurements

In this section we present the architectural level simulations performed to obtain an empirical definition of the function $CPI(S, d, t_{CPU})$ for a range of values of cache size (S), pipeline depth (d) and cycle time (t_{CPU}). We do not yet restrict the function to feasible points in the $\langle S, d, t_{CPU} \rangle$ space.

3.1 The effect of branch delays

When a control transfer instruction (CTI) is executed, the next instruction to be fetched must be delayed until its address is known. The pipeline depth of the i-cache, d_I , is the number of “branch delay” cycles or slots by which the next instruction is delayed. A number of hardware and software schemes have been proposed to limit this effect on CPI [17][18][19]. Two representative schemes are evaluated here: a software-based delayed branch with optional squashing, and a hardware-based branch-target buffer.

In the software approach, we assumed delayed branches with squashing and a static pre-

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

Table 3: BTB prediction performance

diction policy in which backward branches and unconditional jumps are predicted to be taken, and forward branches are predicted to be not taken. Based on the prediction policy, the compiler tries to fill the branch delay slots with useful instructions from: (1) before the branch, (2) after the branch, or (3) the branch target. 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. Squashing inserts `noop` instructions into delay slots, and there is a code expansion penalty associated with using instructions from the branch target because they must be replicated. This is accounted for in our simulations.

Our experiments show that, as expected, CPI increases with the number of branch delay slots as a result of: (1) increases in the number of 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. Table 2 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 illustrates that 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, due to effective branch prediction, the increase is only 8.7%.

In the hardware approach we use a 256 entry branch-target buffer (BTB) with each entry being an address tag, a branch target address, and a 2-bit saturating counter for branch prediction [17]. If the BTB correctly predicts a branch taken, the target address is used as the next instruction address without a stall. When the BTB mispredicts a branch we assume there is a

one cycle stall to fill the BTB in addition to the stall cycles necessary for branch delay.

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 we assumed 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 3.

Table 2 and Table 3 can be used to compare the two approaches. The static scheme performs better because our simulations showed that it 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, while 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. The BTB was restricted to 256 entries as a result of the integration levels available to us. Implementations that favored higher levels of integration could alter the conclusions. However, other researchers have also shown that static branch prediction techniques using sophisticated program profiling and fetch strategies are competitive with much larger BTBs [19]. Of course, for static prediction, the additional CPI due to increased i-cache misses must be considered. Although this is outside of this discussion, we have shown elsewhere that for small cache sizes and large miss penalties, this would give the performance edge to the BTB approach [9]. Nevertheless, because its performance is roughly comparable and its hardware cost is lower, the static prediction scheme is used in the remainder of the cache experiments.

Figure 3 plots the total CPI for a range of i-cache size and delay slot numbers. This figure shows that for i-cache sizes of 1 to 16KW, 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, because, in this region of 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 (0.05–0.20).

Finally, an example of the dependence of CPI on t_{CPU} is illustrated in Figure 4 which

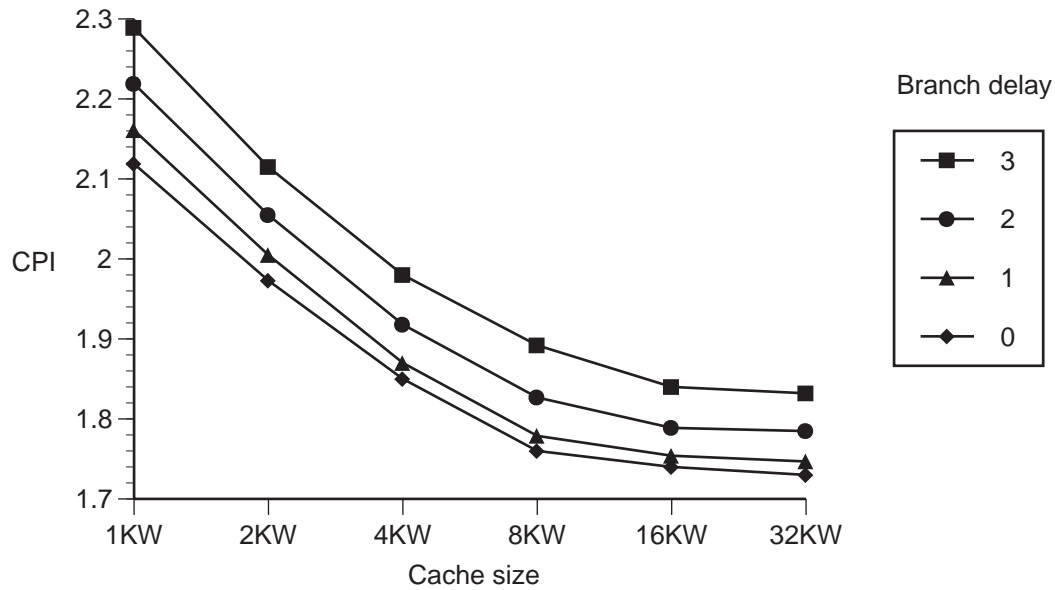


Figure 3: CPI vs. i-cache size

CPI versus i-cache size for different numbers of branch delay slots, d_I . This establishes $CPI(S, d)$, the dependence of CPI on S and d . The value of t_{CPU} is held constant at 4 ns and the d-cache is fixed at 4KW with $d_D=0$.

plots CPI versus t_{CPU} for various cache sizes in a system having two branch delay slots and a miss penalty of 40 ns. 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 in cycles is increased. Figure 4 shows that CPI decreases as t_{CPU} increases, because the miss penalty (in cycles) decreases with increasing t_{CPU} . Figure 4 and similar plots for the remaining values of d_I in the range 0 to 3 can be computed directly from Figure 3 when the miss penalty in nanoseconds is given. In our example MIPS-based processor, the miss penalty is 40 ns, and thus the penalty in cycles is given by $(40/t_{CPU})$. Combining this with the data in Figure 3 yields the empirical definition of the function $CPI(S, d_I, t_{CPU})$ for our range of values of cache size (S), i-cache pipeline depth (d_I) and cycle time (t_{CPU}). We now turn to the d-cache.

3.2 The effect of load delays

The data cache supplies data to the CPU when load instructions are executed. The MIPS ISA has only one memory addressing mode for all load instructions. This mode, usually called register plus displacement, uses a 16 bit signed displacement from a 32 bit general purpose

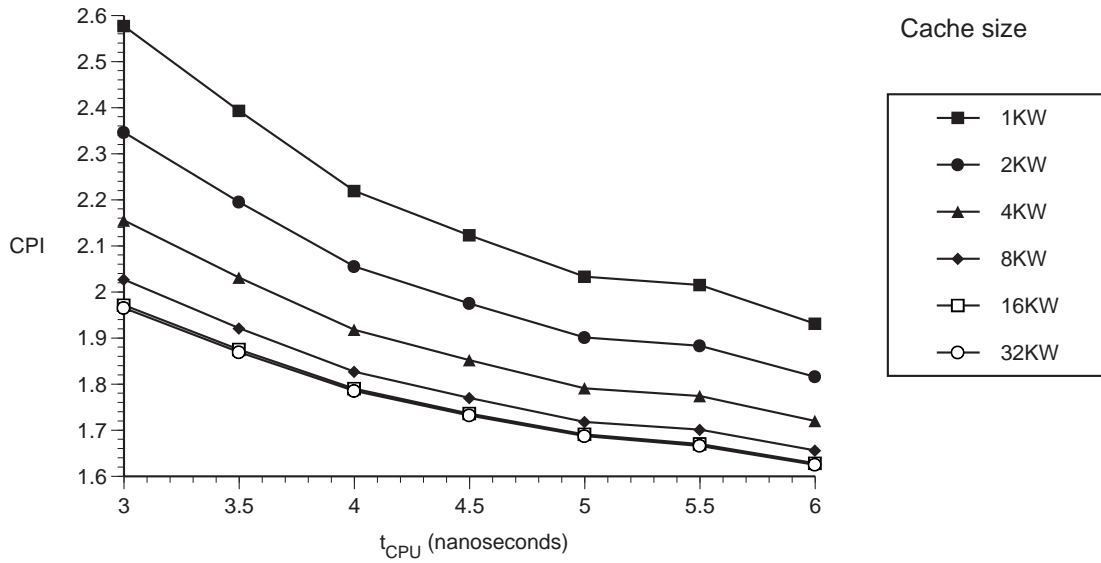


Figure 4: CPI vs. i-cache size

CPI versus t_{CPU} for different i-cache size and with $d_I = 2$. The curves are not smooth because of the need to make the miss penalty an integer number of cycles.

Delay slots	Static		Dynamic	
	Delay cycles per load	Extra CPI	Delay cycles per load	Extra 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 4: The increase in CPI due to load delay cycles

register. The number of CPU cycles, or load delay slots, 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 d-cache, d_D .

Table 4 shows the CPI increase that results from 1, 2 and 3 load delay cycles. The data in this table was again produced using our simulator in [9]. Two sets of simulations were performed to measure the impact of software scheduling and hardware scheduling to fill load

delay slots. In the first set of simulations, load delay slots were filled, where possible, by instructions from within basic blocks to simulate the effect of simple static software scheduling. In the second simulation, delay slots were filled, where possible, by instructions, without regard to basic block boundaries, to simulate dynamic or hardware supported slot filling. Though the data in Table 4 shows that dynamic 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 4 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 5 shows CPI versus d-cache size for 0 through 3 delay cycles. As in the i-cache experiments, the block sizes of the caches have been optimized for refill latency and miss penalty, see (3). This figure also shows the effect that load delay cycles have on CPI. We have assumed that load instructions are interlocked. This avoids code expansion when load delay slots cannot be filled with useful instructions. The figure shows that the performance impact of load delays becomes more pronounced for 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 6 combines the effects of varying t_{CPU} with the variation of d-cache size. This plot shows CPI for d-caches with 2 delay cycles and perfect compile time instruction scheduling to hide load delays, i.e., $d_I = 0$. The curves in this figure will be shifted up or down by the same distances as the curves in Figure 5 for different numbers of delay cycles.

The CPI analysis of pipelined caches shows that for low degrees of pipelining it is possible to decrease overall CPI by trading off increased pipeline depth for a larger cache size. This trade-off is more beneficial for the i-cache than for the d-cache because techniques for mitigating the effect of delay cycles are more successful on the instruction stream.

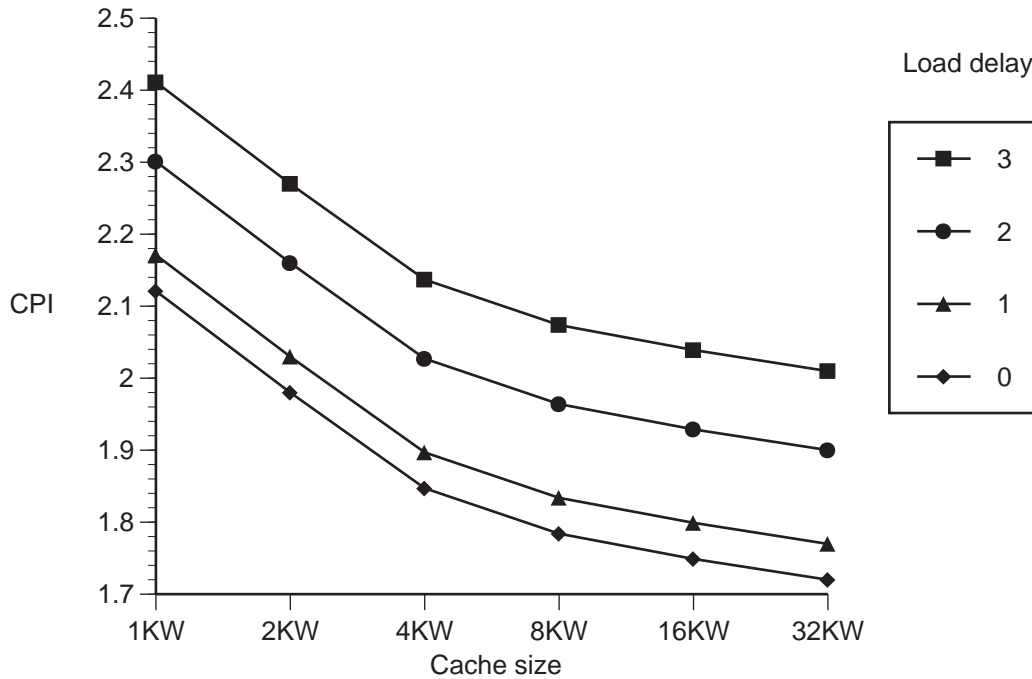


Figure 5: CPI vs. d-cache size

CPI versus d-cache size for different numbers of branch delay slots, d_D . This establishes $CPI(S, d)$, the dependence of CPI on S and d . The value of t_{CPU} is held constant at 4 ns and the i-cache is fixed at 4KW with $d_I = 0$.

4 Calculating feasible CPU cycle times

Calculating feasible values for $t_{CPU}(S, d)$ requires an expression for the cache access time as a function of its size, $t_{CACHE}(S)$. Intuitively, we would expect this function to yield an increasing value of t_{CACHE} as S grows larger. The exact nature of this function is highly dependent on implementation details. To provide a concrete example, in the next subsection we develop a function for t_{CACHE} for a direct-mapped cache that is implemented from GaAs SRAM chips mounted as bare die directly on a multichip module (MCM) [9]. Such functions, derived in terms of basic electrical properties and circuit geometries, are usually referred to as “macromodels” [15].

4.1 Cache Access Time

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

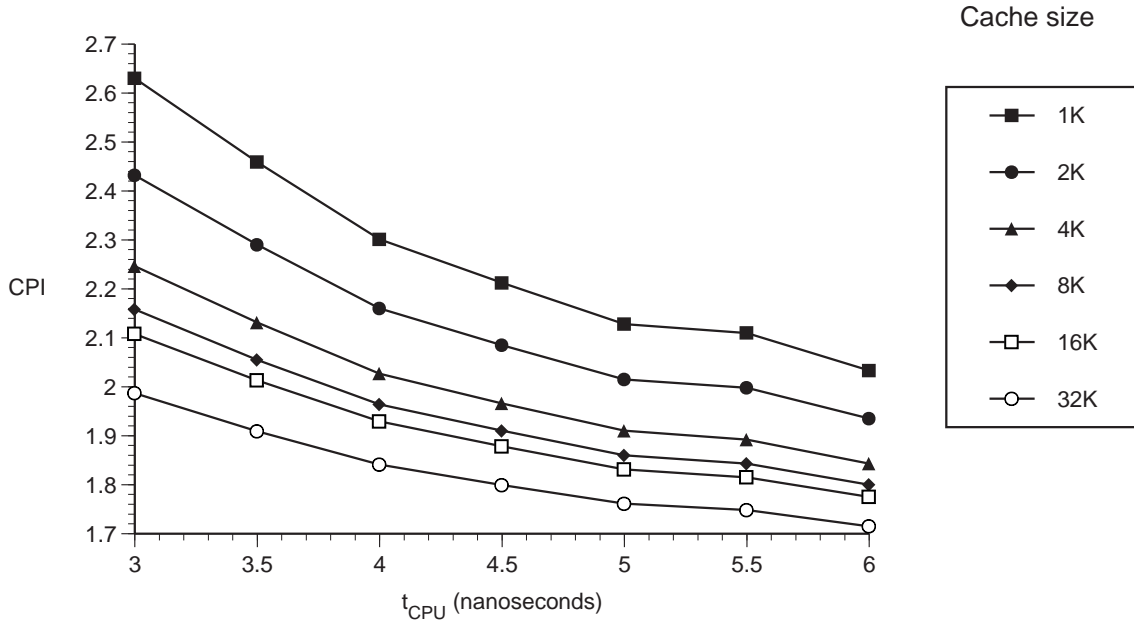


Figure 6: CPI vs. t_{cpu}

CPI versus t_{CPU} for different d-cache sizes and with $d_D = 2$. The curves are not smooth because of the need to make the miss penalty an integer number of cycles.

$$t_{CACHE} = t_{SRAM} + 2t_{MCM} \quad (4)$$

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 chip. Given n , the number of SRAM chips in the i-cache or d-cache, t_{MCM} can be approximated by the following linear equation

$$t_{MCM} = k_0 + k_1 n \quad (5)$$

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. For example, if n is the number of SRAM chips in the i-cache or d-cache, one might arrange the chips in a $\sqrt{n/2} \times \sqrt{2n}$ rectangle as shown in Figure 7. If the CPU is placed in the middle of the long side of this rectangle, the maximum length of a wire from the CPU to any chip is $\bar{p}\sqrt{2n}$ where

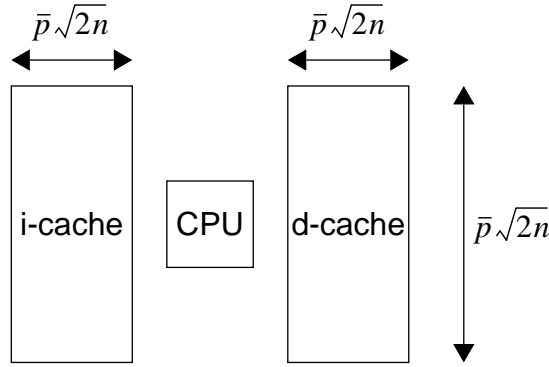


Figure 7: Instruction and data cache layout

The minimum delay arrangement of $2n$ cache SRAM chips.

horizontal and vertical pitches of the chip, including the width of adjacent wiring channels. The value of the linear coefficient can be expressed as

$$k_1 = Z_0 C_{bond} + 2\bar{p}^2 R_{MCM} C_{MCM} \quad (6)$$

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 [20]. The first term of (6) is the delay due to parasitic capacitance, C_{bond} , of the bonding method and the pad that connect 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 (6) assumes the interconnect is quite lossy and so neglects any delay from transmission line behavior. The value of k_1 calculated using (6) is within 1% of the value calculated using SPICE circuit analysis of actual layouts [9].

If the SRAMs have a size of S_{SRAM} bits each, then n can be replaced in the above equations by S/S_{SRAM} and equations (4) through (6) can be combined to produce the following expression for t_{CACHE} ,

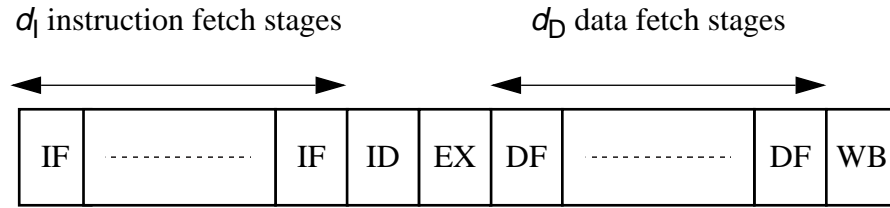


Figure 8: Pipeline stages for i- and d- caches. See Figure 1.

$$t_{CACHE} = t_{SRAM} + 2k_0 + \frac{2S}{S_{SRAM}} [Z_0 C_{cond} + 2\bar{p}^2 R_{MCM} C_{MCM}] \quad (7)$$

Equation (7) thus defines t_{CACHE} for an MCM-based direct-mapped cache as a function of its size S . In this case it is a linear function of cache size.

The macromodel for t_{CACHE} defined by (7) is, of course, specific to particular SRAM chips and MCM technology. In general, the cache may be on a PCB, MCM, or on the same chip as the CPU. More recently, functions for a variety of on-chip cache organizations have been developed [21] that could be used for a similar analysis of on-chip caches. Whatever the case, the important point from the multilevel optimization viewpoint is to develop a macromodel for t_{CACHE} as a function of cache size.

4.2 Tabulating $t_{CPU}(S, d)$

Figure 8 shows the multi-stage instruction execution pipeline of our example processor. This pipeline consists of d_I instruction fetch stages (the IFs), an instruction decode stage (ID), an execution stage (EX), d_D data fetch stages (the DFs), and a write back stage (WB). In the system there are three interlocking circular pipelines that could potentially set the minimum CPU cycle time. Two of these circular pipelines are shown in Figure 1. The top one corresponds to the IF and ID stages in Figure 8, and the bottom one corresponds to the EX and DF stages in Figure 8. The third circular pipeline is simply the EX stage and arises because of the need to recycle ALU results directly back into the ALU when data dependencies exist between two consecutive instructions. If we assume, for the moment, that both the i- and d-caches have

the same pipeline depth, d , then the minimum CPU cycle time t_{CPU} , will have to satisfy the following inequalities as a result of the constraints imposed by the pipelines,

$$t_{CPU} \geq \frac{t_{CACHE} + t_{ADDR}}{d + 1} \quad (8)$$

$$t_{CPU} \geq t_{ALU}$$

where t_{ADDR} is the time it takes to compute the cache address in the ID or EX stages of the pipeline, and t_{ALU} is the delay through the ALU (typically t_{ALU} and t_{ADDR} are equal). From (8) it follows that fixing d causes t_{CPU} to get longer as t_{CACHE} increases. In particular, larger caches which have longer access times result in longer CPU cycle times. Increasing the value of d by placing more latches along the cache access path to increase the number of pipeline stages makes it possible to reduce t_{CPU} .

To more accurately determine how t_{CPU} varies with cache size and pipeline depth we used the timing analyzer developed in [14] to estimate t_{CPU} for cache sizes of 1 to 32KW and cache pipeline depth values of 0 to 3. The value of t_{CACHE} is estimated using the delay model for MCM-based caches that was developed above. 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 $\frac{1}{d+1}$ for each unit increase in the cache access time, t_{CACHE} . 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 5. 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 data in Table 5 shows that for a pipeline depth of 0 the i- and d-caches limit t_{CPU} to more than 10 ns. Clearly, requiring the 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 depths of the i- and d-caches are increased to 3, the feedback loop around the ALU is critical for all cache sizes, and the cycle time is only 3.5 ns.

Cache size	Instruction cache				Data cache			
	d= 0	d= 1	d= 2	d= 3	d= 0	d= 1	d= 2	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 5: Optimal cycle times, $t_{CPU}(S, d)$, for i- and d-caches in nanoseconds

5 Calculating TPI

The cache experiments have been presented above in terms of i-cache or d-cache. In order to combine a particular i-cache organization and a d-cache organization we take the maximum t_{CPU} of each, as the new system cycle time t_{CPU} . These are combined with the results of Figure 3 through Figure 6 to give Figure 9 for the case when the i- and d-caches are equal. Note that the CPI values in Figure 3 through Figure 6 includes the effects of 4KW zero delay d- and i-caches respectively. Their effect was first subtracted before the combined CPI figure was calculated.

A number of conclusions can be drawn from Figure 9. It shows that when the primary cache is divided equally between instruction and data, performance is maximized when the number of branch delay slots is equal to the number of load delay slots, i.e., $d_I = d_D$. (This is not true when the i- and d-caches are of different sizes.) The reason for this is that pipelining the different sides of the 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., CPI will increase without reducing t_{CPU} . Figure 9 also shows that for every combination of load and branch delay slots, there is a

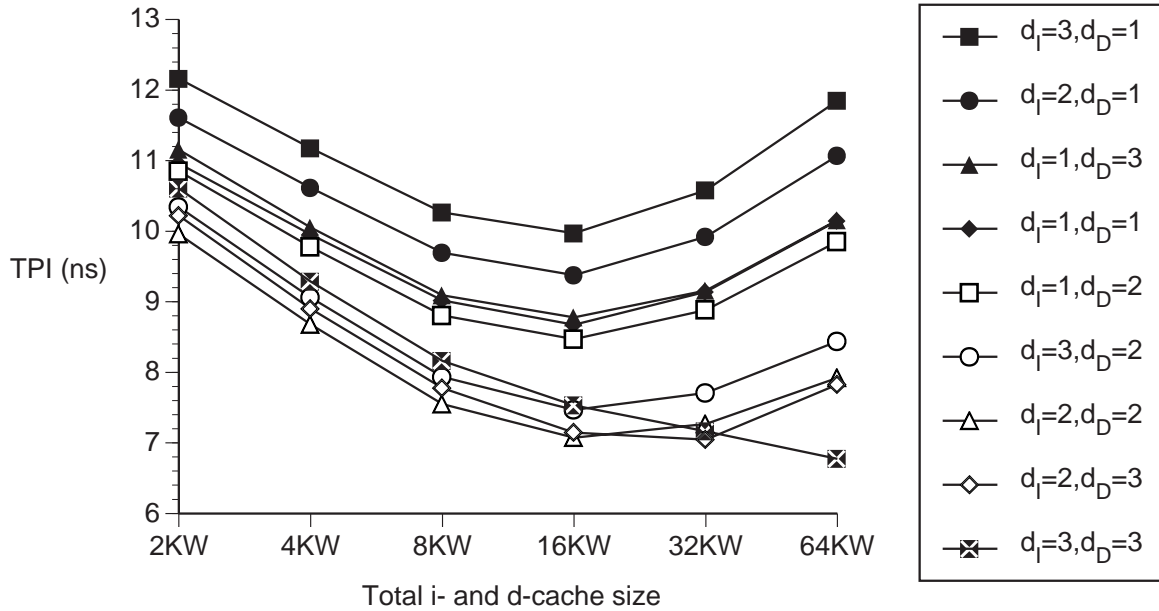


Figure 9: TPI for a system with pipelined caches

TPI versus total cache size for various instruction and data cache delay-slot combinations. The graph for $d_I = 3$, $d_D = 3$ reaches its minimum at 64KW; it starts to increase with larger caches.

cache size that maximizes performance. Maximum performance is reached for medium size caches at TPI = 6.8 ns, when $d_I = 3$, $d_D = 3$, $S = 64KW$, and $t_{CPU} = 3.5$ ns. However, a more economical design point of $d_I = 2$, $d_D = 2$, $S = 16KW$, and $t_{CPU} = 3.6$ ns, results in a value of TPI only marginally greater (7.0 ns) than that obtained with the 64KW cache. The difference is well within experimental variance and so this would appear to be a better estimate for the optimal point.

If dynamic out-of-order load execution were used instead of static load instruction scheduling, a new maximum TPI = 6.2 ns could be reached with the number of branch and load delay slots both equal to three and a combined cache size of 64KW. 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 implementation.

Different cache miss penalties change the performance and location of the optimal

design points: higher penalties require an increase in both the cache size and pipeline depth. Lower penalties have the opposite effect [9]. Increasing the number of branch delay slots increases CPI less than a comparable increase in load delay slots. Smaller refill penalties make it possible to take advantage of this fact by using a larger size i-cache than d-cache and pipelining the access of the i-cache more deeply.

6 Conclusions

This paper has studied the design of pipelined primary caches using a multilevel design optimization procedure. The objective of this optimization is to find the cache size and cache pipeline depth that maximizes system performance. The methodology fits naturally into the “early on” design studies that should be undertaken when commencing a design.

Trace driven simulation is a well-known technique. We have demonstrated that it is possible to accurately estimate the impact of cache size and cache pipeline depth on CPU cycle time using macromodels based on the implementation technology and timing analysis. Combining trace driven simulation with timing analysis enables us to provide the important link between processor architecture and implementation technology with processor performance. This is a link that has been recognized [22], but it is often ignored in the literature on processor architecture.

The multilevel optimization technique is illustrated with the design of a pipelined cache for a high clock rate MIPS-based architecture. The results of this design exercise show that because processors with pipelined caches can have shorter CPU cycle times and larger caches, a significant performance advantage is gained by using two or three pipeline stages to fetch data from the cache. 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.

We re-emphasize that the results are only optimal for the implementation technologies chosen for the design exercise; other choices could result in quite different optimal designs. The exercise is primarily to illustrate the steps in the design of pipelined caches using multi-

level optimization, however, it does exemplify the importance of pipelined caches if high clock rate processors are to achieve high performance, because pipelining the cache can make it feasible to improve CPU cycle time and CPI at the same time. Pipelining reduces or eliminates the dependence of the cycle time on cache access time. This makes it possible to use larger caches, which improve CPI, without affecting the cycle time. Such a situation suggests that the cache size versus set-associativity trade-off may need to be re-examined. If CPU cycle time is not dependent on the access time of a pipelined cache, then increasing the associativity of the cache to lower the miss ratio will provide a larger performance improvement.

References

- [1] A. J. Smith, "A comparative study of set associative memory mapping algorithms and their use for cache and main memory," *IEEE Trans. Software Engineering*, SE-4(2):121-130, 1978.
- [2] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. thesis, University of California, 1987.
- [3] S. A. Przybylski, *Cache and Memory Hierarchy Design*, 1990, San Mateo, California: Morgan Kaufman Publishers, Inc. 1990.
- [4] 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 Jour. Solid-State Circuits*, 16(5):763-767, May 1991.
- [5] O. A. Olukotun, T. N. Mudge, and R. B. Brown, "Performance optimization of pipelined primary caches," in *19th Annual Int. Symp. Computer Architecture*, Goldcoast, Australia, May 1992, pp. 181-190.
- [6] G. Kane and J. Heinrich, *MIPS RISC Architecture*, 1992, Englewood Cliffs, New Jersey: Prentice Hall. 1992.
- [7] R. Brown, M. Upton, A. Chandna, T. Huff, T. Mudge, and R. Oettel, "Gallium arsenide process evaluation based on a RISC microprocessor example," *IEEE Journal of Solid-State Circuits*, vol. 28, no. 10, Oct. 1993, pp. 1030-1037.
- [8] 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*, :1577-1585, 1991.
- [9] O. A. Olukotun, *Technology-Organization Trade-offs in the Architecture of a High Performance Processor*, Ph.D. thesis, The University of Michigan, 1991.
- [10] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach*, 1990, San Mateo, California: Morgan Kaufman Publishers, Inc. 1990.
- [11] O. A. Olukotun, T. N. Mudge, and R. B. Brown, "Implementing a cache for high-performance GaAs microprocessor," in *Proc. 18th Annual Int. Symp. Computer Architecture*, 1991.
- [12] M. D. Smith, *Tracing with Pixie*, Technical Report CSL-TR-91-497, Computer Systems Laboratory, November 1991.
- [13] *MIPS RISC Compiler Languages Programmer's Guide*, MIPS Computer Systems, Inc, December 1988.

-
- [14] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, "Check T_c and min T_c : Timing verification and optimal clocking of synchronous digital circuits," in *Proc. IEEE Conf. Computer-Aided Design*, Santa Clara, California, Nov. 1990, pp. 552–555.
- [15] A. I. Kayssi, *A Methodology for the Construction of Accurate Timing Macromodels for Digital Circuits*, Ph.D., The University of Michigan, 1993.
- [16] R. Brown, *et al.*, "Synthesis and verification of a GaAs microprocessor from a Verilog hardware description," in *Open Verilog International User Group Meeting*, 1992.
- [17] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annual Int. Symp. Computer Architecture*, 1981.
- [18] D. J. Lilja, "Reducing the branch penalty in pipelined processors," in *IEEE Computer Magazine*, 1988.
- [19] 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*, 1989.
- [20] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, 1990, Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.
- [21] T. Wada, S. Rajan, and S. A. Przybylski, "An analytical access time model for on-chip cache memories," *IEEE Journal of Solid-State Circuits*, 27(8):1147 - 1156, 1992.
- [22] J. L. Hennessy and N. P. Jouppi, "Computer technology and architecture: An evolving interaction," in *Computer*, pp. 18-29, 1991.