A Study of an Alternative Split Cache Organization

Jack W. Davidson

Computer Science Technical Report 85-04a
November 30, 1984

A Study of an Alternative Split Cache Organization

Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22901

ABSTRACT

Split caches are normally divided into two parts, one for data and one for instructions. Split caches provide all the advantages of a unified cache (i.e., reduced memory access time and reduced memory bus traffic), and have several additional advantages. The bandwidth is effectively doubled since a request for data and a request for an instruction can be processed simultaneously, and the access time for a split cache may be lower than a unified cache of the same size. In addition, the individual caches can be tailored to the specific reference streams they will encounter. In this paper we discuss a split cache organization motivated by the execution stack found in many contemporary architectures and high-level languages. Trace-driven simulations are used to evaluate the proposed design. Based on these simulations, it is shown that such an organization can significantly improve memory performance at a small cost. In particular, the miss ratio and memory bus traffic are substantially reduced.

November 30, 1984

Department of Computer Science

The University of Virginia

Charlottesville, VA 22901

A Study of an Alternative Split Cache Organization

1. Introduction

A cache memory is a small (relatively), high-speed memory that maintains a copy of recently used instructions and data. Data and instructions held in cache memory can be referenced in 10 to 25 percent of the time required to access main memory. Computer systems with good price/performance ratios are possible using cache memories. Consequently, most all modern, medium to large-scale computer systems include some type of cache memory.

There are two aspects of program behavior that make cache memories effective. They are:

1. Temporal locality, which is the tendency of a program to reference in the near future those locations referenced in the recent past. Loops are one type of construct that contribute to this phenomena.
2. Spatial locality, which is the tendency of a program to reference in the near future locations near those referenced in the recent past. Processing arrays and sequential portions of code lead to this phenomena.

The cache makes use of these phenomena by bringing in extra words on each access to main memory (look ahead) and keeping copies of recently used words (look behind).

The main metric used to measure a cache memory's effectiveness is hit rate. The hit rate is the ratio of the total number of memory references to the cache and the total number of memory references expressed as a percentage. Typically cache memories operate at hit rates of 85 to 90 percent or greater. A small increase in the hit rate can mean substantially faster execution times. For an excellent introduction to cache memories and survey of research in the design of cache memories up to approximately 1981 see [Smit82].

One method of improving certain aspects of cache performance is to split the cache into two

separate caches. Normally one cache holds instructions while the other holds data. Such an approach has several advantages. The bandwidth of the cache is essentially doubled since the two caches can service requests simultaneously. As the size of a cache increases so may its access time. This effect is caused by several factors. A very large cache may require more complex access circuitry. Further, with a large unified cache it may not be possible to place the cache near all the components that need access to the cache. Splitting the cache may allow each portion to be placed at its most advantageous position, and since the individual caches are smaller they may have less complicated access circuitry (hence faster).

A third possible advantage of a split cache organization is that the instruction cache can be simpler (hence faster or cheaper) if no stores are allowed into the instruction cache [Smit84]. A fourth advantage is that the instruction cache and data cache can be tailored to the specific referencing patterns found in their respective reference streams. A fifth advantage is that a split cache allows more flexibility in choosing the total capacity of cache. Since most caches employ bit selection to choose a set, caches usually come in sizes that are a power of two. For example, with a unified cache of size 8K bytes, the next increment would be 16K. With a split cache it would be possible to have total capacities of 9K (8K + 1K), 10K, and 12K.

The main disadvantage of the split instruction/data cache is its inefficient use of the cache memory. A unified cache is in some sense self-managing. It can devote space to the portions of a program (instruction or data) in the most demand. On the other hand, in a split cache instructions and data cannot share the total resource available. Simulation studies have shown that where instruction and data can appear in the same cache lines, split caches may have lower hit rates than unified caches. The effect depends on the particular workload [Smit82].

This paper examines the performance of a split cache organization that is motivated by the execution behavior of high-level languages. In particular we examine the performance of a cache that is split into a section that holds references to the runtime stack, and a section that holds references to instructions and all other data.

The rest of this paper is organized in the following manner. This section concludes with a

review of some of the current work in cache memory design. Section 2 describes the motivation for splitting a cache into a section for the runtime stack and a section for instructions and all other data. Section 3 describes the proposed design. The trace-driven simulations conducted to evaluate the proposed design are described in Section 4.. The results of the simulations are presented in Section 5. Section 6 analyzes the simulation results and discusses the architectural ramifications of the proposed cache organization. Conclusions are presented in Section 7.

1.1 Recent Work

In his PhD dissertation, Borgwardt describes the development of cache structures to speed up memory referencing for block structured languages such as Pascal [Borg81]. Most of his proposals are for additional hardware aimed at reducing the number of microinstructions needed to emulate an instruction rather than at attempts to improve the cache hit rate. To test the various schemes proposed, six different microengines were coded to emulate Pascal P-code, and their performances were compared on a variety of Pascal test programs.

He concludes that the use of special hardware to generate stack addresses based on the Pascal runtime environment can result in significant speedups. He also tests the effects of various write policies, the use of top of stack registers, and a 64 word top of stack minicache, and program prefetching. Each one of these provided a small amount of improvement in performance.

A recent study by Hill and Smith, describes the design of on-chip caches for microprocessors [Hill84]. Placing the cache on the processor chip can reduce both memory access time and bus traffic. Using traces from four machines, two 16-bit architectures and two 32-bit architectures, they determined that a sector cache organization is a good choice for small on-chip caches. The use of a sector cache allows tradeoffs between the chip area required, the hit rate, and the amount of traffic on the memory bus.

Smith and Goodman analyze, both theoretically and experimentally, the cache placement policies for instruction caches [Smit84]. The theoretical analyses are based on a new model for cache references - the loop model. The loop model assumes the instruction stream consists of an unbounded sequence of address references formed by periodically repeating the same finite sequence

of new address references. Their experimental results indicate that the loop model is a good predictor of observed cache behavior.

Several techniques for reducing processor–memory traffic are discussed in [Good83]. He shows that when process switches are infrequent, an effective way to reduce memory–processor traffic is to have the cache exploit primarily temporal locality. The traffic ratio can be reduced further by using a new write strategy — *write once*. With the write once strategy, initially write–through is employed. All other caches are purged of the block being written. On successive writes to the block, copy–back is employed. Goodman shows how this policy reduces memory traffic, yet provides a general solution to the cache consistency problem. Trace–driven simulations were performed for the VAX–11 and PDP–11 architectures to test these ideas.

In a recent paper by Clark, cache performance of the VAX–11/780 is studied by direct measurement of the hardware [Clar83]. Two different types of workloads were measured using the hardware monitor. One was measurement of the machine in normal use at a Digital Equipment Engineering Site. The second was a synthetic workload produced by a remote terminal emulator using canned scripts to simulate a general timesharing load in an educational program–development environment. Besides characterizing the behavior of the 780's cache, this study shows that trace–driven simulations appear to be overly optimistic in their characterization of cache performance.

2. Motivation

The implementations of many high–level languages use a central stack for storing procedure and function environments. Pascal, C, Algol, Modula, and Ada are all examples of such languages. Consequently, for this reason and others, many contemporary architectures support a hardware stack. Besides providing support for the implementation of high–level languages, hardware stacks allow interrupts to be nested and they support shareable or reentrant routines. The PDP–11 and VAX–11 processors, the Motorola 68000, the National Semiconductor 32016, the Prime processors, and the Pyramid processor, to name a few, all support hardware stacks.

This suggests that a possible division of the cache may be to split it into a section for stack references, and one for instruction references and non–stack data references. To study how

programs reference stack data and non-stack data, address traces for five programs run on a VAX-11/780 running UNIX were produced with tags to discriminate between instruction references ('I-stream'), data references ('D-stream'), and stack references ('S-stream'). These programs are described in Section 3. Table 1 shows the percentage of references for each of these streams broken down by whether they were reads or writes. It should be noted that the I-stream percentages are adjusted to reflect that individual bytes are not fetched from memory, but that an instruction buffer fetches ahead of the program counter. This is, of course, only an approximation to the way the VAX actually operates.

Stream Distribution							
program	instruction stream	data stream			stack stream		
	reads	reads	writes	total	reads	writes	total
pargen	73.0	6.6	0.4	7.0	7.8	12.3	20.1
ed	48.7	18.1	12.2	30.3	10.6	10.4	21.0
nroff	52.3	12.6	5.1	17.7	14.8	15.2	30.0
zeros	39.9	9.3	5.5	14.8	21.4	23.9	45.3
cc	58.3	14.5	3.1	17.6	12.6	11.5	24.1
average	54.4	12.2	5.2	17.5	13.4	14.7	28.1

Table 1. Distribution of Address References (percent)

The table shows that in all cases but one, the percentage of S-stream references is greater than the D-stream references. D-stream references are 17.5 percent of the total number of references, while S-stream references make up 28.1 percent of the total. It is interesting to note that references to the S-stream are split about equally between reads and writes, while for the D-stream the reads are 12.2 percent of the total and the writes are only 5.2 percent of the total.

3. Proposed Design

Based on the statistics in Table 1, a cache split into a section for all stack references, and a section for instructions and all non-stack data could be a viable organization for machines similar to the VAX-11. Such an organization would provide increased bandwidth, and the stack cache's design could be driven by its referencing patterns.

A stack's referencing pattern is very predictable. By its very nature, stack references are very local. Consider how programs reference data. Most procedures and subroutines reference local

variables that are found in the procedure's activation record, which is located at the top of the stack. This reduces the possibility for conflicts, and indicates that a set-associative organization could be an unnecessary expense for a stack cache. A more natural organization would be a direct-mapped cache. This has the advantage of being cheaper and faster.

Approximately 50 percent of the references to the stack are writes. This is not surprising if one considers how runtime stacks are used in programming languages. On a procedure or subroutine call the caller's state must be saved on the stack. For complicated calling sequences this may involve saving significant amounts of information. Even for the simplest calling sequence three to four words must be saved. Similarly when a new procedure begins execution, its local variables are allocated on the stack. At each invocation of the procedure, these variables must be initialized (i.e. written) before they are used. Caches for many processors use a write-through and no write-allocate policy because the percentage of data reads (our D-stream plus S-stream) is much greater than data writes. The extra expense and complexity of write-allocate can not be justified. Because of the behavior shown in Table 1, it is clear that a stack cache should employ write-allocate. This increases the hit rate, and if used in conjunction with a copy-back scheme, it significantly reduces the number of memory writes.

4. Trace-Driven Simulations

To test the above ideas, several controlled experiments were performed using address traces gathered on a VAX-11/780 running UNIX. Simulations were performed to determine: 1) the optimal size of the stack cache, and 2) the effectiveness of the split cache organization,

The traces were generated by modifying and instrumenting the UNIX debugger, adb, to record every address referenced by the program. Each address was tagged with information to identify the stream, the type of object referenced, and the operation to be performed. The possible streams are: I-stream, D-stream, and S-stream. They are as previously described.

The possible types of objects are: byte, word, longword and float, and quadword and double. These types have lengths of 8, 16, 32, and 64 bits respectively. The possible operations are: read,

write, and modify. A modify operation is essentially a read then a write of the same location. The address along with its tag allows reasonably realistic simulations of the VAX architecture to be performed.

The programs traced were:

1) Pargen – a LR(1) parser generator written in Pascal.

2) Ed – a simple text editor written in C.

3) Nroff – a text formatting program written in C.

4) Zeros – a program for finding the zeros of functions written in FORTRAN–77.

5) Cc – the C compiler written in C.

Ed, nroff, and cc are production programs found on most UNIX systems. The trace for each program consisted of approximately 250,000 address references. While it is felt that these programs are reasonably representative of the types of programs run under UNIX on a VAX–11, different programs could yield different results. In addition, as in the case of most trace–driven simulations, traces of the operating system were not included. Consequently, the performance reported here is likely to be lower in practice.

Each simulation used all five address traces to simulate multiprogramming. Each trace was used for 10,000 time units before switching to the next trace. The simulation clock was incremented by one time unit for each cache hit, and ten time units on each cache miss. No distinctions were made between the time required for a read and write miss, or read and write hits. Scheduling of which trace to use next was done on a round–robin basis. The simulations were run for approximately 1 million address references. Because the address traces contain the virtual addresses fetched by the programs, each trace was mapped to its own distinct address space before being presented to the cache.

Because the address traces were generated on a VAX–11/780, the raw trace files contain a large number of I–stream references. This is because the VAX uses an eight byte instruction buffer ('IB') to prefetch bytes ahead of the program counter. The IB operates in the following manner. It

fetches ahead of the program counter. When a jump occurs, several bytes may have been fetched that will not be used. The IB fetches an address-aligned four bytes whenever there is room in the IB for at least one byte. When these four bytes arrive (possibly much later), the IB takes as many bytes as it has room for then [Emer84].

Simulation of the actual behavior of the IB would be difficult due to the timing dependencies of its operation. In the trace-driven simulations described here, the IB fetched a four byte address-aligned longword only when the IB was empty. It is interesting to note that Clark reports in his paper on the measurement of the VAX that 66 percent of references to the cache are due to the I-stream [Clar83]. Our simulations with a cache organization identical to the VAX's show an average of about 55 percent of the references to the cache are due to the I-stream. Part of this difference is due to the conservative fetch policy used in our simulation of the IB.

The first set of simulations were conducted to determine, for aggregate cache sizes ranging from 1K bytes to 128K bytes, the optimal size for the stack cache. The instruction/data cache had the following organization: a) two-way set-associative, b) random replacement, c) write-through and no write-allocate, and d) a line size of eight bytes. The stack cache was direct-mapped with a line size of eight bytes. For each aggregate size, several experiments were run to determine the best split.

To evaluate the performance of the split cache, five sets of experiments were performed. Each experiment had a different cache organization. In all experiments involving a stack cache, only the optimal size stack cache was tested. For each experiment, caches with total capacities ranging from 1K bytes to 128K bytes were tested. The five organizations tested were:

1) a unified, two-way set-associative cache, random replacement, write-through and no write-allocate, and a line size of eight bytes. This organization is the one found in the VAX-11/780.

2) as in 1 above, except that on S-stream references write allocation is performed.

Optimal Stack Cache Size			
Total Cache Size	I/D Cache Size	Stack Cache Size	Hit Rate
1k	.25k	.75k	0.84467
1k	.50k	.50k	0.88290
*1k	.75k	.25k	0.89112
1k	.825k	.175k	0.88845
2k	1.k	1k	0.90600
2k	1.5k	.5k	0.91445
*2k	1.75k	.25k	0.91451
2k	1.825k	.125k	0.90844
4k	2k	2k	0.92418
4k	2k	2k	0.92418
*4k	3k	1k	0.93046
4k	3.5k	.5k	0.93029
8k	6k	2k	0.94878
*8k	7k	1k	0.94977
8k	7.5k	.5k	0.94880
16k	14k	2k	0.96136
*16k	15k	1k	0.96201
16k	15.5k	.5k	0.96188
32k	30k	2k	0.97294
*32k	31k	1k	0.97300
32k	31.5k	.5k	0.97247
64k	59k	5k	0.98006
*64k	60k	4k	0.98034
64k	62k	2k	0.97914
64k	63k	1k	0.97834
64k	63.5k	.5k	0.97740
128k	123k	5k	0.98216
*128k	124k	4k	0.98231
128k	126k	2k	0.98116
128k	127k	1k	0.98014

Table 2. Experiments to Determine Optimal Stack Cache Size

3) a split cache, where the I/D cache is organized as in 1 above. The stack cache is a directed-mapped cache with a line size of eight bytes.

4) as in 3 above, except that on S-stream references write allocation is performed.

5) as in 4 above, except that the stack cache employs copy-back.

Experiments 1 and 3, and 2 and 4 serve to isolate the effects of the split cache organization. Experiment 5 serves to determine the effectiveness of using copy-back in the stack cache.

Throughout the simulations several parameters were kept constant. For instance, the line size and the number of elements per set were not varied. The motivation for this was to stay as close

as possible to the organization used in the VAX. This allows comparisons between our results and results in several reports in the literature that described both the simulated and actual behavior of the VAX-11/780's cache. This provides some degree of verification that the simulations are reliable. Whenever possible we compare the results reported here with the results from other sources.

5. Results

The results of the experiments to determine the optimal stack cache size are shown in Table 2. The starred entries mark the optimal split for each total size. For aggregate cache sizes of 1K and 2K bytes, a stack size of 256 bytes produced the best hit rate. For aggregate sizes ranging from 4K to 32K bytes, a stack cache size of 1K bytes produced the best hit rate. For caches larger than 32K bytes, a 4K stack cache is required for optimal performance.

Table 3 shows the hit rates for experiments 1, 2, 3, and 4. Experiment 5's hit rate is identical to experiment 4's. These statistics are plotted in Figure 1. As expected, there is a substantial difference between the hit rates of the experiments where the independent variable is write-allocate or no write-allocate. We note that that splitting the cache into an instruction/data cache and a stack cache incurred no miss ratio penalty. In fact, in both experiments where the independent variable was the split cache, the split cache, in general, had a slightly higher hit ratio.

Cache Hit Rate Statistics						
Unified Cache			Split Cache(Optimal Split)			
	No Write Alloc Experiment 1	Write Alloc Experiment 2	(I/D)	(Stack)	No Write Alloc Experiment 3	Write Alloc Experiment 4
Size (bytes)	Hit Rate	Hit Rate	Size (bytes)	Size (bytes)	Hit Rate	Hit Rate
1k	0.84441	0.88708	.75k	.25k	0.86166	0.89112
2k	0.88232	0.91634	1.75k	.25k	0.88663	0.91451
4k	0.90024	0.92911	3k	1k	0.91280	0.93047
8k	0.92665	0.94856	7k	1k	0.93381	0.94977
16k	0.94575	0.96129	15k	1k	0.94671	0.96202
32k	0.96179	0.97298	31k	1k	0.95976	0.97300
64k	0.96948	0.97847	60k	4k	0.97237	0.98034
128k	0.97379	0.98174	124k	4k	0.97503	0.98232

Table 3. Hit Ratios for Experiments 1, 2, 3, and 4

We note that the 8K cache in experiment 1 is the organization found on the VAX. Its hit ratio

is 92.6 percent. In Clark's paper on the measured performance of the VAX, he notes that J. S. Emer performed some simulations of the 780's cache organization [Clar83]. Emer reported hit ratios of 93 percent.

Experiment 5 was conducted to determine the effects of using a copy-back mechanism in the stack cache. One set of simulations were conducted that used an optimally split cache that employed write-allocate, but no copy-back. A second set of simulations were run with the same organization except that the stack cache employed write-allocate with copy-back. Table 4 contains the results of this experiment.

Write Traffic Reduction (Experiment 5)					
Total Cache Size (bytes)	I/D (bytes)	Stack (bytes)	Write Through (# of Writes)	Copy Back (# of Writes)	Reduction (percent)
1k	.75k	.25k	164954	46477	71
2k	1.75k	.25k	177611	48559	72
4k	3k	1k	181627	46275	74
8k	7k	1k	202704	49575	75
16k	15k	1k	226653	54193	76
32k	31k	1k	247529	58816	76
64k	60k	4k	255132	57251	77
128k	124k	4k	260185	58101	77

Table 4. Comparison of Memory Writes Copy Back vs. No Copy Back

The number of memory writes is reduced by anywhere from 71 percent for an aggregate cache size of 1K bytes to 77 percent for an aggregate cache size of 128K bytes. The extra complexity and expense of a copy-back cache would seem to be worthwhile.

6. Analysis

The proposed split cache design offers all the advantages previously mentioned for a split cache. The bandwidth is doubled since the cache can service two requests simultaneously if the requests are not to the same cache. Another advantage is that the individual caches can be placed near the components that need access to them. In a later section we discuss placing the stack cache on the chip with the processor. The split cache provides flexibility in choosing total cache size. It is possible to have a cache memories with capacities other than powers of two. The following sections discuss specific areas of analysis.

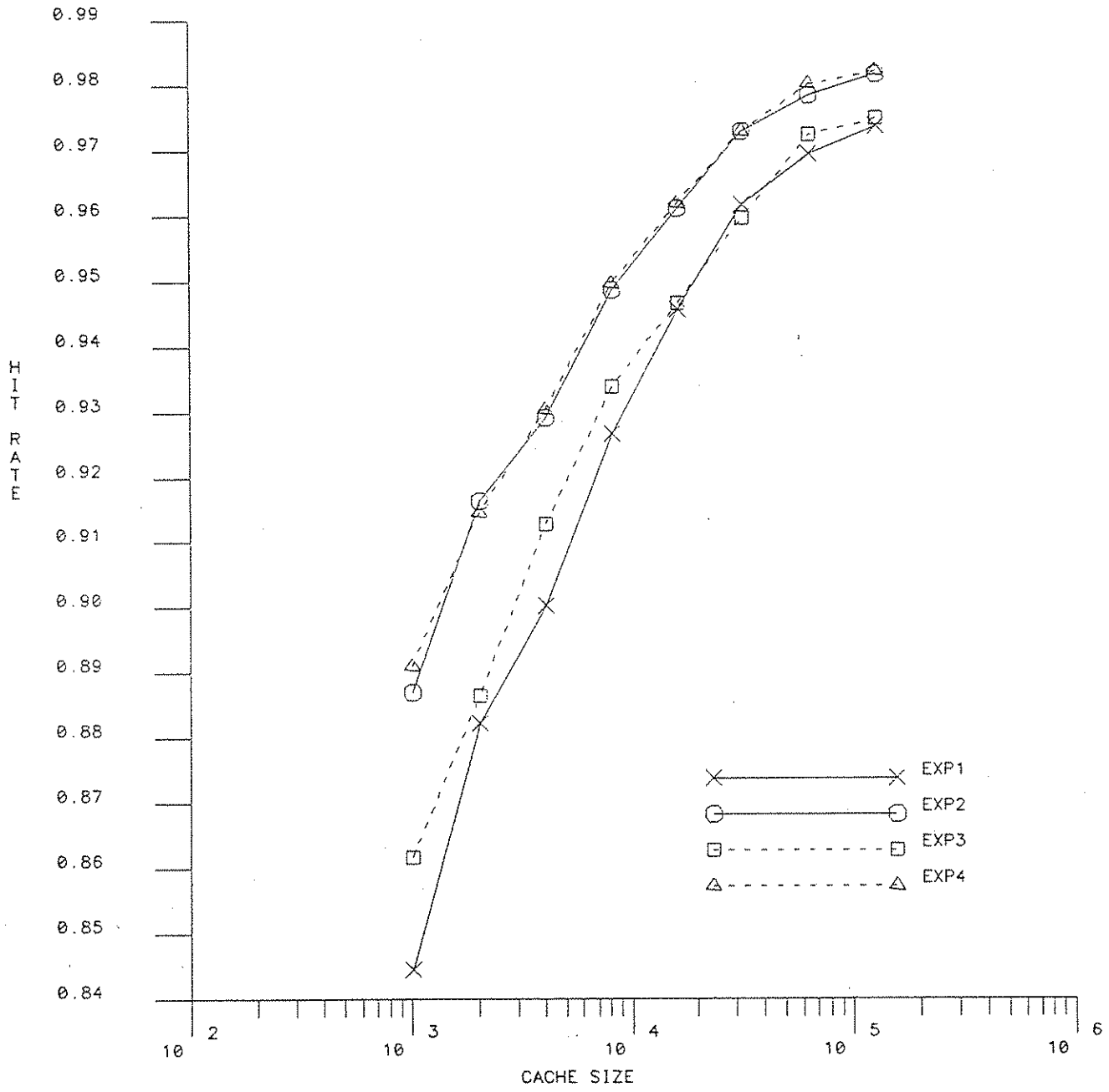

Figure 1. Hit Rate Comparisons of Experiments 1, 2, 3, and 4

6.1 Hit Ratio

Comparison of the results of Experiment 1 and Experiment 3 show that with all other design parameters being the same, the overall hit rate for a split cache was *higher* than the hit rate for a unified cache of the same size (see Figure 1). The difference, however, is small.

The major impact was caused by tailoring the stack cache's organization to handle the referenc-

ing patterns it encountered. Using a write-allocate policy in the stack cache significantly improved the hit rate. It is interesting to note that in his simulations of the PDP-11/70 cache, Strecker reports neligible performance differences between write-allocate and no write-allocate for a unified cache of 1K bytes, two-way set-associative, and random replacement [Stre76]. Use of this organization is so effective that a split cache of a specified size out performs a unified cache that has *double* the capacity. For example in Table 3, the hit rate of the 8K split cache (7K + 1K) has a hit rate of 94.9 percent. The unified cache of 16K has a hit rate of 94.5 percent. This improvement would be even higher in an actual configuration where the size of the I/D cache would be 8K bytes.

It is interesting to note that the recently announced VAX 8600 employs a 16K copy-back cache [Desm84]. Our simulation studies indicate that comparable performance could be obtained by employing an 8K I/D cache with write-through and a 1K stack cache with copy-back. Such a configuration would be cheaper providing a significantly better cost/performance ratio.

Table 5 shows a breakdown of the hit ratios by stream. The hit rates for the I-stream and S-stream are significantly higher than the hit rate for the D-stream. This is not surprising considering that the referencing patterns for the I-stream and S-stream are predictable. This points out several avenues of experimentation. Is it possible to significantly improve the hit rate of the D-stream? One possible way would be to have the software (i.e. the compiler) predict the referencing pattern for the D-stream. Perhaps performance could be improved further by splitting the cach into three parts: a cache for the I-stream, a cache for the D-stream, and a cache for the S-stream. In this way the data cache could be tailored to its referencing stream (if there is one). If there is no discernible pattern to the D-stream, it may be that a small fully associative cache could provide better performance.

6.2 Cache Consistency

While using a copy-back mechanism reduced the number of writes to memory by approximately 77 percent, a copy-back cache exacerbates the problem of cache consistency in multiprocessor systems [Cens78, Dubo82, Smit82]. If write-through is used, main memory always contains an up-to-date copy of all information. With copy-back, however, different versions of the same

Breakdown of Hit Ratios by Stream						
Total Cache Size (bytes)	I/D (bytes)	Stack (bytes)	I-stream	D-stream	S-stream	Total
1k	.75k	.25k	0.90560	0.72102	0.84851	0.89112
2k	1.75k	.25k	0.92692	0.78264	0.88164	0.91451
4k	3k	1k	0.94096	0.80847	0.89647	0.93047
8k	7k	1k	0.95948	0.85680	0.92455	0.94977
16k	15k	1k	0.97182	0.88794	0.94170	0.96202
32k	31k	1k	0.98307	0.91603	0.95671	0.97300
64k	60k	4k	0.98819	0.92967	0.96410	0.98034
128k	124k	4k	0.98968	0.93658	0.96775	0.98232

Table 5. Hit Ratios by Stream

piece of information may exist in several different places at the same time. There are several solutions to this problem. One approach was discussed in the review of recent work. A second approach is to create a special bus to broadcast the address of a write by a processor to all other processors [Cens78]. The other processors can then invalidate their copy. A third solution is to use a directory of main memory lines, and use it to ensure that no lines are write-shared. This method is very complex and may be expensive in terms of hardware.

The use of a simple convention and only using copy-back in the stack cache and *not* in the I/D cache, allows us to avoid many of these complications. The convention that must be followed is that information that can be shared among processors must not be stored on any processor's stack. While this convention must be enforced by the designer's of the system software, it is unlikely that global shared data would have been allocated on the stack in the first place. Global shared data may still be cached, but it will be entered into the I/D cache which employs write-through.

6.3 On-Chip Caches

For a given aggregate cache size, the optimal size of the stack cache is quite small. For example for aggregate cache sizes of 1K and 2K bytes, 256 words of stack cache provides optimal performance. For cache sizes ranging from 4K to 32K bytes a 1K byte stack cache is sufficient. Hill and Smith note that placing a cache on the processor chip can reduce both mean memory access time and bus traffic [Hill84]. They contend that these on-chip caches will initially be small, ranging from 32 to 2048 bytes. The stack cache certainly falls into this range.

6.4 Architectural Requirements

The only real requirement for the proposed scheme to be implemented is the ability to distinguish between a stack reference and other types of references. For instance, the VAX-11/780 uses one bit of the address to distinguish between references to the stack segment (known as the P1 space), and all other references. This capability could easily be added to other machines.

7. Summary

We believe that the simulations presented in this paper show that a split cache organization based on an instruction/data cache and a stack cache is a viable split cache organization. Such an organization provides excellent performance at a small incremental cost. While the simulations presented here are for the VAX-11/780 only, we believe that similar results will be obtained for other contemporary architectures. The Motorola 68000 and the National Semiconductor both have hardware stacks and instruction similar to the VAX to support high-level languages. Simulation experiments are currently being conducted to test this hypothesis.

8. Acknowledgements

We would like to thank Tim Sigmon for many helpful observations.

9. References

[Borg81] P. A. Borgwardt, *Cache Structures Based on the Execution Stack for High Level Languages*, PhD Dissertation, University of Washington, 1981.

[Cens78] L. Censier and P. A. Feautrier, A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers TC-27*, 12 (December 1978), 1112-1118.

[Clar83] D. W. Clark, Cache Performance in the VAX-11/780, *ACM Transactions on Computer Systems 1*, 1 (February 1983), 24-37.

[Desm84] J. Desmond, Dec. Launches VAX 8600, *Computerworld 18*, 45 (November 1984), 1.

[Dubo82] M. Dubois and F. A. Briggs, Effects of cache concurrency in multiprocessors, *Proceedings of the 9th Annual Symposium on Computer Architecture*, Austin, TX, April 1982, 299-308.

[Emer84] J. S. Emer and D. W. Clark, A Characterization of Processor Performance in the VAX-11/780, *Proceedings of the 11th Annual Symposium on Computer Architecture*, Ann Arbor, June 1984, 301-310.

[Good83] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, *Proceedings of the 10th Annual Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, 124-131.

[Hill84] M. D. Hill and A. J. Smith, Experimental Evaluation of On-Chip Microprocessor Cache Memories, *Proceedings of the 11th Annual Symposium on Computer Architecture*, Ann Arbor, June 1984, 158-166.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys 14*, 3 (September 1982), 473-530.

[Smit84] J. E. Smith and J. R. Goodman, Instruction Cache Replacement Policies and Organizations, 549, University of Wisconsin, Madison, WI, July 1984.

[Stre76] W. D. Strecker, Cache Memories for PDP-11 Family Computers, *Proceedings of the Third Annual Symposium on Computer Architecture*, New York, January 1976, 155-158.