Proc. 12'th Annual International Symposium on Computer Architecture, June 17-19, 1985, Boston, Massachusetts, USA. pp. 64-73.

Cache Evaluation and the Impact of Workload Choice*

Alan Jay Smith
Computer Science Division
EECS Department
University of California
Berkeley, California 94720

Abstract

The selection of the "best" parameters for a cache design, such as size, mapping algorithm, fetch algorithm, line size, etc., is dependent on the expected workload. Similarly, the machine performance is sensitive to the cache performance which itself depends on the workload.

Most cache designers have been greatly handicapped in their designs by the lack of realistic cache performance estimates. Published research generally presents data which is unrealistic in some respects, and available traces are often not representative.

In this paper, we present measurements from a very wide variety of traces: there are 49 traces, taken from 6 machine architectures, (370, 360, VAX, M68000, Z8000, CDC 6400), coded in 7 source languages. Statistics are shown for miss ratios, the effectiveness of prefetching in terms of both miss ratio and its effect on bus traffic, the frequency of writes, reads and instruction fetches, and the frequency of branches. Some general observations are made and a "design estimate" set of miss ratios are proposed. Some "fudge" factors are proposed by which statistics for workloads for one machine architecture can be used to estimate corresponding parameters for another (as yet unrealized) architecture.

1. Introduction

Almost all medium and high performance machines and most high performance microprocessors now being designed will include cache memories to be used for instructions, for data or for both. There are a number of choices to be made regarding the cache including size, line size (block size), mapping algorithm, replacement algorithm, writeback algorithm, split (instructions/data) vs. unified, fetch algorithm, et cetera; see [Smit82] for a detailed discussion of these issues. Making the "best" choices and selecting the "best" parameters (with respect to cost and performance) depends greatly on the workload to be expected [Macd84]. For example, a cache which achieves a 99% hit ratio may cost 80% more than one which achieves 98%, may increase the CPU cost by 25% and may only boost overall CPU performance by 8%; that suggests that the higher performing cache is not cost effective. However, if the same two designs yield hit ratios of 90% and 80% respectively, and if the performance increase would be 50%, then different conclusions might well be reached.

Computer architects have been handicapped by the lack of generally available realistic cache workload estimates. While there are hundreds of published papers on cache memories (see [Smit82] for a partial bibliography), only a few present usable data. A large fraction contain no measurements at all. Almost all of the papers that do present measurements rely on trace driven simulation using a small set of traces, and for reasons explained further below, those traces are likely to be unrepresentative of the results to be expected in practice. There do exist some realistic numbers, as we note below, but they are hardly enough to constitute a design database.

The purpose of this paper is discuss and explain workload selection as it relates to cache memory design, and to present data from which the designer can work. We have used 49 program address traces taken from 6 (or 5, if the 360 and 370 are the same) machine architectures (VAX, 370, 360/91, Z8000, CDC 6400, M68000), derived from 7 programming languages (Fortran, 370 Assembler, APL, C, LISP, AlgolW, Cobol) to compute overall, instruction and data miss ratios and bus traffic rates for various cache designs; these experiments show the variety of workload behavior possible. Characteristics of the traces are tabulated and the effects of some design choices are evaluated. Finally, we present what we consider to be a "reasonable" set of numbers with which we believe designers can comfortably work. In that discussion, we also suggest some "fudge" factors, which indicate how realistic (or available) numbers for machine architecture M1 under workload conditions W1 can be used to estimate similar parameters for architecture M2 under workload W1*.

In the remainder of this section, we discuss additional background for our measurement results. First we consider the advantages and disadvantages of trace driven simulation. Then we review some (possible) cases of performance misprediction and also discuss some published and valid miss ratio figures. The second section discusses the traces used. The measurement results and analysis are in section 3, and in section 4 we propose target workload values and factors by which one workload can be used to estimate another. Section 5 summarizes our findings.

1.1. Trace Driven Simulation

A **program address trace** is a trace of the sequence of (virtual) addresses accessed by a computer program or programs. **Trace driven simulation** involves driving a simulation model of a system with a trace of external stimuli rather than with a random number generator. Trace driven simulation is a very good way to study many aspects of cache design and performance, for a number of reasons. First, it is superior to either pure mathematical models or random number driven simulation because there do not currently exist any generally accepted or believable models for those characteristics of program behavior that determine cache performance; thus it is not possible to specify a realistic model nor to drive a simulator with a good representation of a program. A trace properly

*The material presented here is based on research supported in part by the National Science Foundation under grant DCR-8202591 and by the Defense Advanced Research Projects Agency under contract N00039-82-C-0235. Computer time has been provided by the Stanford Linear Accelerator Center under Department of Energy contract DE-AC03-76SF-00515.

represents at least one real program, and in certain respects can be expected to drive the simulator correctly.

It is important to note that a trace reflects not only the program traced and the functional architecture of the machine (instruction set) but also the **design architecture** (higher level implementation). In particular, **the number of memory references is affected by the width of the data path to memory**: fetching two four-byte instructions requires 4, 2 or 1 memory reference, depending on whether the memory interface is 2, 4 or 8 bytes wide. It also depends on how much "memory" the interface itself has; if one request is for 4 bytes, the next request is for the next four bytes, and the interface is 8 bytes wide, then fewer fetches will result if the interface "remembers" that it has the target four bytes of the second fetch rather than redoing the fetch. The interface can be quite complex, as with the Ifetch buffer in the VAX 11/780 [Clar83] and can behave differently for instructions and data. (A trace should reflect, to the greatest possible extent, only the functional architecture; the design architecture should and usually can be emulated in the simulator.)

A simulator is also much better in many ways than the construction of prototype designs. It is far faster to build a simulator, and the design being simulated can be varied easily, sometimes by just changing an input parameter. Conversely, a hardware prototype can require man-years to build and can be varied little if at all. Also, the results of a live workload tend to yield slightly different results (e.g. 1% to 3%) from run to run, depending on the random setting of initial conditions such as the angular position of the disks [Curr75].

For the reasons given above, trace driven simulation has been used for almost every research paper which presents cache measurements, with a few exceptions discussed below. There are, however, several reasons why the results of trace driven simulations should be taken with a grain of salt. (1) A trace driven simulation of a million memory addresses, which is fairly long, represents about 1/30 of a second for a machine such as the IBM 3081, and only about one second for an M68000; thus a trace is only a very small sample of a real workload. (2) Traces seldom are taken from the "messiest" parts of large programs; more often they are traces of the initial portions of small programs. (3) It is very difficult to trace the operating system (OS) and few OS traces are available. On many machines, however, the OS dominates the workload. (4) Most real machines switch every few thousand instructions and are constantly taking interrupts. It is difficult to include this effect accurately in a trace driven simulation and many simulators don't try. (5) The sequence of memory addresses presented to the cache can vary with hardware buffers such as prefetch buffers and loop buffers, and is certainly sensitive to the data path width. Thus the trace itself may not be completely accurate with respect to the implementation of the architecture. (6) In running machines, a certain (usually small) fraction of the cache activity is due to input/output; this effect is seldom included in trace driven simulations.

In this paper we are primarily concerned with items 1-3 immediately above. By presenting the results of a very large number of simulations, one can get an idea of the range of program behavior. Included are two traces of IBM's MVS operating system, which should have performance that is close to the worst likely to be observed.

1.2. Real Workloads and Questionable Estimates

There are only a small number of papers in which provide measurements taken by hardware monitors from running machines. In [Mila75] it is reported that a 16K cache on an IBM 370/165-2 running VS2 had a 0.94 hit ratio, with 1.6 fetches per instruction and .22 stores/instruction; it is also found that 73% of the CPU cycles were used in supervisor state. Merrill [Merr74] found cache hit ratios for a 16K cache in the 370/168 of 0.932 to 0.997 for six applications programs, and also reports that the performance (MIP rate) of the machine increased for one benchmark from 2.07 to 2.34 MIPS when the cache hit ratio went from 0.969 to 0.9867. In [Hard80] (see [Smit82] for the data) it is found that the supervisor miss ratio and problem (user) state miss ratio can respectively be described by the curves 0.3249K**(-0.5309) and 0.03K**(-0.1892) for an IBM 370/MVS workload. (Hit ratio = 1 - miss ratio.) Supervisor and problem state hit ratios are thus approximately (0.925, 0.948, 0.964) and (0.982, 0.984, 0.986) respectively at (16K, 32K, 64K) bytes. These machines (IBM 165, 168, Amdahl 470V) all use 32 byte lines. For a 64K cache, 64 byte lines and workloads of small scientific programs, large scientific programs, business (Cobol) programs, and timesharing, the misses per instruction were found to be 0.0015, 0.0114, 0.035 and 0.042 respectively on the Fujitsu M380 [Hatt83]; assuming two memory references per instruction, that works out to hit ratios respectively of 0.9992, 0.9943, 0.9825 and 0.979. The Synapse machines [Fran84], based on the M68000 and with a 16K cache per processor and 16 byte lines report a hit ratio above 0.95.

A very thorough set of measurements is presented in [Clar83] for the Digital Equipment Corp. VAX 11/780, which has an 8Kbyte cache and 8 byte lines. The mean hit ratio for data was found to be 83.5% and for instructions 91.4%; overall, the figure was about 89.7%. Most interestingly, it is reported in [Clar83] that a simulation study at DEC using VAX traces predicted about a 93% hit ratio, significantly higher than the 89.7% observed.

The inspiration for this paper came from the first article to describe the forthcoming Zilog Z80000 [Alpe83] in which the projected hit ratios for the Z80000's 256 bytes of storage are reported to be 0.82, 0.75 or 0.88 depending on whether the (effective) block size is 2, 4 or 16 bytes. (The machine uses a sector cache (block/subblock), with a 16 byte sector (large block) and then fetches either 2 bytes, 4 bytes or 16 bytes (called a block or subblock)). These figures are considerably better than this author would expect to occur in practice, since the results from most other traces are considerably less favorable. These predictions are off, we believe, not because of programming errors but because of a poor workload selection: (1) The traces used to estimate the miss ratios were taken from a Z8000, which is a 16 bit machine, and the estimates are for a 32-bit machine. (2) The traces are of a version of Unix that runs on the Z8000 and which was ported from the PDP-11; thus the code and data sizes are small. (3) The C compiler used to compile the programs traced was not very sophisticated and appears to generate an inordinately large number of sequential instructions between loads, stores and branches. (Presumably the compiler and other software has matured since the time the traces were generated.) (4) Many of the programs traced were small, tightly coded utilities.

It is also worth noting, although we do not further consider the issue in this paper, that projected instruction rates for some machines are far too high. It is common to estimate the instruction rate of a microprocessor by assuming that all instructions execute in the minimum time and memory delays cause no processor wait states. Such estimates are also (sometimes deliberately) based on faulty workloads.

2. The Traces (Our Workload)

We have selected 49 traces from those available to us for analysis. These traces are derived from 6 machine architectures: the IBM 370, the IBM 360/91, the DEC VAX, the Zilog

Z8000, the CDC 6400, and the Motorola 68000; we refer to these as 5.5 different architectures due to the close similarity between the 360/91 and the 370. Two of the traces were split into 5 sections each, so there are up to 57 sets of data for some measurements. (The two traces which were sectioned were split for two reasons: (a) they were very long and therefore could be easily subdivided, and (b) the use of garbage collection in Lisp programs suggested the possibility of anomalous behavior within those sections containing garbage collection.) The traces used are listed and described in this section; it should be noted that in addition to functional architecture differences between the machines traced, there are also design architecture differences embodied in the traces, which are also described below.

For the IBM 370, nine traces were examined: FCOMP1 (Fortran compile of program that solves Reynolds partial differential equations (2330 lines)), CCOMP1 (Cobol compile, 240 lines, accounting report), FGO1 (Fortran Go [execution] step, factor analysis, 1249 lines, single precision), FGO2 (Fortran Go step, double precision analysis of satellite information, 2057 lines, FortG compiler), FGO3 (Fortran Go step, double precision numerical analysis, 840 lines, FortG compiler), CGO1 (Cobol Go step, fixed assets program doing tax transaction selection), CGO2 (Cobol Go Step, projects depreciation of fixed assets), and MVS1 and MVS2 (different standard MVS workloads at Amdahl Corp.). These traces were generated at Amdahl, and correctly reflect the architecture of the Amdahl 470: a cache with a four byte interface to the CPU.

Thirteen traces for the Zilog Z8000 microprocessor were also analyzed; these are the same traces as were analyzed in [Alpe83]. Each trace is for a program which is part of the UNIX system software. These Z8000 traces are: ZOD (octal dump of core images), ZSORT (sort program), ZVI (screen editor), ZGREP (search a file for a pattern), ZPR (format a file for the line printer), ZCPP (C compiler preprocessor), ZC3 (C compiler third pass - optimizer), ZC4 (C compiler fourth pass - lister), ZDIFF (compare files), ZED (line editor), ZSED (stream editor), ZNM (load module name listing) and ZLD (link editor). These traces were generated at Zilog Corporation and correctly reflect the Z8000: a two byte memory interface is assumed.

Fourteen traces for the DEC VAX were used. These traces were: VCCOM (the C compiler compiling a C program of 125 lines, written in C), VTROFF (the photo typesetter text formatting system, written in C), VPUZZLE (the well known "puzzle" program; used by Baskett to test raw CPU power, written in C), VOTMDL (parser/constructor, written in Pascal, uses set operations), VSPICE (the Spice circuit simulator, written in Fortran), VLS (Unix utility which lists files, in C), VAWK (text processing language, in C), VC2 (assembly language peep hole optimizer, in C), VSEDX (text processing stream editor, in C), VQSORT (internal quick sort of 10,000 integers, in C), VYMERGE (a parse table compacting program, in C), VTOWERS (Towers of Hanoi, 14 disks, in C), LISPC (Vax Lisp Compiler, written in LISP) and VAXIMA (VAXIMA symbolic algebraic manipulation program derived from Macsyma, written in LISP). Both the LISPC and VAXIMA traces have been cut into many sections of 1,000,000 memory references each; LISPC2...LISPC14 are the initial portions of sections 2, 5, 8, 11 and 14 of the trace and similarly for VAXIMA1...VAXIMA25. A four byte memory interface is used for all VAX traces, but without any "memory"; i.e. after each data or instruction fetch, all unused bytes are discarded; this lack of memory will overstate the frequency of instruction fetches relative to a more efficient design architecture. (The actual VAX 11/780 implementation is quite a bit more complicated than this, and we don't attempt to simulate it.)

There are five traces for the CDC 6400: TWOD1 (Fortran Go of a program that solves the two dimensional scattering problem of an infinite circular cylinder), PPAS (start up portion of Fortran Go of a phase plane analysis program solving a set of two simultaneous differential equations), PPAL (same as PPAS, except that tracing began after program had gone into iteration loops), DIPOLE (Fortran Go of a program that solves a three dimensional scattering problem for a cube using the dipole approximation technique) and MOTIS (Fortran Go of an MOS circuit analysis program). These traces assume a one word (60 bit) memory interface for data and a one instruction (15 or 30 bits) interface for instructions; i.e. there is no memory in the instruction interface.

There were four traces for the IBM 360/91 used: WATEX (execution of a Fortran program compiled using the Watfiv compiler; the program is a combinatorial search routine), WATFIV (Fortran compilation of the WATEX program using the Watfiv Fortran compiler, the compiler presumably written in assembler), APL (execution of an APL program which does plots at a terminal; interpreter presumably written in assembler), and FFT (execution of FFT program written in Algol, compiled with the AlgolW compiler). These programs have been extensively analyzed in [Smit76], [Smit79] and [Smit82]. These traces assume an 8 byte interface with memory, but with no memory; all bytes are discarded after each individual fetch.

Finally, there are four short traces for the Motorola 68000. The programs are: PL0 (the PL0 program from Wirth, "Algorithms + Data Structures = Programs"), MATCH (pattern matching program from Kernighan and Plauger, "Software Tools in Pascal"), SORT (quicksort) and STAT (trace statistics program); each source program is written in PASCAL. These traces were gathered with a hardware monitor from a working 68000 in real time, and only differentiate between fetches (reads and ifetches) and writes; they reflect the actual implementation of the 68000.

3. Experimental Results

A number of different cache designs were simulated, each for some or all of the traces; not all experiments were run with all traces because of the large number of cases. In this section, we present some of the results of those simulations.

3.1. Overall Miss Ratios

Table 1 shows the miss ratios for 57 traces (treating the LISPC and VAXIMA traces as five each) for a fully associative cache managed with LRU replacement, demand fetch, no task switch purges, copy back with fetch on write, and 16 byte lines. The length of each trace is shown in table 2. The same data is plotted in figure 1. The full associativity and the lack of task switching indicate that in a real machine, performance would be lower. There are many interesting observations to be made regarding this data, and we do so an item at a time.

The worst performance (highest miss ratio) is observed for the MVS1 and MVS2 traces and for the CGO1, CGO2, WATFIV, FCOMP1 and CCOMP1 traces. The first two are from the world's largest operating system, which is known to have poor locality. The Fortran (FCOMP1), Cobol (CCOMP1) and Watfiv (WATFIV) compilers are also large, mature pieces of software. It is worth comparing these results with those from [Hard80], which are reproduced in figure 2. The MVS2 trace corresponds fairly well with the MVS trace miss ratios from [Hard80], although the line size for [Hard80] is 32 bytes as compared with 16 bytes here. (In the range of memory sizes from 16K to 64K, the miss ratio drops rapidly with increasing line size [Smit82], [Kuma79], which suggests that even the MVS2 trace doesn't perform as badly as MVS does in practice.) The problem state results from [Hard80] are

Trace: Capacity	FGO1	FGO2	CGO1	FCOMP1	CCOMP1	CGO2	FGO3	MVS1	MVS2	WATFIV	WATEX	APL	FFT1
32	.478	.479	.438	.470	.495	.495	.438	.449	.455	.523	.466	.468	.545
64	.410	.403	.359	.380	.331	.331	.366	.392	.388	.455	.405	.375	.489
128	.368	.235	.335	.326	.290	.290	.338	.349	.340	.404	.359	.223	.436
256	.292	.131	.301	.284	.230	.230	.320	.318	.302	.323	.261	.146	.222
512	.245	.120	.278	.227	.205	.205	.287	.279	.269	.267	.180	.089	.179
1024	.219	.100	.228	.179	.169	.170	.206	.236	.218	.216	.079	.054	.133
2048	.173	.046	.161	.150	.134	.136	.135	.201	.183	.181	.053	.045	.004
4096	.011	.007	.053	.095	.109	.113	.068	.152	.148	.149	.020	.029	.003
8192	.0054	.0036	.044	.068	.031	.049	.050	.119	.121	.060	.006	.022	.003
16384	.0049	.0036	.041	.039	.0066	.034	.047	.057	.080	.013	.004	.019	.003
32768	.0047	.0036	.039	.022	.0066	.034	.044	.023	.055	.008	.004	.016	.003
65536	.0047	.0036	.021	.019	.0066	.034	.043	.016	.038	.008	.004	.013	.003
131072	.0047	.0036	.021	.019	.0066	.034	.043	.016	.032	.008	.004	.013	.003

Trace: Capacity	LISP2	LISP5	LISP8	LISP11	LISP14	VAXMA1	VAXMA7	VAXMA13	VAXMA19	VAXMA25	TWOD	PPAL	DPOLE	MOTIS	PPAS
32	.462	.451	.461	.467	.440	.406	.467	.469	.466	.408	.345	.337	.356	.440	.356
64	.364	.347	.364	.364	.336	.330	.355	.359	.355	.319	.317	.313	.334	.334	.332
128	.307	.291	.307	.305	.270	.306	.309	.306	.306	.290	.280	.272	.303	.159	.286
256	.251	.231	.253	.245	.215	.093	.244	.244	.250	.141	.182	.191	.255	.106	.241
512	.205	.170	.204	.198	.142	.082	.198	.195	.211	.067	.095	.125	.197	.087	.175
1024	.117	.104	.111	.117	.098	.074	.155	.136	.175	.028	.066	.088	.142	.074	.124
2048	.093	.058	.091	.097	.041	.066	.116	.102	.118	.023	.046	.063	.079	.052	.069
4096	.069	.041	.067	.072	.024	.059	.095	.076	.035	.022	.042	.057	.028	.045	.029
8192	.045	.027	.042	.050	.016	.051	.065	.042	.016	.020	.011	.008	.0018	.018	.024
16384	.019	.017	.017	.025	.010	.044	.051	.029	.0098	.019	.0071	.007	.0018	.018	.024
32768	.009	.011	.009	.010	.009	.040	.038	.025	.0081	.018	.0066	.007	.0018	.018	.024
65536	.009	.011	.009	.010	.009	.037	.026	.019	.0081	.017	.0066	.007	.0018	.017	.024

Trace: Capacity	ZVI	ZSORT	ZGREP	ZPR	ZOD	ZC3	ZC4	ZDIFF	ZSED	ZCPP	ZED	ZLD	ZNN	MATCH	PLO
32	.305	.299	.335	.298	.351	.334	.313	.362	.322	.340	.288	.297	.338	.277	.331
64	.221	.215	.277	.202	.280	.217	.227	.159	.256	.208	.239	.214	.261	.199	.202
128	.175	.158	.200	.139	.170	.155	.179	.094	.163	.170	.113	.161	.138	.144	.155
256	.082	.071	.139	.024	.064	.119	.135	.044	.077	.132	.038	.072	.057	.083	.102
512	.070	.048	.044	.006	.052	.074	.114	.035	.048	.080	.005	.055	.048	.040	.069
1024	.050	.0039	.036	.0032	.010	.051	.096	.016	.026	.056	.004	.009	.038	.014	.047
2048	.021	.0023	.022	.0014	.003	.027	.059	.011	.0039	.025	.0026	.0013	.029	.008	.016
4096	.010	.0013	.0023	.0011	.0015	.013	.040	.009	.0023	.009	.0024	.0010	.0086	.007	.010
8192	.0039	.0013	.0022	.0011	.0015	.0038	.026	.007	.0019	.006	.0024	.0010	.0066	.007	.010
16384	.0033	.0013	.0022	.0011	.0015	.0032	.014	.005	.0019	.006	.0024	.0010	.0047	.007	.010
32768	.0033	.0013	.0022	.0011	.0015	.0032	.0082	.005	.0019	.006	.0024	.0010	.0047	.007	.010
65536	.0033	.0013	.0022	.0011	.0015	.0032	.0082	.005	.0019	.006	.0024	.0010	.0047	.007	.010

Trace: Capacity	VCCOM	VSPICE	VOTMD1	VPUZZLE	VTROFF	VLO	VQSORT	VSEDX	VTOWERS	VAWK	VYMRGE	VC2	QSORT	STAT
32	.385	.480	.445	.355	.402	.473	.580	.653	.551	.448	.425	.370	.278	.373
64	.269	.368	.308	.285	.318	.316	.383	.308	.273	.327	.355	.278	.146	.264
128	.134	.263	.232	.087	.285	.167	.170	.170	.141	.296	.174	.216	.085	.066
256	.101	.199	.182	.019	.257	.122	.084	.070	.018	.254	.041	.106	.046	.045
512	.090	.102	.139	.015	.217	.079	.006	.048	.0004	.219	.024	.081	.005	.036
1024	.073	.045	.0020	.012	.149	.042	.003	.024	.0002	.146	.015	.060	.0015	.004
2048	.055	.028	.0017	.0100	.041	.019	.003	.0035	.0002	.082	.013	.012	.0015	.004
4096	.038	.020	.0015	.0090	.012	.013	.003	.0022	.0002	.026	.011	.006	.0015	.004
8192	.027	.013	.0015	.0089	.0052	.0065	.003	.0022	.0002	.008	.011	.005	.0015	.004
16384	.024	.0075	.0015	.0086	.0043	.0047	.003	.0022	.0002	.004	.010	.005	.0015	.004
32768	.023	.0073	.0015	.0080	.0043	.0047	.003	.0022	.0002	.004	.0075	.005	.0015	.004
65536	.022	.0073	.0015	.0080	.0043	.0047	.003	.0022	.0002	.004	.0075	.005	.0015	.004

Table 1

Trace	Machine	%Read	%Write	%Ifetch	Source	#Ilines	#Dlines	%Branch	Aspace	Length
ZVI	Z8000	14.3	5.7	80.0	C	626	203	9.2	13264	250K
ZSORT	Z8000	17.9	9.0	73.1	C	167	147	6.9	5024	250K
ZGREP	Z8000	16.7	8.9	74.4	C	394	145	9.3	8624	250K
ZPR	Z8000	15.2	6.4	78.5	C	196	75	13.2	4336	250K
ZOD	Z8000	14.0	9.5	76.5	C	253	122	12.9	6000	250K
ZCPP	Z8000	17.0	8.5	74.5	C	433	307	9.6	11840	130K
ZC3	Z8000	16.6	8.5	74.9	C	441	364	10.0	12880	250K
ZC4	Z8000	15.4	9.1	75.5	C	713	1338	10.3	32816	250K
ZDIFF	Z8000	19.6	4.5	75.9	C	953	277	16.0	19680	250K
ZSED	Z8000	18.9	12.6	68.5	C	211	270	7.1	7696	250K
ZED	Z8000	16.7	9.2	74.1	C	103	81	8.3	2944	78K
ZLD	Z8000	17.1	12.5	70.4	C	163	90	9.2	4048	250K
ZNN	Z8000	16.1	3.6	80.2	C	436	723	13.8	18544	250K
Average	Z8000	16.6	8.3	75.1		391	319	10.5	11361	
VCCOM	VAX	25.8	7.6	66.6	C	1022	4572	16.2	89504	250K
VPUZZLE	VAX	25.8	7.9	66.3	C	132	1847	8.5	31664	250K
VTROFF	VAX	25.8	12.8	61.4	C	578	502	13.6	17280	250K
VLS	VAX	27.5	16.7	55.8	C	444	496	18.1	15040	202K
VQSORT	VAX	29.1	27.3	43.7	C	31	689	19.8	11520	250K
VSEDX	VAX	27.4	15.7	56.9	C	292	248	30.7	8640	250K
VTOWERS	VAX	30.5	27.8	41.7	C	9	47	25.0	896	250K
VAWK	VAX	26.1	13.6	60.2	C	328	721	10.8	16784	250K
VYMRGE	VAX	25.4	20.4	54.3	C	207	1668	17.4	30000	250K
VC2	VAX	22.9	11.3	65.7	C	238	1007	17.2	19920	250K
VSPICE	VAX	30.2	18.6	51.3	Fortran	1111	719	14.0	29280	250K
VOTMD1	VAX	31.3	22.2	46.5	Pascal	147	219	18.1	5856	250K
Average	VAX	27.3	16.8	55.9		378	1061	17.5	23032	
LISPC2	VAX	29.6	17.9	52.5	Lisp	1227	1067	15.2	36704	250K
LISPC5	VAX	29.1	17.2	53.7	Lisp	1479	1335	15.2	45024	250K
LISPC8	VAX	29.5	17.8	52.5	Lisp	1127	1057	15.4	34944	250K
LISPC11	VAX	29.7	17.8	52.5	Lisp	1229	1138	15.2	37872	250K
LISPC14	VAX	28.8	16.5	54.7	Lisp	988	1310	15.4	37768	250K
VAXIMA1	VAX	28.5	4.9	66.6	Lisp	624	8431	9.7	144880	250K
VAXIMA7	VAX	31.0	16.4	52.6	Lisp	4399	2037	13.6	102976	250K
VAXIMA13	VAX	30.3	16.7	53.0	Lisp	2984	1724	14.1	75328	250K
VAXIMA19	VAX	28.5	18.3	53.1	Lisp	947	1078	13.9	32400	250K
VAXIMA25	VAX	31.5	12.6	55.9	Lisp	15	4240	13.6	68080	250K
Average	VAX (lisp)	29.7	15.6	54.7		1502	2342	14.1	61598	
MVS1	370	31.2	18.3	50.5	Assem	1381	2726	12.8	65712	500K
MVS2	370	30.2	18.1	51.6	Assem.	5555	8823	14.2	230048	500K
FGO1	370	30.0	17.6	52.5	Fortran	495	670	15.0	18640	250K
FGO2	370	30.6	5.7	63.7	Fortran	303	591	20.1	14304	250K
FGO3	370	30.0	12.8	57.2	Fortran	1215	806	6.4	32336	250K
CGO1	370	41.6	34.2	24.3	Cobol	365	4060	13.3	70800	250K
CGO2	370	41.1	32.4	26.5	Cobol	449	4783	13.4	83712	250K
FCOMP1	370	29.5	20.7	49.8	Assem?	2110	2532	14.9	74272	250K
CCOMP1	370	30.8	9.9	59.3	Assem?	637	996	15.9	26128	250K
Average	370	32.8	18.9	48.4		1390	2887	14.0	68439	
WATFIV	360/91	23.3	16.5	60.1	Assem?	1181	776	13.8	31312	250K
WATEX	360/91	24.5	9.1	66.4	Fortran	598	454	11.5	16832	250K
APL	360/91	21.5	8.2	70.3	APL	763	2472	18.3	51760	250K
FFT1	360/91	23.4	7.7	68.9	AlgolW	147	708	20.6	13680	250K
Average	360/91	23.2	10.4	66.4		672	1103	16.1	28396	
Match	68000	83.4*	16.6		Pascal				3440	30K
PLO	68000	85.4*	14.6		Pascal				4800	30K
QSORT	68000	90.4*	9.6		Pascal				1408	30K
STAT	68000	93.4*	6.6		Pascal				1824	30K
Average	68000	88.2*	11.8						2868	
TWOD	6400	15.4	8.4	76.2	Fortran	753	895	2.0	26368	250K
DPOLE	6400	15.4	6.6	78.0	Fortran	370	90	3.1	7360	250K
MOTIS	6400	11.9	12.1	75.9	Fortran	431	3866	9.6	68752	250K
PPAS	6400	15.3	6.6	78.1	Fortran	380	90	3.1	7520	200K
PPAL	6400	16.1	6.0	77.9	Fortran	653	1005	3.2	26528	250K
Average	6400	14.8	7.9	77.2		517	1189	4.2	27306	

Table 2

continued from last page

[Smit78] Alan Jay Smith, "Sequential Program Prefetching in Memory Hierarchies", IEEE Computer, 11, 12, December, 1978, pp. 7-21.

[Smit79] Alan Jay Smith, "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write-Through", JACM, 26, 1, January, 1979, pp. 6-27.

[Smit82] Alan Jay Smith, "Cache Memories", ACM Computing Surveys, 14, 3, September, 1982, pp. 473-530. (Revised version, 1984, entitled "CPU Cache Memories" to appear, and available from author.)

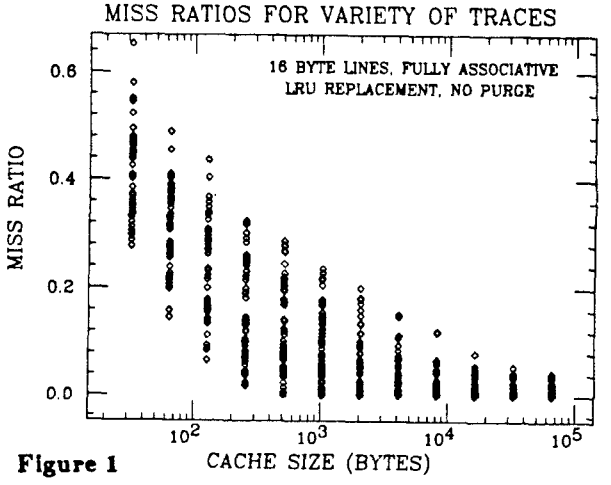

Figure 1

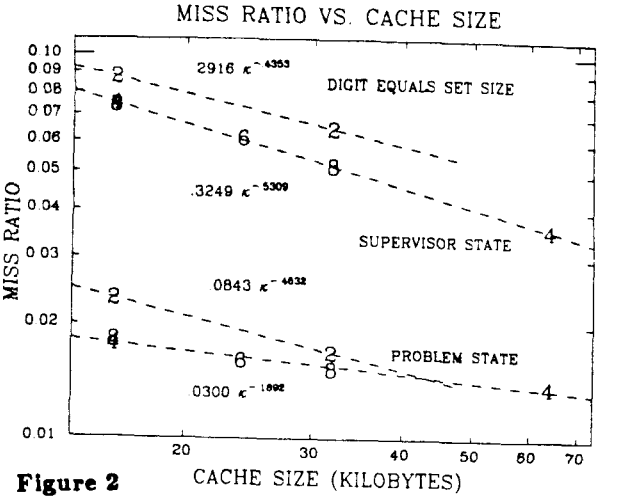

Figure 2

in rough correspondence with the miss ratios from the traces taken from the 370 and 360/91 when adjustment is made for the line size change.

The very best results occur for the 68000 traces, which have an average miss ratio at, 1024 bytes, for example, of 1.7%, but those are very short traces of very small toy programs. The next best results generally occur for the Z8000 traces, which average out to a miss ratio of 3.1% at 1024 bytes, as compared to an average miss ratio for the 370 and 360 programs of 17% at 1K; similar disparities are evident over the range of cache sizes. For reasons explained above, we are inclined to believe that the Z8000 traces (and the M68000 traces) will be poor predictors of the performance to be expected from the Z80000. The VAX programs, except those written in LISP, average to a miss ratio of 4.8% at 1K, but many of those traces also come from small, tightly coded Unix utilities, and some (VPUZZLE and VTOWERS) are toy programs.

The programs written in LISP have average miss ratios of (11.1%, 5.5%, 2.4%, 1.55%) at (1K, 4K, 16K, 64K) respectively. While those miss ratios are worse than for the other VAX traces, they are better than for the 370 and 360 traces and are not distressingly high. This result may be somewhat surprising, since it has been claimed or speculated *by some* that LISP programs would have very poor locality and would run poorly on machines with caches.

The traces from the 6400 have miss ratios near the middle of the group of all of the traces.

3.2. Trace Characteristics

In table 2 we have tabulated the characteristics of each trace. In this section we discuss that table.

The frequency of instruction fetches is significantly higher for the Z8000 traces at 75.1% and the CDC 6400 traces at 77.2 than for any of the others, and the frequency of writes is correspondingly lower. (Because the 6400 traces suppose an Ifetch sequence with no memory, it significantly overstates the number of fetches to memory for instructions that would take place in most reasonable machine implementations. Our data shows one fetch per instruction, where in most implementations, 2 to 4 instructions would be loaded each time.) Although for the M68000, we can't differentiate between reads and instruction fetches, even the M68000 traces have higher frequencies of writes than the Z8000 traces. (The sample is too small to be very sure.) This very high frequency of instruction fetches suggests one of two possibilities: either the compiler is generating poor code, such that it takes a large number of instructions to accomplish something, or the instructions aren't very powerful. It also suggests a reason why the miss ratios are so low: long sequential sequences of short instructions yield high hit ratios in a cache with a line size significantly longer than the fetch size. All of these factors can be expected to change with the move to the 32 bit Z80000: (a) The instructions will be more powerful, so the frequency of memory references due to instruction fetches should drop. (b) The compiler should be more mature and better, with the same result. (c) With the larger fetch size, the benefits of sequentiality on the hit ratio will be lessened.

Within the set of traces for each machine architecture, **the fraction of memory references which are data reads is remarkably stable**, while the fraction of writes varies widely and inversely with the frequency of instruction fetches. With the exception of the Cobol traces for the 370, the fraction of reads is insensitive to the source language. We can suggest no good explanation of this result, but find it very interesting and possibly significant.

From our data, **for both the 370 and the VAX it is reasonable to use as a rule of thumb that half of the memory references are instruction fetches**, or conversely, there are two memory references per instruction. For all of the architectures, it seems to be satisfactory to say that **reads (on the average) outnumber writes by about 2 to 1.**

The columns in table 2 labeled "#Ilines", "#Dlines" and "Aspace" are, during the period traced, respectively the number of (16 byte) distinct lines referenced by an instruction fetch, the number of lines referenced by a data read or write and the total number of bytes in the lines referenced (16*(#Ilines + #Dlines)). The purpose of the Aspace (address space) column is to indicate the size of the programs traced, as an aid to determining whether the programs are in any way representative. As can be seen, the average sizes of 2868 bytes for the M68000 traces and 11361 bytes for the Z8000 traces suggest that those programs are small. Somewhat larger are the averages for the VAX traces - 23032 (excluding the Lisp programs), the 360/91 programs - 28396, and the CDC 6400 programs (21306). The Lisp programs (61598 bytes) and the 370 programs (68439 bytes) are the largest. The latter two sets of data represent large batch programs and a large operating system. The middle group is a mixture of batch programs (CDC, 360, VAX) and utilities (VAX). The first group contains primarily utilities and toy programs.

From the Aspace figures, one can see the limitations of using traces, or at least small, short traces, to evaluate the performance of large caches. These trace runs extend at most to 500,000 memory references, and most are for 250,000 memory references. (The traces themselves are, with a few exceptions, much longer, but computer time is a limited resource. The last column shows the trace length used in each case.) With only a few exceptions the traces reference less than 64Kbytes of memory, and it makes little sense to estimate miss ratios for caches over 32K with this data, unless the traces are run for much longer periods and also unless multiple traces are combined in a realistic simulation of multiprogramming.

Comparing the #Ilines and #Dlines column shows the great variability in the ratio of the two measures. Some traces, such as CGO1, CGO2, VAXIMA1, VAXIMA25, VQSORT, and VYMRGE, show small numbers of instructions manipulating large amounts of data. Others, such as ZVI, ZGREP, and VAXIMA7, show a large ratio of data to instructions, although that condition is less common. An examination of the averages for those columns shows that **in most cases, the size of the data space is larger than that for the instruction space**. Specifically, 34 of the 57 traces show larger numbers of data lines than instruction lines; 10 of the 23 showing the converse are for the Z8000. The average number of data lines is significantly larger than the number of instruction lines for each set of traces except for the Z8000.

The column labeled "%Branch" gives the fraction of instruction fetch references that appear to be successful branches. This is determined by comparing the addresses of successive instruction fetches. If the second one is either less than the first or is more than 8 (bytes) greater, then the first is counted as a branch. This counting mechanism is needed because the address traces do not otherwise identify branch instructions. This mechanism will miss a few branches which jump over fewer than 8 bytes.

The frequency of branches is highest (17.5%) for the VAX traces (excluding Lisp) with the frequency for the 360/91 (16.%), VAX/Lisp traces (14.1%), and 370 (14.0%) traces close behind. The results for the Z8000 (10.5%) and the CDC6400 (4.2%) are much lower. For a few cases, the same or a similar program were traced for both the Z8000 and the VAX, and in those cases, the frequency of branches for the VAX is 2 to 4 times as high. The higher frequency of Z8000 instructions, and the lower rate of loads, stores and branches suggests that the VAX instructions individually do a lot more than the Z8000 instructions.

The frequency of branches does not seem to be related to the source language. Within the set of traces for each machine, there is substantial variation in the frequency of branches.

3.3. Write Back Activity

An important parameter affecting the design of the cache and memory system of a computer is the frequency of writes to memory. For a machine which uses write through, by which memory is written to on every store instruction, the write frequency is usually just the frequency in the trace of stores to memory. (The exception would be an implementation in which adjacent short writes are combined into a longer write, as when two 2-byte writes are combined into a four byte write to a memory with at least a 4 byte wide interface.) If the machine uses copy-back, however, the frequency of writes to memory is the miss ratio times the probability that a line to be pushed is dirty. The number of bytes transferred is that quantity times the line size.

In table 3, we present some data on the probability that a line to be pushed is dirty. The simulations on which these measurements are based are somewhat different than those discussed above. In this case, a 32Kbyte memory is simulated, partitioned into a 16Kbyte data cache and 16Kbyte instruction cache, and every 20,000 memory references, the cache is purged to simulate multiprogramming. The total number of lines pushed comprises those that are pushed as part of a line fetch (replacement), and also those pushed when the cache is artificially purged. The "fraction data pushes dirty" is just the fraction of all data lines that are pushed that have been modified (written to) since the time they were fetched. Four of the entries in that table (LISP Compiler, VAXIMA, Z8000-assorted, CDC 6400 - assorted) represent multiprogramming simulations, in which the traces were run through the simulator in a round robin manner, switching and purging every 20,000 memory references. (We believe that the value 20,000 is reasonable and representative, but the results are definitely sensitive to that figure.) The Z8000 assortment consists of ZVI, ZGREP, ZPR, ZOD, ZSORT; the CDC 6400 assortment includes all five CDC 6400 traces; the LISP Compiler and VAXIMA mixtures include the five trace sections described earlier.

Trace(s)	Fraction Data Line Pushes Dirty
LISP Compiler - 5 Sections	.26
VAXIMA - 5 Sections	.23
VCCOM	.63
VSPICE	.37
VOTMD1	.49
VPUZZLE	.77
VTROFF	.27
FGO1	.56
FGO2	.43
CGO1	.35
FCOMP1	.63
CCOMP1	.22
MVS1	.48
MVS2	.56
Z8000 - Assorted	.48
CDC 6400 - Assorted	.80
Average	.47

Table 3

Averaging over all of the results presented, the probability of a data push being dirty is 0.47, which is close enough to 0.5 to say that as a rule of thumb, **half of the data lines pushed will be dirty**. The standard deviation of the numbers in this table, however, is 0.18, and a quick examination shows that the range is 0.22 to 0.80, which is quite wide. Thus while 50% is a reasonable target to which to design, in practice the variation will be wide. There also appears to be no pattern to the numbers; even two different compilers for the same machine (FCOMP1, CCOMP1) have widely different probabilities of pushing a dirty data line.

It is possible to deduce a variation with cache size in the probability that a pushed line will be dirty. Clearly, the larger the cache, the longer the mean residence time of a line before it gets pushed and the higher the probability that it will be dirty. Likewise, if the mean time between cache purges (task switches) could be extended, the probability that a data line which is removed is dirty would increase. We have not collected the data necessary to characterize this trend, but we would expect it to be small, since a line that is to be modified is usually modified shortly after it is first referenced.

3.4. Instruction and Data Miss Ratios

From the same set of simulations used to generate table 3, we collected the miss ratios for the instructions in the instruction cache and the data references in the data cache; those miss ratios appear in figures 3 and 4. (Please note that the vertical scale differs between figures 3 and 4.)

Again, there is a very wide range of miss ratios among the various traces. The data miss ratios tend to be higher for small cache sizes; thereafter, the instruction or data miss ratio may be lower.

It is interesting to use this data to speculate on the performance to be expected from the 256 byte, 4 bytes/block instruction cache in the Motorola 68020 [Macg84]. The miss ratios for a 256 byte, 16bytes/block, cache can be seen from figure 3 to range from almost 0.0 to about 0.32. Miss ratios for a block size of 4 should be greater because it takes 4 fetches to load 16 bytes, rather than 1 fetch. If the average instruction is 3 bytes long, and one executes 7 instructions between branches (reasonable figures), then on the average 21 bytes of instructions will be fetched sequentially; the 4 byte block size captures little of this sequentiality. I would be inclined to predict miss ratios in the range of 0.2 to 0.6 with this design for most workloads. (This estimate is derived by extrapolating from the miss ratios in table 1, adjusting for the smaller line size. We also take into account miss ratio data for 4 byte lines, collected for another paper in preparation and not presented here. In section 4, we suggest that 0.25 is a reasonable point estimate for a 256 byte instruction cache with 16 byte lines, when used with a 32-bit architecture.)

3.5. The Effect of Prefetching

Prefetching has been strongly advocated previously by this author as a means to significantly reduce the miss ratio of a cache [Smit82, Smit78]. An additional set of simulations was run to evaluate the effectiveness of prefetching over a large variety of traces. These simulations use the same sets of traces and parameter values as were used to generate table 3: Two cache organizations were simulated, a unified (instructions and data) and a split (separate instruction and data caches) design. Each was simulated with and without prefetch always; *prefetch always* verifies that line i+1 is in the cache at the time line i is referenced, and if it is not in the cache, then it prefetches it. At intervals of 20,000 memory references (except for the M68000 traces, where the interval was 15,000), the cache is purged, to simulate multiprogramming.

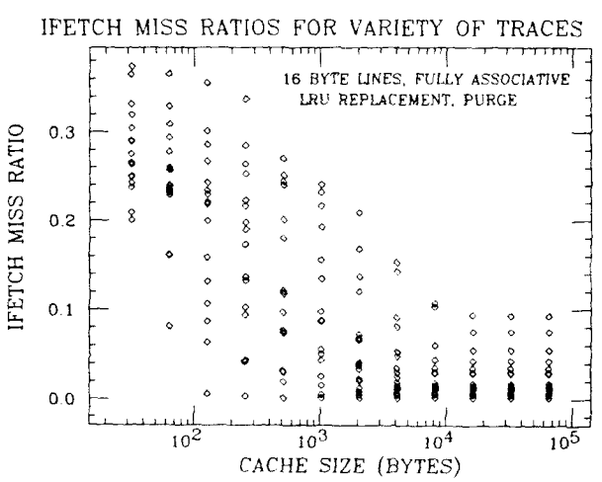

Figure 3

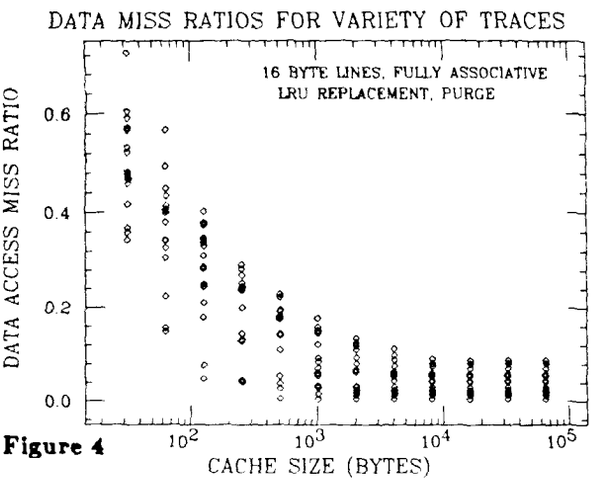

Figure 4

3.5.1. Prefetch Miss Ratios

The miss ratios for the unified cache, and for the instruction and data caches, both with and without prefetching, were tabulated and the ratio of the miss ratios with to without prefetching are plotted in figures 5, 6 and 7. The vertical scale is logarithmic and is different in each case.

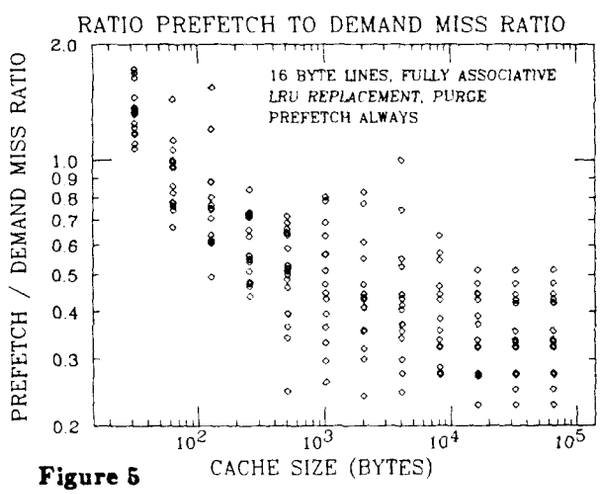

Figure 5

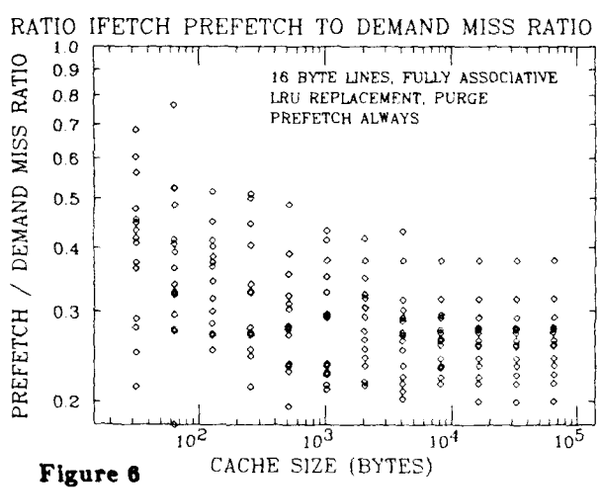

Figure 6

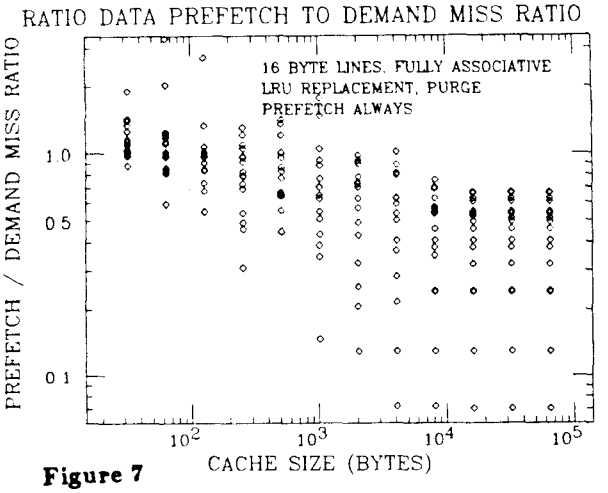

Figure 7

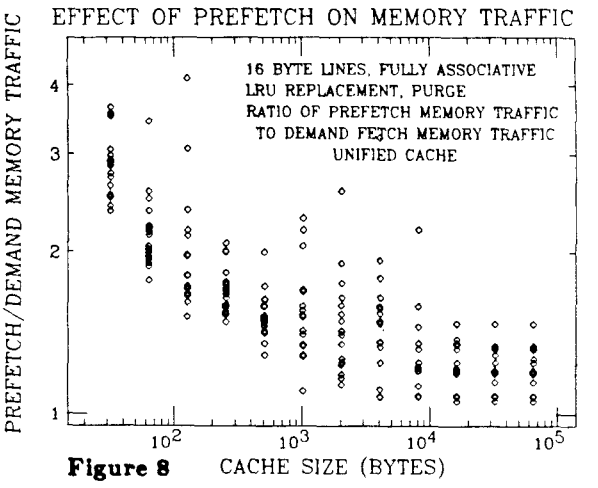

Figure 8

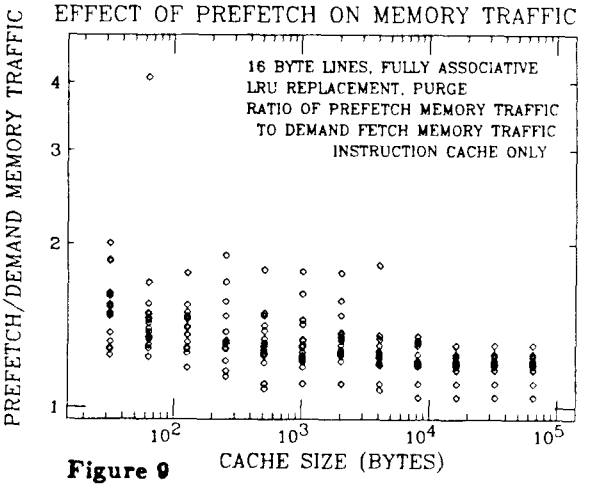

Figure 9

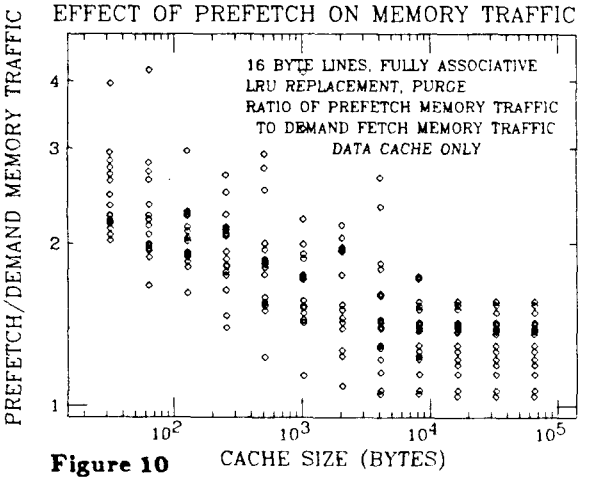

Figure 10

There are several observations to be made from figures 5, 6 and 7. First, it can be seen that **prefetching is increasingly useful with increasing cache size**. The reason for this is simple: as the cache becomes larger, the residence times of lines increases, and the probability that a prefetched line will be used increases; conversely, the probability that the line removed to make space for the prefetched line will soon be needed again decreases. The prefetch miss ratio is almost always below the demand fetch miss ratio once the cache is above 256 bytes, but the benefits aren't significant and consistent until the cache is somewhat larger. Note that the results in figure 5 are not monotonic; it is possible for the demand fetch miss ratio to drop faster with increasing cache size than the prefetch miss ratio, and what seems to have happened in a few cases is that with demand fetch an important loop becomes cache resident, while for prefetch, it doesn't quite fit.

From figure 6, we observe that **prefetching seems to always cut the instruction fetch miss ratio**, and for large cache sizes ($\geq$8K) always by more than 60%. This result is not surprising, since most instruction execution is sequential, as is prefetching. In most cases, the miss ratio for large caches is less than 30% of its level for demand fetch. It is also worth observing that the effectiveness of instruction prefetching increases with decreasing block (line) size, so with its small 4 byte line size, the M68000 instruction cache could expect a dramatically lower miss ratio with the use of prefetching.

Data prefetching can also be observed to be effective for large caches, and **for data caches of 8Kbytes or more, prefetching always causes the data miss ratio to drop**, with the average drop on the order of 50%. For smaller cache sizes, prefetching is much less effective, and in some cases increases the miss ratio. That prefetching is effective for data has been previously observed [Smit78]; data is often stored and referenced sequentially. On the other hand, there is certainly less locality in data than in instructions, leading to the lesser effectiveness.

3.5.2. Prefetch Memory Traffic Rates

Associated with prefetching are some problems, difficulties and disadvantages, discussed in some detail in [Smit82]. One of the most important costs when using prefetching is an unavoidable increase in the traffic between the cache and main memory, since prefetching not only fetches what is needed, but sometimes fetches that which is not needed and will not be referenced. In a microprocessor based system with a shared bus, the traffic capacity of the bus limits the number of microprocessors that can be used, and thus although prefetching cuts the miss ratio of each processor and presumably increases its performance, the increase in traffic can lower the maximum possible system performance level.

In figures 8, 9 and 10 we have computed the factor by which the memory traffic increases when "prefetch always" is used. The average increase is shown in table 4. (The average is computed by summing the prefetch traffic for all of the traces and dividing it by the demand fetch traffic; it is not just

Cache Size	Unified Cache	Instruction Cache	Data Cache
32	2.870	1.519	2.440
64	2.138	1.463	2.349
128	1.879	1.368	2.064
256	1.679	1.356	1.942
512	1.547	1.407	1.949
1024	1.602	1.313	1.851
2048	1.476	1.309	1.707
4096	1.537	1.246	1.621
8192	1.389	1.258	1.411
16384	1.269	1.194	1.340
32768	1.213	1.191	1.335
65536	1.209	1.191	1.335

Table 4

Average ratio of memory traffic for prefetch to demand fetch. (Sum of prefetch memory traffic divided by sum of demand fetch memory traffic.)

an average of the numbers in figures 8, 9 and 10, which would be misleading.) It can be seen that the traffic increases less for the instruction cache than for the data cache. **For the instruction cache, the percent by which memory traffic increases due to prefetching levels off at about 20% (factor of 1.2 as much traffic) beyond 4K bytes, and for the data cache, beyond 8K bytes, it stablises at about 30% to 40%.** Whether or not this level of increase is acceptable depends on the specific system design.

4. Rules of Thumb, Fudge Factors and Consensus Miss Ratios

In the preceding section (3), we examined a variety of measurements from trace driven simulations using a large number and variety of program address traces. One of purposes of that discussion was to show the range of behavior observed, and to see if behavior could be related to facts about the traces, such as the machine architecture or the source language. In this section, we take the data in the previous section, combine it with data previously published (e.g. [Smit82], and figure 2), add to that the author's additional experience with cache memory studies and consulting projects, and produce a set of *opinions*. Some of these opinions could be further justified with additional simulations, but those are beyond the scope of this paper. The purpose of these opinions is not to match exactly any one trace or the characteristics of any specific existing workload, but to provide guidance to the designer as to (approximately) what workload to expect.

4.1. Overall Miss Ratios

In table 5, we have created what we consider to be reasonable miss ratios to use as a *design estimate* for a 32 bit architecture running fairly large programs and a mature (i.e. large) operating system. The unified miss ratios were estimated from data in figures 1 and 2, and the instruction and data miss ratios came from figures 3 and 4. In each case, the number picked is towards the worst of the values observed, perhaps at the 85'th percentile or so. In the range of 32 bytes to 512 bytes, doubling the cache size seems to cut the miss ratio by about 14%; from 512 to 64K, by about 27%, and overall, by about 23%.

For validation of the data in table 5, we can compare with the results reported in [Clar83]. First, however, it is important to take note of the complex VAX 11/780 design architecture. That machine has an instruction buffer which prefetches in a complicated manner, not matching ours exactly. That machine also uses a set associative cache with 2

Design Target Miss Ratios

Cache Size	Unified	Instructions	Data
32	.50	.33	.55
64	.40	.30	.45
128	.35	.27	.35
256	.30	.25	.28
512	.27	.23	.20
1024	.21	.20	.16
2048	.17	.15	.12
4096	.12	.10	.10
8192	.08	.06	.08
16384	.06	.05	.06
32768	.04	.03	.04
65536	.03	.02	.03

With 16 byte line. **Table 5**

elements per set. (The effect of the latter on the miss ratio should be small.) Thus, while we compare figures from [Clar83] with ours, they do not represent exactly same thing.

Clark reports data miss ratios of 16.5% and instruction miss ratios of 8.6%, for an 8 byte line. He also reports an overall read miss ratio of 10.3%. (Instruction miss ratios are for a design with an 8 byte data path and memory.) For a cache size of 8Kbytes, the miss ratio can usually be close to halved by changing to 16 byte lines [Smit82], so our figure of 8% would be closer to 12% to 16% for an 8Kbyte cache with 8 byte lines. This compares to the actual 10.3% or 12.5% ((16.5+ 8.6)/2) figure, which is not out of line. Clark also reports the results of an experiment in which he effectively halved the cache size, to 4Kbytes. The data, instruction and overall miss ratios are reported to be 23.1%, 15.7% and 17.5%, which compares to our prediction of 12% for twice the line size, or about 18% - 20% for an 8 byte line size. (We are less close for the instruction and data figures, where we claim miss ratios for the two that are approximately equal, and Clark finds better results for instruction fetch than for data reference. This is likely due to the very different fetch mechanisms for instructions and data in the 11/780.)

It is also interesting to compare our predictions with those in [Alpe83] for the Z80000. There, it is suggested that with 16 byte blocks, the 256 byte cache will have a miss ratio of 12%; we predict about 30%.

4.2. Architecture Type

Although it is generally seems clear that a machine with a small, simple instruction set (like the RISC machine [Patt85]) will require more instructions to accomplish a given amount of work, it is hard to document that from table 2 because of the varying workloads, source languages, design architectures and compilers. One would expect that the frequency of instructions would be lowest for the VAX, which is the most complicated architecture and has the most powerful instructions, next lowest for the 360/370 and highest for the CDC6400 which has few and simple instructions. (We are omitting the Z8000 from this discussion since it is a 16-bit architecture.) If one looks at table 2, the frequency of instructions increases in this order: 370, VAX, 360/91 and CDC 6400. The VAX traces were all generated under Unix, and Unix compilers are generally not considered to produce highly efficient code. Likewise, the Watfiv and AlgolW compilers produce poor code. Also, as noted earlier, the VAX traces may overstate the frequency of instructions, due to the lack of "memory" in the instruction fetch mechanism. The IBM 370 and the CDC6400 compilers are mature and should produce efficient, optimized code.

Comparing the IBM 370 traces (55% instructions, excluding the Cobol traces) and the CDC 6400 traces (77% instructions) we find the expected result. Based on these two data points, **we claim that the ratio of instructions to data loads & stores will range from about 1:1 for relatively complex (32 bit) architectures up to about 3:1 for extremely simplified architectures,** assuming a standard (single) register set.

It should be evident that with simplified architectures, instruction sequences will be longer. Thus, large block sizes and sequential prefetching will be relatively more useful than for complex architectures. On the other hand, code sizes will be larger, suggesting that for the same block size, the miss ratio may be higher for the simple architecture. (In [Patt83] is it suggested that this problem can be overcome by using data compression techniques on the code.)

The frequencies of loads and stores reflects both the number of registers and the degree of register use optimization in the compiler. More registers mean that more information can be kept in the registers, rather than being loaded and unloaded, but that advantage may be more than offset by a compiler which loads and stores all the registers when a procedure call occurs. These two factors have to be balanced off in predicting the change when moving to a new architecture; our data in table 2 isn't sufficient to make any definite statement. The approach taken in "RISC" machines [Patt85] is that of multiple register sets, arranged as a stack; this should significantly cut the amount of memory traffic relative to a similar architecture with only one register set.

It is also worth noting that the frequency of branches seems to and should have the same trends with architecture type as the frequency of instructions. If instructions are simple and not very powerful, the distance between successful branches should be large, and conversely. The data in table 2 tends to confirm this impression, with higher frequencies of successful branches for the VAX and 370, and lower frequencies for the Z8000 and CDC6400. That data can be used to make reasonable estimates of branch frequencies in an as yet unimplemented architecture by interpolating among the machines for which we show information, based on the architecture complexity.

5. Conclusions

There have been two purposes to this paper. First, we have presented the results of a very large number of trace driven simulations along with a discussion of the advantages and disadvantages of that approach. That has been in order to give the reader **an understanding of the value and meaning of workload selection for cache evaluation.** The second goal of this paper has been to **propose some miss ratios and other parameter values which can be used by the computer architect in designing a new machine and in predicting its performance.** The latter has been based not only on the data presented here but on the professional opinion and experience of the author, and should be taken with suitable "error bounds." When in doubt, it is better (at least for one's own use, as opposed to marketing) to lean in the pessimistic direction and make conservative estimates. It is also worth noting that from the data here, and without any known case otherwise, **caches always work**; a cache of any reasonable size always has a hit ratio high enough to make it worth while. The traffic ratio, however, may not be lower than 1.0 [Hill84] and that parameter needs to be carefully watched. (Traffic ratio is the ratio of the memory traffic with a cache to that without a cache.)

There are two principal ways in which this work needs to be extended. First, the effect of line size on miss ratio needs to be quantified beyond the general statements made here and the results in [Smit82] and [Kuma79]; research on this topic is in progress. Second, our sample workloads here could be usefully augmented in two ways; first the quantity could be increased by increasing the number of traces per machine and the number of machines traced; second, the quality could be improved by obtaining traces that are comparable with respect to assumptions about instruction buffering strategies and with regard to the quality of the compilers used.

Acknowledgements

The traces used for this research have come from many sources. My thanks to Len Shustek for generating the IBM 360/91 traces at SLAC, to Amdahl Corporation and Bill Harding for the IBM 370 traces, to Zilog Corporation, John Banning and Juan Porcar for the Z8000 traces, to Signetics Corporation and Martin Freeman for the M68000 traces, to John Lee for the CDC 6400 traces, and to Robert Henry and George Taylor for the VAX 11/780 traces. Thanks also to Howard Sachs who asked the kind of questions that prompted this research.

Bibliography

[Alpe83] Donald Alpert, Dean Carberry, Mike Yamamura, Ying Chow and Phil Mak, Electronics, July 14, 1983, pp. 113-119.

[Clar83] Douglas Clark, "Cache Performance in the VAX-11/780", ACM TOCS, 1, 1, February, 1983, pp. 24-37.

[Curr75] Brian Currah, "Some Causes of Variability in CPU Time", Share Inc., Computer Measurement and Evaluation, selected papers from the SHARE Project, volume III, Dec. 1973-March, 1975, pp. 389-392.

[Fran84] Steven Frank, "Tightly coupled multiprocessor system speeds memory-access times", Electronics, January 12, 1984, pp. 164-169.

[Hard80] William Harding, M. H. MacDougall and William Raymond, "Empirical Estimation of Cache Miss Ratios as a Function of Cache Size", unpublished internal report, Amdahl Corp., Sunnyvale, Ca., September, 1980. (Relevant data appears in [Smit82].)

[Hatt83] Akira Hattori, Minoru Koshino and Shigemi Kamimoto, "Three Level Hierarchical Storage System for Facom M380/382", Proc. IFIP Conf., September, 1983, pp. 693-697.

[Hill84] Mark Hill and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proc. 11'th Ann. Symp. on Computer Architecture, June, 1984, Ann Arbor, Michigan, pp. 158-166.

[Kuma79] B. Kumar, "A Model of Spatial Locality and Its Application to Cache Design", unpublished technical report, Computer Systems Laboratory, Stanford University, Stanford, Ca., 1979.

[Macd84] Myron H. MacDougall, "Instruction Level Program and Processor Modeling", IEEE Computer, July, 1984, pp. 14-24.

[Macg84] Doug MacGregor, Dave Mothersole and Bill Moyer, "The Motorola MC68020", IEEE Micro, August, 1984, pp. 101-118.

[Merr74] Barry Merrill, "370/168 Cache Memory Performance", SHARE Computer Measurement and Evaluation Newsletter, No. 26, July, 1974, pp. 98-101.

[Mila75] G. Milandre and R. Mikkor, "VS2-R2 Experience at the University of Toronto Computer Centre", SHARE 44 Proc., Los Angeles, Ca., March, 1975, pp. 165-178.

[Patt83] David A. Patterson, Phil Garrison, Mark Hill, Dimitris Lioupis, Chris Nyberg, Tim Sippel and Korbin Van Dyke, "Architecture of a VLSI Instruction Cache for a RISC", Proc. 10'th Ann. Int. Symp. on Computer Architecture, June, 1983, pp. 108-116.

[Patt85] David Patterson, "Reduced Instruction Set Computers", CACM, 28, 1, January, 1985.

[Smit76] Alan Jay Smith, "A Modified Working Set Paging Algorithm", IEEETC, C-25, 9, September, 1976, pp. 907-914.

continued after tables 1 & 2