Evaluation of a "Stall" Cache: An Efficient Restricted On-chip Instruction Cache

Klaus Erik Schauser
Krste Asanović
David A. Patterson

Computer Science Division, EECS Department
University of California, Berkeley
Berkeley, CA 94720

Edward H. Frank

Sun Microsystems
Mountain View, California

Abstract

In this report we compare the cost and performance of a new kind of restricted instruction cache architecture — the stall cache — against several other conventional cache architectures. The stall cache minimizes the size of an on-chip instruction cache by caching only those instructions whose instruction fetch phase collides with the memory access phase of a preceding load or store instruction.

Many existing machines provide a single cycle external cache memory [6, 17, 2]. Our results show that, under this assumption, the stall cache always outperforms an equivalent sized on-chip instruction cache, reducing external memory access stalls by approximately 10%. In addition we present results for a system using an on-chip data cache, and for one with a double width data bus and short instruction prefetch buffer.

1 Introduction

RISC instruction sets are designed to facilitate pipelining, with many architectures aiming to issue at least one instruction per clock cycle [19, 20, 8, 18]. To sustain this instruction bandwidth, most RISC machines employ some form of instruction cache. Current device technology does not allow large on-chip caches, and so many designs employ a large external cache to achieve low miss-rates over a wide range of applications. Since instruction fetch rate limits processor execution rate, processor cycle time is often the same as the external cache cycle time. It is likely that this style of design will remain popular for device technologies where large on-chip caches are not feasible. In particular, device technologies such as GaAs offer high speed but relatively low integration density [23].

On most RISC architectures, the only instructions that access memory are loads and stores. Usually variables and intermediate results can be kept in registers, reducing the number of data accesses as compared to memory-memory or register-memory architectures. However, studies have shown that loads and stores still account for around 25–40% of all instructions executed [9].

If a processor is to avoid memory access stalls, the memory subsystem must be capable of delivering at least one instruction word every cycle while servicing data accesses. A CPU that has only a single bus to external cache memory might encounter a structural hazard and have to stall during the memory access phase of a load or store instruction. There are several ways to reduce the number of these load and store stalls. One solution is to have separate data and instruction busses; another possibility is to have caches on chip.

In this report we evaluate the cost and performance of a number of existing solutions for removing these stalls against a new proposal: the stall cache [7]. We first present the architectural alternatives, then develop a cost and performance model for each, and finally present experimental results for traces taken from programs in the SPEC benchmark suite.

2 Architectural Alternatives

We considered the following seven processor alternatives, each of which is connected to a single cycle external cache system:

1. No caching on chip with a single memory port.
2. No caching on chip with dual instruction and data ports.

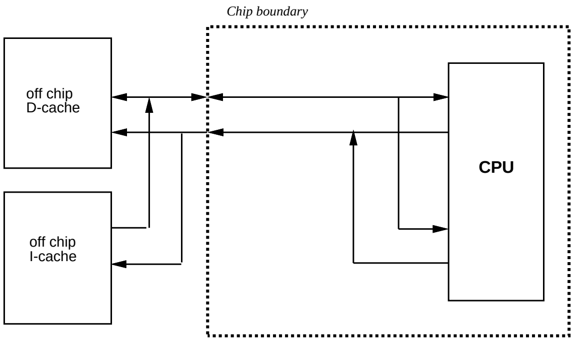


Figure 1: Unenhanced Processor

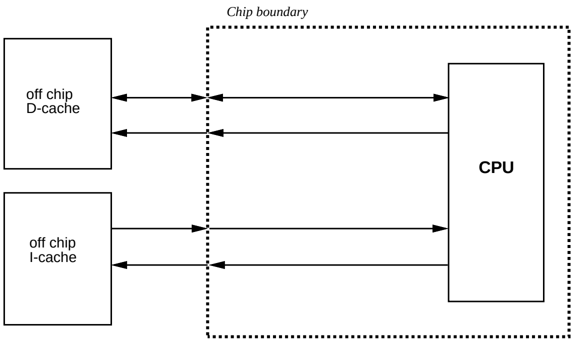


Figure 2: Harvard Architecture

3. Full instruction cache on chip with a single memory port.

4. Full data cache on chip with a single memory port.

5. Full instruction and data cache on chip with a single memory port.

6. Stall cache on chip with a single memory port.

7. Single double-width memory port with an on-chip instruction buffer.

The first alternative is shown in Figure 1. This unenhanced processor has a single external bus but no caches on chip, therefore every load or store causes a stall. We include separate off-chip data and instruction caches so that we will have consistent external cache miss rates for all options. The second alternative — the Harvard architecture (Figure 2) — has separate instruction and data busses, and hence never has a memory access conflict. Of all our alternatives, the first and second schemes represent the worst and best case performance respectively.

The next four alternatives add some form of caching on chip. An on-chip cache attempts to reduce the number of access conflicts by avoiding coincident use of the single external memory bus. However, memory access

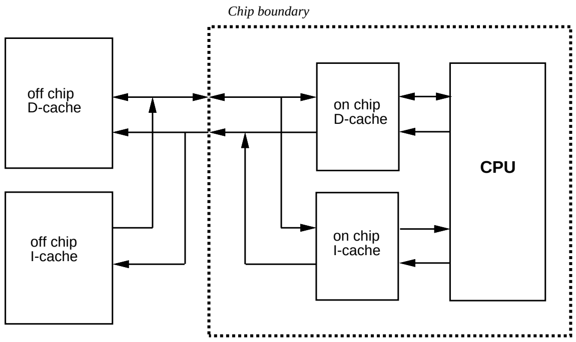


Figure 3: Single Bus Processor with Instruction and Data Cache

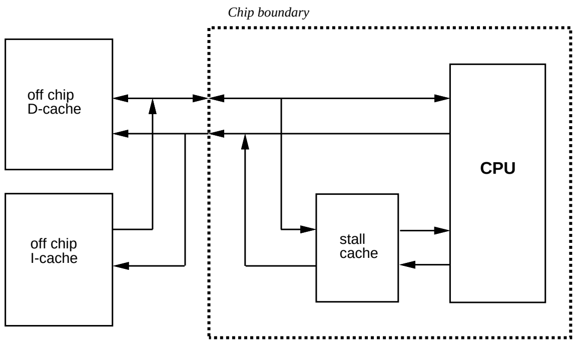


Figure 4: Stall Cache Architecture

conflicts can still occur on cache misses. The most general scheme, as shown in Figure 3, has both instruction and data caches on chip. Two further variants are to have only an on-chip instruction cache, or only an on-chip data cache.

Figure 4 shows the stall cache. This is a variant of the on-chip instruction cache with a single external memory port: Instead of providing a full instruction cache only those instructions whose instruction fetch phase coincides with the memory access phase of a load or store instruction are cached. For the majority of execution time, the instruction fetch unit can monopolize the memory port. When a load or store is encountered, the next instruction is fetched from the stall cache thereby freeing the memory port for the data access. If the next instruction is not in the stall cache, we add a stall cycle to the pipeline to fetch the missing instruction from external cache. Therefore the stall cache is a restricted instruction cache employing a similar philosophy as a branch target buffer [15], but for data access stalls instead of branch stalls.

The single double-width memory port, as shown in Figure 5 provides the bandwidth of two separate ports without the overhead of two sets of address lines [3]. We assume that instructions occupy only one word each, and hence that every external cache access can return

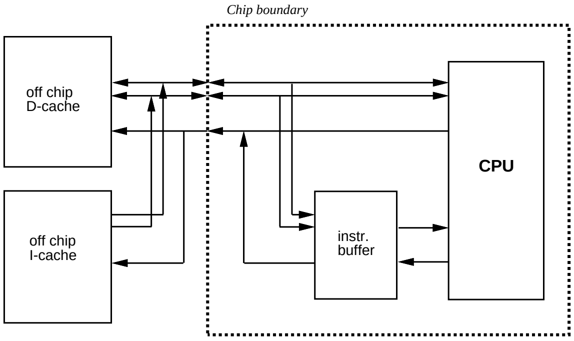


Figure 5: Double Width Bus Processor with Instruction Buffer

two instructions. The instruction prefetch buffer can fetch ahead of the current pair of instructions on those cycles where the memory port is not required for a data access, allowing execution to continue from the instruction buffer during a data access. A big advantage with this approach is that data accesses can make use of the extra data bus width to allow single cycle double precision loads and stores.

3 Cost and Performance Models

In this section we describe the cost and performance models we used to compare these architectures.

3.1 Cost

A complete CPU cost model would include chip area and packaging costs. A formula to compute the total cost for a CPU chip is

$$\text{Cost} = f(\text{Basic Area} + \text{Cache Area} + \text{Pad Area}) +$$

$$g(\text{Unenhanced pin count} + \text{Extra Pins})$$

Ideally, one would derive for each architecture the die area required for additional circuitry and for any extra bond pads. This area would be added to the die area required for the simplest system with a single memory port and no cache. For the caching schemes, die area is required for cache memory and control logic. The instruction buffer consumes die area for the buffer memory and the associated control logic. Each extra pin we add to the system will add at least the area of a bonding pad and signal buffers to the die area.

The additional pins required can be added to the pin count of the simplest chip to give a total pin count. A Harvard architecture requires pins for an additional address and data bus, hence nearly doubling the number of pins required. The instruction buffer requires double the number of data bus pins, or approximately 50% more pins in total than the unenhanced system.

In practice, it is difficult to relate cost of pins to cost of die area since this relationship is very technology dependent. In addition, pin count may already be determined by higher level system considerations. We chose to only directly compare the costs of the cache alternatives, where the increase in die area is roughly proportional to the gross cache capacity. We therefore use a simplified cost model that takes into account only the total storage requirement for a cache including tag and valid bits.

3.2 Performance

We measure performance by counting the total number of clock cycles taken to execute our benchmark programs, as given by the expression

$$Cycles = I\text{-}count + Stall\ cycles$$

where *I-count* is the dynamic instruction count. Sources of stall cycles include load delays, branch delays, and floating point stalls in addition to memory system stalls.

All our results are for the MIPS R3000 [13]. Compilers for this architecture must fill unused load and branch delay slots with `NOP` instructions since there are no hardware interlocks. Thus for the MIPS architecture the instruction count includes these delay stall cycles.

The number of floating point stalls and external cache miss stalls will be common to all of our alternatives, and will dilute the benefit of any scheme to remove memory access stalls. Since the sources of these stalls are very implementation dependent, we have chosen to present our results without including these stalls. Hence we assume all MIPS instructions take one clock cycle. We later present figures for some typical system configurations that indicate the impact of these stalls on the speedups in our results.

The *Harvard* architecture has no memory access stalls, hence run time is given by

$$Cycles_{Harvard} = I\text{-}count$$

The *unenhanced* system has no on-chip caches and a single port to memory. Every load or store instruction generates a memory stall giving a total run time

$$Cycles_{Unenhanced} = I\text{-}count + Loads + Stores$$

For the system with a *full I-cache* on chip, we note that we can fetch in parallel from external memory whenever the instruction currently in the memory access phase of the pipeline is not a load or store. Hence we only incur a penalty if an I-cache miss coincides with a data access. The resulting run time is

$$Cycles_{I\text{-}Cache} = I\text{-}count + Coincident\ misses$$

A *write-back D-cache* on chip with write-allocate only needs to use the external memory bus if there is a load miss, in which case we need to stall in order to fetch the missing word, and whenever we need to copy back dirty words either on a load miss or store miss. There is no I-cache so instruction fetches always use the external bus. Here the total run time is

$$Cycles_{D\text{-}Cache_{wb}} = I\text{-}count + Load\ misses + Copybacks$$

An on-chip *write-through D-cache* needs to use the external bus on every store and on load misses resulting in a total run time

$$Cycle_{D\text{-}Cache_{wt}} = I\text{-}count + Load\ misses + Stores$$

We could consider implementing a system that has both an instruction cache and a data cache on chip. When the instruction in the memory access phase of the pipeline is not a load/store, the instruction fetch unit can access the external cache, avoiding stalls due to misses of the on-chip instruction cache. When the instruction in the memory access phase of the pipeline is a load/store, either an instruction fetch or a data access could use the port to the external cache. If the instruction fetch unit uses the port to the external cache, then we must stall whenever the on-chip data cache misses and needs to transfer data. Alternatively, if we allow the data access to use the port, a stall occurs whenever the on-chip instruction cache misses. The number of stalls is the same as if we had only a data cache or instruction cache respectively. We therefore discount the case of having both caches on chip.

The *stall cache* architecture incurs a penalty whenever there is a miss in the stall cache. The stall cache is only accessed during the memory access phase of a load or store, and must then always wait a cycle before fetching a missing instruction.

$$Cycles_{Stall\ Cache} = I\text{-}count + Stall\ Cache\ misses$$

The *instruction buffer* fetches two instructions at a time over the double width bus whenever the bus is free, providing there is space in the buffer. The buffer is flushed on taken branches. Stall cycles only occur if the buffer is empty when a load or store instruction requires a memory access. The execution time is

$$Cycles_{I\text{-}buffer} = I\text{-}count + Buffer\ stalls$$

4 Methodology

We used trace driven simulation to evaluate the different configurations [21].

We took our traces from the SPEC benchmark suite running on a MIPS R3000 [4, 5]. The SPEC benchmark suite consists of ten different programs, 4 written in C and 6 in FORTRAN. The number of references generated by execution of these programs reach into the tens of billions on the MIPS R3000. We wanted to include all the different programs in the suite to cover a wide range of program behaviors. We also needed to simulate hundreds of different cache configurations.

The conventional approach for trace driven simulations would be to generate the entire address trace once for each benchmark and then use the trace as input to many simulations of different cache configurations. However, it is not practical to store the entire trace for programs in the SPEC benchmark suite since the number of references is far too large. Even if we could store traces of this size, or generate them on the fly, we would still have been limited by the speed of our cache simulators given that we needed to perform thousands of cache simulations.

Our solution was to adopt a sampling technique similar to Laha *et al* [14] to generate a shorter — and hence storable — trace. At evenly spaced intervals over the entire execution of the benchmark we take samples of the address trace. The results in [14] show that as few as 35 samples can give accurate estimates of both the mean value and the distribution of the miss ratio. The advantage of this scheme is that with only a small fraction of the total address trace the original program behavior can be reproduced. In [1] and [24] alternative schemes are presented for trace compaction and efficient trace-driven simulation. We chose the simpler approach because it was satisfactory for our purposes, saving both disk space and cache simulation time.

For the results in this report we took 50 samples for each benchmark with 200,000 instructions per sample, for a total of 10,000,000 instructions per benchmark program. We chose the sample length to be approximately the duration of a Unix time slice and each sample was preceded by a cache flush to simulate multiprogramming. The caches under study are small, and their contents are unlikely to survive context switches. This sampling represents a compression by a factor of between 100-2000, with a corresponding reduction in simulation time. Each simulation still required around 25 minutes of CPU time on a Sun Sparcstation-1+, and the results in this report represent over 1600 such simulations.

This scheme worked well for all the benchmark programs except `tomcatv`. This is a very small FORTRAN program, exhibiting strong periodicity that unfortunately correlated with our sampling frequency. In [14] this problem is mentioned and sampling at random intervals is recommended for these cases. We repeated the sampling of `tomcatv` using 37 equally spaced samples, and this simple alternative approach gave good results. The `espresso` benchmark is the only SPEC program to require multiple runs using different inputs. To simplify our sample gathering, we selected only the longest running of the four input files, `tial`. We include results for 8 out of the 10 SPEC benchmarks; we did not manage to obtain traces for `gcc` and `fppp`. Note that for this study, we are not overly concerned with determining the absolute performance of our alternative architectures on the SPEC benchmarks. We merely require a representative sample set with which to make comparisons between the alternatives.

We compiled the benchmark programs using version 2.11 of the MIPS `cc` and `FORTRAN` compilers for a MIPS M/2000 running RISC/os 4.50. We used `pixie` to instrument the object code to generate the instruction and data references [16]. We ran each instrumented benchmark program twice; once to count the total number of instructions executed, and again to take the evenly spaced samples. We then used this stored sequence of samples as input to our cache simulations.

The main simulation tool used was the `dinero` cache simulator [10, 11]. We had to make a number of modifications to `dinero` to record coincident instruction and data misses. We also wrote an instruction buffer simulator that reads `dinero` format traces.

5 Results

In Section 5.1 we present dynamic memory reference statistics for the individual benchmarks. Section 5.2 discusses our selection of cache configurations, and Section 5.3 our speedup calculations. Section 5.4 presents result curves obtained by combining the individual benchmark results, and Section 5.5 considers the benchmark results individually.

5.1 Memory Reference Statistics

Table 1 presents dynamic memory reference statistics for the benchmark programs. For each program the upper line gives figures for a complete execution, the lower for the sampled trace. The columns in the table reading left to right contain: number of instructions, percentage of data references, percentage loads, percentage stores, and load/store ratio. The percentages are relative to the total number of instructions.

The instruction count for a complete benchmark run varies from a little over 1.1 billion instructions for `espresso` up to nearly 22 billion instructions for `spice2g6`. The samples always total 10 million instruction references, except for `tomcatv` where we took 7.4 million. Data references account for between 17% and 51% of all instructions executed. The C programs in this benchmark suite tend to have fewer data accesses than the FORTRAN programs. Overall the ratio of loads to stores is approximately 2.8:1.

This table gives an indication that our sampling technique has managed to capture the overall behavior of these benchmark programs. The relative ratios of types of data access agree to within 1% except for `tomcatv` with 50 samples. We used the 37 sample trace for our simulations of `tomcatv`.

5.2 Cache Configurations

We focused on smaller caches as these are more relevant to designs for which a stall cache might be considered. We simulated cache sizes from 64 to 64K bytes. Since external memory accesses are single cycle, the cache miss penalty will be the same as the cache fill time for single word blocks. Fetching other than individual words on demand will cause a drop in performance. Hence we restricted ourselves to single word blocks, or larger blocks with single word sub-block placement. We investigated the effect of changing block size from 4 to 32 bytes on direct mapped caches.

To assess the effects of changing associativity we varied associativity from direct mapped to 8-way for caches with single word blocks. Previous studies have shown little performance improvement for degrees of associativity greater than 8-way [21].

5.3 Speedup Calculation

We measure performance as the speedup of a configuration relative to the time required for the unenhanced processor

$$Speedup_{Configuration} = \frac{\text{Cycles}_{Unenhanced}}{\text{Cycles}_{Configuration}}$$

This speedup should be reduced to take into account floating point stalls and external cache misses. Our simulations show that a system using the MIPS R2010 FPU would add between 10–15% floating point stall cycles to the basic instruction count for the FORTRAN programs, in addition to 10% stalls from other sources. We simulated a direct mapped external cache consisting of 64Kbytes of instruction cache and 64Kbytes of data cache, with 32-byte blocks. For the C programs the miss ratio was less than 1%, with a 15-cycle cache miss penalty we need to add around 15% stall cycles to the basic instruction count. For the FORTRAN programs the miss ratio was between 3-5%, with a 15-cycle cache miss penalty we need to add around 60% stall cycles to the basic instruction count. These stalls will add the same number of cycles to the execution times of each of our architectural alternatives. This will reduce the maximum obtainable speedup, and reduce the relative advantage of those schemes that are more successful at eliminating memory access stalls. We have not included the effect of these stalls in our results, since they are extremely implementation dependent.

Benchmark	Instructions	Data	Loads	Stores	L/S Ratio
`espresso`	1,160,836,913	22.04 %	17.62 %	4.42 %	3.98
	10,000,000	21.84 %	17.42 %	4.42 %	3.94
`spice2g6`	21,967,654,691	29.11 %	24.14 %	4.97 %	4.86
	10,000,000	30.04 %	24.63 %	5.41 %	4.55
`doduc`	1,645,045,552	38.19 %	28.21 %	9.98 %	2.83
	10,000,000	38.35 %	28.28 %	10.07 %	2.81
`nasa7`	9,261,864,986	50.97 %	36.83 %	14.15 %	2.60
	10,000,000	50.03 %	36.35 %	13.68 %	2.66
`li`	6,036,802,819	33.86 %	22.00 %	11.86 %	1.85
	10,000,000	33.84 %	21.92 %	11.92 %	1.84
`eqntott`	1,248,622,981	17.44 %	16.50 %	0.94 %	17.55
	10,000,000	17.48 %	16.28 %	1.20 %	13.57
`matrix300`	2,775,939,665	47.11 %	31.39 %	15.72 %	2.00
	10,000,000	47.75 %	30.78 %	16.97 %	1.81
`tomcatv`	1,812,801,534	49.59 %	36.64 %	12.95 %	2.83
50 samples	10,000,000	70.94 %	47.95 %	23.00 %	2.08
37 samples	7,400,000	50.49 %	37.07 %	13.42 %	2.76
arith. average		36.04 %	26.67 %	9.37 %	2.85
		36.23 %	26.59 %	9.64 %	2.76

Table 1: Dynamic Memory References

5.4 Combined Results

The results for the SPEC benchmarks are summarized in Figure 6 for direct mapped caches with a block size of one word. This figure plots performance versus cost for each of our alternative configurations. The overall speedup is computed as the geometric average over the benchmark set. The cost represents the gross cache capacity for the caching alternatives.

The cost of the Harvard architecture is constant. Its speedup is the maximum possible for any of our alternatives since there are no memory access stalls. Rather than plotting a single cost/performance point for this architecture we ignore cost and show its performance as a horizontal line. The Harvard architecture achieves a speedup of 1.36 over the unenhanced architecture.

The main aim of this study was to compare the stall cache with a full instruction cache. If we consider horizontal lines of constant speedup on this graph, we see that a full instruction cache architecture requires roughly 1.5–2 times the capacity of a stall cache architecture to give the same speed-up. This shows how restricting the entries stored in a cache, as is done in the stall cache, can substantially reduce the required capacity. If we consider vertical lines of constant cost, we see that a stall cache organization can add 3–5% to the speedup achieved by a conventional instruction cache organization with the same area. Note that stall detection logic has to be present in both cases. A 512 byte stall cache (gross cost nearly 1 Kbyte) achieves an average speedup of 1.26 over the unenhanced processor. In practice this could eliminate nearly 75% of the stalls due to loads and stores across a single port.

We simulated the same cache configurations for data caches, using a write-through cache. As can be seen on the graph, its average performance is much worse than any of the other architectural alternatives. Even a 64K byte data cache achieves less than 50% of the potential increase in speedup. Further disadvantages are that data caches are more complex than instruction caches, given that they must handle writes as well as reads.

The instruction buffer achieves impressive performance, attaining a maximum speedup of 1.28. This represents nearly 80% of the potential increase in speedup. The MIPS R3000 instruction set has no double word load and store instructions so we couldn't evaluate the extra speed-up possible due to the expanded data bus width. For this graph we have shown the cost of the instruction buffer as solely that of the buffer storage required. In reality, the cost of the instruction buffer is substantially higher due to technology dependent pin and packaging costs. We simulated buffer lengths ranging from 4–512 bytes, but the results show that increasing buffer length beyond 32 bytes results in only slight improvement. This is because the buffer must be flushed on taken branches, limiting the effectiveness of larger buffers. Our findings are consistent with those for the ECL SPARC [3]. Small instruction buffers can give the same performance improvement as a 1 Kbyte instruction cache.

We were interested in determining the performance/cost of larger block sizes for direct mapped caches with single word sub-block placement. Larger blocks have the advantage of reducing tag storage requirements but will increase the miss rate, as on a miss a whole block may be invalidated. Since we are using single word sub-block placement the miss penalty is still

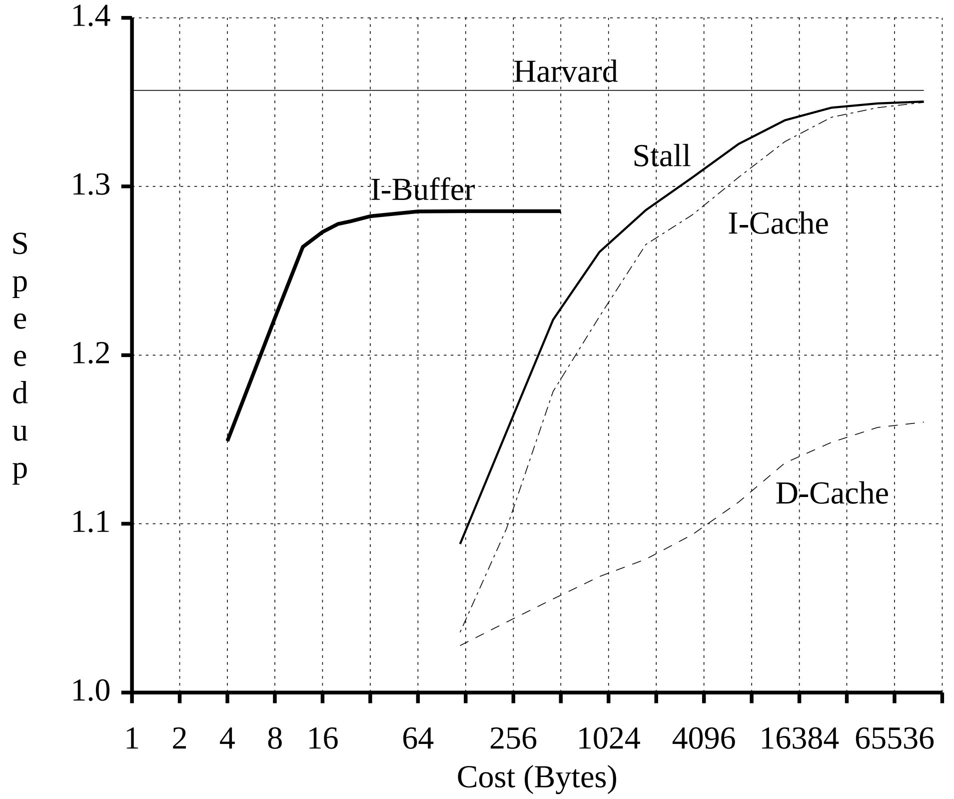


Figure 6: **Geometric Average of Speedup vs. Gross Cost for SPEC benchmarks.** The cache alternatives are direct mapped with 4 byte blocks. The gross cost includes data and tag storage.

only one cycle, but we require an additional valid bit per word. Increasing block size affected our alternatives in different ways. Figure 7 plots speedup against gross cache cost, including tag and valid bits.

For the stall cache we see that increasing block size is not beneficial to performance/cost. This is explained by noting that on average we expect only every third instruction to be cached, so larger blocks cause a serious drop in performance that is not compensated by the reduction in tag storage. For the instruction cache we see that using larger blocks improves its performance/cost ratio substantially. Performance/cost increases as we change the block size from 4 to 8, and from 8 to 16 bytes, but drops off again as we move from 16 to 32 byte blocks. Here, the drop in performance due to larger block size is small compared to the cost savings from reducing tag storage. However, even the best instruction cache configuration is worse than the worst stall cache configuration. For the data cache, we see that larger block sizes lower performance. This is explained by the poor spatial locality of the data reference streams.

Increasing associativity can improve cache performance, but adds to cost by increasing tag area and complicating control logic. In [12] Hill shows how the added logic may slow cache cycle time and negate the benefits of increasing associativity, here we did not attempt to account for these effects. In Figure 8 we present the speedup for direct-mapped, 2-way, 4-way, and 8-way associative caches with a block size of one word for each of the cache architectures. The replacement policy was LRU.

Increasing associativity improves performance/cost for stall caches larger than 512 bytes, with negligible effect on smaller caches. The performance/cost of the instruction cache decreases with increasing associativity for small caches. We believe this is due to the same reasons as cited in [22]. A high degree of associativity combined with an LRU replacement policy can cause a higher miss rate in instruction loops that exceed the cache capacity. The data cache shows the greatest improvement with increased associativity, but is still much worse than the other alternatives.

5.5 Individual Results

We examined the results for the benchmarks individually, as shown in Figures 9–32, and found that the stall cache was consistently better than the instruction cache.

The three benchmarks `espresso`, `spice2g6`, and `li` exhibit overall behavior similar to the geometric aver-

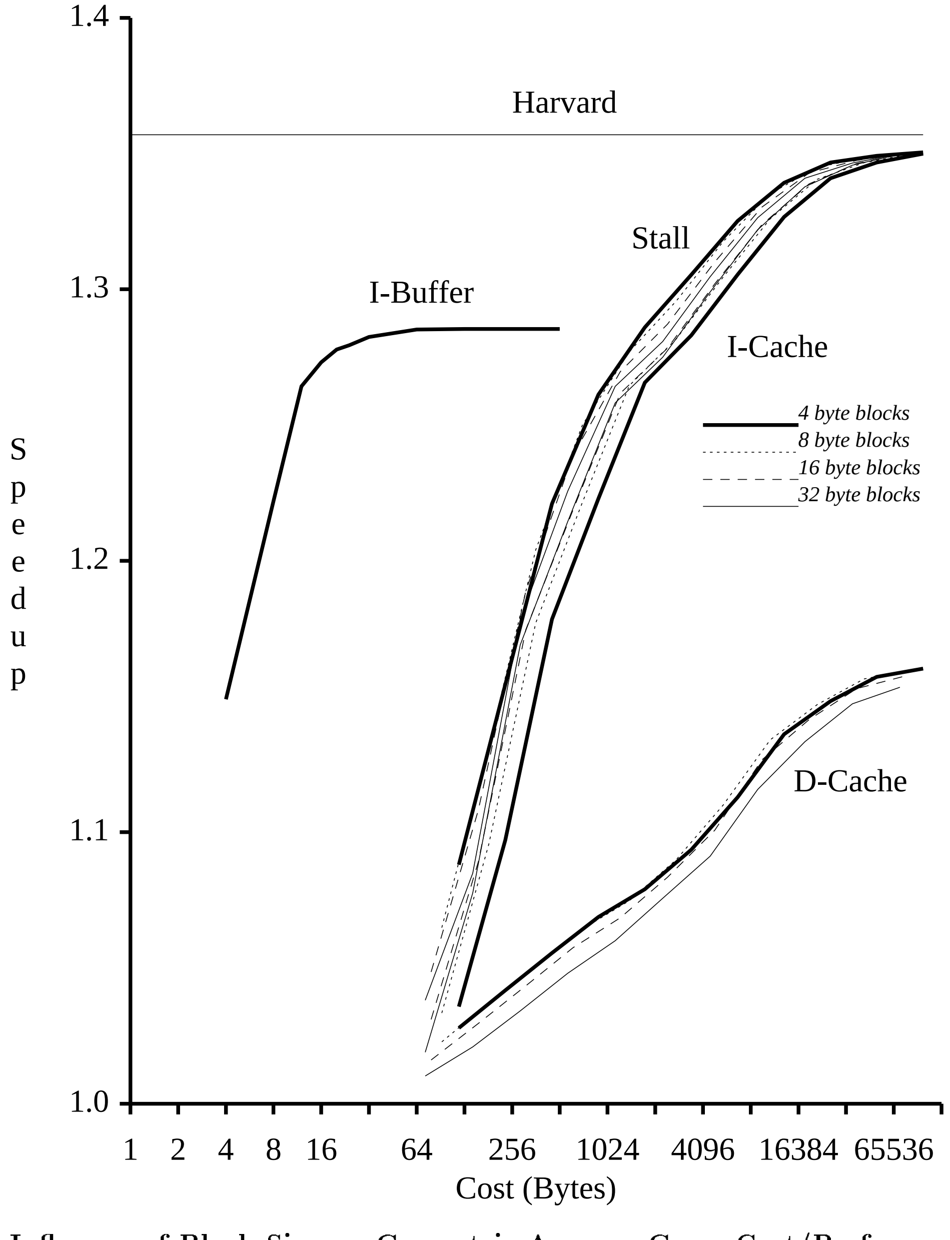


Figure 7: **Influence of Block Size on Geometric Average Gross Cost/Performance.** All cache alternatives are direct mapped; for each we give curves for 4, 8, 16, and 32 byte blocks with single word sub-block placement. The gross cost includes data and tag storage.

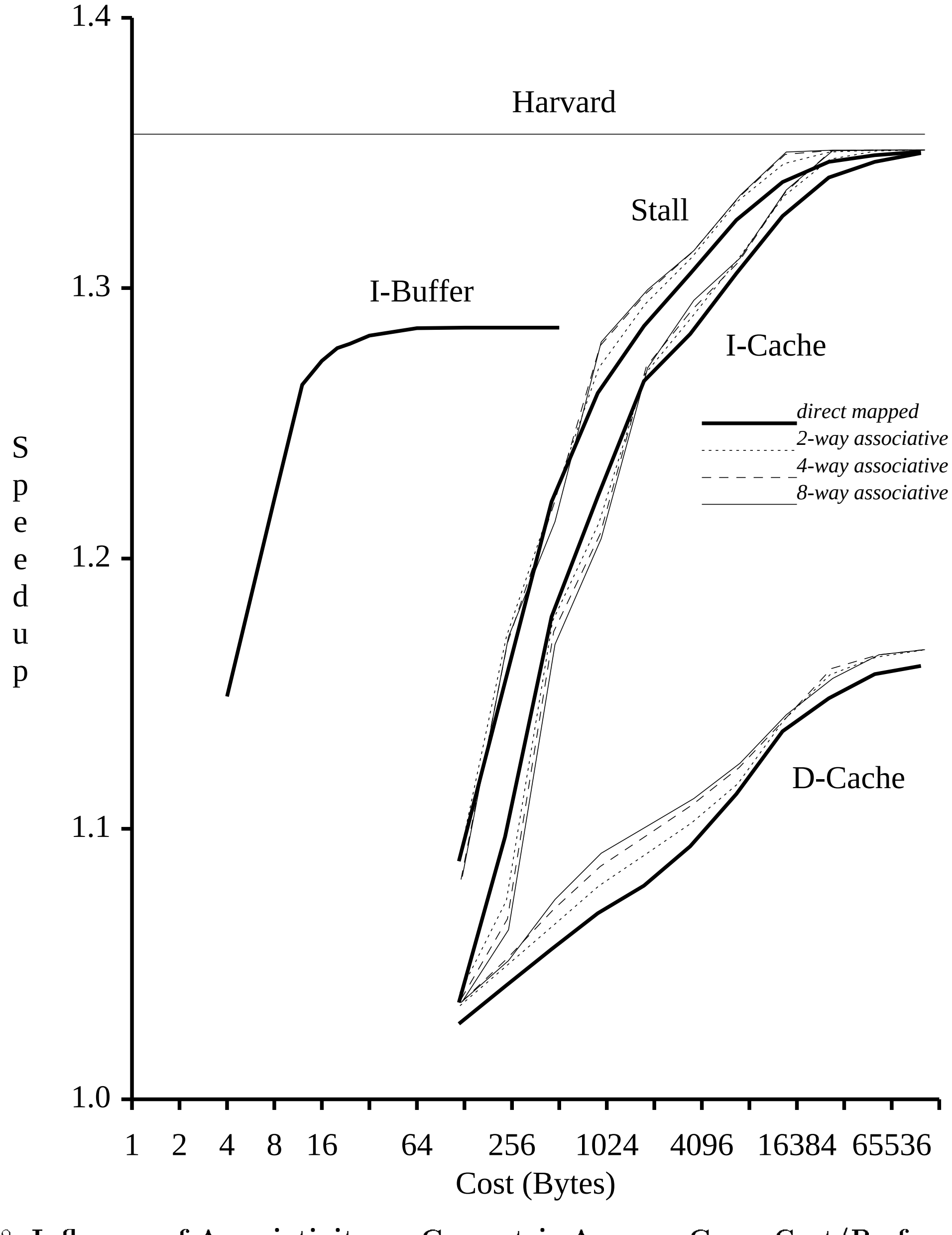


Figure 8: **Influence of Associativity on Geometric Average Gross Cost/Performance**. For each cache alternative we give curves for direct mapped, 2-way, 4-way, and 8-way associativity; all have single word blocks. The gross cost includes data and tag storage.

age. For `espresso`, the percentage of data references is small and the instruction buffer performs well. For `spice2g6`, there is a more gradual increase in performance with increasing instruction and data cache sizes indicating larger working sets.

`Eqntott` spends much of its time performing quicksort. Consequently, even small instruction buffers and caches perform well. For very small cache sizes, the stall cache has a pronounced advantage over the full instruction cache.

`Doduc` is the only benchmark where the data cache outperforms both stall and instruction caches. This only occurs for caches smaller than 4Kbytes. In two other benchmarks, `li` and `espresso`, a small data cache also outperforms small instruction caches, but the effect is strongest in `doduc`.

Three of the FORTRAN programs, `nasa7`, `matrix300`, and `tomcatv`, exhibit very similar behavior. These are all scientific codes, with small loops operating on large data arrays. As can be expected, small data caches perform very poorly. However, small instruction or stall caches catch most of the dynamic instruction references. The instruction buffer performs significantly worse than the instruction caches. The buffer attempts to exploit unused external memory access cycles to maintain a full instruction queue, but these three benchmarks have the highest data memory traffic of those in this study. The small kernels in these codes also require frequent flushes of the buffer contents.

6 Conclusion and Future Work

For CPUs with a single port to an external single cycle cache system, a stall cache is more cost effective than an on-chip instruction cache or an on-chip data cache. For the same speedup, an on-chip instruction cache needs approximately 1.5–2 times the capacity of the stall cache. A 512 byte stall cache can eliminate over 75% of the memory access stalls caused by fetching both instructions and data over a single external memory port. Stall caches larger than 4 Kbytes can eliminate over 90% of these stalls, and hence achieve a speedup close to that of a Harvard architecture.

An on-chip data cache always performs much worse than the other alternatives. The instruction buffer performs well in architectures where instruction fetch bandwidth is less than memory port bandwidth. A 32 byte buffer eliminates up to 75% of the memory access stalls, larger buffers show little additional improvement.

This study has shown that with intelligent cache design nearly the same performance as a full Harvard architecture can be achieved without the extra pins and pads. This is important for current high-speed, low-integration designs. It is also very important for future systems, since moving to 64-bit processors could make a Harvard architecture much more expensive.

Further work would be to evaluate the possible speedup on architectures that have register windows such as SPARC. Register windows can reduce the total number of loads and stores, at least for integer programs. This reduction would further increase the efficiency of the stall cache.

The stall cache can be further restricted by taking advantage of other sources of stalls. For example, on architectures such as MIPS II and SPARC that have hardware interlocks, there is no need to cache instructions whose instruction fetch phase coincides with the memory access phase of a load that causes a load delay stall. The resulting delay cycle frees the external cache port for an instruction access. Similarly, it is possible to exploit floating point stalls.

7 Acknowledgements

We thank Mark Hill for taking time to discuss this work with us. John Mashey and Earl Killian of MIPS promptly answered several of our questions regarding `pixie` and cache simulation of the SPEC programs. Rafael H. Saavedra-Barrera helped us to compile the FORTRAN programs and provided many useful comments on an earlier draft of the report. Krste Asanović is supported by the International Computer Science Institute. Klaus Erik Schauser is supported by D. Culler's PYI Award (CCR-9058342). Computational resources were provided, in part, under NSF Infrastructure Grant CDA-8722788.

References

[1] A. Agarwal and M. Huffman. Blocking: Exploiting Spatial Locality for Trace Compaction. In *Conference on Measurement and Modeling of Computer Systems*, pages 48–57. ACM SIGMETRIC, May 1990.

[2] A. V. Bechtolsheim and E. H. Frank. Sun's SPARCstation 1: A Workstation for the 1990s. In *COMPCON*, pages 184–188, February 1990.

[3] E. W. Brown, A. Agrawal, T. Creary, M. F. Klein, D. Murata, and J. Petolino. Implementing SPARC in ECL. *MICRO*, 10(2):10–22, February 1990.

[4] Systems Performance Evaluation Cooperative. SPEC Benchmark Suite Release 1.0. *SPEC Newsletter*, 1(1), Fall 1989.

[5] K. Dixit and V. Metha. Analyzing Performance using SPEC Release 1.0 Benchmarks. *SPEC Newsletter*, 1(2), Spring 1990.

[6] M. Forsyth, S. Mangelsdorf, E. DeLano, C. Gleason, J. Yetter, and D. Steiss. CMOS PA-RISC Processor for a New Family of Workstations. In *COMPCON*, pages 202–207, February 1991.

[7] E. H. Frank and M. Namjoo. Apparatus and Method for Providing a Stall Cache — Patent Disclosure. Technical report, SUN Microsystems, 1989.

[8] J. L. Hennessy. VLSI Processor Architecture. *IEEE Trans. on Computers*, C-33(11):1221–1246, December 1984.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantative Approach.* Morgan Kaufmann, 1990.

[10] M. D. Hill. *DineroIII Documentation, Unpublished Unix-style Man Pages.* Computer Science Division (EECS), University of California, Berkeley, October 1985.

[11] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, Computer Science Division (EECS), University of California, Berkeley, November 1987. Available as Tech. Report No. UCB/CSD 87/381.

[12] M. D. Hill. A Case for Direct Mapped Caches. *Computer*, 21(12):25–40, December 1988.

[13] G. Kane. *MIPS RISC Architecture (R2000/R3000).* Prentice Hall, 1989.

[14] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. on Computers*, 37(11):1325–1336, November 1988.

[15] J. K. F. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *Computer*, 17(1):6–22, January 1984.

[16] MIPS. `pixie` documentation. Unpublished Unix-style man pages.

[17] M. J. K Nielsen. DECstation 5000 Model 200. In *COMPCON*, pages 220–225, February 1991.

[18] D. A. Patterson. Reduced Instruction Set Computers. *Communications of the Association for Computing Machinery*, 28(1):8–21, Januar 1985.

[19] D. A. Patterson and D. R. Ditzel. The Case for the Reduced Instruction Set Computer. *Computer Architecture News*, 8(6):25–33, October 1980.

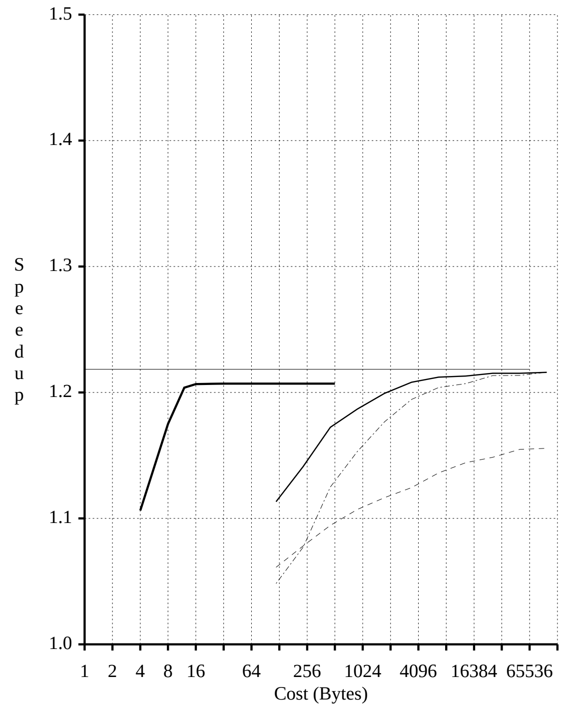
[20] G. Radin. The 801 Minicomputer. In *Proc. Symposium Architectural Support for Programming Languages and Operating Systems*, pages 39–47, March 1982.

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

[22] J. E. Smith and J. R. Goodman. A Study of Instruction Cache Organizations and Replacement Policies. In *Proc. Tenth Symposium on Computer Architecture*, pages 132–137, June 1983.

[23] H. Vlahos and V. Milutinović. GaAs Microprocessors and Digital Systems. *IEEE Micro*, pages 28–56, February 1988.

[24] W. Wang and J. Baer. Efficient Trace-Driven Simulation Methods for Cache Performance Analysis. In *Conference on Measurement and Modeling of Computer Systems*, pages 27–36. ACM SIGMETRIC, May 1990.



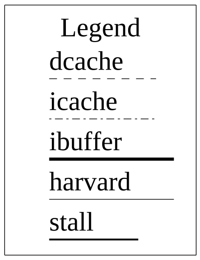


Figure 9: **Speedup vs. Gross Cost for `espresso`.** All cache alternatives are direct mapped with 4 byte blocks.

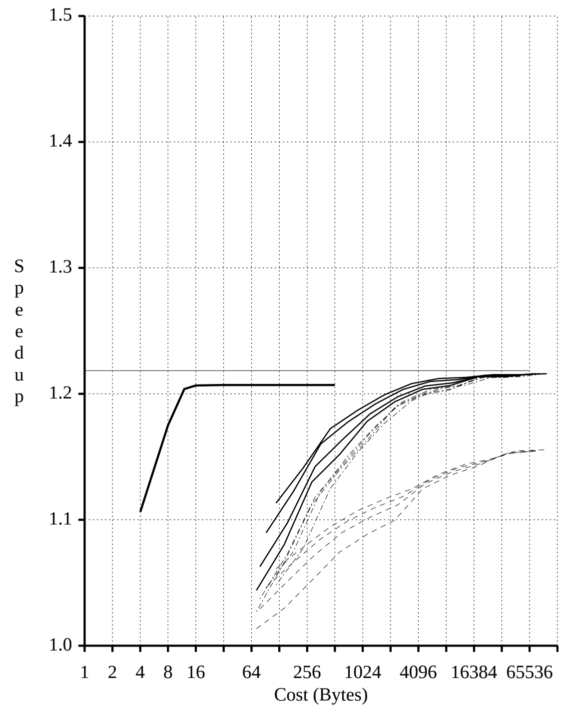


Figure 10: **Influence of Block Size on Speedup vs. Gross Cost for `espresso`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

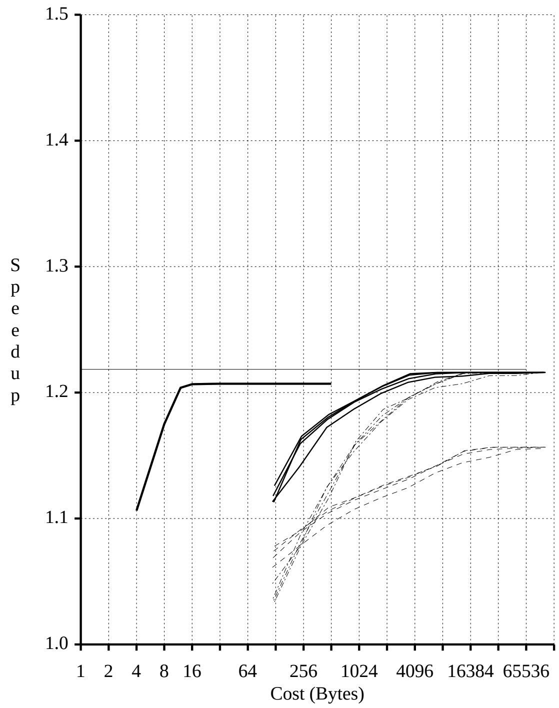
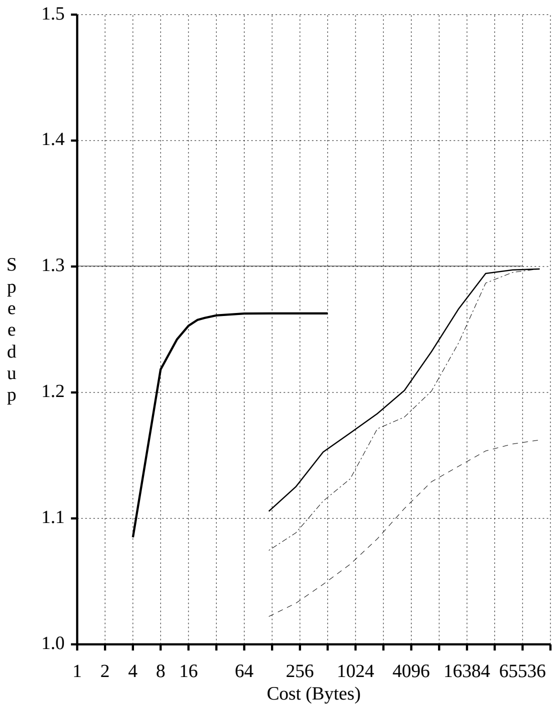


Figure 11: **Influence of Associativity on Speedup vs. Gross Cost for `espresso`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.



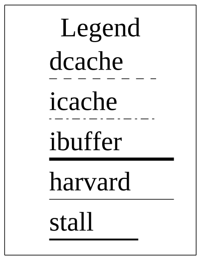


Figure 12: **Speedup vs. Gross Cost for `spice2g6`.** All cache alternatives are direct mapped with 4 byte blocks.

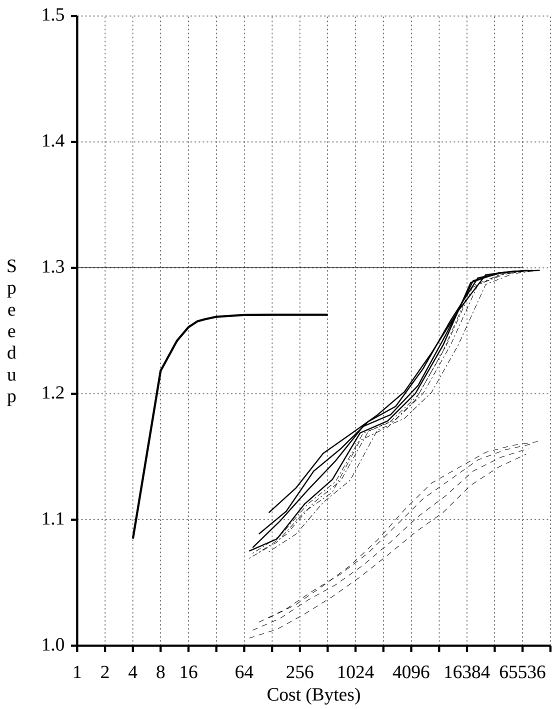


Figure 13: **Influence of Block Size on Speedup vs. Gross Cost for `spice2g6`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

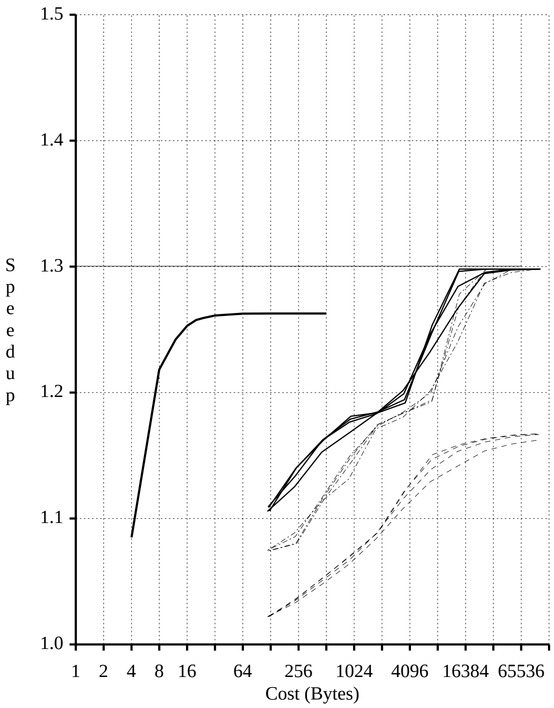
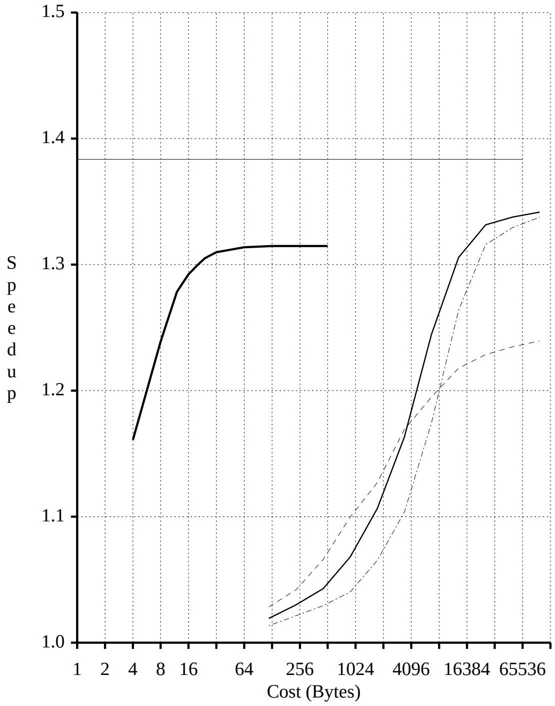


Figure 14: **Influence of Associativity on Speedup vs. Gross Cost for `spice2g6`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.



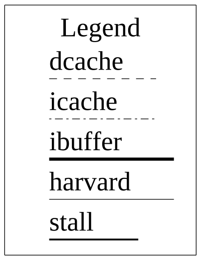


Figure 15: **Speedup vs. Gross Cost for `doduc`.** All cache alternatives are direct mapped with 4 byte blocks.

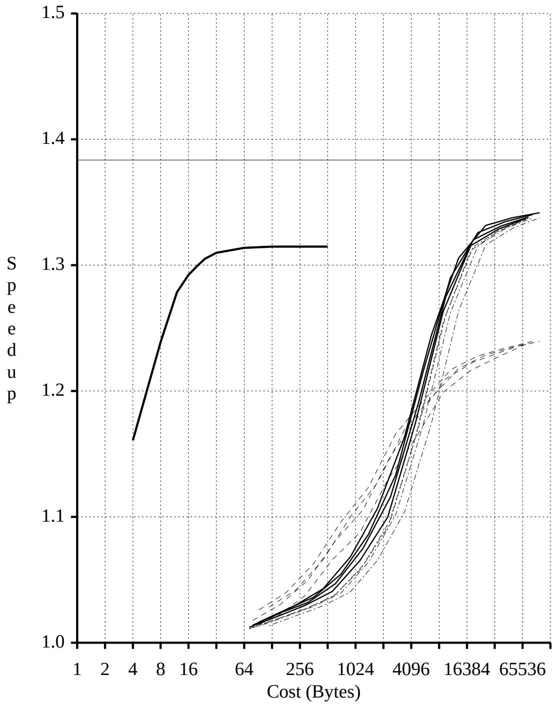


Figure 16: **Influence of Block Size on Speedup vs. Gross Cost for `doduc`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

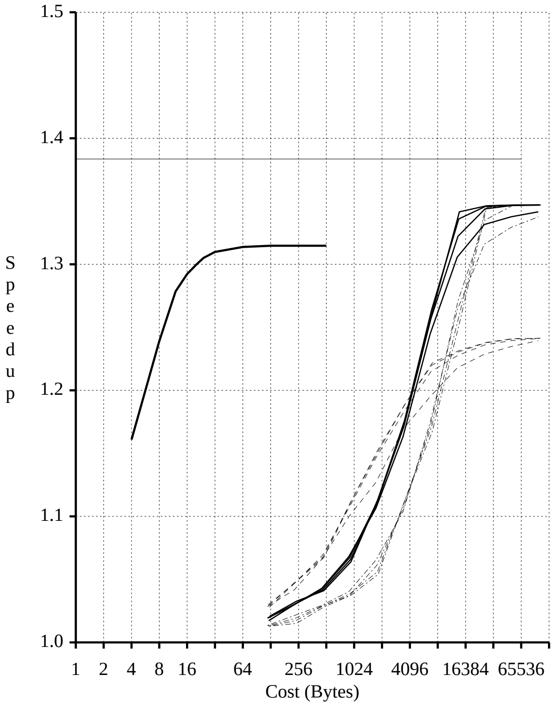
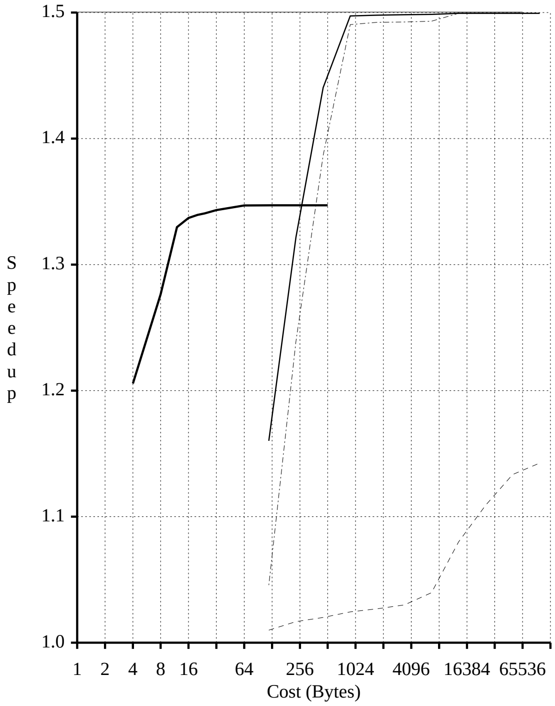


Figure 17: **Influence of Associativity on Speedup vs. Gross Cost for `doduc`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.



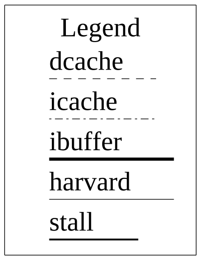


Figure 18: **Speedup vs. Gross Cost for `nasa7`.** All cache alternatives are direct mapped with 4 byte blocks.

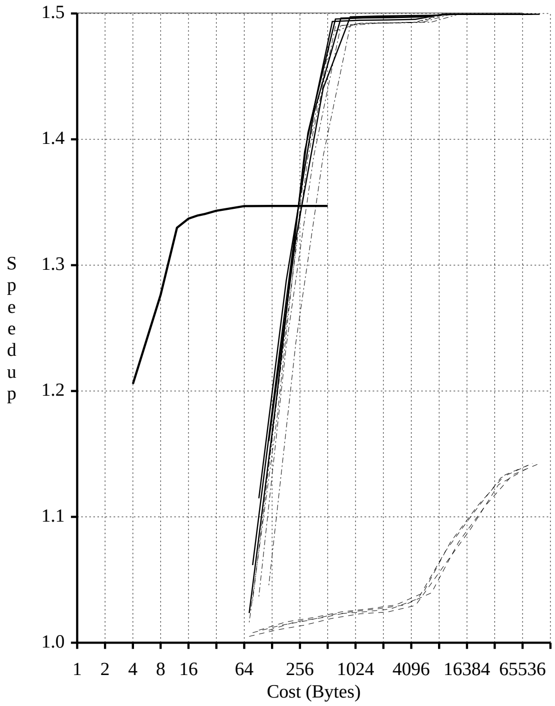


Figure 19: **Influence of Block Size on Speedup vs. Gross Cost for `nasa7`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

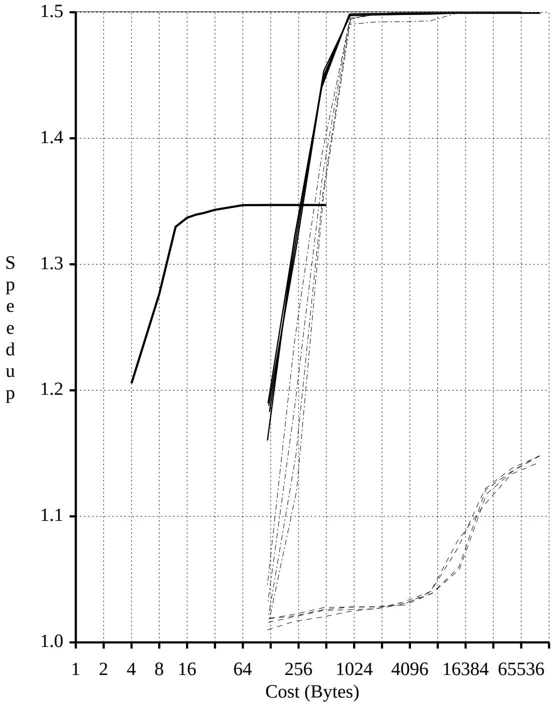


Figure 20: **Influence of Associativity on Speedup vs. Gross Cost for `nasa7`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.

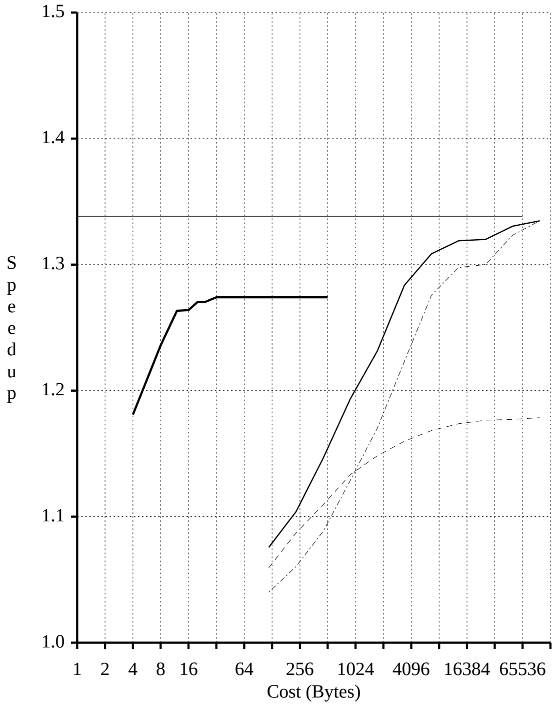
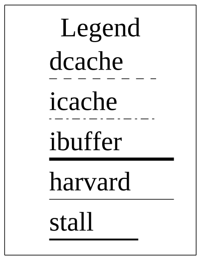


Figure 21: **Speedup vs. Gross Cost for `li`.** All cache alternatives are direct mapped with 4 byte blocks.



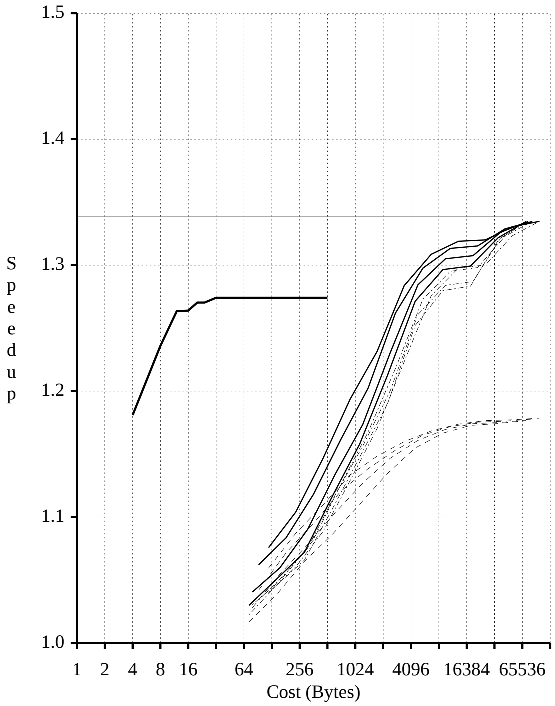


Figure 22: **Influence of Block Size on Speedup vs. Gross Cost for `li`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

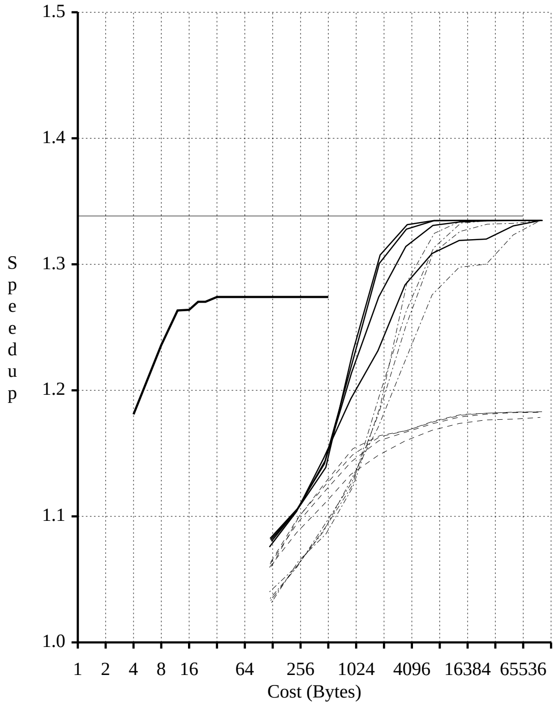
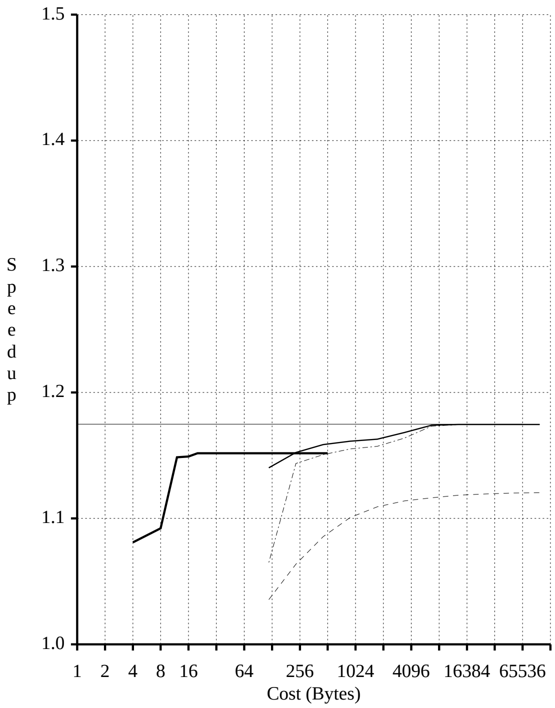


Figure 23: **Influence of Associativity on Speedup vs. Gross Cost for `li`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.



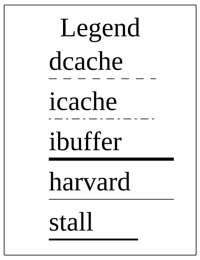


Figure 24: **Speedup vs. Gross Cost for `eqntott`.** All cache alternatives are direct mapped with 4 byte blocks.

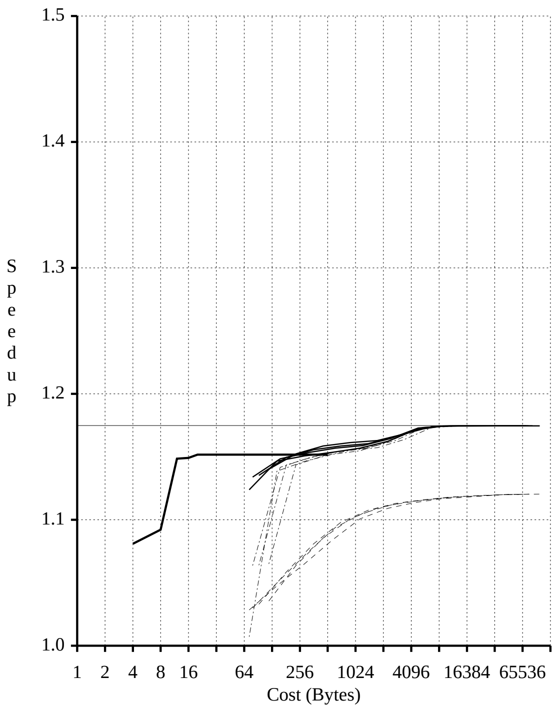


Figure 25: **Influence of Block Size on Speedup vs. Gross Cost for `eqntott`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

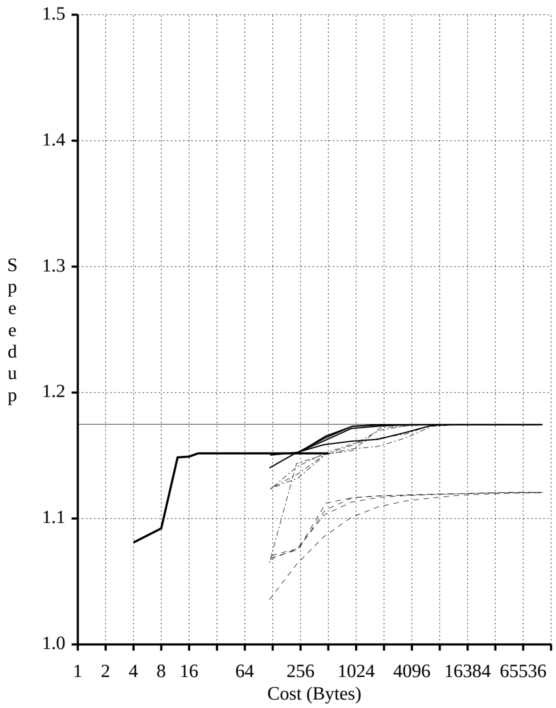
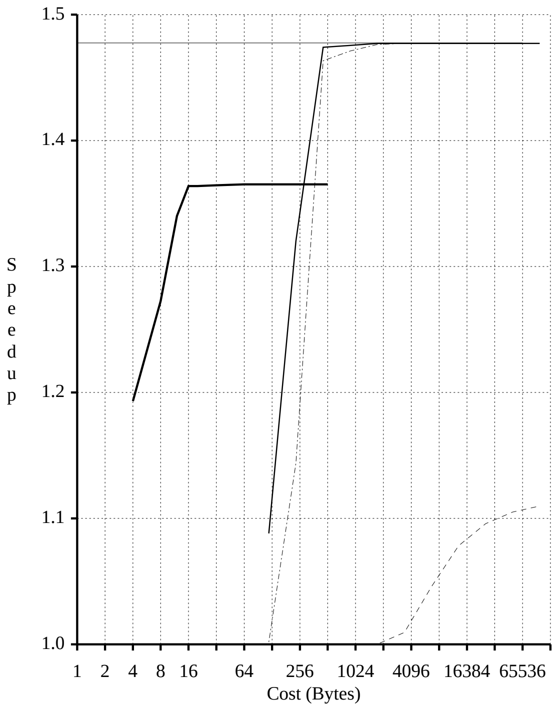


Figure 26: **Influence of Associativity on Speedup vs. Gross Cost for `eqntott`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.



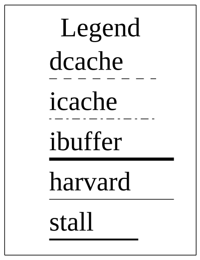


Figure 27: **Speedup vs. Gross Cost for `matrix300`.** All cache alternatives are direct mapped with 4 byte blocks.

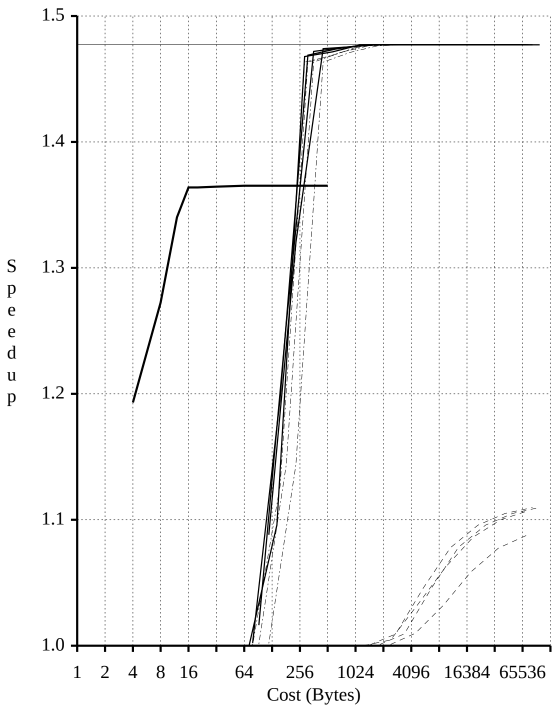


Figure 28: **Influence of Block Size on Speedup vs. Gross Cost for `matrix300`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

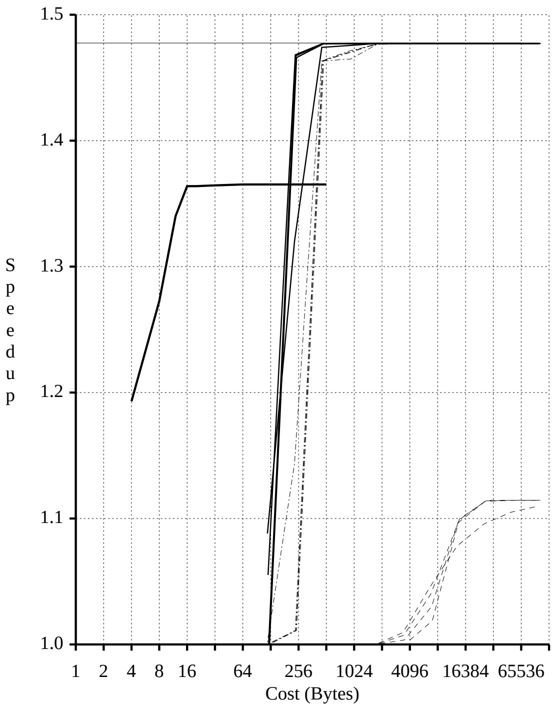
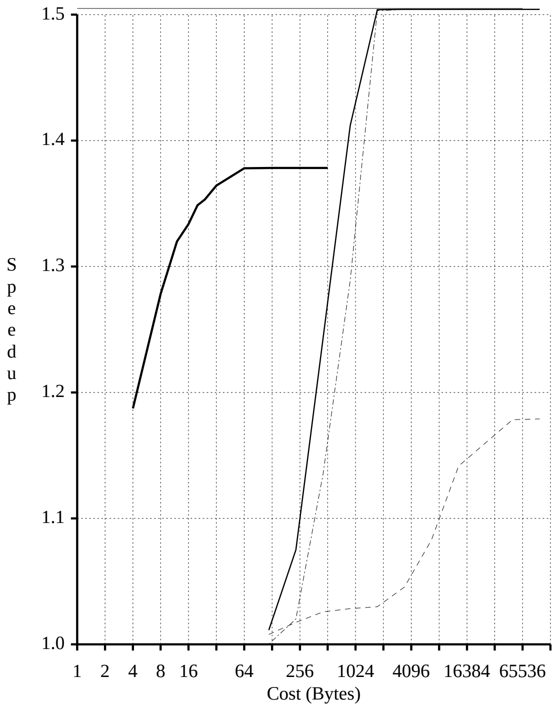


Figure 29: **Influence of Associativity on Speedup vs. Gross Cost for `matrix300`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.



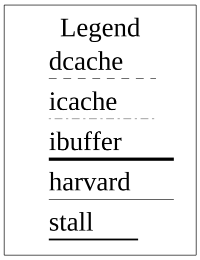


Figure 30: **Speedup vs. Gross Cost for `tomcatv`.** All cache alternatives are direct mapped with 4 byte blocks.

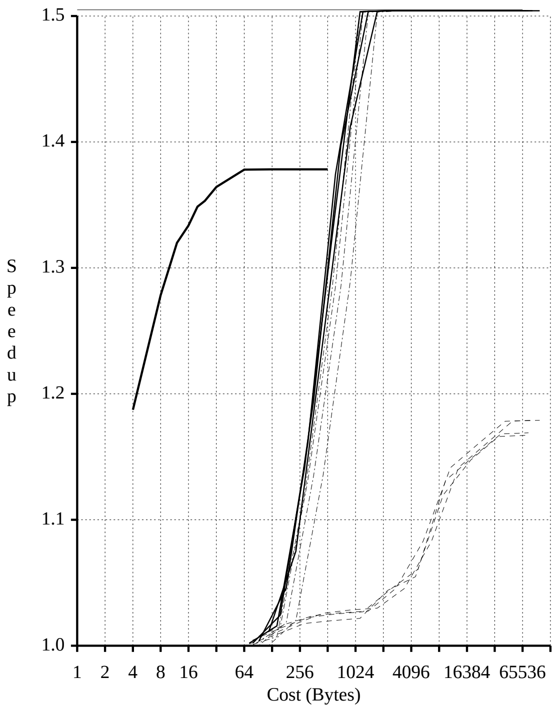


Figure 31: **Influence of Block Size on Speedup vs. Gross Cost for `tomcatv`.** All caches are direct mapped with single word sub-block placement. For each we present curves with 4, 8, 16, and 32 byte blocks.

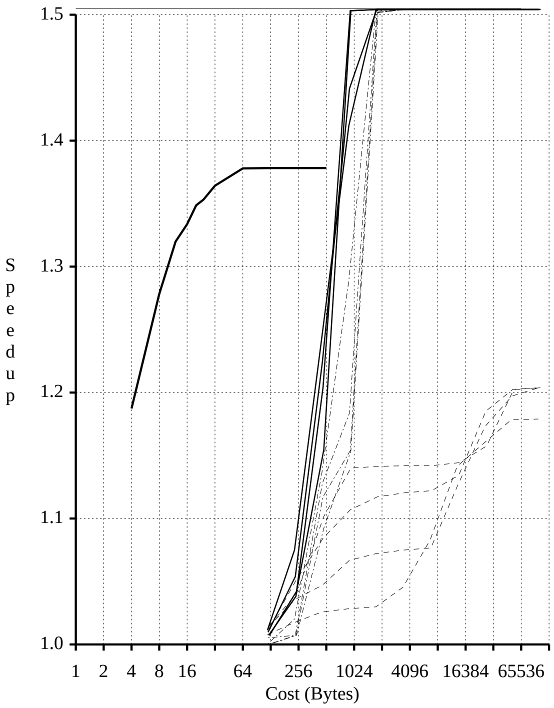


Figure 32: **Influence of Associativity on Speedup vs. Gross Cost for `tomcatv`.** We present curves for direct mapped, 2-way, 4-way, and 8-way associative caches each with single word blocks.