Vector Processor Caches*

Jeffrey D. Gee
Alan Jay Smith

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

Vector processors have typically used vector registers, interleaved memory, and pipelined access to data to provide sufficient memory system performance. Caches have been used mainly for instructions, while data references are usually uncached, presumably partially because of the belief that there is insufficient data locality in vector workloads. In this study we use memory address traces from Cray X-MP and Ardent Titan machines to examine both reference locality and cache performance in a vector processing environment. Many of the Titan traces in particular are from real vectorized applications which reference large amounts of data. We have found that vector references contain somewhat less temporal locality, but large amounts of spatial locality compared to instruction and scalar references. Cache miss ratios are found to be comparable to those measured and published previously for various non-vectorized workloads. We provide analyses of trace behavior with regard to parameters of interest to cache designers. Calculations based on our measured miss ratios indicate that caches will improve average access times, which in turn can be expected to translate into significant improvements in machine performance.

October 21, 1992

* The material presented here is based on research supported in part by the National Science Foundation under grants MIP-8713274 and MIP-9116578, by NASA under consortium agreement NCA2-128, by the State of California under the MICRO program, and by Philips Laboratories/Signetics, the International Business Machines Corporation, Digital Equipment Corporation, the Intel Corporation, Mitsubishi Electric Company and Sun Microsystems,

1. Introduction

Caches are small, high speed memories which improve performance by reducing memory access time from tens of processor cycles to at most a few processor cycles. Caches are effective because of the *principle of locality*, which states that over small periods of time, programs tend to cluster their references to a small subset of the address space. *Temporal locality* implies that data currently in use will likely be used again in the near future (i.e. program loops and local variables). *Spatial locality* implies that data near recently referenced data is also likely to be used (i.e. instructions, array data and stack data).

Most commercial vector processors do not fully utilize caches in their memory hierarchies. Vector processors from Cray, Hitachi, Fujitsu, and NEC use caches only as high speed instruction buffers [Eoya88, Lazo88, Lube85]. The IBM 3090 VF [Tuck86] and Alliant FX-8 [Abu86] are among the few machines which cache vector references. The IBM 3090, however, is a commercial mainframe with cache memory. The add-on Vector Facility simply shares the common memory interface with the general purpose processor. The Alliant FX-8 is a vector multiprocessor which requires caches to support the memory bandwidth requirements of its eight processors.

One reason for the lack of data caches in most vector processors is the (incorrect) assumption that there is little locality in vector workloads. We will show that sufficient locality exists in these workloads to well justify the use of caches. Furthermore, current technology allows the implementation of much larger caches compared to the cache sizes available when current supercomputers were developed. Caches of a megabyte or more may reasonably contain the working sets of real vector applications.

Caches have also been avoided because it has been possible to obtain good performance through careful programming. By overlapping data access with useful computation and using long vectors to minimize the effect of large memory latencies, programs can be coded to run well on supercomputers. Nevertheless, there is the cost in programmer time to tune programs on vector machines, which follows what we believe is the obsolete paradigm of expending human time rather than improving the machine hardware or software. We also point out that coding around memory latencies is becoming increasingly difficult as memory latencies continue to increase [Neve89]. By improving locality in vector applications [McKe69, Triv77] and using vector caches to speed access to data, it may be possible to obtain higher performance levels with less programmer effort.

This research makes extensive use of trace-driven simulation to explore the feasibility of using caches in vector processors. We drive cache models with a large number of vector traces, including many from real production applications running on Ardent Titan machines. Our

results indicate that caches can significantly improve the performance of a vector processor. Vector references contain large amounts of spatial locality and significant temporal locality. Miss ratios are in the same range as miss ratios for scalar machines, and are especially low for the large cache sizes possible with current technology. Vector caches reduce average access times by large amounts, which should translate into improved execution rates.

The remainder of this paper is organized as follows. Section 2 provides some background information and summarizes prior research on vector processor caches. Section 3 describes our methodology and examines the address traces used in this study. The bulk of our work is contained in Sections 4 and 5, in which we evaluate the locality present in the traces and analyze cache miss ratios across a wide range of cache parameters. Section 6 uses these miss ratios to estimate the performance effect of a vector cache, and compares these estimates to detailed timing simulations carried out in a separate effort [Gee92]. Section 7 presents our conclusions.

2. Background

2.1. Caches and Vector Processors

Very few commercial vector machines cache vector references. One such machine is the IBM 3090 VF [Tuck86], which caches vector references in a 64 Kbyte cache with a 128-byte block size and 4-way associativity. The cache is a write-back cache with LRU replacement and a data bandwidth of one 64-bit word per processor cycle. The single-ported cache is shared between the 3090 general purpose unit and the Vector Facility. The Alliant FX/8 multiprocessor [Abu86] also caches vector references in a shared, 128 Kbyte cache. The FX/8 consists of eight processors, each capable of 12 Mflops peak execution rate, connected to the shared cache via a crossbar switch. The cache is direct-mapped, with four-way interleaving and a peak bandwidth of 47 million words per second.

The fastest supercomputers only cache instructions, or at most scalar data references. Cray machines contain four to eight instruction buffers to store program code fragments currently in use. Cray 1 processors contain four 128-byte buffers; Cray X-MP and Y-MP processors contain four 256-byte buffers; Cray 2 processors contain eight 128-byte buffers. The Cray-2 also has a 128 Kbyte programmable local memory which can hold scalar as well as vector data currently in use. This local memory is not a cache, and its performance hinges on the ability of the compiler or programmer to keep frequently used data resident. The Hitachi S-810/20, Fujitsu VP-200, and NEC SX-2 contain 256 Kbyte, 64 Kbyte, and 64 Kbyte caches for instruction and scalar references respectively [Lazo88]. Successors to those Japanese machines continue to avoid using caches for vector references [Neve89].

Personal supercomputer and minisupercomputers such as the Ardent Titan [Died88], Stardent ST3000, and Convex C-240 [Chas88] use caches to improve integer or scalar operations, but have no cache for vector operations. The Ardent Titan consists of a commercial microprocessor (the MIPS R2000) performing integer operations, with a custom floating-point vector unit capable of delivering 16 Mflops peak performance. A Titan processor has 16 Kbytes of instruction and 16 Kbytes of data cache for MIPS scalar references. The Stardent ST3000, the successor to the Ardent Titan, runs at twice the clock speed of the Titan and contains 32 Kbyte instruction and scalar data caches. The Convex C-240 multiprocessor supports up to four 50-Mflop processors which execute a Cray-like instruction set. Each processor has 8 Kbytes of instruction cache and 4 Kbytes of data cache for scalar data references.

2.2. Previous Work

Although caches for vector data are currently not used in many supercomputers, continuing improvements in processor speed relative to memory speed have led to renewed interest. Several studies have recently looked at vector cache performance by measuring cache miss ratios using traces from vectorized applications. A few studies have also looked at the more important issue of processor performance with a vector cache. This section summarizes these earlier efforts, all of which tend to indicate that vector caches perform reasonably well. Results from our research supplement and extend the previous work.

Clark and Wilson [Clar86] measured cache miss ratios for an IBM 3090 VF running both scalar and vectorized versions of a benchmark set. The 3090 has a 64 Kbyte cache, with a 128 byte block size and a set size of four. Clark and Wilson found that vectorized versions of a program have higher cache miss rates, because vectorized programs execute and reference fewer instructions. The absolute number of cache misses was fairly constant.

So and Zecca [So88a,So88b] and Callahan and Porterfield [Call90] measured miss ratios for scientific programs running on IBM vector machines. Both studies started from a baseline cache organization and selectively varied parameters such as cache size, block size, and associativity. So and Zecca simulated cache sizes up to 2 Mbytes, while Callahan and Porterfield simulated cache sizes up to 256 Kbytes. Miss ratios were found to be approximately 5% or less for caches larger than 64 Kbytes using large block sizes (i.e. 64-128 bytes). Callahan and Porterfield estimated processor performance with a 32 Kbyte data cache with a four-byte block size, and found that approximately one-third of total execution time is spent waiting for cache misses. Those authors suggest that compilation techniques can be used to reduce this waiting time, although it is possible that larger caches with larger blocks would be as effective, and potentially more useful given the large size of typical vector applications.

Abu-Sufah and Malony [Abu86] timed kernels of varying vector length on the Alliant FX/8 to evaluate the performance effect of its 64 Kbyte data cache. Caching vectors was found to improve performance significantly, as programs executing on a uniprocessor system ran 1.4 to 2.3 times faster when vector data fits into the cache. In an eight-processor system, observed speedups from caching vectors are even larger, because the cache alleviates to a large extent the bottleneck at shared main memory.

Our research extends earlier work by first providing a detailed evaluation of the spatial and temporal locality within the reference stream of a vector processor. This analysis provides useful insight as to the potential benefit of vector caches. We then measure cache miss ratios over a much wider range of cache parameters, as compared to earlier efforts. These miss ratios are in turn used to estimate the performance effect of a vector cache. Our estimates show that very large caches can improve execution speed by a factor of two when memory is slow (e.g. 50 processor cycles). Studies which have previously looked at machine performance [Abu86, Call90] examined cache sizes (32-64 Kbytes) which are considerably smaller than the cache sizes that can be obtained with current technology. Our performance estimates are also validated against results from an accurate timing simulator of a vector cache machine [Gee92].

3. Methodology and Trace Characteristics

This research makes extensive use of *trace-driven simulation*. Trace-driven simulation involves the recording, via either hardware or software means, of the sequence of memory addresses referenced by a program. These addresses are later used as input to a cache simulator.

We have collected two sets of vectorized traces, one from programs running on a Cray X-MP, and another from programs running on an Ardent Titan. Most of the applications traced on the Cray are small programs such as the Livermore Loops and NAS Kernels. These programs are designed more for evaluating processor pipelines rather than large vector caches. We have other Cray traces from scalar UNIX applications, also not typical of large vectorized workloads. The Ardent Titan traces are more representative of true memory reference behavior, as they were gathered from large real production applications. The Titan traces are also quite long (20 million references), enabling us to more thoroughly evaluate the large caches that we simulate.

The Cray X-MP address traces were collected by a modified Cray Research Inc. simulator of Cray machines [Cast87]. In this case, the simulator was configured to simulate the X-MP and was also run on an X-MP. The simulator was modified to record reference addresses into a trace file while simulating X-MP execution. The traces come from both vectorizable and non-vectorizable benchmarks. Traces from vector benchmarks include four of the NAS Kernels [Bail85] and four of the Livermore Loops [McMa86]. The NAS Kernels are specifically designed to evaluate vector processor performance, while the Livermore Loops are Fortran

computations frequently used at the Lawrence Livermore National Laboratory. The remaining traces were collected from the UNIX utilities *diff*, *grep*, and *qsort*, and the *dhrystone*, *linpack*, and *whetstone* benchmarks. None of the Cray benchmarks references very much address space. As mentioned earlier, the vectorized kernels are designed to represent computational rather than memory reference behavior.

The Cray traces include all instruction, scalar data, and vector data references. Data references access eight-byte quantities, although references involving Cray 24-bit address registers use only the low three bytes of an eight-byte word. Instruction fetches reference two-byte instruction parcels, with an instruction being either one or two parcels long. Separate trace records are generated for each parcel of a two-parcel instruction. Each benchmark was traced for up to one million instruction executions.

The Ardent Titan is a graphics supercomputer which combines the scalar performance of a MIPS R2000 processor with a custom floating point unit (FPU) which executes scalar and vector floating-point operations. The FPU has a peak performance of 16 Mflops, and is fully pipelined with chaining support, vector register storage for 8K 64-bit words, and three memory access pipelines. A Titan can be configured with up to four processor boards, where each board contains a 16 MHz R2000 and an 8 MHz FPU.

The Titan traces were collected by using an Ardent *postloader* (similar to the MIPS *pixie* tool) to instrument object code. Calls to tracing routines were inserted in front of each load, store, and branch in the original object code. The tracing routines are implemented in Unix System V shared libraries, which are attached to the code at runtime. When the calls to the tracing routines are inserted, care is taken to ensure that instruction addresses passed to the tracer are the same as in the original, unmodified code. While it is working, the postloader maintains original addresses for each basic block. Calls to tracing routines pass these original addresses as parameters, along with the number of instructions in the basic block. Data reference addresses are unaffected by the postloader. Traces include instruction and scalar references made by the MIPS R2000, as well as scalar and vector references made by the FPU. The FPU can also perform scatter/gather references, which are seen as a sequence of scalar references. MIPS instruction and data references are four bytes wide, while FPU data references are four or eight bytes wide, depending on the floating-point operation precision.

The Ardent benchmarks, all of which are in Fortran, are listed in Table 1. Three (*bmk1*, *bmk11a*, and *bmk21b*) come from the Los Alamos benchmark set [Grif84]. Although these codes are self-contained applications, and not just program kernels, they are smaller and somewhat less ‘‘real’’ than the other programs.

Ardent Benchmarks	
Benchmark	Description
ampac	molecular orbital package
arc3d	3-D fluid flow using finite difference analysis
bmk1	monte carlo simulation
bmk11a	particle in a cell
bmk21b	photon transport
born	molecular mechanics minimization
flo82	airfoil flow analysis using Euler equations
lapack	1000x1000 linear equation solver (BLAS level 3)
mopac	molecular orbital package
simple	hydrodynamic and thermal behavior of fluids
wake	calculates free wake of rotor

Table 1: Ardent Benchmarks

The other programs are real production applications used for benchmarking at Ardent Computer, and come from areas such as computational chemistry, computational fluid dynamics and linear analysis. While not as large as programs routinely run on Cray machines, the Ardent programs are large enough to stress the memory system of the Titan, and are very similar *qualitatively* to the types of programs run on Crays. Thus, results observed on the Ardent programs should apply directly to Cray applications, provided we scale our results to correspond to the larger problem sizes.

Table 2 lists memory reference and address space characteristics of our traces. The memory reference counts were measured using two techniques:

(1) Each architectural memory reference is counted as one access, irrespective of whether it references a one, two, four, or eight-byte quantity. These reference counts, being *unnormalized* to the word size of the machine, are more representative of the machine architecture rather than the machine implementation.

(2) Certain program references are merged into one or split into two accesses, depending on the nominal word size of the machine. This corresponds more to how data is actually referenced. For the Cray, with a word size of eight bytes, consecutive two-byte instruction parcels in the same basic block are packed into a single eight-byte word reference. For the Ardent Titan, with a word size of four bytes, each double precision data reference is split into two single precision references. These are referred to as *normalized* reference counts.

In Table 2, *unnormalized* instruction fetch percentages are highest for the Cray scalar workload and smallest for the highly vectorized kernels in the Cray vector workload. Cray instruction fetch percentages are reduced significantly once the counts are normalized to word size. Table 2 also lists the fraction of data references that are vectorized, which ranges from a low 10 percent in the Cray scalar workload to over 80 percent in the Cray vector workload. Average basic block size ranges from 17 to 25 instructions, corresponding to branch taken frequencies of 4 to 6 percent. Prior studies have measured branch taken frequencies of 7% and 6% for IBM 370 and CDC 6400 scientific workloads [Lee84] and 10% in a MIPS floating point workload [Perl89]. The fraction of taken branches is much larger in non-scientific workloads, and ranges from 10 to nearly 30 percent [Lee84, Perl89].

Table 2 also shows workload averages from [Smit85], including data from a VAX workload of ten integer C programs, an IBM 370 workload of three FORTRAN scientific applications, and an IBM 360/91 workload consisting of four mixed applications. All results from [Smit85] are normalized to machine word size, and thus are best compared to our normalized statistics. The VAX and IBM 370 workloads assume a constant four-byte reference width, while the 360/91 workload assumes a constant eight-byte reference width.

Normalized averages for the Ardent workload agree quite well with the workloads from [Smit85]. In the Cray vector workload, instruction fetch percentages are much lower because the highly vectorized kernels require fewer instructions to execute the application. In the Cray scalar workload, the ratio of data reads to writes is only one-to-one, much lower than the typical two-to-one ratio. This is possible because Cray machines provide programmer-visible *backup scalar registers* which buffer data between primary scalar registers and main memory. Transfers between these backup registers and main memory are carried out in block mode, thus equalizing the ratio of reads to writes.

Table 2 also shows that the Cray traces are quite small, as none references much more than 128 Kbytes of memory space. In comparison, the Ardent traces average over one megabyte of referenced space, and many of the Ardent programs have not even reached their full size by the end of the trace.

insert table2.ps here.

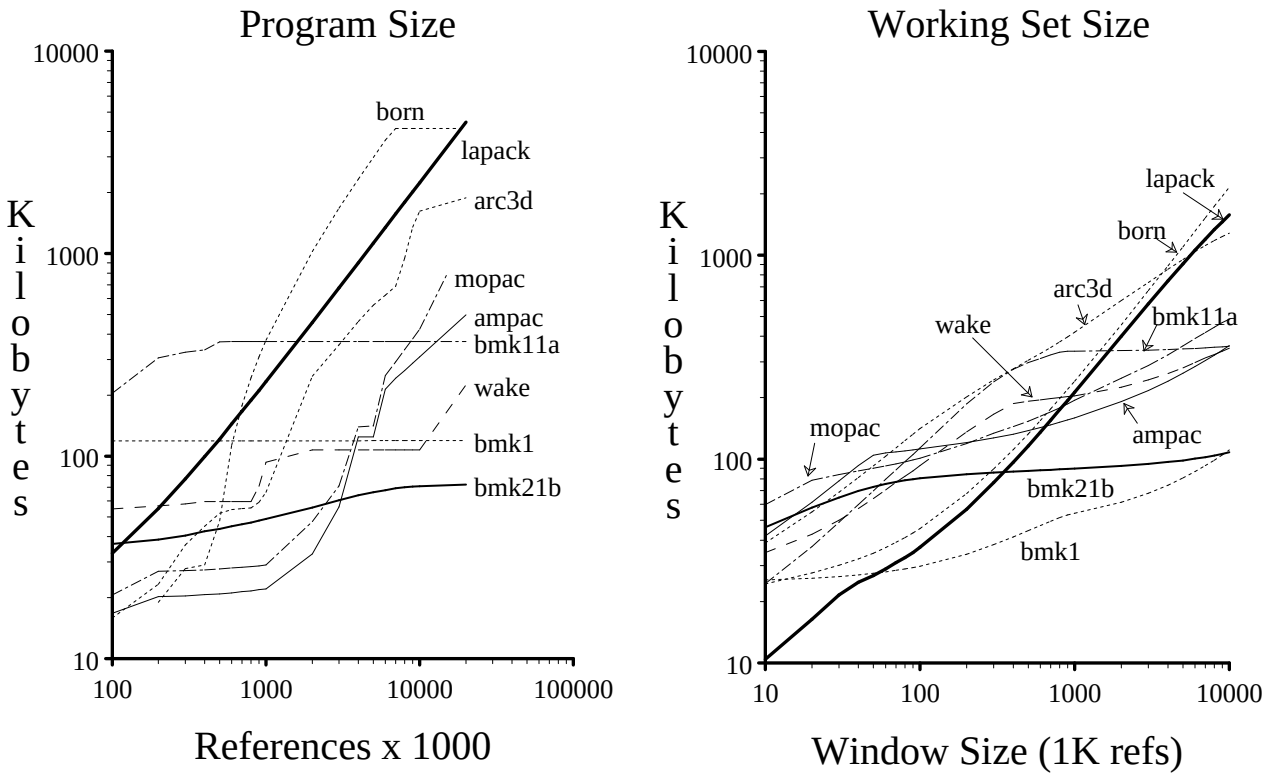

Figure 1: Ardent program sizes as a function of time (top) and working set sizes as a function of the window size parameter (bottom).

Figure 1 displays the cumulative address space size referenced as a function of time for various Ardent programs. Programs such as *born*, *bmk1*, *bmk11a*, and *bmk21b* appear to have reached maximum size by the end of the trace, while *arc3d*, *ampac*, *lapack*, *mopac*, and *wake* continue to reference new data. Working set sizes in the same figure increase with window size, but remain for the most part below 500 kilobytes for window sizes as large as 10 million references.

Tables 3 and 4 list the vector lengths and strides observed in all of the the traces containing vector references. Averages and distributions are provided for individual traces and for all traces collected on a given machine. In this context, vector lengths represent the length of individual vector load and store instructions, which is constrained to some maximum number (64 in the Cray X-MP, 32 in the Ardent Titan). Actual vector lengths coded in the applications may even be larger, but such vectors are processed in strips. Vector strides are the distances (in words) between memory addresses of successive vector elements; vectors of unit stride are contiguous. Geometric as well as linear averages are listed for strides, as the presence of only a few large strides can greatly skew linear averages.

Vector Lengths in the Cray Traces									
length	loop14 %	loop16 %	loop21 %	loop8 %	nas1 %	nas2 %	nas5 %	nas7 %	All Programs %
3	2.1	45.3	-	-	-	-	1.8	-	0.2
5	-	22.1	-	-	-	-	2.3	-	0.1
7	-	10.5	-	-	-	-	2.0	-	0.1
8	-	-	-	-	-	73.6	1.8	-	10.8
16	-	-	-	-	-	24.9	1.5	-	3.7
19	-	-	-	-	-	-	19.6	-	0.8
20	-	-	-	-	-	-	3.4	-	0.1
25	-	-	99.2	-	-	-	2.3	-	12.5
32	-	-	-	-	98.2	-	3.5	91.6	43.9
35	-	-	-	49.3	-	-	3.3	-	11.3
41	5.6	-	-	-	-	-	-	-	0.1
64	90.1	9.3	-	49.8	-	-	-	7.7	16.1
averages	60	11	25	49	32	10	24	34	33
Vector Strides in the Cray Traces									
stride	loops14 %	loops16 %	loops21 %	loops8 %	nas1 %	nas2 %	nas5 %	nas7 %	All Programs %
1	99.8	90.7	0.7	15.4	99.6	17.7	18.9	8.3	24.9
2	-	-	-	-	-	61.5	5.7	-	8.9
5	-	-	-	84.6	-	-	-	-	18.5
8	0.2	9.3	-	-	0.4	0.4	1.3	0.2	0.2
16	-	-	-	-	-	20.4	-	-	2.9
25	-	-	99.2	-	-	-	-	-	12.0
32	-	-	-	-	-	-	-	91.6	29.4
40	-	-	-	-	-	-	74.2	-	3.0
lin. avg.	1.0	1.7	24.8	4.4	1.0	4.7	30.1	29.4	15.5
geo. avg.	1.0	1.2	24.4	3.9	1.0	2.7	16.5	24.0	7.1

Table 3: Vector lengths and strides in the Cray traces.

The Cray benchmarks tend to operate on only a few different vector lengths, with the exception of *nas5*. Cray vector lengths are also short for the most part, averaging less than half the maximum vector length on the X-MP. The average Cray stride is quite large due to large values in *loops21*, *nas5*, and *nas7*, but geometric averages are lower due to short strides in *loops14*, *loops16*, and *nas1*. Geometric and linear averages for *individual* Cray traces are approximately equal, as strides within a Cray kernel are fairly uniform.

Average vector lengths in the Ardent traces are close to the maximum vector length on the Titan. Ardent strides, in comparison to Cray strides, are less uniform. Ardent linear averages are larger than Cray linear averages, but this is due more to the presence of outlying values rather than to poorer locality in the Ardent programs. Since the Ardent programs are real applications, they are presumably more representative of the relevant workload.

Vector Lengths in the Ardent Traces								
length	arc3d %	bmk1 %	bmk11a %	flo82 %	lapack %	simple %	wake %	All Programs %
2	-	-	-	5.7	-	-	-	1.1
4	-	-	-	13.7	-	-	-	2.7
8	-	-	-	30.3	3.1	-	-	6.1
9	-	-	-	3.7	-	42.4	-	11.5
10	-	-	-	4.5	-	-	-	0.9
15	-	-	-	-	-	-	3.8	0.3
20	-	98.6	-	14.1	-	-	-	12.6
21	-	-	-	3.2	-	-	-	0.6
26	5.5	-	-	-	-	-	-	0.9
28	55.0	-	-	-	-	-	-	8.7
29	10.1	-	-	-	-	-	-	1.6
30	26.1	-	-	-	-	-	-	4.1
31	-	-	-	-	-	46.2	11.9	12.6
32	-	-	100.0	20.6	96.9	11.2	83.9	36.3
averages	28	20	32	15	31	22	31	24
Vector Strides in the Ardent Traces								
stride	arc3d %	bmk1 %	bmk11a %	flo82 %	lapack %	simple %	wake %	All Programs %
1	98.6	50.4	100.0	74.5	100.0	55.4	99.5	78.8
2	-	-	-	0.8	-	-	-	0.2
12	-	-	-	-	-	21.2	-	5.3
20	-	24.8	-	-	-	-	-	2.5
30	0.9	-	-	-	-	-	-	0.1
34	-	-	-	-	-	23.4	-	5.9
48	-	-	-	-	-	-	0.2	0.0
90	0.5	-	-	-	-	-	-	0.0
385	-	-	-	3.9	-	-	-	0.8
386	-	-	-	20.7	-	-	-	4.0
961	-	-	-	-	-	-	0.3	0.0
1351	-	24.7	-	-	-	-	-	2.4
lin. avg.	1.9	339.8	1.0	95.6	1.0	11.1	4.4	55.4
geo. avg.	1.1	12.5	1.0	4.4	1.0	3.9	1.0	2.4

Table 4: Vector lengths and strides in the Ardent traces.

4. Locality in Vectorized Applications

This section directly examines the temporal and spatial locality present in the address traces. We characterize temporal locality by using *reference counts* and *reference intervals*, which respectively represent (a) the number of references to an address, and (b) the times between references to an address. Applications containing temporal locality have high reference counts and short reference intervals, as data is referenced repeatedly within small windows of time. We characterize spatial locality by examining the decrease in cache miss ratio with increasing block size. The greater the degree of spatial locality, the more quickly the miss ratio

will decrease with increasing block size.

Our various analyses have been run on each individual trace; we've condensed those analyses into three workloads from the Cray traces and two from the Ardent traces. The Cray traces are divided into a *Livermore Loops* workload, a *NAS Kernel* workload, and a *scalar* workload consisting of the remaining traces. The first two workloads are heavily vectorized, while the third workload consists mainly of scalar benchmarks plus *linpack*. The Cray linpack trace was generated for a small 10x10 dataset and thus is predominantly scalar. Throughout this and remaining sections of the paper, Cray references are analyzed with consecutive instruction parcels packed into eight-byte quantities, i.e. an instruction parcel fetch does not occur until parcels outside the current eight-byte word are needed. This packing of instructions is consistent with the actual X-MP hardware, which fetches both instructions and data in eight-byte quantities.

The Ardent traces with significant vectorization (arc3d, bmk1, bmk11a, flo82, lapack, simple, and wake) comprise an Ardent vectorized workload, while the remaining Ardent traces (ampac, bmk21b, born, mopac) comprise an Ardent scalar workload. Unlike the Cray traces, we analyze the Ardent traces without normalizing to a single reference width, as the Ardent Titan references both four and eight-byte data items over its eight-byte bus. The Titan fetches instructions four bytes at a time from its instruction cache, which is how we model the instruction stream.

4.1. Reference Counts

Reference counts were obtained by measuring the number of individual (1) instruction, (2) scalar, and (3) vector references to each memory location. This separates reference counts for what we call *shared* data (Table 2) into scalar and vector components, allowing us to draw comparisons between scalar and vector reference locality. We later combine counts for all reference types to examine the overall temporal locality in vector applications.

Figure 2 presents reference counts for the individual Ardent workloads, the entire set of Ardent traces, and the entire set of Cray traces. (Results for individual Cray workloads, are left to the Appendix). Each graph in Figure 2 shows reference counts for instruction, scalar, vector, and all references. The point (X,Y) on the graph signifies that a fraction **Y** of all memory locations are referenced up to **X** times during the course of the trace.

Vector reference counts in the Cray benchmarks are very high, but are an artifact of the large loop indices required to benchmark accurately with the Livermore and NAS kernels. Vector reference counts in the Ardent traces, while lower, are more likely to reflect the true amount of data reuse in vectorized workloads. Note that there is reuse of vector data, as vector reference counts in the Ardent vector workload actually exceed scalar reference counts. Mean, median, and 90 percentile reference counts for all workloads are summarized in Table 5.

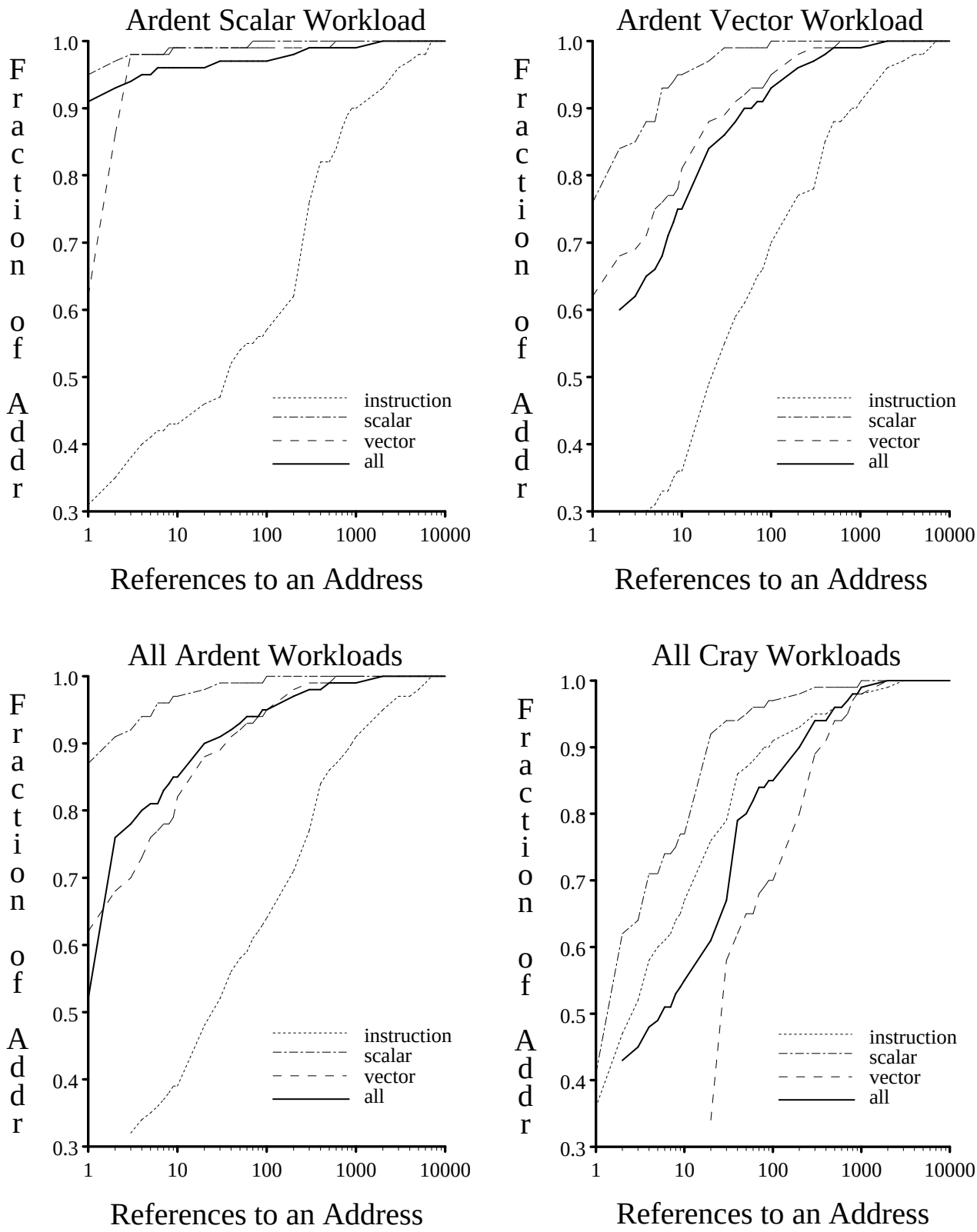

Figure 2: Cumulative reference count distributions for individual Ardent workloads, the combined Ardent workload, and the combined Cray workload.

Reference Count Summary by Workload						
Workload	*Instruction*			*Scalar*		
	Mean	Median	90 pct.	Mean	Median	90 pct.
Cray loops	55	3	23	17	4	20
Cray nas	54	4	55	14	1	16
Cray scalar	144	3	270	53	2	58
Cray (all)	99	3	80	26	2	20
Ardent vector	352	21	928	8	1	6
Ardent scalar	464	37	920	6	1	1
Ardent (all)	397	23	928	7	1	2
Workload	*Vector*			*All*		
	Mean	Median	90 pct.	Mean	Median	90 pct.
Cray loops	157	20	479	83	4	90
Cray nas	156	40	360	99	40	360
Cray scalar	59	2	169	89	2	104
Cray (all)	135	20	400	87	6	208
Ardent vector	22	1	32	51	2	60
Ardent scalar	8	1	3	30	1	1
Ardent (all)	22	1	32	40	1	20

Table 5: Reference Count Summary

Listed are mean, median, and 90 percentile *reference counts* (number of references to each memory location) for the Cray and Ardent workloads. Results are provided for individual workloads and for all traces collected on a given machine. Summaries are presented for instruction references, scalar references, vector references, and all references combined.

4.2. Reference Intervals

Reference intervals are the number of references between successive accesses to the same memory location. As a measure of temporal locality, reference intervals are more useful than reference counts because references that are closely clustered in time define a different type of locality of reference than the same number of references uniformly spaced over the traced interval. Programs contain large amounts of temporal locality if reference intervals are short, and low amounts if the opposite holds true. Data will remain cached during short reference intervals, but will often be pushed from the cache during longer intervals. One study [Sang84] found that data must be referenced more than 600 times per second to remain cached in the Amdahl 580.

We analyze reference intervals *separately* for instructions, scalar references, and vector references. For data referenced by both scalar and vector instructions, scalar reference intervals are the times between the *current* and *previous scalar* reference to that data. Similarly, vector

reference intervals are the times between the current and previous *vector* reference to that data. For *overall* results, we do not differentiate between data reference types and measure reference intervals between any two references to that data.

Figure 3 shows cumulative reference interval distributions for the individual Ardent workloads and the complete sets of Ardent and Cray traces (results for the individual Cray workloads are in the Appendix). Each graph contains reference interval distributions for instruction, scalar, vector, and all references. Mean, median, and 90 percentile reference intervals are listed in Table 6.

Reference Interval Summary by Workload

Workload	*Instruction*			*Scalar*		
	Mean	Median	90 pct.	Mean	Median	90 pct.
Cray loops	2034	46	873	13714	45	38395
Cray nas	2478	237	984	3094	169	6515
Cray scalar	1049	48	1086	1267	48	892
Cray (all)	1628	49	1086	6347	48	5472
Ardent vector	1992	98	1713	6456	103	3962
Ardent scalar	3915	234	9643	3719	138	4339
Ardent (all)	2776	156	3332	5243	121	4231
Workload	*Vector*			*All*		
	Mean	Median	90 pct.	Mean	Median	90 pct.
Cray loops	5058	2217	4444	4181	292	4444
Cray nas	9240	699	46521	7608	490	37089
Cray scalar	1740	410	1461	1109	48	1086
Cray (all)	7165	1086	32887	4502	237	8996
Ardent vector	45655	3917	268165	10456	150	9213
Ardent scalar	23613	8435	238069	3911	201	8620
Ardent (all)	45294	3962	265333	8072	160	8759

Table 6. Mean, median, and 90 percentile reference intervals (number of references between rereferences to data) for the Cray and Ardent workloads.

In the Cray programs, reference intervals for instructions and scalar data are short, while vector reference intervals are roughly an order of magnitude larger. Since vectorized applications operate on large data structures in a serial fashion, the times between successive references to each individual vector data item are consequently high. In processing each vector element however, the same instructions and scalar variables are being reused, which accounts for the higher temporal locality in these reference classes. Note, however, that median vector reference intervals are only a few thousand references, which is sufficiently short that vector data should remain in the cache between loop iterations.

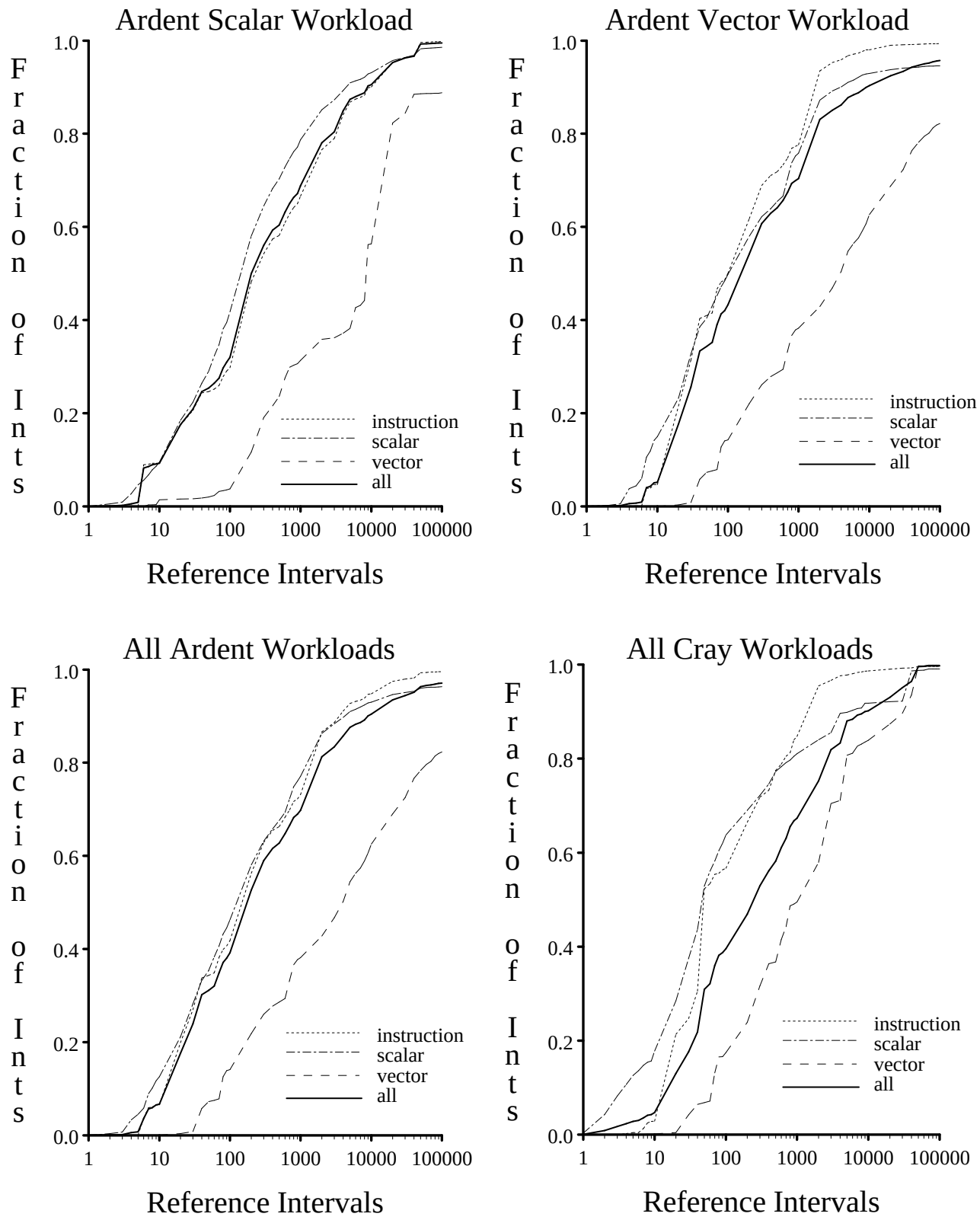

Figure 3: Cumulative distribution of time between successive references to the same memory location.

Results for the Ardent workloads are similar to the Cray results. Instruction and scalar reference intervals are both short, while vector reference intervals are roughly an order of magnitude larger than both. Reference intervals in the Ardent traces are also larger relative to the Cray traces, as the Ardent programs are real, long-running codes and not tightly-coded kernels or small benchmarks.

4.3. Ratios of Miss Ratios

To quantify spatial locality, we measured *ratios of miss ratios* [Smit87], which, for a particular cache and block size, is the ratio of its miss ratio to the miss ratio for the same-size cache with one-half the block size. Mathematically, the ratios of miss ratios for a cache of size C and block size B is given by $\frac{mr(C,B)}{mr(C,\frac{B}{2})}$ where *mr(C,B)* is the miss rate for a cache of size C with block size B. The ratio of miss ratios can range from a low of 0.5, when the level of spatial locality is very high, to well above 1.0 if there is little spatial locality; the ratio of miss ratios can be above one if the extra data fetched in large blocks is not used and only displaces useful data from the cache. This latter phenomenon, known as *memory pollution*, is most noticeable in smaller caches since increasing block size reduces the number of cache blocks below the number of distinct regions in concurrent use.

To obtain ratios of miss ratios, we measured cache miss ratios using LRU stack techniques. Simulations were run for block sizes from 16 to 128 bytes, covering a wide range of microprocessor and mainframe cache implementations. Fully-associative caches were simulated using fairly deep LRU stacks (500 elements). For still larger caches, set-associative simulations were performed to reduce overhead searching LRU stacks. Each set-associative cache consisted of 64 fully-associative stacks, with each stack containing up to 500 elements. Prior work [Smit82, Hill89] had found that set sizes beyond eight do little to improve cache miss ratios; thus our set-associative results should be virtually identical to results for fully-associative caches.

Figures showing ratios of miss ratios for various reference classes in the individual Cray and Ardent workloads are left to the Appendix. In the Cray workloads, ratios of miss ratios are low for instruction and scalar data references. Vector references, however, contain little spatial locality due to large strides in the Livermore and NAS kernels. We have noted that these kernels are designed for evaluating processor pipelines, and do not necessarily reflect the typical memory reference behavior of vector applications.

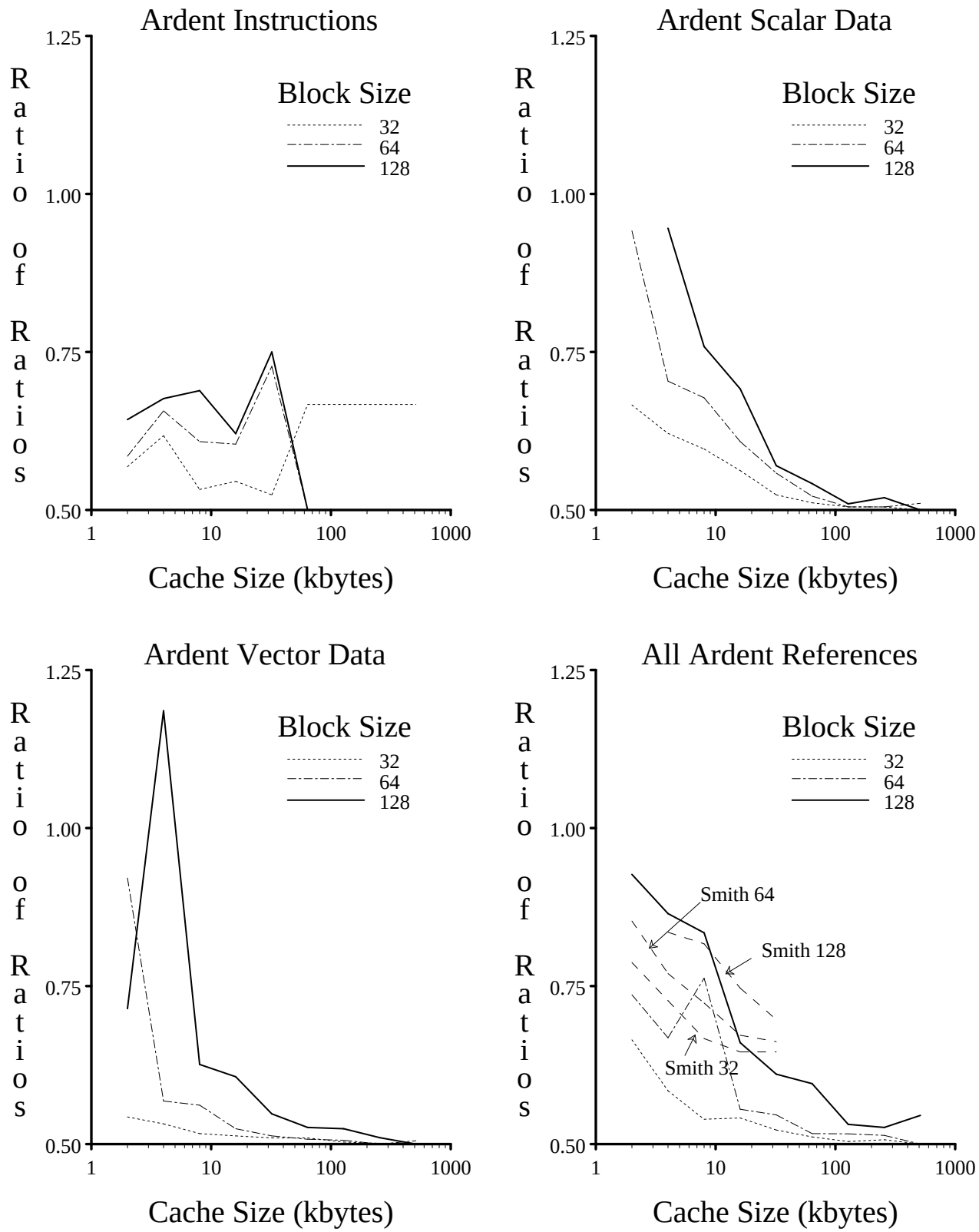

Figure 4: Ardent ratios of miss ratios for all workloads plotted on the basis of instructions, scalar data, vector data, and all references. The last graph also contains ratios of miss ratios from [Smit87].

Figure 4 shows the ratios of miss ratios across all Ardent traces, plotted on the basis of instruction, scalar, vector, and all references. The last graph in the figure also contains ratios of miss ratios measured for general-purpose, multiprogrammed workloads [Smit87]. Unlike the Cray results, spatial locality is present in all reference classes, and is particularly strong for vector references. This improved locality is mainly due to the shorter strides in the Ardent programs. Ratios of miss ratios for scalar and vector data become quite low once the cache size exceeds 16 Kbytes. In Figure 4d, measured ratios of miss ratios across all references are actually lower than results from [Smit87]; i.e. we find more locality in our workloads than was found previously for a non-vectorized workload.

In summary, in our analysis of locality we have found large amounts of spatial and temporal locality in instruction and scalar references within the Ardent and Cray traces. Vector references contain lesser amounts of temporal locality, although median vector reference intervals are short enough (several thousand references in the Ardent programs) that the bulk of the vector data should remain cached between periods of use. The Ardent traces show large amounts of spatial locality; there is less spatial locality in the Cray traces. We believe the Cray results to be anomalous due to the artificial memory reference behavior in the small kernels, whereas the Ardent results reflect the considerable spatial locality in real vector applications.

5. Cache Miss Ratios

In this section, we examine cache miss ratios over a wide range of cache parameters, to see if the measurements of locality yield the expected low miss ratios. Simulations were conducted using both the Ardent and Cray traces, although as in prior sections we will emphasize the Ardent results over the Cray results. To approximate actual hardware implementations, references to consecutive two-byte Cray instruction parcels are merged into one eight-byte word reference. The Ardent Titan, unlike the Cray X-MP, references both single-precision as well as double-precision data, thus Titan references are not normalized to a constant word size. We do not implement periodic cache flushing in our cache simulator, as uniprogrammed vector machines typically execute for long periods of time between interrupts.

5.1. Fully-Associative Caches

Three types of fully-associative caches were simulated: *instruction caches*, *data caches* (containing scalar and vector data), and *unified caches* (containing instructions, scalar data, and vector data). Block sizes range from 16 to 128 bytes, and the maximum cache size simulated is 4 Mbytes. To keep stack distances manageable, we did not simulate some combinations of large cache size and small block size.

5.1.1. Miss Ratios

Figure 5 shows miss ratios for fully-associative Ardent instruction, data, and unified caches, as well as for fully-associative Cray unified caches. Figures containing results for Cray instruction and data caches are left to the Appendix.

Figure 5 also shows (a) design target miss ratios [Smit87] and (b) SPEC miss ratios [Gee91] averaged over the SPEC [SPEC89] floating-point application subset. All miss ratios are for fully-associative caches except for the SPEC results, which are plotted for a set size of eight, the maximum measured. The design target miss ratios, measured from address traces for a variety of scalar machines, represent the cache performance in general-purpose, multiprogrammed environments. The SPEC miss ratios were measured in a uniprogrammed environment, and should provide a better basis for comparison against our uniprogrammed vector workloads. We note that the SPEC floating-point miss ratios are significantly higher than miss ratios for the SPEC integer benchmarks [Pnev90].

In the figure, both the Ardent and SPEC instruction cache miss ratios fall more rapidly than design target miss ratios as cache size increases. Unlike the workloads used to measure design target miss rates, the Ardent and SPEC workloads are uniprogrammed, and thus do not contain any multiprogramming cache misses.

Ardent data cache miss ratios agree reasonably well with the design targets, while SPEC data cache miss ratios tend to be larger by as much as a factor of three. The higher SPEC miss rates are likely due to inherent differences between workloads. What is more interesting is that data cache miss ratios for real vector workloads are actually quite low; they compare favorably with miss ratios for general-purpose workloads (the design targets) and with miss ratios for the popular floating-point benchmark suite (SPEC).

For unified caches, design target miss ratios are slightly higher than Ardent miss ratios, again due to multiprogramming effects. SPEC unified cache miss ratios are also higher than Ardent miss ratios, due to a higher data component of the miss rate. For the Cray programs, unified cache miss ratios are larger than SPEC and design target miss rates at small cache sizes, since the Cray kernels contain little spatial locality. We also observe that Cray miss ratios drop dramatically once these kernels completely fit into the cache.

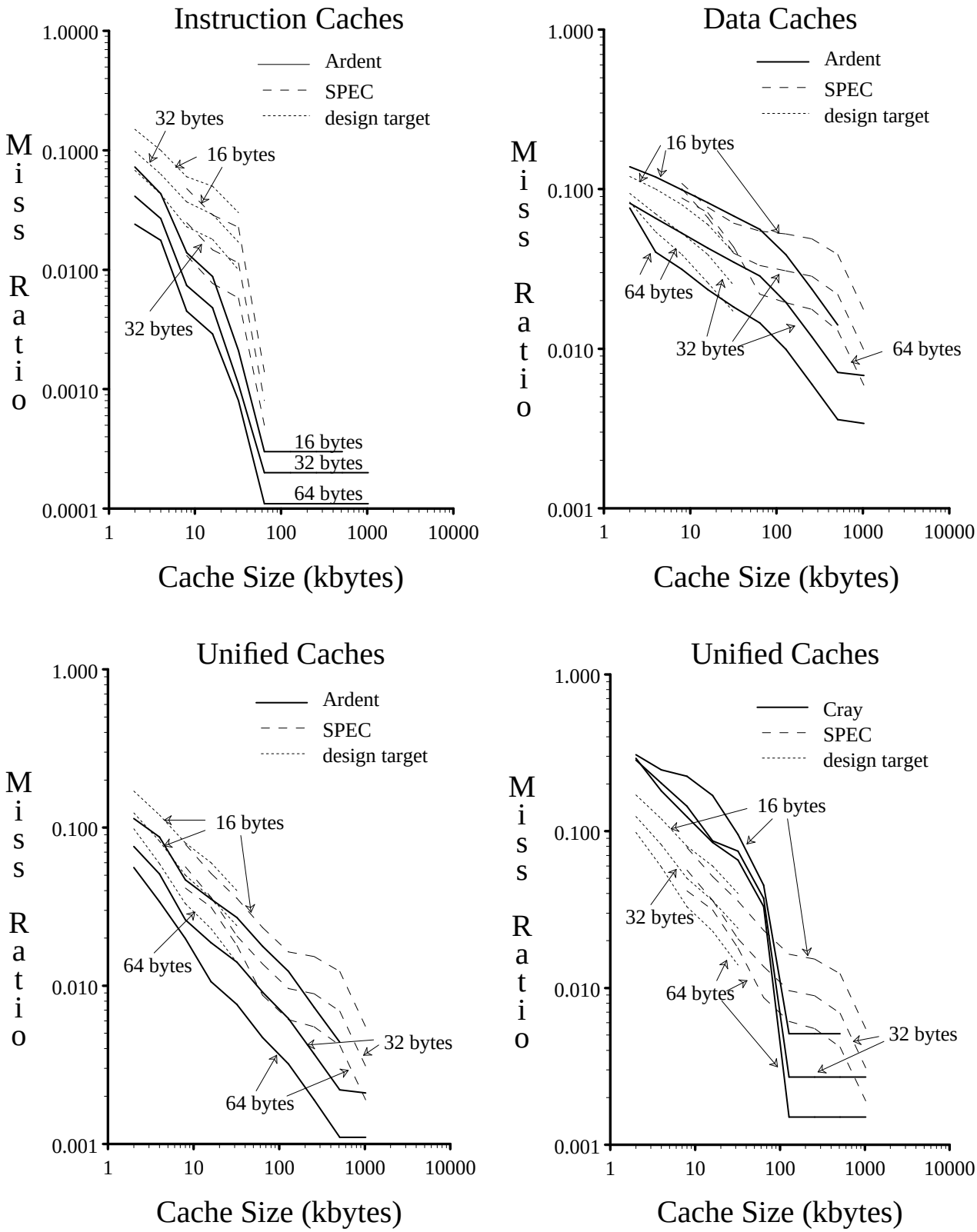

Figure 5: Fully-associative cache miss ratios plotted against design target and SPEC floating-point miss rates.

We can make a couple of observations from these results:

(a) The Ardent traces reference significant amounts of data, as data and unified cache miss ratios continue to decrease until the cache size exceeds 512K. Traces used in previous cache simulation efforts [Hill87, Hill89, Pnev90, Przy88, Smit82, Smit85, Smit87] have referenced much less data. Bypassing the trace storage process completely and simulating references during trace generation [Borg90, Gee92, Gee91, So88a, So88b] can allow even larger problem sizes (e.g. [Borg90]) to be observed. However, the large increase in CPU time to simulate these longer traces limits the design space that can be explored.

(b) Ardent miss ratios level off beyond caches larger than one half megabyte. This cache size is large enough to capture most, if not all of the locality within these programs. There is even sufficient locality in the Ardent programs to make use of smaller caches. Data cache miss ratios fall below 1 percent for cache sizes as small as 64 or 128 Kbytes.

In generating these results, we have assumed that Ardent data references can be either four or eight-byte quantities. To compare Ardent results more directly with design target and SPEC miss ratios, Ardent Titan simulations were repeated with data references *normalized* to a four-byte reference width, i.e. double precision data references are split into two four-byte halves. Results for Ardent data and unified caches, listed in the Appendix, show that normalizing data references to a four-byte word size reduces data and unified cache miss ratios by some 30 and 10%, respectively. The reduction arises because the second half of each double-precision data reference always hits in the cache. Ardent data cache miss ratios fall to roughly 70% of design target miss ratios and less than 50% of SPEC data cache miss ratios, while unified cache miss ratios fall to less than 50% of design target and SPEC miss rates.

5.1.2. Ratios of Miss Ratios

Cache miss ratios are often workload dependent. A more stable measure of cache performance is the *ratio of miss ratios*, which we used in Section 4 to analyze spatial locality. Ratios of miss ratios less than one indicate that doubling the cache block size decreases miss ratios. However, whenever the ratio of ratios exceeds 0.5, any decrease in cache miss ratio is also accompanied by increased fetch traffic.

Ratios of miss ratios for fully-associative unified caches are shown in Table 7 (instruction and unified cache results are in the Appendix). The table also includes ratios of ratios from [Smit87] for comparison. Ardent ratios of miss ratios are consistently 12 to 15 percent *smaller* than ratios of miss ratios from [Smit87], due to the large amount of spatial locality in this

Unified Cache Ratios of Miss Ratios									
Cache Size	*Ardent*			*Cray*			*[Smit87]*		
	Block Size			Block Size			Block Size		
	32	64	128	32	64	128	32	64	128
128	0.753	0.941	1.801	0.861	0.871	1.110	0.942	1.184	2.046
256	0.685	0.942	1.078	0.860	0.836	0.952	0.822	1.162	1.453
512	0.668	0.776	0.967	0.950	0.819	0.878	0.767	0.914	1.451
1024	0.732	0.729	0.886	0.955	0.856	0.876	0.831	0.880	1.023
2048	0.665	0.736	0.927	0.923	1.028	0.930	0.731	0.787	1.132
4096	0.584	0.668	0.865	0.824	0.893	1.187	0.719	0.746	0.809
8192	0.558	0.763	0.834	0.646	0.857	1.028	0.645	0.661	0.753
16384	0.530	0.567	0.660	0.513	0.979	1.007	0.616	0.633	0.685
32768	0.522	0.539	0.618	0.784	0.197	2.878	0.601	0.601	0.660
65536	0.511	0.517	0.553	0.825	0.887	0.160	-	-	-
131072	0.504	0.516	0.531	0.529	0.556	0.533	-	-	-
262144	0.507	0.514	0.526	0.529	0.556	0.533	-	-	-
524288	0.500	0.500	0.546	0.529	0.556	0.533	-	-	-

Table 7: Unified Cache Ratios of Miss Ratios

workload. Cray ratios of miss ratios are about 15 percent larger for unified caches, but agree fairly well with [Smit87] for instruction and data caches. These results, in conjunction with the miss ratios presented earlier, indicate that the cache performance of vector workloads is comparable to that of any other workload which normally benefits from data caching.

5.2. Set-Associative Caches

Set-associative caches have been studied in detail in the literature (see e.g. [Smit82], [Hill89]); it has been observed that caches with moderate levels of associativity (4-way to 8-way) have miss rations nearly as low as for fully-associative caches. In [Hill89], it was found that reducing associativity from eight-way to four-way, four-way to two-way, and from two-way to direct-mapped causes relative increases of 5, 10, and 30 percent in miss ratio respectively. These increases in miss ratio were found to be consistent over a wide range of cache and block sizes. As discussed, the benefits of increasing associativity clearly diminish beyond a size size of eight.

The question is whether these results also apply for vector workloads. Two prior studies using vector applications [So88a, So88b] simulated three programs on a 64 Kbyte cache with set size varying from one to four. On average, their results agree with [Hill89]. As set size varied from four to two, and two to one, miss ratios increased by averages of 10 and 28 percent, respectively. Another study [Call90], however, found that increasing associativity has little effect on vector cache performance, and could even reduce performance in certain cases.

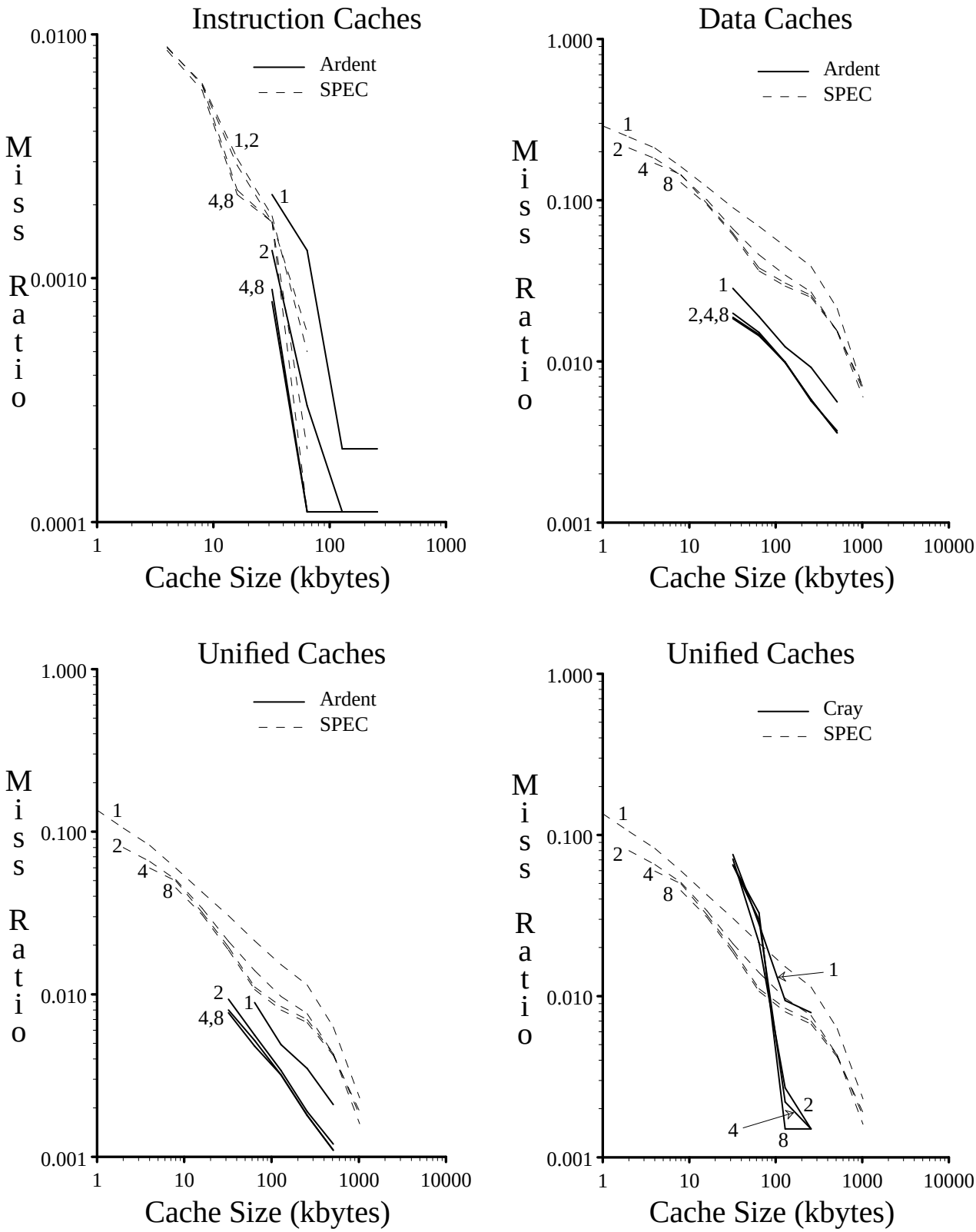

Figure 6: Set-associative Ardent and Cray cache miss ratios for a 64 byte block size, plotted against SPEC floating point miss ratios. Curves are labeled by set size.

In this section, we evaluate the effect of set-associativity in vector processor caches over a wide range of cache parameters, covering cache designs not examined in previous studies. Instruction, data, and unified caches up to four megabytes in size were simulated, with set size ranging from one to eight. We also compare LRU replacement to random replacement.

5.2.1. Miss Ratios

Figure 6 shows set-associative Ardent and Cray miss ratios vs. set-associative SPEC floating-point miss ratios for a block size of 64 bytes. Full tables of set-associative Ardent and Cray cache miss ratios are available in the Appendix.

Set-associative miss ratios for the Ardent and Cray traces decrease significantly when the set size is increased from one to two. Set sizes beyond two have much less effect, as only instruction cache performance improves noticeably. In contrast, SPEC floating-point miss ratios continue to drop as associativity increases, although most of the improvement is realized with a set size of two.

Ardent and Cray instruction cache miss ratios are roughly equal to SPEC instruction cache miss ratios, For data and unified caches, Ardent and Cray miss ratios are both lower than the SPEC miss ratios. The smaller Cray miss ratios, as we have noted before, are due to the small kernels being completely contained in caches larger than 64 Kbytes. The smaller Ardent miss ratios are likely due to workload differences, as we had observed similar results for fully-associative caches.

One interesting observation is that miss ratios for 128 Kbyte and 256 Kbyte Ardent data caches, as well as for 64 Kbyte Cray data caches, actually *increase* as set size increases beyond two. This same phenomenon was observed in one study [Call90], but not in another [So88a]. We suspect that the LRU replacement algorithm used in the simulations is poorly matched to the reference patterns typical of vector applications. When references are highly sequential (i.e. instruction and vector references) and the set size is not large enough to manage collisions, then LRU replacement will often remove blocks that are most needed in the near future [Smit83]. We compare LRU to random replacement in Section 5.2.3.

5.2.2. Miss Ratio Spreads

Miss ratio spreads enable us to isolate the relative effect of increasing associativity on cache miss ratios. The miss ratio spread between a 2n-way set associative cache and an n-way set associative cache is defined as $m(n)/m(2n) - 1$. Here $m(2n)$ is the miss ratio for the 2n-way set associative cache, and $m(n)$ is the miss ratio for the n-way set associative cache. Table 8 list miss ratio spreads for Ardent unified caches as set size varies from eight-to-four, four-to-two, and two-to-one. The data is smoothed with a weighted average of adjacent spreads as recommended

by [Cham83]. If *mrs*(*c*) is the miss ratio spread for a cache of size *c*, then the smoothed spread *mrs'*(*c*) is equal to 0.15**mrs*(*c*/4) + 0.20**mrs*(*c*/2) + 0.30**mrs*(*c*) + 0.20**mrs*(2*c*) + 0.15**mrs*(4*c*). Spreads for endpoint cache sizes are calculated with weights increased proportionately to sum to 1.0.

In a previous study [Hill89], smoothed miss ratio spreads were found to be fairly constant across cache size, block size, and cache type, averaging 5, 10, and 25 percent for changes in associativity from eight-to-four, four-to-two, and two-to-one, respectively. Our results are not as consistent, as miss ratio spreads for unified caches clearly increase with increasing block size. The spreads going from two-way to one-way associativity (direct mapping) are much larger compared to [Hill89], but spreads from eight-way to four-way and four-way to two-way associativity are much smaller. Ardent instruction caches (see Appendix) do seem to benefit more from increased associativity than Ardent data or unified caches, but instruction miss ratios are so low that any reduction in the number of misses appears insignificant.

Miss ratio spreads for the Cray traces, which are not given, are smaller than Ardent miss ratio spreads for instruction caches but larger for data and unified caches. Like the Ardent spreads, the Cray spreads vary significantly with cache and block size, but also suggest that set sizes beyond two or four are unnecessary in vector applications. This differs from earlier work in both scalar [Smit82, Hill89] and vector [So88a, So88b] environments, where increasing associativity beyond two continued to yield decreases in miss ratio.

Ardent Smoothed Miss Ratio Spreads for Unified Caches

Cache Size	Block Size: 16 Bytes			Block Size: 32 Bytes		
	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1
32K	0.019	0.056	0.296	0.020	0.080	0.381
64K	0.020	0.044	0.370	0.024	0.059	0.450
128K	0.010	0.040	0.414	0.014	0.050	0.488
256K	0.003	0.030	0.433	0.014	0.028	0.498
512K	-0.006	0.030	0.370	0.003	0.021	0.426
Cache Size	Block Size: 64 Bytes			Block Size: 128 Bytes		
	8-to-4	4-to-2	2-to-1	8-to-4	4-to-2	2-to-1
32K	0.044	0.113	0.521	0.051	0.197	1.292
64K	0.039	0.090	0.582	0.060	0.141	1.316
128K	0.023	0.083	0.609	0.046	0.104	1.295
256K	0.012	0.059	0.607	0.044	0.044	1.124
512K	0.000	0.048	0.526	0.022	0.018	0.905

Table 8: Smoothed miss ratio spreads for Ardent unified caches

5.2.3. Replacement Algorithms

While studying the effects of set-associativity, we observed that data cache miss rates for certain combinations of cache and block size data caches can actually *increase* with increasing associativity. For the Cray kernels, this increase is as large as 30 percent as the set size increases from two to four to eight. For the Ardent workload, based on real applications rather than small kernels, the increase is slight but apparent.

One paper [Smit83] noted that LRU and FIFO replacement can lead to 100 percent miss ratios in small, fully-associative instruction caches when program loops are larger than the cache size. Under such conditions, random replacement provides superior performance compared to both LRU and FIFO. While the data caches we study are much larger than the small instruction buffers analyzed in that study, we believe that similar effects are occurring within cache sets when the number of blocks which map into a set is larger than the set size.

To test this theory, set-associative data cache simulations were repeated for the Ardent traces using the stack-based implementation of random replacement described in [Matt70]. Figure 7 compares Ardent data cache miss ratios with random replacement vs. data cache miss ratios with LRU replacement. The various curves in the figure are parameterized by cache size. The figure shows that there is effectively *no consistent difference* in performance between random and LRU replacement. With random replacement, miss ratios no longer increase with increasing associativity, but neither is there a significant improvement in performance. LRU replacement performs slightly better than random replacement when the set size is small, while random replacement is slightly better for larger set sizes.

6. Estimated Performance Impact

So far we have provided fairly strong evidence supporting the use of vector caches. This evidence, however, consists mainly of time-independent metrics such as cache miss ratios and ratios of miss ratios. The performance impact of caching vector references on machine performance is the actual issue, and we consider this by using our measured cache miss ratios to estimate average memory access delays. Our results suggest that vector caches can significantly reduce the time needed to gain access to data in memory.

Before proceeding, we should mention that reducing memory access delays may not always translate into large performance gains in a vector machine. Vector machines typically reference long vectors in a pipelined fashion, which reduces the average delay seen per individual reference. Vector processors can also request data well before it is used, although this generally requires significant programming effort. Still, applications can often be coded to perform well in an environment where memory is many cycles away.

6.1. Mean Delay Per Memory Reference

We can compare the memory system performance of caching vs. non-caching vector processors by estimating the *average delay seen by each memory reference*. In a machine using a cache, the mean delay is approximately the product of the cache miss ratio and the time to service a cache miss. This estimate assumes that copy-back caches with write buffering are employed to minimize the effect of data writes on average delays [Smit79]. The time to service a cache miss is the sum of (a) memory system latency, and (b) cache block size divided by total memory bandwidth. Given a memory latency of L cycles, a block size of B bytes, a bus width of W bytes, and C cycles per bus transfer, the miss service time (in cycles) is equal to $L + C(\frac{B}{W})$. This calculation assumes no fetch bypass or wrap-around load; instead, the entire line is fetched and then accessed.

A non-caching vector processor encounters main memory delays at: (1) branches in the instruction stream, (2) the first element of a vector load, and (3) scalar loads. Instruction references following a branch and vector loads following the first load cause no additional delay, as these accesses are overlapped in a pipelined memory system. Note that assumption (1) is somewhat unfair by ignoring the use of commonly used instruction buffers. We make this assumption, however, to show that even small instruction caches are extremely effective relative to a machine which does not cache instructions. Similarly, (2) and (3) assume that a vector machine stalls on each pending scalar or vector load, which is clearly a worst-case scenario. We also assume that stores to memory are buffered, and only the occasional store which overflows the store buffers will cause delays. We assume that such stores make up 10 percent of the total [Smit79].

Two sets of parameters were used to compute a range of mean delays per memory reference for the Ardent traces. The first set of parameters assumes a memory latency of 14 cycles, an eight-byte bus, and a transfer rate of eight bytes per cycle. These values correspond closely to Cray X-MP and Ardent Titan memory system parameters. The second set of parameters assumes a memory latency of 45 cycles, an eight-byte bus, and a transfer rate of eight bytes per cycle. The longer memory latency corresponds roughly to the memory system parameters of a Cray 2.

Figures 8 and 9 show mean delays per memory reference for Ardent fully-associative instruction, data, and unified caches. The figures show average delay vs. cache block size, with curves in each figure parameterized by cache size. Each graph also shows mean delays per reference without caches for comparison. For data and unified caches, which support combinations of instruction, scalar, and vector references, mean delays without caches are separately plotted for the different reference classes.

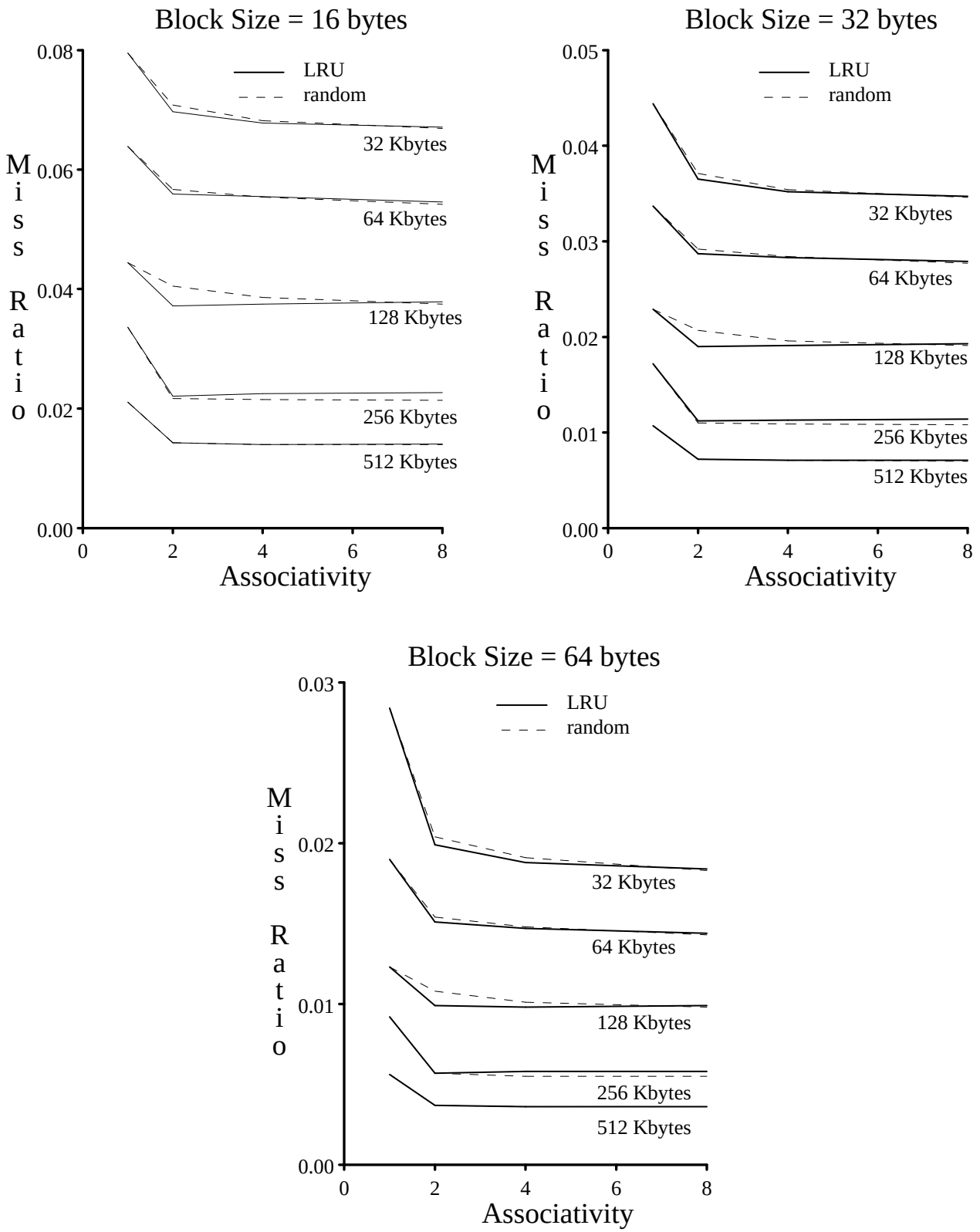

Figure 7: Set-associative Ardent data cache miss ratios for block sizes of 16, 32, and 64 bytes. For each cache size (labeled in kbytes), miss ratios are plotted for LRU and random replacement.

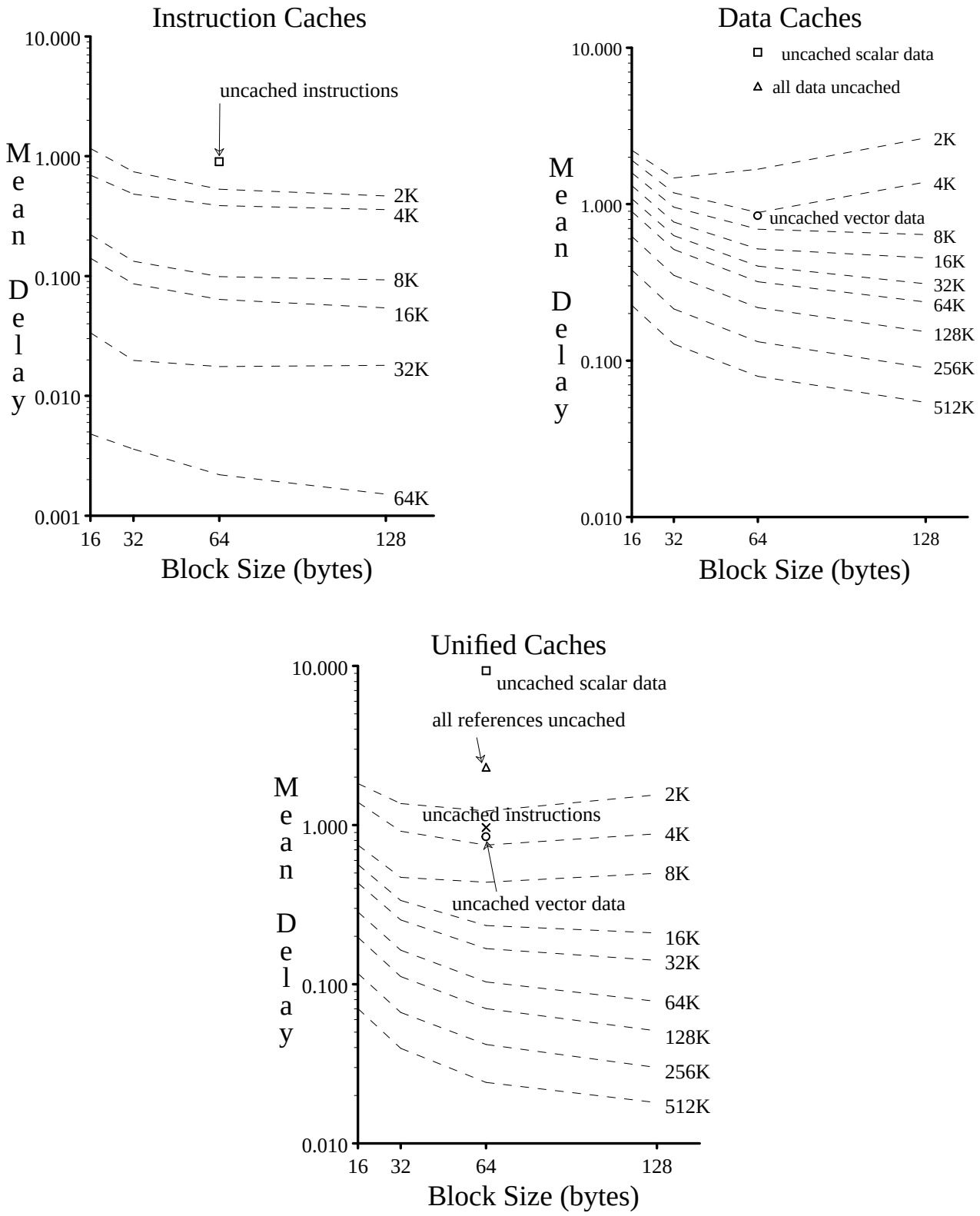

Figure 8: Mean delay per memory reference (cycles) for Ardent fully-associative instruction, data, and unified caches. Plots are labeled by cache size (Kbytes), and delays without caches are also shown for the different reference types. Memory latency is 14 cycles, and transfer time is 1 cycle per 8-byte word.

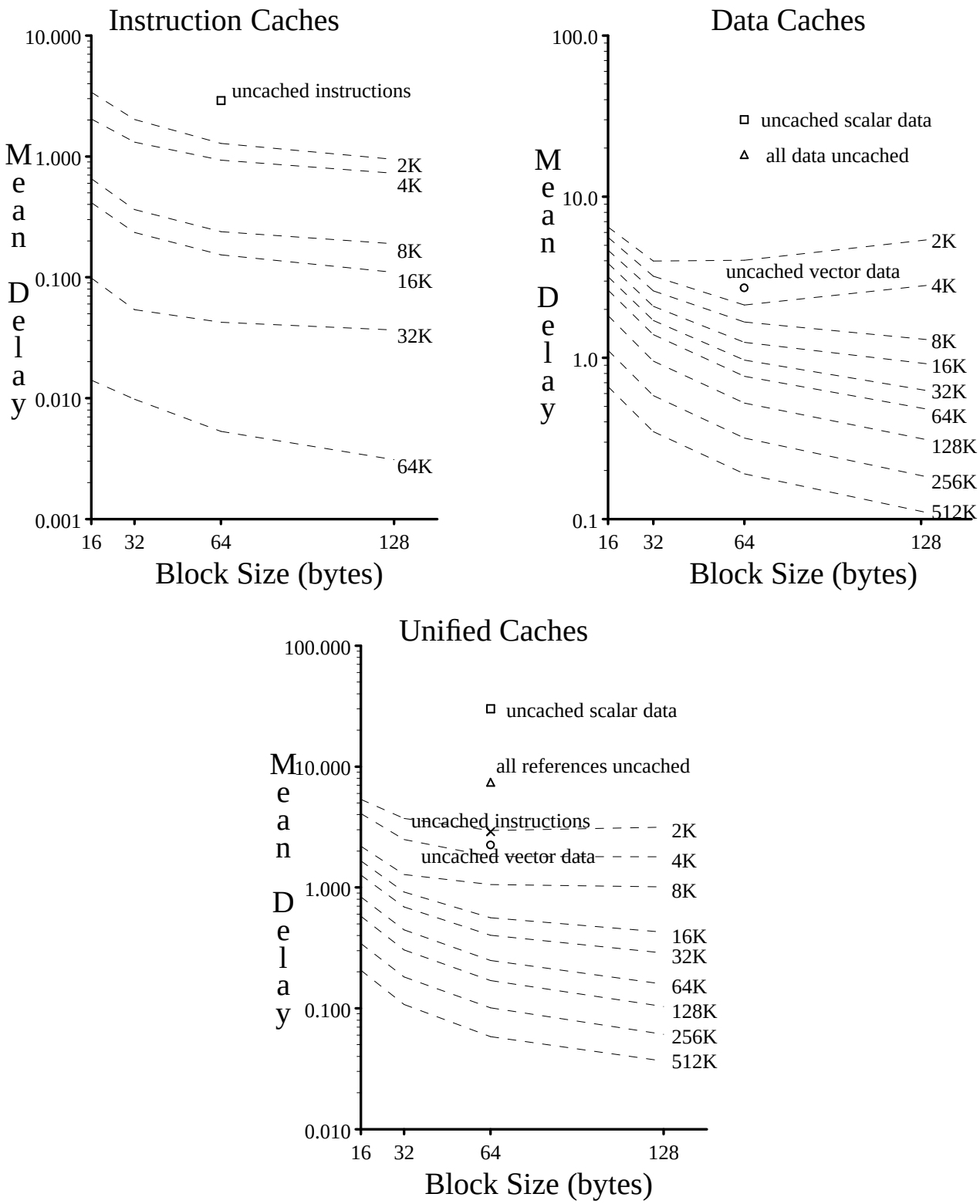

Figure 9: Mean delay per memory reference (cycles) for Ardent fully-associative instruction, data, and unified caches. Plots are labeled by cache size (Kbytes), and delays without caches are also shown for the different reference types. Memory latency is 45 cycles, and transfer time is 1 cycle per 8-byte word.

As the figures show, caches of even a few kilobytes can improve memory access performance relative to systems without caches. Scalar references benefit most, while instruction and vector references can be pipelined and thus require larger caches to realize performance improvements. Note that the average delay per uncached vector reference is actually quite low, as one might expect in carefully tuned vectorized applications. We note again that the Ardent traces are drawn from real production applications, and that care has (presumably) been taken to minimize memory access delays.

It is also interesting to note that because of very low miss rates, memory access delays in large caches are quite tolerant of increasing memory latency. Access delays remain reasonably stable as memory latency increases from 14 cycles to 45 cycles.

Block Sizes Which Minimize Mean Delay						
Cache Size	Ardent			[Smit87]		
	Instruction	*Data*	*Unified*	*Instruction*	*Data*	*Unified*
1K	128	32	64	64	32	32
2K	128	32	64	64	32	64
4K	128	64	64	128	64	64
8K	128	128	64	128	64	128
16K	128	128	128	128	64	128
32K	128	128	128	128	128	128
64K	128	128	128	-	-	-
128K	128	128	128	-	-	-
256K	128	128	128	-	-	-
512K	128	128	128	-	-	-

Table 9: Block sizes which minimize mean access delays for instruction, data, and unified caches. Results for the Ardent traces are identical for memory latencies of 14 and 45 cycles. Results from [Smit87] are for a memory latency of approximately 40 cycles.

We can also use average access delays to determine the best choice of block size for a particular cache size. While increasing block size usually reduces miss ratios, the improvement in miss ratio is offset somewhat by the increase in service time for a cache miss. Eventually, we reach a point where delays are minimized, and further increases in block size reduce performance. The block sizes which maximize performance, which are identical for both 14 and 45 cycle memory latencies, are presented in Table 9, along with results from [Smit87] for a memory latency of approximately 40 cycles. Our optimal block sizes are slightly larger than measured in [Smit87], due to the increased spatial locality in vector applications and our wider bus width (8 bytes vs. 4 bytes). Note that in both this and the earlier study, the maximum block size considered was 128-bytes; better performance might be obtained with still larger block sizes.

6.2. Machine Performance

Translating these observed reductions in mean access delay into performance speedups is difficult, as vector machines have the ability to overlap memory system delays with useful work. This section provides estimates for a worst-case scenario in which a vector machine always halts until the completion of each memory reference. Our model thus gives an upper bound on the performance effect of a vector cache; in another study which uses a detailed timing simulation [Gee92], we find quantitatively similar results.

At peak performance, a vector processor can produce and store floating-point results back to memory on every cycle. The average performance of a vector machine is well below peak due to a number of factors, including (1) nonvectorized and integer execution, (2) memory access delays, and (3) paging and I/O, We create a simple model which assumes that average performance is limited mainly by vector access delays. In other words, a vector machine can, on average, produce one result every $1 + D_{vector}$ cycles, where D_{vector} is the mean delay per vector reference. Therefore the average performance of a vector processor, as a fraction of peak performance, is given by the following equation:

$$\frac{P_{ave}}{P_{peak}} = \frac{1}{1 + D_{vector}}$$

While this model is clearly oversimplified, it has the useful property that it can be used to estimate the *maximum possible speedup* brought about by a vector cache. This maximum speedup is the ratio of the average performance with cache to the average performance without cache:

$$S_{\max} = \frac{1 + D_{vector\,(memory)}}{1 + D_{vector\,(cache)}}$$

This maximum speedup can be calculated using mean vector access delays from the previous section. From Table 10, the mean delay accessing each vector element from memory is (a) 0.85 cycles and (b) 2.7 cycles for memory latencies of (a) 14 cycles and (b) 45 cycles. A 512 KByte data cache can reduce the mean delay per vector reference to roughly 0.1 cycles. Substituting 0.85 cycles for $D_{vector\,(memory)}$ and 0.1 cycles for $D_{vector\,(cache)}$, we get a maximum cache-induced speedup of 1.7 for a memory latency of 14 cycles. For a memory latency of 45 cycles, substituting 2.7 cycles for $D_{vector\,(memory)}$ results in a maximum speedup of 3.4. Given the trend towards longer memory latencies, vector caches could improve performance by even larger factors in future machines.

Ardent Mean Delay Per Memory Reference Without Cache (cycles)		
Reference Type	Memory Latency (cycles)	
	14	45
instructions	0.8996	2.8916
scalar data	9.3525	30.0615
vector data	0.8471	2.7229
scalar and vector data	5.6728	18.2340
instructions and all data	2.3041	7.4061

Table 10: Mean delay per memory reference (cycles) without caches

We can validate these estimates against results from [Gee92], where we presented the results of detailed timing simulations of proposed vector cache machines based on the Ardent Titan. The proposed machine had a maximum execution rate of 128 megaflops, based on a 64 MHz vector pipeline. The memory latency was roughly 40 cycles. Machine models with caches up to 4 Mbytes were simulated, along with a model which referenced all data from memory. The average speedup measured over a wide range of applications was roughly 2x, and speedups up to 4x were measured on applications making large numbers of scalar references. This agrees reasonably well with the 3.4x speedup estimated here for a machine with a 45-cycle memory latency. Note that in one case the addition of a cache caused a decrease in performance; that was for an FFT which used a dataset larger than the cache size. Generally, such poor results can be avoided by using tiling type algorithms rather than ones that assume a uniform flat memory system.

7. Conclusions

We have provided a detailed evaluation of cache performance in a vector processor. Workloads were constructed from a large number of address traces from CRAY X-MP and Ardent Titan vector machines. The Titan traces in particular came from real workloads which were heavily used at Ardent Computer. Both vectorized as well as scalar workloads were analyzed. We began by examining the spatial and temporal locality in the workloads. We then measured cache miss ratios over a range of cache sizes, block sizes, and associativities. Finally, we estimated the performance impact of a vector cache using memory system parameters typical of current vector machines.

Temporal locality was characterized with reference counts and reference intervals; spatial locality was characterized with ratios of cache miss ratios. The results are reasonably consistent

between the Cray and Ardent traces. Large amounts of temporal and spatial locality are present in instruction and scalar references. Vector references contain lesser, but still significant amounts of temporal locality. Spatial locality in vector references is very strong in the Ardent traces, but weaker in the Cray traces due to large strides in the Cray kernels. Since the Cray kernels are small programs used to test CPU performance, we have more confidence in the Ardent results, which were measured from real production applications.

Cache miss ratios were measured for fully-associative caches ranging from 128 bytes to 4 Mbytes, with block size varying from 16 to 128 bytes. Compared to the *design target miss ratios* [Smit87], Ardent and Cray instruction and unified cache miss ratios are significantly lower, as multiprogramming misses are absent from uniprogrammed vector workloads. Ardent data cache miss ratios agree fairly well with the design targets, while Cray data cache miss ratios are nearly five times larger due to low spatial locality in Cray vector references. Ardent miss ratios are also lower than observed results for the SPEC floating-point suite of benchmarks, which were also analyzed in a uniprogramming environment.

We also examined the effect of set-associativity on cache performance, and found that set sizes larger than two do not have a significant effect on miss ratios. Under certain circumstances, data cache miss ratios actually *increased* slightly with increasing set size. We found that random replacement eliminates this phenomenon, although the average performance difference between the two replacement schemes is negligible.

Finally, we evaluated the performance effect of a vector cache by comparing *mean delays per reference*, which correspond to the average delay seen by each reference due to cache misses and memory system latency. Two sets of memory system parameters were chosen to represent a range of vector machines. Scalar references benefit most from data caches, as, unlike instruction or vector references, memory latency cannot be amortized over a string of contiguous references. Delays for instruction and vector references, while low, can also be reduced significantly by caching. The mean delays were then used to estimate the maximum performance improvement due to a vector cache. Estimated maximum speedups ranged from 1.7 to 3.4, based on memory latencies of 14 and 45 cycles. These estimates are in line with results from a detailed timing simulation study of vector cache machines [Gee92].

In summary, caches can significantly improve the performance of a vector processor. Locality in vector applications is sufficiently strong that even moderately-sized caches can improve the performance of vector references. Although it is possible to avoid caches by programming around long memory latencies, the time and cost of this process can be large, and counters the modern practice of substituting machine time for people time. Memory latencies are also increasing relative to processor speeds [Neve89], which complicates the tuning process and further supports the use of vector caches.

Bibliography

[Abu86] W. Abu-Sufah and A.D. Malony, ‘‘Experimental Results for Vector Processing on the Alliant FX/8,’’ CSRD Rpt. No. 539, University of Illinois, Urbana, IL, 1986.

[Bail85] D.H. Bailey and J.T. Barton, ‘‘The NAS Kernel Benchmark Program,’’ NASA Technical Memorandum 86711, August 1985.

[Borg90] A. Borg, R.E. Kessler, and D.W. Wall, ‘‘Generation and Analysis of Very Long Address Traces,’’ *Proc. 17th Int’l Symp. Comp. Arch.*, May, 1990, Seattle, WA, pp. 270-279.

[Cala85] D.A. Calahan, ‘‘An Analysis and Simulation of the CRAY X-MP Memory System,’’ *Proceedings of the First International Conference on Supercomputing Systems*, December, 1985, St. Petersburg, FL, pp. 568-574.

[Cala88] D.A. Calahan, ‘‘Performance Evaluation of Static and Dynamic Memory Systems on the CRAY-2,’’ *Proceedings of the 1988 International Conference on Supercomputing*, July, 1988, St. Malo, France, pp. 519-524.

[Call90] D. Callahan and A. Porterfield, ‘‘Data Cache Performance of Supercomputer Applications,’’ *Proceedings Supercomputing ‘90*, November, 1990, New York, NY, pp. 564-572.

[Cast87] M.J. Castillo, *Instruction and Address Tracing and Analysis For The CRAY X-MP*, Master’s Report, University of California, Berkeley, 1987.

[Cham83] J.M. Chambers, W.S. Cleveland, B. Kleiner, and P.A. Tukey, *Graphical Methods for Data Analysis*, Duxbury Press, Boston, 1983.

[Chas88] M. Chastain, G. Gostin, J. Mankovich, and S. Wallach, ‘‘The Convex C240 Architecture,’’ *Proc. Spring Compcon ’88*, February, 1988, San Francisco, CA, pp. 321-329.

[Cheu84] T. Cheung and J. Smith, ‘‘An Analysis of the CRAY X-MP Memory System,’’ *Proceedings of the 1984 International Conference on Parallel Processing*, August, 1984, Bellaire, MI, pp. 499-505.

[Clar86] R.S. Clark, and T.L. Wilson, ‘‘Vector System Performance of the IBM 3090,’’ *IBM Systems Journal*, vol. 25, no. 1, 1986, pp. 63-82.

[Died88] T. Diede, C. Hagenmaier, G. Miranker, J. Rubinstein, and W. Worley, ‘‘The Titan Graphics Supercomputer Architecture,’’ *Computer*, September 1988, pp. 13-30.

[Eoya88] C. Eoyang, R.H. Mendez, and O.M. Lubeck, ‘‘The Birth of the Second Generation: The Hitachi S-820/80,’’ *Proceedings Supercomputing ’88*, November, 1988, pp. 296-303.

[Gee91] J. Gee, M.D. Hill, D. Pnevmatikatos, and A.J. Smith, ‘‘Cache Performance of the SPEC Benchmark Suite,’’ U.C. Berkeley Technical Report no. UCB/CSD 91/648 and University of Wisconsin at Madison Computer Sciences Department Technical Report no. 1049, September, 1991.

[Gee92] J. Gee and A.J. Smith, ‘‘The Performance Impact of Vector Processor Caches,’’ *Proc. of the 25th Hawaii Int’l Conf. on System Sciences*, Kauai, HI, Jan. 1992, pp. I:437-448.

[Grif84] J.H. Griffin and M.L. Simmons, *Los Alamos National Laboratory Computer Benchmarking 1983.* Tech. Report LA-10151-MS, Los Alamos National Laboratory, June 1984.

[Hill87] M.D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, Ph.D. Dissertation, University of California, Berkeley, 1987.

[Hill89] M.D. Hill and A.J. Smith, ‘‘Evaluating Associativity in CPU Caches,’’ *IEEE Transactions on Computers*, vol. 38, no. 12, December, 1989, pp. 1612-1630.

[Lazo88] C. Lazou, *Supercomputers and their Use*, Oxford University Press, 1988.

[Lee84] J.K.F. Lee and A.J. Smith, ‘‘Branch Prediction Strategies and Branch Target Buffer Design,’’ *Computer*, January, 1984, pp. 6-22.

[Lube85] O. Lubeck, J. Moore, and R. Mendez, ‘‘A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2,’’ *Proceedings of the First International Conference on Supercomputing Systems*, December, 1985, St. Petersburg, FL, pp. 320-327.

[Matt70] R.L. Mattson, J. Gecsei, D.R. Slutz, and L.L. Traiger, ‘‘Evaluation Techniques for Storage Hierarchies,’’ *IBM Systems Journal*, 2, 1970, pp. 78-117.

[McKe69] McKellar and E.G. Coffman, Jr., ‘‘Organizing Matrices and Matrix Operations for Paged Memory Systems,’’ *Comm. ACM*, vol. 12, no. 3, 1969, pp. 153-165.

[McMa86] F.H. McMahon, ‘‘The Livermore Fortran Kernels: A Computer Test of the Floating-Point Performance Range,’’ Lawrence Livermore National

Laboratory, UCRL-53745, December 1986.

[Neve89] K.W. Neves, ‘‘Supercomputers: The Next Generation,’’ *Scientific Information Bulletin*, vol. 14, no. 4, 1989.

[Perl89] C.H. Perleberg and A.J. Smith, ‘‘Branch Target Buffer Design and Optimization,’’ Technical Report No. UCB/CSD 89/552, University of California, Berkeley, 1989.

[Pnev90] D. Pnevmatikatos, M. D. Hill, ‘‘Cache Performance of the Integer SPEC Benchmarks on a RISC,’’ *Computer Architecture News*, vol. 18, 2, June, 1990, pp. 53-68.

[Przy88] S. Przybylski, M. Horowitz, J. Hennesy, ‘‘Performance Tradeoffs in Cache Design,’’ *Proc. 15th Int’l Symp. Comp. Arch.*, May, 1988, Honolulu, HI, pp. 290-298.

[Sang84] J. Sanguinetti, ‘‘Program Optimization for a Pipelined Machine,’’ *Proc. 1984 Sigmetrics*, August, 1984, Cambridge, MA, pp. 88-93.

[Smit79] A.J. Smith, ‘‘Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write Through,’’ *Journal of the ACM*, vol. 26, no. 1, January, 1979, pp. 6-27.

[Smit82] A.J. Smith, ‘‘Cache Memories,’’ *ACM Computing Surveys*, vol. 14, no. 3, September, 1982, pp. 474-529.

[Smit85] A.J. Smith, ‘‘Cache Evaluation and the Impact of Workload Choice,’’ *Proceedings of the 12th International Symposium on Computer Architecture*, June, 1985, Boston, MA, pp. 64-73.

[Smit87] A.J. Smith, ‘‘Line (Block) Size Choices for CPU Cache Memories,’’ *IEEE Transactions on Computers*, vol. C-36, no. 9, September, 1987, pp. 1063-1075.

[Smit83] J.E. Smith, and J.R. Goodman, ‘‘A Study of Instruction Cache Organizations and Replacement Policies,’’ *Proc. 10th Int’l Symp. Comp. Arch.*, June, 1983, pp. 132-137.

[So88a] K. So and V. Zecca, ‘‘Cache Performance of Vector Processors,’’ *Proc. 15th Int’l Symp. Comp. Arch.*, May, 1988, Honolulu, HI, pp. 261-268.

[So88b] K. So and V. Zecca, ‘‘Program Locality of Vectorized Applications Running on the IBM 3090 with Vector Facility,’’ *IBM Systems Journal*, vol. 27, no. 4, 1988, pp. 436-452.

[SPEC89] SPEC newsletter, Vol 1, 1, Fall 1989.

[Triv77] K.S. Trivedi, ‘‘On the Paging Performance of Array Algorithms,’’ *IEEE Trans. Comp.*, vol. C-26, no. 10, October, 1977, pp. 938-947.

[Tuck86] S.G. Tucker, ‘‘The IBM 3090 System: An overview,’’ *IBM Systems Journal*, vol. 25, no. 1, 1986, pp. 4-19.

Appendix

This appendix contains various figures and tables omitted from the main text due to space limitations. Included are tables of miss ratios, ratios of miss ratios, and memory access delays. Some of this data is also present in the main text in graphical form.

In addition, there are a number of results here which pertain only to the Cray traces. As we considered these traces to be far less realistic than the Ardent traces, we omitted much of the Cray results from the main text but present them here for completeness.

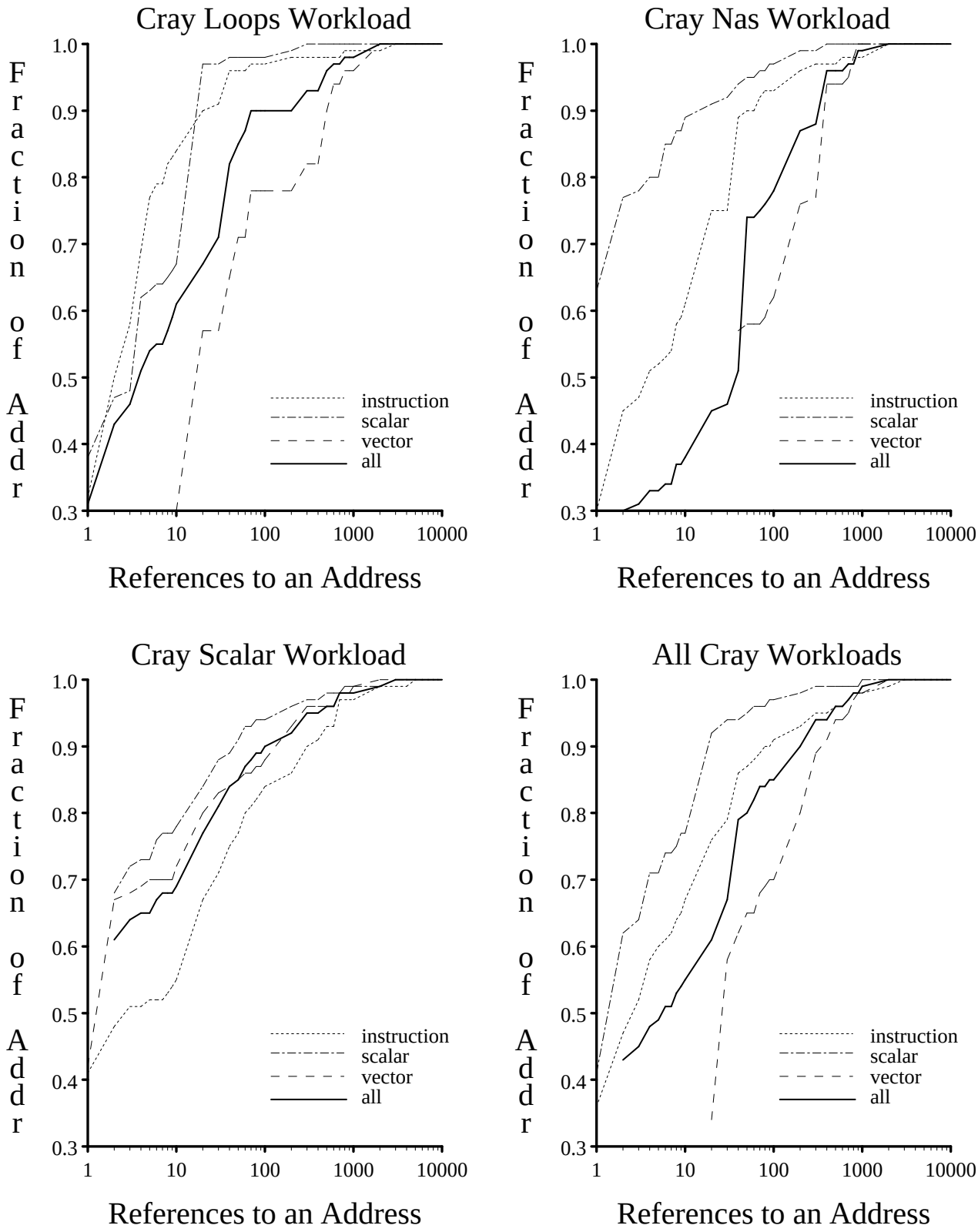

Figure A1: Cray reference counts for individual workloads and for all workloads combined.

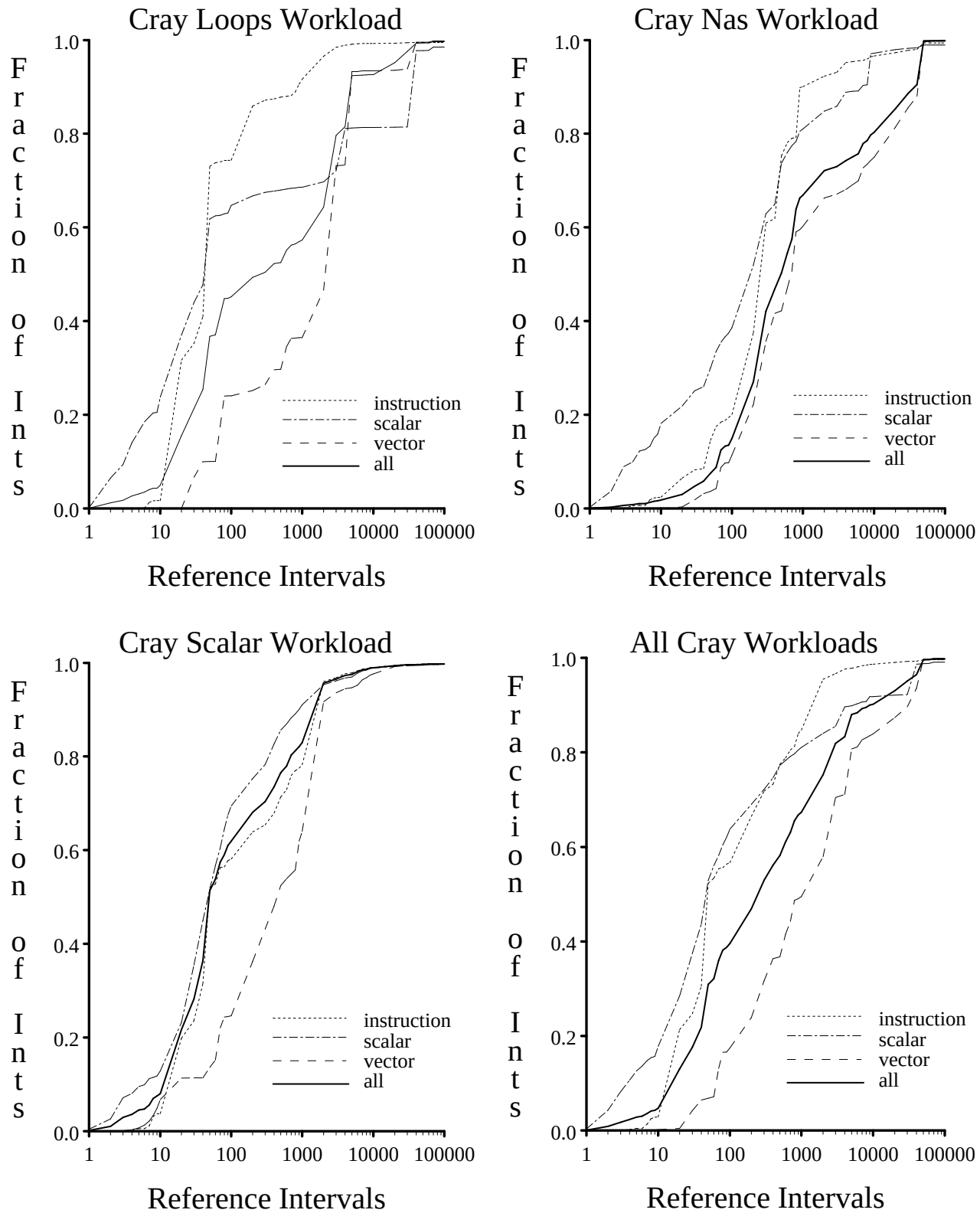

Figure A2: Cray reference intervals for individual workloads and for all workloads combined.

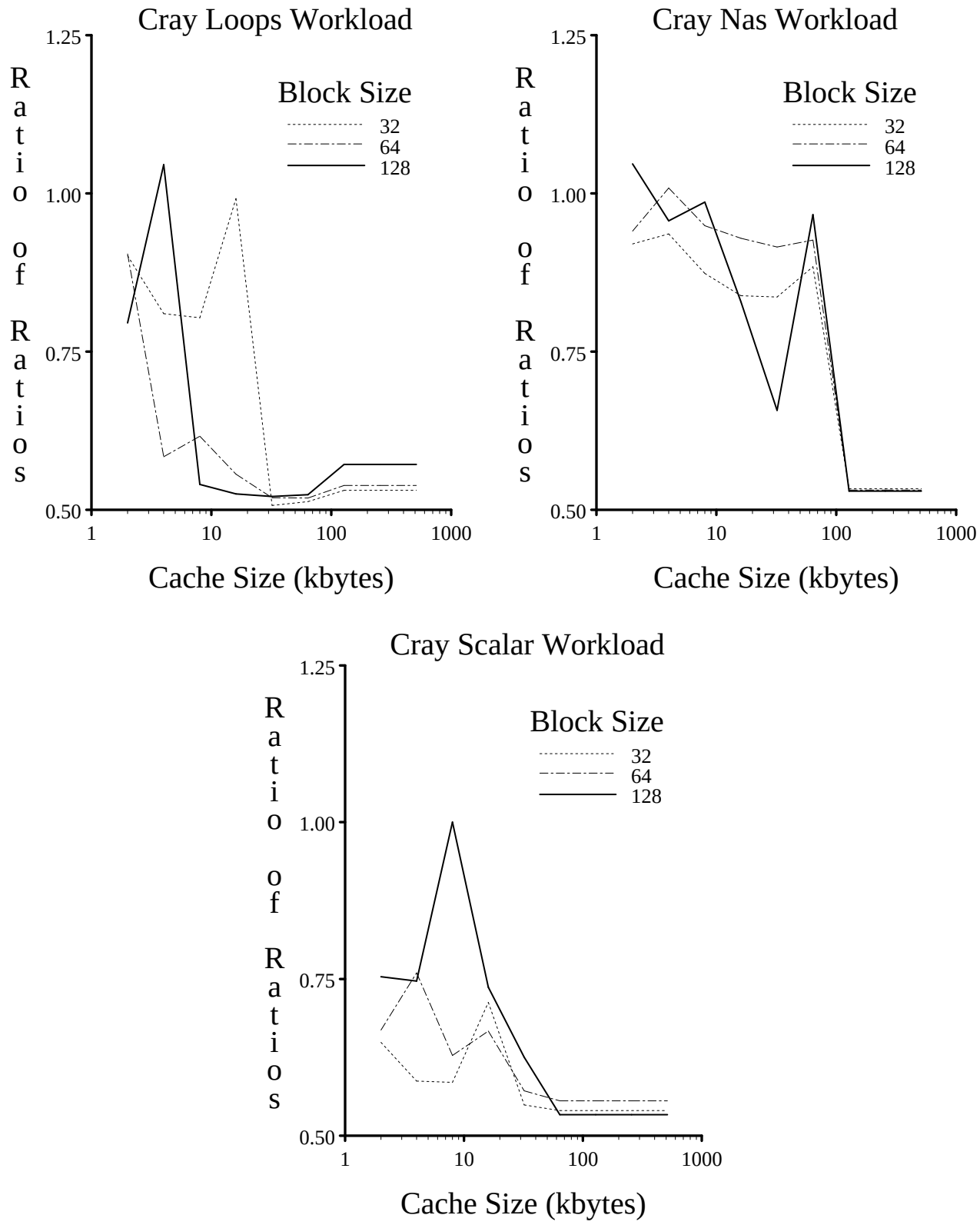

Figure A3: Cray ratios of miss ratios for individual workloads and for all workloads combined.

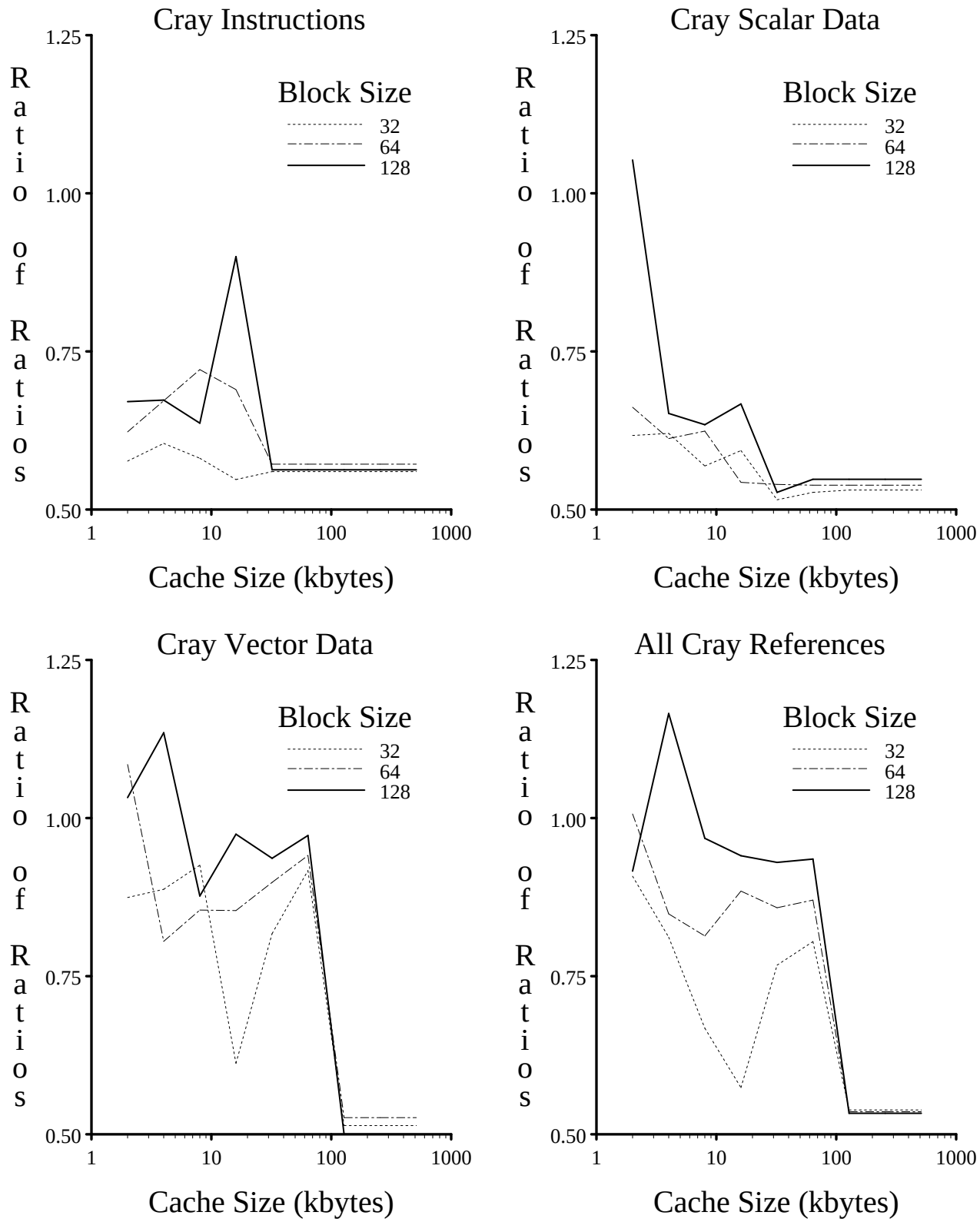

Figure A4: Cray ratios of miss ratios for all workloads plotted on the basis of instructions (upper left), scalar data (upper right), vector data (lower left), and all references (lower right).

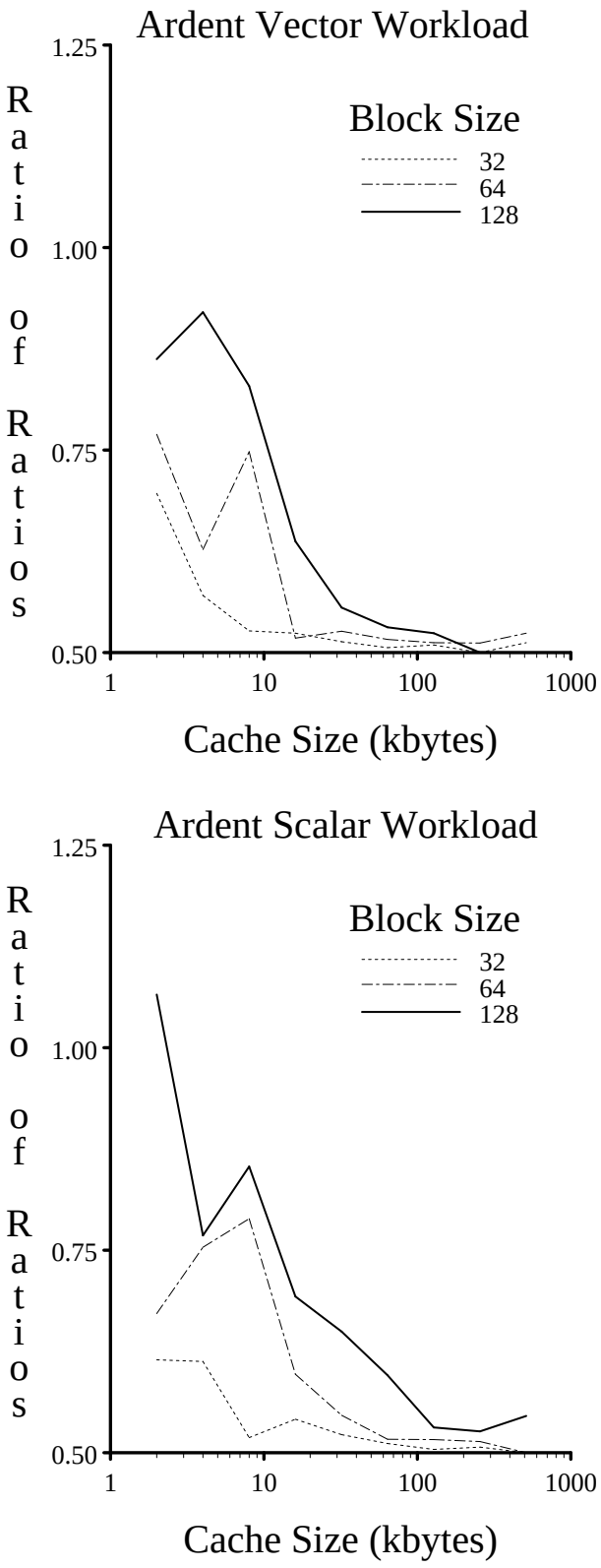

Figure A5: Ardent ratios of miss ratios for individual workloads.

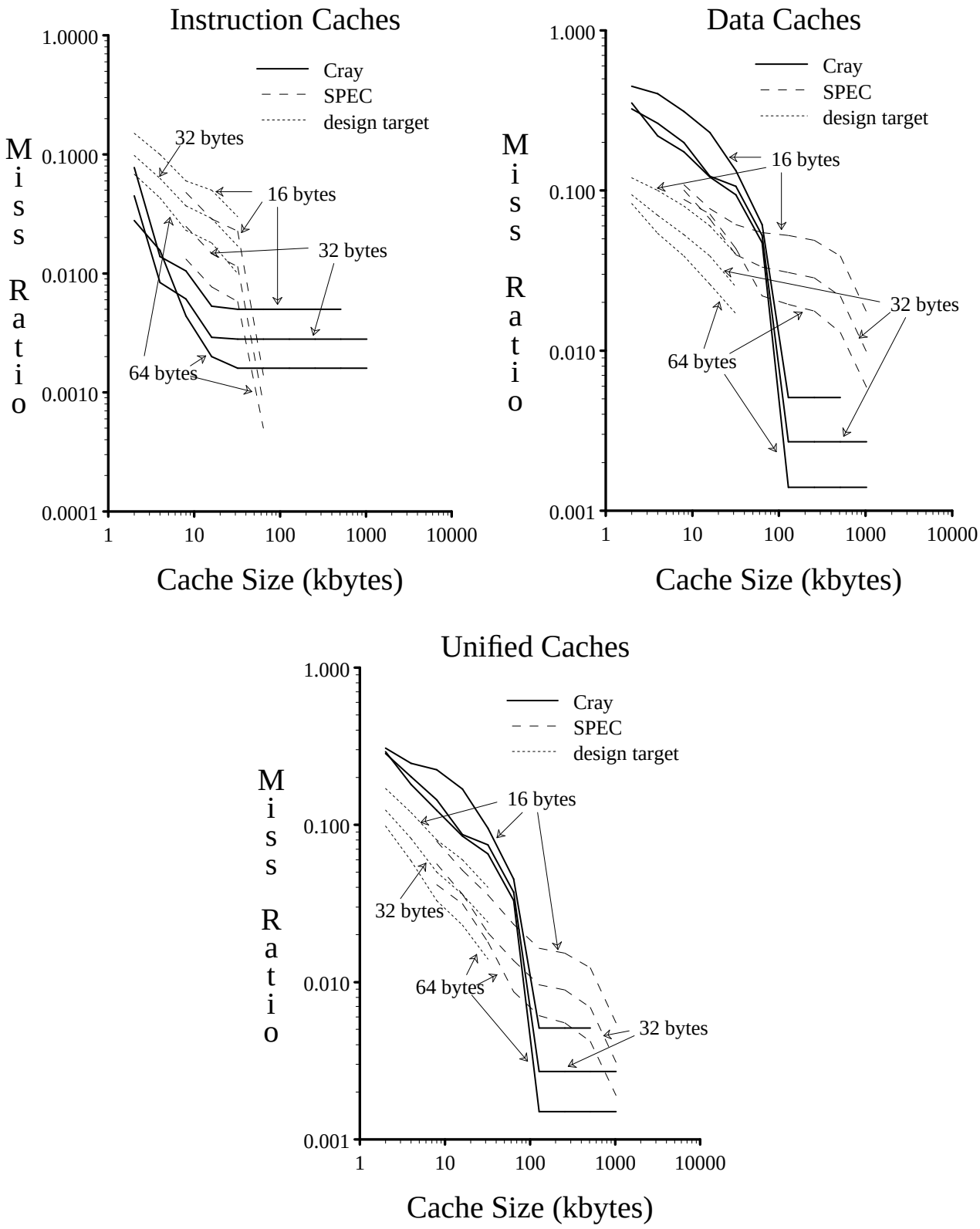

Figure A7: Fully-associative Cray vs. design target and SPEC floating point miss ratios. Curves are parameterized by block size.

Fully Associative Cache Miss Ratios								
Cache Size	*Ardent Instruction*				*Cray Instruction*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.1859	0.1059	0.0783	0.0625	0.4263	0.2835	0.1820	0.1251
256	0.1614	0.0910	0.0583	0.0406	0.2720	0.1997	0.1339	0.0915
512	0.1216	0.0725	0.0460	0.0300	0.1479	0.0845	0.0724	0.0465
1K	0.0883	0.0505	0.0307	0.0238	0.1132	0.0648	0.0398	0.0272
2K	0.0725	0.0412	0.0241	0.0155	0.0777	0.0448	0.0279	0.0187
4K	0.0434	0.0268	0.0176	0.0119	0.0139	0.0084	0.0156	0.0105
8K	0.0139	0.0074	0.0045	0.0031	0.0105	0.0061	0.0044	0.0028
16K	0.0088	0.0048	0.0029	0.0018	0.0053	0.0029	0.0020	0.0018
32K	0.0021	0.0011	0.0008	0.0006	0.0050	0.0028	0.0016	0.0009
64K	0.0003	0.0002	0.0001	0.0001	0.0050	0.0028	0.0016	0.0009
Cache Size	*Ardent Data*				*Cray Data*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.3881	0.3138	0.3443	0.3985	0.7129	0.6619	0.5714	0.5155
256	0.3173	0.2509	0.2283	0.2709	0.6617	0.6128	0.5319	0.4719
512	0.2491	0.1963	0.1710	0.1713	0.5945	0.5708	0.4822	0.4386
1K	0.1749	0.1320	0.1270	0.1185	0.4641	0.4982	0.4571	0.3919
2K	0.1381	0.0816	0.0762	0.0884	0.4488	0.3236	0.3515	0.3656
4K	0.1191	0.0657	0.0402	0.0461	0.4023	0.2630	0.2194	0.2578
8K	0.0990	0.0531	0.0315	0.0213	0.3119	0.1992	0.1740	0.1565
16K	0.0821	0.0427	0.0235	0.0151	0.2307	0.1227	0.1212	0.1230
32K	0.0678	0.0349	0.0183	0.0103	0.1329	0.1063	0.0941	0.0613
64K	0.0561	0.0285	0.0145	0.0079	0.0611	0.0524	0.0473	0.0073
128K	0.0389	0.0195	0.0099	0.0051	0.0051	0.0027	0.0014	0.0008
256K	0.0237	0.0119	0.0060	0.0030	0.0051	0.0027	0.0014	0.0008
512K	0.0141	0.0071	0.0036	0.0018	0.0051	0.0027	0.0014	0.0008
1M		0.0068	0.0034	0.0017		0.0027	0.0014	0.0008
2M			0.0032	0.0016			0.0014	0.0008
Cache Size	*Ardent Unified*				*Cray Unified*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.3061	0.2306	0.2170	0.3908	0.6818	0.5867	0.5110	0.5672
256	0.2455	0.1682	0.1584	0.1707	0.6187	0.5318	0.4446	0.4231
512	0.2096	0.1399	0.1086	0.1050	0.5108	0.4852	0.3975	0.3491
1K	0.1536	0.1124	0.0819	0.0726	0.4267	0.4076	0.3489	0.3056
2K	0.1140	0.0758	0.0558	0.0517	0.3075	0.2838	0.2917	0.2712
4K	0.0871	0.0509	0.0340	0.0294	0.2457	0.2025	0.1809	0.2147
8K	0.0468	0.0261	0.0199	0.0166	0.2237	0.1444	0.1237	0.1271
16K	0.0353	0.0187	0.0106	0.0070	0.1687	0.0865	0.0847	0.0853
32K	0.0270	0.0141	0.0076	0.0047	0.0951	0.0746	0.0652	0.0423
64K	0.0178	0.0091	0.0047	0.0026	0.0452	0.0373	0.0331	0.0053
128K	0.0123	0.0062	0.0032	0.0017	0.0051	0.0027	0.0015	0.0008
256K	0.0073	0.0037	0.0019	0.0010	0.0051	0.0027	0.0015	0.0008
512K	0.0044	0.0022	0.0011	0.0006	0.0051	0.0027	0.0015	0.0008
1M		0.0021	0.0011	0.0005		0.0027	0.0015	0.0008
2M			0.0010	0.0005			0.0015	0.0008

Table A1: Fully-associative cache miss ratios

Data Cache Miss Ratios by Reference Type								
Cache Size	*Ardent Scalar Data*				*Cray Scalar Data*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.3405	0.3346	0.4521	0.5896	0.4060	0.4163	0.4213	0.4793
256	0.2358	0.2335	0.2515	0.3651	0.2641	0.2575	0.3014	0.3434
512	0.1579	0.1492	0.1574	0.1921	0.1886	0.1416	0.1421	0.2372
1K	0.0915	0.0834	0.0939	0.1077	0.1375	0.1121	0.0858	0.0834
2K	0.0605	0.0413	0.0440	0.0649	0.1076	0.0704	0.0612	0.0597
4K	0.0493	0.0303	0.0216	0.0245	0.0737	0.0507	0.0368	0.0368
8K	0.0392	0.0231	0.0157	0.0120	0.0601	0.0387	0.0289	0.0217
16K	0.0320	0.0173	0.0109	0.0077	0.0531	0.0320	0.0195	0.0166
32K	0.0294	0.0155	0.0084	0.0050	0.0427	0.0223	0.0117	0.0070
64K	0.0273	0.0138	0.0072	0.0038	0.0229	0.0119	0.0065	0.0034
128K	0.0240	0.0120	0.0061	0.0031	0.0134	0.0071	0.0039	0.0021
256K	0.0193	0.0097	0.0049	0.0025	0.0134	0.0071	0.0039	0.0021
512K	0.0186	0.0093	0.0047	0.0024	0.0134	0.0071	0.0039	0.0021
1M		0.0092	0.0046	0.0023		0.0071	0.0039	0.0021
Cache Size	*Ardent Vector Data*				*Cray Vector Data*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.4488	0.2874	0.2069	0.1550	0.8398	0.7635	0.6334	0.5305
256	0.4212	0.2731	0.1988	0.1508	0.8262	0.7597	0.6272	0.5250
512	0.3652	0.2563	0.1884	0.1448	0.7623	0.7483	0.6229	0.5219
1K	0.2812	0.1939	0.1692	0.1323	0.6807	0.6579	0.6107	0.5194
2K	0.2369	0.1330	0.1173	0.1183	0.5850	0.4283	0.4715	0.4921
4K	0.2080	0.1107	0.0638	0.0736	0.5369	0.3508	0.2949	0.3492
8K	0.1752	0.0914	0.0515	0.0332	0.4154	0.2656	0.2340	0.2122
16K	0.1458	0.0750	0.0396	0.0245	0.3041	0.1601	0.1633	0.1669
32K	0.1169	0.0597	0.0310	0.0171	0.1703	0.1410	0.1278	0.0838
64K	0.0928	0.0472	0.0240	0.0131	0.0769	0.0691	0.0641	0.0089
128K	0.0578	0.0292	0.0148	0.0077	0.0016	0.0008	0.0004	0.0002
256K	0.0294	0.0147	0.0074	0.0038	0.0016	0.0008	0.0004	0.0002
512K	0.0082	0.0042	0.0021	0.0011	0.0016	0.0008	0.0004	0.0002
1M		0.0037	0.0019	0.0010		0.0008	0.0004	0.0002
2M			0.0014	0.0007			0.0004	0.0002

Table A2: Fully-associative data cache miss ratios by reference type (scalar and vector data).

Unified Cache Miss Ratios by Reference Type								
Cache Size	*Ardent Instructions*				*Cray Instructions*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.2280	0.1360	0.1057	0.2905	0.5029	0.3423	0.2930	0.5208
256	0.1808	0.1079	0.0819	0.0663	0.4130	0.2694	0.2013	0.2159
512	0.1578	0.0926	0.0602	0.0432	0.2561	0.2282	0.1603	0.1219
1K	0.1125	0.0770	0.0488	0.0343	0.2079	0.1390	0.0966	0.0874
2K	0.0841	0.0513	0.0334	0.0275	0.1312	0.0825	0.0682	0.0560
4K	0.0623	0.0358	0.0235	0.0161	0.0938	0.0525	0.0378	0.0361
8K	0.0189	0.0109	0.0114	0.0104	0.0351	0.0227	0.0142	0.0109
16K	0.0118	0.0063	0.0038	0.0025	0.0147	0.0099	0.0090	0.0061
32K	0.0079	0.0041	0.0022	0.0015	0.0093	0.0056	0.0039	0.0031
64K	0.0011	0.0006	0.0004	0.0002	0.0073	0.0043	0.0028	0.0012
128K	0.0007	0.0004	0.0002	0.0001	0.0050	0.0028	0.0016	0.0009
256K	0.0004	0.0002	0.0001	0.0001	0.0050	0.0027	0.0016	0.0009
512K	0.0003	0.0002	0.0001	0.0000	0.0050	0.0027	0.0016	0.0009
Cache Size	*Ardent Scalar Data*				*Cray Scalar Data*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.5205	0.5851	0.6965	1.0000	0.5673	0.5452	0.5693	0.7231
256	0.3686	0.3350	0.4476	0.6263	0.4373	0.4136	0.4036	0.5072
512	0.2747	0.2443	0.2482	0.3336	0.2652	0.2565	0.2452	0.3040
1K	0.1890	0.1672	0.1491	0.1847	0.1948	0.1433	0.1157	0.1518
2K	0.1305	0.0981	0.0879	0.0991	0.1460	0.1045	0.0849	0.0752
4K	0.0907	0.0616	0.0497	0.0481	0.0920	0.0636	0.0502	0.0513
8K	0.0535	0.0330	0.0262	0.0271	0.0710	0.0430	0.0320	0.0246
16K	0.0422	0.0243	0.0156	0.0113	0.0549	0.0335	0.0208	0.0183
32K	0.0345	0.0192	0.0113	0.0076	0.0445	0.0233	0.0121	0.0073
64K	0.0278	0.0143	0.0077	0.0043	0.0230	0.0119	0.0067	0.0034
128K	0.0243	0.0122	0.0062	0.0032	0.0134	0.0071	0.0038	0.0021
256K	0.0194	0.0097	0.0049	0.0025	0.0134	0.0071	0.0038	0.0021
512K	0.0186	0.0093	0.0047	0.0024	0.0134	0.0071	0.0038	0.0021
1M		0.0092	0.0046	0.0023	0.0134	0.0071	0.0038	0.0021
Cache Size	*Ardent Vector Data*				*Cray Vector Data*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	0.4572	0.2925	0.2099	0.1589	0.8528	0.7728	0.6374	0.5348
256	0.4404	0.2835	0.2047	0.1566	0.8358	0.7618	0.6295	0.5315
512	0.4079	0.2639	0.1937	0.1487	0.7883	0.7573	0.6243	0.5247
1K	0.3314	0.2347	0.1755	0.1380	0.6957	0.7024	0.6196	0.5198
2K	0.2551	0.1810	0.1366	0.1230	0.6040	0.5664	0.5316	0.5008
4K	0.2174	0.1189	0.0713	0.0777	0.5467	0.4871	0.3337	0.4056
8K	0.1903	0.1000	0.0577	0.0368	0.4199	0.3567	0.3019	0.2497
16K	0.1541	0.0787	0.0413	0.0260	0.3221	0.2907	0.2296	0.1677
32K	0.1214	0.0621	0.0324	0.0179	0.1753	0.1435	0.1289	0.0838
64K	0.0959	0.0485	0.0247	0.0134	0.0805	0.0707	0.0650	0.0089
128K	0.0599	0.0303	0.0154	0.0081	0.0016	0.0008	0.0004	0.0002
256K	0.0297	0.0149	0.0075	0.0038	0.0016	0.0008	0.0004	0.0002
512K	0.0083	0.0042	0.0021	0.0011	0.0016	0.0008	0.0004	0.0002
1M		0.0038	0.0020	0.0010		0.0008	0.0004	0.0002
2M			0.0014	0.0007			0.0004	0.0002

Table A3: Fully-associative unified cache miss ratios by reference type (instruction, scalar, vector).

Fully Associative Ardent Cache Miss Ratios (4-byte memory interface)				
Data Caches				
Cache	Block Size			
Size	16	32	64	128
128	0.2865	0.2317	0.2542	0.2941
256	0.2342	0.1852	0.1685	0.1999
512	0.1838	0.1449	0.1262	0.1265
1K	0.1291	0.0974	0.0938	0.0875
2K	0.1019	0.0602	0.0563	0.0652
4K	0.0879	0.0485	0.0297	0.0340
8K	0.0730	0.0392	0.0232	0.0157
16K	0.0606	0.0315	0.0173	0.0111
32K	0.0501	0.0258	0.0135	0.0076
64K	0.0414	0.0210	0.0107	0.0058
128K	0.0287	0.0144	0.0073	0.0038
256K	0.0175	0.0088	0.0044	0.0022
512K	0.0104	0.0052	0.0026	0.0013
Unified Caches				
Cache	Block Size			
Size	16	32	64	128
128	0.2771	0.2087	0.1964	0.3537
256	0.2222	0.1523	0.1433	0.1545
512	0.1897	0.1267	0.0983	0.0950
1K	0.1390	0.1017	0.0741	0.0658
2K	0.1032	0.0686	0.0505	0.0468
4K	0.0789	0.0461	0.0308	0.0266
8K	0.0424	0.0236	0.0180	0.0151
16K	0.0320	0.0169	0.0096	0.0063
32K	0.0245	0.0128	0.0069	0.0042
64K	0.0162	0.0082	0.0043	0.0023
128K	0.0111	0.0056	0.0029	0.0015
256K	0.0066	0.0033	0.0017	0.0009
512K	0.0039	0.0020	0.0010	0.0005

Table A4: Fully-associative Ardent cache miss ratios assuming a 4-byte cache interface

Instruction Cache Ratios of Miss Ratios									
Cache Size	*Ardent*			*Cray*			*[Smit87]*		
	Block Size			Block Size			Block Size		
	32	64	128	32	64	128	32	64	128
128	0.570	0.739	0.798	0.665	0.642	0.687	0.687	0.955	0.929
256	0.564	0.641	0.696	0.734	0.671	0.683	0.723	0.754	0.931
512	0.596	0.635	0.652	0.571	0.857	0.642	0.735	0.705	0.910
1K	0.572	0.608	0.775	0.572	0.614	0.683	0.693	0.790	0.801
2K	0.568	0.585	0.643	0.577	0.623	0.670	0.737	0.701	0.832
4K	0.618	0.657	0.676	0.604	1.857	0.673	0.651	0.660	0.831
8K	0.532	0.608	0.689	0.581	0.721	0.636	0.598	0.667	0.690
16K	0.546	0.604	0.621	0.547	0.690	0.900	0.581	0.624	0.657
32K	0.524	0.727	0.750	0.560	0.571	0.563	0.581	0.624	0.634
64K	0.667	0.500	1.000	0.560	0.571	0.563	-	-	-
128K	0.667	0.500	1.000	0.560	0.571	0.563	-	-	-

Table A5: Instruction Cache Ratios of Miss Ratios

Data Cache Ratios of Miss Ratios									
Cache Size	*Ardent*			*Cray*			*[Smit87]*		
	Block Size			Block Size			Block Size		
	32	64	128	32	64	128	32	64	128
128	0.809	1.097	1.157	0.929	0.863	0.902	1.004	1.328	1.439
256	0.791	0.910	1.187	0.926	0.868	0.887	0.944	1.124	1.577
512	0.788	0.871	1.002	0.960	0.845	0.910	0.956	1.122	1.314
1K	0.755	0.962	0.933	1.074	0.918	0.857	0.836	1.015	1.499
2K	0.591	0.934	1.160	0.721	1.086	1.040	0.787	0.853	1.071
4K	0.552	0.612	1.147	0.654	0.834	1.175	0.727	0.770	0.835
8K	0.536	0.593	0.676	0.639	0.874	0.899	0.667	0.723	0.817
16K	0.520	0.550	0.643	0.532	0.988	1.015	0.646	0.672	0.747
32K	0.515	0.524	0.563	0.800	0.885	0.651	0.646	0.662	0.697
64K	0.508	0.509	0.545	0.858	0.903	0.154	-	-	-
128K	0.501	0.508	0.515	0.529	0.519	0.571	-	-	-
256K	0.502	0.504	0.500	0.529	0.519	0.571	-	-	-
512K	0.504	0.507	0.500	0.529	0.519	0.571	-	-	-

Table A6: Data Cache Ratios of Miss Ratios

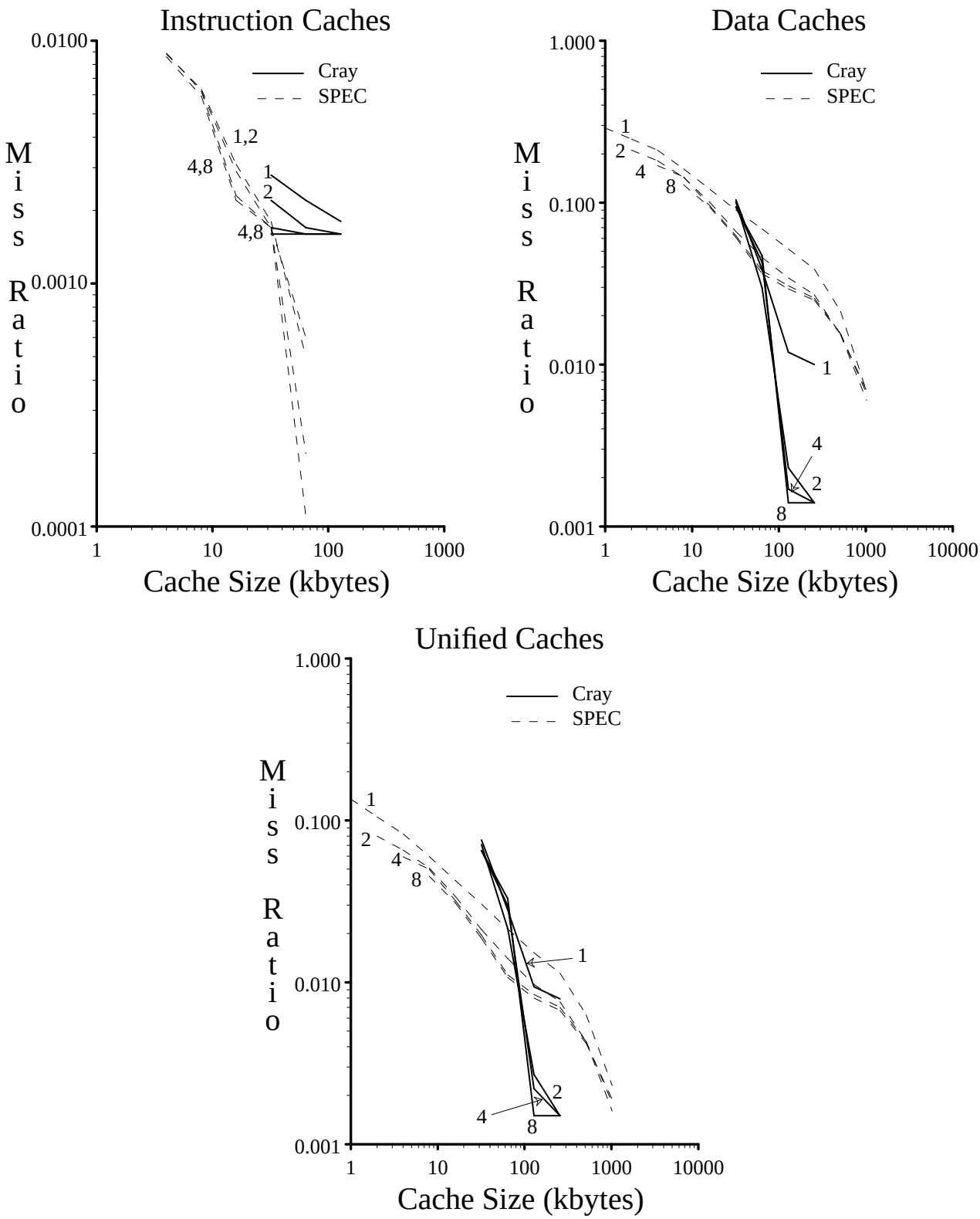

Figure A8: Set-associative Cray and SPEC cache miss ratios for a 64 byte block size. Curves are labeled by set size.

Set-Associative Instruction Cache Miss Ratios									
Cache Size	Set Size	*Ardent* Block Size				*Cray* Block Size			
		16	32	64	128	16	32	64	128
32K	1	0.0070	0.0039	0.0022	0.0014	0.0077	0.0045	0.0028	0.0020
32K	2	0.0042	0.0023	0.0013	0.0008	0.0061	0.0036	0.0022	0.0015
32K	4	0.0029	0.0016	0.0009	0.0006	0.0052	0.0029	0.0017	0.0011
32K	8	0.0024	0.0013	0.0008	0.0005	0.0050	0.0028	0.0016	0.0010
64K	1	0.0044	0.0023	0.0013	0.0007	0.0065	0.0037	0.0022	0.0014
64K	2	0.0011	0.0006	0.0003	0.0002	0.0052	0.0029	0.0017	0.0010
64K	4	0.0004	0.0002	0.0001	0.0001	0.0050	0.0028	0.0016	0.0010
64K	8	0.0003	0.0002	0.0001	0.0001	0.0050	0.0028	0.0016	0.0009
128K	1	0.0007	0.0004	0.0002	0.0001	0.0055	0.0031	0.0018	0.0011
128K	2	0.0003	0.0002	0.0001	0.0001	0.0051	0.0028	0.0016	0.0010
128K	4	0.0003	0.0002	0.0001	0.0000	0.0050	0.0028	0.0016	0.0009
128K	8	0.0003	0.0002	0.0001	0.0000	0.0050	0.0028	0.0016	0.0009

Table A7: Set-associative Ardent and Cray instruction cache miss ratios

Set-Associative Data Cache Miss Ratios									
Cache Size	Set Size	*Ardent* Block Size				*Cray* Block Size			
		16	32	64	128	16	32	64	128
32K	1	0.0795	0.0444	0.0284	0.0221	0.1473	0.1207	0.1042	0.0967
32K	2	0.0697	0.0365	0.0199	0.0122	0.1393	0.1186	0.1012	0.0910
32K	4	0.0678	0.0352	0.0188	0.0109	0.1287	0.1048	0.0942	0.0886
32K	8	0.0671	0.0347	0.0184	0.0105	0.1324	0.1061	0.0941	0.0879
64K	1	0.0639	0.0337	0.0190	0.0119	0.0626	0.0470	0.0385	0.0344
64K	2	0.0559	0.0287	0.0151	0.0083	0.0504	0.0366	0.0296	0.0262
64K	4	0.0555	0.0283	0.0147	0.0080	0.0570	0.0477	0.0427	0.0403
64K	8	0.0546	0.0279	0.0144	0.0078	0.0583	0.0512	0.0470	0.0451
128K	1	0.0444	0.0229	0.0123	0.0071	0.0241	0.0161	0.0119	0.0100
128K	2	0.0372	0.0190	0.0099	0.0053	0.0079	0.0042	0.0023	0.0014
128K	4	0.0375	0.0191	0.0098	0.0052	0.0056	0.0030	0.0017	0.0009
128K	8	0.0379	0.0193	0.0099	0.0052	0.0051	0.0027	0.0014	0.0008
256K	1	0.0336	0.0172	0.0092	0.0052	0.0165	0.0122	0.0100	0.0090
256K	2	0.0221	0.0112	0.0057	0.0030	0.0051	0.0027	0.0014	0.0008
256K	4	0.0225	0.0113	0.0058	0.0030	0.0051	0.0027	0.0014	0.0008
256K	8	0.0227	0.0114	0.0058	0.0030	0.0051	0.0027	0.0014	0.0008
512K	1	0.0211	0.0107	0.0056	0.0030	0.0161	0.0120	0.0097	0.0086
512K	2	0.0143	0.0072	0.0037	0.0019	0.0051	0.0027	0.0014	0.0008
512K	4	0.0140	0.0071	0.0036	0.0018	0.0051	0.0027	0.0014	0.0008
512K	8	0.0140	0.0071	0.0036	0.0018	0.0051	0.0027	0.0014	0.0008

Table A8: Set-associative Ardent and Cray data cache miss ratios

Set-Associative Unified Cache Miss Ratios									
Cache Size	Set Size	*Ardent* Block Size				*Cray* Block Size			
		16	32	64	128	16	32	64	128
32K	1	0.0355	0.0211	0.0141	0.0132	0.1077	0.0876	0.0757	0.0712
32K	2	0.0282	0.0156	0.0093	0.0064	0.0995	0.0835	0.0708	0.0636
32K	4	0.0262	0.0141	0.0080	0.0050	0.0907	0.0730	0.0651	0.0610
32K	8	0.0260	0.0139	0.0077	0.0047	0.0935	0.0740	0.0652	0.0607
64K	1	0.0263	0.0148	0.0089	0.0081	0.0463	0.0343	0.0279	0.0248
64K	2	0.0193	0.0102	0.0056	0.0034	0.0371	0.0267	0.0215	0.0190
64K	4	0.0185	0.0096	0.0052	0.0030	0.0411	0.0337	0.0298	0.0280
64K	8	0.0175	0.0091	0.0048	0.0028	0.0423	0.0363	0.0328	0.0312
128K	1	0.0159	0.0086	0.0049	0.0050	0.0195	0.0129	0.0094	0.0078
128K	2	0.0124	0.0064	0.0034	0.0019	0.0088	0.0048	0.0027	0.0016
128K	4	0.0120	0.0061	0.0032	0.0017	0.0076	0.0041	0.0022	0.0012
128K	8	0.0121	0.0062	0.0032	0.0017	0.0051	0.0027	0.0015	0.0008
256K	1	0.0117	0.0062	0.0035	0.0021	0.0140	0.0100	0.0079	0.0070
256K	2	0.0071	0.0036	0.0019	0.0010	0.0051	0.0027	0.0015	0.0008
256K	4	0.0070	0.0036	0.0018	0.0010	0.0051	0.0027	0.0015	0.0008
256K	8	0.0070	0.0035	0.0018	0.0009	0.0051	0.0027	0.0015	0.0008
512K	1	0.0073	0.0038	0.0021	0.0012	0.0125	0.0090	0.0070	0.0061
512K	2	0.0046	0.0023	0.0012	0.0006	0.0051	0.0027	0.0015	0.0008
512K	4	0.0044	0.0022	0.0011	0.0006	0.0051	0.0027	0.0015	0.0008
512K	8	0.0044	0.0022	0.0011	0.0006	0.0051	0.0027	0.0015	0.0008

Table A9: Set-associative Ardent and Cray unified cache miss ratios

Ardent Smoothed Miss Ratio Spreads for Instruction Caches						
Cache Size	Block Size: 16 Bytes			Block Size: 32 Bytes		
	8-to-4	4-to-2	to-1	8-to-4	4-to-2	2-to-1
32K	0.199	0.745	1.538	0.107	0.817	1.424
64K	0.167	0.723	1.647	0.054	0.809	1.487
128K	0.098	0.417	1.233	0.035	0.466	1.071
256K	0.050	0.262	0.917	0.000	0.300	0.775
512K	0.000	0.000	0.333	0.000	0.000	0.250
Cache Size	Block Size: 64 Bytes			Block Size: 128 Bytes		
	8-to-4	4-to-2	to-1	8-to-4	4-to-2	2-to-1
32K	0.058	0.821	1.576	0.092	0.462	1.115
64K	0.029	0.810	1.751	0.047	0.431	1.059
128K	0.019	0.467	1.271	0.030	0.250	0.613
256K	0.000	0.300	1.000	0.000	0.150	0.375
512K	0.000	0.000	0.350	0.000	0.000	0.000

Table A10: Smoothed miss ratio spreads for Ardent instruction caches

Ardent Smoothed Miss Ratio Spreads for Data Caches						
Cache Size	Block Size: 16 Bytes			Block Size: 32 Bytes		
	8-to-4	4-to-2	to-1	8-to-4	4-to-2	2-to-1
32K	0.007	0.013	0.154	0.009	0.020	0.201
64K	0.004	0.004	0.221	0.004	0.011	0.255
128K	-0.001	0.003	0.283	0.000	0.007	0.309
256K	-0.005	-0.002	0.318	-0.003	0.001	0.334
512K	-0.007	0.003	0.288	-0.003	0.002	0.300
Cache Size	Block Size: 64 Bytes			Block Size: 128 Bytes		
	8-to-4	4-to-2	to-1	8-to-4	4-to-2	2-to-1
32K	0.014	0.038	0.333	0.025	0.071	0.586
64K	0.010	0.023	0.357	0.018	0.046	0.553
128K	0.004	0.018	0.388	0.011	0.039	0.544
256K	0.001	0.007	0.383	0.004	0.021	0.486
512K	-0.002	0.006	0.334	0.000	0.020	0.404

Table A11: Smoothed miss ratio spreads for Ardent data caches

Performance of Different Replacement Algorithms (Ardent Data Caches)							
		Miss Ratios (Random)			Ratios of Ratios (Random / LRU)		
Cache	Set	Block Size			Block Size		
Size	Size	16	32	64	16	32	64
32K	1	0.0795	0.0444	0.0284	1.0000	1.0000	1.0000
32K	2	0.0708	0.0371	0.0204	1.0158	1.0164	1.0251
32K	4	0.0682	0.0354	0.0191	1.0059	1.0057	1.0160
32K	8	0.0669	0.0346	0.0183	0.9970	0.9971	0.9946
64K	1	0.0639	0.0337	0.0190	1.0000	1.0000	1.0000
64K	2	0.0567	0.0292	0.0154	1.0143	1.0174	1.0199
64K	4	0.0554	0.0284	0.0148	0.9982	1.0035	1.0068
64K	8	0.0542	0.0277	0.0143	0.9927	0.9928	0.9931
128K	1	0.0444	0.0229	0.0123	1.0000	1.0000	1.0000
128K	2	0.0405	0.0207	0.0108	1.0887	1.0895	1.0909
128K	4	0.0386	0.0196	0.0101	1.0293	1.0262	1.0306
128K	8	0.0375	0.0191	0.0098	0.9894	0.9896	0.9899
256K	1	0.0336	0.0172	0.0092	1.0000	1.0000	1.0000
256K	2	0.0217	0.0110	0.0057	0.9819	0.9821	1.0000
256K	4	0.0215	0.0109	0.0055	0.9556	0.9646	0.9483
256K	8	0.0214	0.0108	0.0055	0.9427	0.9474	0.9483
512K	1	0.0211	0.0107	0.0056	1.0000	1.0000	1.0000
512K	2	0.0143	0.0072	0.0037	1.0000	1.0000	1.0000
512K	4	0.0140	0.0071	0.0036	1.0000	1.0000	1.0000
512K	8	0.0140	0.0070	0.0036	0.9929	0.9859	1.0000
Geometric Average					0.9999	1.0006	1.0028

Table A12: Ardent set-associative data cache miss ratios using random replacement, along with ratios of miss ratios (random / LRU). Values are averages across all Ardent traces.

Ardent Mean Delay Per Memory Reference (cycles)								
Cache Size	*Instruction Caches: 14 Cycle Latency*				*Instruction Caches: 45 Cycle Latency*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	2.9744	1.9062	1.7226	1.8750	8.7373	5.1891	4.1499	3.8125
256	2.5824	1.6380	1.2826	1.2180	7.5858	4.4590	3.0899	2.4766
512	1.9456	1.3050	1.0120	0.9000	5.7152	3.5525	2.4380	1.8300
1K	1.4128	0.9090	0.6754	0.7140	4.1501	2.4745	1.6271	1.4518
2K	1.1600	0.7416	0.5302	0.4650	3.4075	2.0188	1.2773	0.9455
4K	0.6944	0.4824	0.3872	0.3570	2.0398	1.3132	0.9328	0.7259
8K	0.2224	0.1332	0.0990	0.0930	0.6533	0.3626	0.2385	0.1891
16K	0.1408	0.0864	0.0638	0.0540	0.4136	0.2352	0.1537	0.1098
32K	0.0336	0.0198	0.0176	0.0180	0.0987	0.0539	0.0424	0.0366
64K	0.0048	0.0036	0.0022	0.0015	0.0141	0.0098	0.0053	0.0030
Cache Size	*Data Caches: 14 Cycle Latency*				*Data Caches: 45 Cycle Latency*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	6.2096	5.6484	7.5746	11.9550	18.2407	15.3762	18.2479	24.3085
256	5.0768	4.5162	5.0226	8.1270	14.9131	12.2941	12.0999	16.5249
512	3.9856	3.5334	3.7620	5.1390	11.7077	9.6187	9.0630	10.4493
1K	2.7984	2.3760	2.7940	3.5550	8.2203	6.4680	6.7310	7.2285
2K	2.2096	1.4688	1.6764	2.6520	6.4907	3.9984	4.0386	5.3924
4K	1.9056	1.1826	0.8844	1.3830	5.5977	3.2193	2.1306	2.8121
8K	1.5840	0.9558	0.6930	0.6390	4.6530	2.6019	1.6695	1.2993
16K	1.3136	0.7686	0.5170	0.4530	3.8587	2.0923	1.2455	0.9211
32K	1.0848	0.6282	0.4026	0.3090	3.1866	1.7101	0.9699	0.6283
64K	0.8976	0.5130	0.3190	0.2370	2.6367	1.3965	0.7685	0.4819
128K	0.6224	0.3510	0.2178	0.1530	1.8283	0.9555	0.5247	0.3111
256K	0.3792	0.2142	0.1320	0.0900	1.1139	0.5831	0.3180	0.1830
512K	0.2256	0.1278	0.0792	0.0540	0.6627	0.3479	0.1908	0.1098
Cache Size	*Unified Caches: 14 Cycle Latency*				*Unified Caches: 45 Cycle Latency*			
	Block Size				Block Size			
	16	32	64	128	16	32	64	128
128	4.8976	4.1508	4.7740	11.7240	14.3867	11.2994	11.5010	23.8388
256	3.9280	3.0276	3.4848	5.1210	11.5385	8.2418	8.3952	10.4127
512	3.3536	2.5182	2.3892	3.1500	9.8512	6.8551	5.7558	6.4050
1K	2.4576	2.0232	1.8018	2.1780	7.2192	5.5076	4.3407	4.4286
2K	1.8240	1.3644	1.2276	1.5510	5.3580	3.7142	2.9574	3.1537
4K	1.3936	0.9162	0.7480	0.8820	4.0937	2.4941	1.8020	1.7934
8K	0.7488	0.4698	0.4378	0.4980	2.1996	1.2789	1.0547	1.0126
16K	0.5648	0.3366	0.2332	0.2100	1.6591	0.9163	0.5618	0.4270
32K	0.4320	0.2538	0.1672	0.1410	1.2690	0.6909	0.4028	0.2867
64K	0.2848	0.1638	0.1034	0.0780	0.8366	0.4459	0.2491	0.1586
128K	0.1968	0.1116	0.0704	0.0510	0.5781	0.3038	0.1696	0.1037
256K	0.1168	0.0666	0.0418	0.0300	0.3431	0.1813	0.1007	0.0610

Table A13: Mean delay per memory reference (cycles) for Ardent fully-associative instruction, data, and unified caches. Results are listed for memory latencies of 14 and 45 cycles. Transfer time is 1 cycle per 8-byte word.

Set-Associative Cache Block Sizes Which Minimize Mean Delay				
Cache Size	Associativity	*Instruction*	*Data*	*Unified*
32K	1	128	64	64
32K	2	128	128	128
32K	4	128	128	128
32K	8	128	128	128
64K	1	128	128	64
64K	2	128	128	128
64K	4	128	128	128
64K	8	128	128	128
128K	1	128	128	64
128K	2	128	128	128
128K	4	128	128	128
128K	8	128	128	128
256K	1	128	128	128
256K	2	128	128	128
256K	4	128	128	128
256K	8	128	128	128
512K	1	128	128	128
512K	2	128	128	128
512K	4	128	128	128
512K	8	128	128	128

Table A14: Block sizes which minimize mean access delays in Ardent set-associative caches. Results are identical for memory latencies of 14 and 45 cycles.