

BACH: BYU Address Collection Hardware, The Collection of Complete Traces

J. Kelly Flanagan, Brent E. Nelson, James K Archibald, Knut Grimsrud

Department of Electrical and Computer Engineering, 459 Clyde Building, Brigham Young University, Provo UT 84602

Abstract

Trace driven simulation is an important tool for computer systems performance analysis and prediction, but its accuracy decreases when incomplete or inaccurate traces are used for input. Nevertheless, many memory hierarchy simulation studies have been published which rely on such traces. In this paper we describe BACH, a hardware monitor developed to capture long, accurate, and complete traces on a variety of hardware and software platforms. BACH traces are long — traces containing over 200 million contiguous references have been collected to date. BACH traces are accurate — in contrast to other techniques such as inlining they contain almost no time and space dilation effects. BACH traces are complete — they contain all references generated by the CPU during tracing, including prefetches and demand fetches from user code, system calls, exceptions, interrupts, and other system code. Finally, the traces produced using BACH are available to members of the general research community. In addition, we demonstrate the usefulness of the traces acquired using BACH through a cache simulation study. The miss rates obtained using BACH traces are shown to be as much as 50 times higher than those obtained using other traces. It is also shown that with a memory access time of 30, predictions of effective access time can be off by as much as a factor of 2 when using inaccurate or incomplete trace data.

1 Introduction and Background

Trace driven simulation is an important tool for computer systems performance analysis and prediction. It has been used extensively in the past to study memory system performance, TLB behavior, network performance, instruction mixes, and operating system performance. As with all modelling techniques, the use of realistic data as input to the model is essential in order to make accurate performance predictions. In the case of trace-driven simulation this input data is a *trace*, or stream of references. For cache modelling, the trace is the stream of memory reads, writes, and instruction fetches generated by the CPU.

A number of trace-driven cache simulation studies have been published [1, 2, 3, 4, 5], many of which have used incomplete or inaccurate traces. For this paper, we define an

incomplete trace to be one which does not contain all CPU-generated memory references including those from user code, system calls, system code, interrupts, and exceptions. Similarly, an *inaccurate* trace is one whose reference stream has been distorted by the tracing mechanism. Finally, for accurate system performance evaluation studies we believe that long traces are necessary as shown in [2].

In this paper we describe an important new tool, BACH, to support computer systems performance evaluation which enables the collection of accurate and complete traces. Traces collected using BACH are unique in the published literature we are aware of:

- They contain a record of all system activity due to user code, system calls, operating system overhead, exceptions, and interrupts.
- They are contiguous and contain all references generated by the CPU during tracing.
- Their length is limited only by available secondary storage.
- When collected in conjunction with lightweight instrumentation running on the traced machine, they contain much more information than conventional CPU address traces.
- They are available to the research community.
- They contain distortion effects (time and space dilation) which have been quantified to be about 1% and which can, in general, be removed from the trace by post processing.

A major thrust of our current research program is to conduct comparative studies on a variety of CPU architectures and operating systems. Accordingly, BACH was designed to be used to trace a number of systems. To date it has been used on an i486 running UNIX SysVR3.2, UNIX SysVR4, MACH 2.6, MACH 3.0, and DOS 5.0 as well as a MC68030 running HP-UX. We are currently working on a SPARC adaptation.

The remainder of the paper is organized into the following sections:

- A survey of existing trace collection methodologies
- An overview of the BACH tracing mechanism
- Contents of current BACH traces
- Discussion of advantages and disadvantages of the BACH system
- A set of case studies which illustrate the usefulness of BACH for systems performance evaluation and which quantify the error resulting from the use of less accurate or incomplete traces.

2 Other Traces and Trace Gathering Techniques

Several address trace collection schemes have been used and described in the literature. These are described in the following sections.

2.1 Instruction Modification Techniques

Instruction modification or *inlining* produces traces by instrumenting source, object, or executable code with instructions that write the trace information to a buffer or file. The instructions that are added are generally placed at the beginning of each basic block of code. This is sufficient to create an address trace using postprocessing. The acquisition of data references requires much more instrumentation which can be difficult to generate for complex addressing modes. Inlining was used in [2, 6, 7].

Traces gathered in this way typically do not contain operating system references and do not include the number of cycles between references. Operating system references are not included because it is difficult to instrument the complete operating system. Likewise, the time between memory references is difficult to determine without making simplifying assumptions about system operation. The instructions added at basic block boundaries must themselves be executed which increases the execution time of the process. This increased execution time is known as *time dilation* and is 10:1 for the implementation reported in [2]. The added instructions also increase the size of the application and produce additional references that pollute the system cache. The increase in the size of the executable code is known as *space dilation*.

2.2 Microcode Modification Methods

Microcode modification techniques generate trace data through the modification of the microroutines that produce memory references. The technique described in [8], ATUM, modifies each microroutine to store address information to a reserved portion of memory in addition to its normal operation. This implementation is operating system independent, requires no additional hardware, and can collect both physical and virtual address streams. ATUM also has disadvantages in that the added microcode takes time to execute which distorts the resulting trace, only short traces can be captured due to the necessity of storing them in the small main memory buffer, and it is only possible on machines that have modifiable microcode such as the VAX.

2.3 Single Stepping or T-bit Methods

Some processors have built-in tracing mechanisms. In the VAX architecture for example, setting the T-bit causes the processor to trap at the beginning of each instruction. A user level tracing routine can then be invoked that records the address of the next instruction. Data references can be obtained by executing the instruction using an interpreter. However, operating system references are not captured and a time dilation of approximately 100:1 is typical [8].

2.4 Processor Simulation

It is possible to create a simulation model which represents an entire computer system. This model can then be used to execute applications and produce address traces as in [9]. This technique is flexible since the software model can be modified to produce traces containing the desired information, but it has several disadvantages: (i) to enable the execution of complicated programs and operating systems the model must be extremely

complex, (ii) the absence of an executing operating system makes the tracing of multi-programming workloads difficult, and (iii) a time dilation of 1000:1 or more is typical [8].

2.5 Hardware Monitors

Hardware monitors collect address traces by passively monitoring the signals generated by the target CPU [10, 11]. These signals are stored in the hardware monitor's local memory until the buffer is full. When full, the buffer is emptied to secondary storage. Since the signals from the CPU are monitored in real time no time dilation is introduced. The resulting traces are *complete*, containing all CPU references. Although hardware monitors capture complete traces they typically have small local memories and therefore produce short traces.

The BACH tracing system uses a hardware monitoring technique combined with software to overcome most of the problems of this approach. The remainder of this paper will describe BACH and its usefulness for performance evaluation studies.

3 The BACH Hardware/Software System

This section describes BACH's basic operation and organization. This discussion is in general terms since BACH can be used to trace a variety of computer systems. The specific details for the i486 system used to generate the traces for this paper will be given in inset sections.

3.1 Overview of BACH's Collection Methodology

A typical BACH setup consists of the machine being traced (TRACEE), BACH, and an extractor (EXTRACT). BACH begins collecting trace data when TRACEE enables tracing. BACH monitors its CPU pins, identifying bus transactions and recording each transaction it is configured to save. When BACH's internal buffer is full it interrupts TRACEE. While TRACEE spins in an interrupt loop BACH's internal buffer is emptied by EXTRACT. The buffer contents may be stored to secondary storage media or processed while being extracted. This process may be repeated as many times as desired, producing a contiguous trace. Tracing is complete when TRACEE disables tracing.

3.2 BACH's Hardware

BACH consists of a backplane, static RAM, and control circuitry. It connects to both TRACEE and EXTRACT via cables.

TRACEE includes two extra hardware items for tracing. First, a small amount of custom circuitry is required to monitor the CPU signals, decode the desired bus cycles, and generate control signals for BACH. Second, an I/O interface device inside TRACEE allows the assertion of other control lines in addition to those available at the CPU pins. For example, a one bit signal is currently used to indicate when the system enters and exits the kernel. This allows each trace record or CPU reference to be attributed to either user or system activity. Finally, EXTRACT is a workstation and is used to download the trace data onto tape, disk, other storage media, or process it immediately.

i486 Specifics:

The i486 CPU is removed and inserted into an interface board that conditions the signals for transmission to BACH. This interface board with the CPU is then plugged into the original CPU socket. The board decodes the i486 bus transactions and latches the appropriate bus information at the proper times.

A commercially available 24 bit parallel I/O board is also installed in the i486 system which allows the assertion of other control lines in addition to those available at the CPU pins. These signals are used for a number of purposes: (i) one is asserted by TRACEE to enable tracing, (ii) another is asserted by TRACEE to mark each reference as being from user or system code, (iii) BACH asserts one to indicate that its buffer is full, (iv) BACH asserts another when its buffer has been emptied and it is ready to proceed.

3.3 BACH's Software

The amount of information that BACH can extract is strongly dependent on what instrumentation software is running on TRACEE. For BACH to collect basic traces, a small amount of required software must be written. Additional lightweight instrumentation can also be used to help collect added information such as system call usage and context switch statistics.

3.3.1 Required Software

The required software performs two tasks: (i) it asserts a line on the I/O interface controller to begin tracing and deasserts this line to finish and (ii) it suspends TRACEE while BACH's trace buffer is emptied. We have developed and tested device driver implementations of this controlling software for MACH 2.6, MACH 3.0, UNIX SysVR3.2, and UNIX SysVR4.

i486 Specifics:

On the i486 system running MACH 2.6 this software is implemented as a device driver with the associated special file */dev/bach*. This device driver has three routines, *bachopen()*, *bachclose()*, and *bachintr()*.

The *bachopen()* routine asserts the control line going to BACH to enable tracing. This is done by loading the appropriate register on the parallel I/O card with a value of '1'. The *bachclose()* routine deasserts this same line to end tracing.

Each time the buffer on BACH fills an interrupt signal is asserted on the i486 system. The *bachintr()* routine is called in response to this interrupt. It turns off interrupts, disables tracing, and then spins in a tight loop. Within the loop a "continue" signal, generated by BACH, is monitored. When the continue signal is asserted the loop is terminated, tracing is enabled, interrupts are restored, and execution continues.

3.3.2 Optional Software for Lightweight Instrumentation

The instrumentation methods described in this section are by no means a complete list, but simply outline what has been used to date with BACH. Our experience to date is that no single method of instrumentation is applicable to all operating systems or even

two different versions of a single operating system. The following discussion of operating system instrumentation describes the general methods used to date. Specific examples are presented in Section 6.

3.3.3 Single-bit Trace Annotation

As mentioned above, one of the two connections between TRACEE and BACH consists of an I/O interface device. This device is used to pass control information between BACH and TRACEE. Other pins on this connection are used to send information to BACH which is not available at the CPU pins. This method has the advantage that no new trace records are created – it simply adds extra information to each trace record.

i486 Specifics:

The parallel I/O device is used to pass control information between BACH and the i486 system. Our main use to date on the i486 for this has been to mark references as being from user or system code, as described above. This was done by instrumenting the kernel entry and exit points. There are several such points located in a file called “*locore.s*”. At each point, an OUT instruction is executed which writes the proper bit into the parallel I/O card.

3.3.4 Full Trace Annotation

To provide more information in the trace, extra CPU references can also be generated. This is accomplished by adding code that writes useful data to unused memory or I/O locations. No harm is caused by writing to unused locations, but the write reference is captured and stored by BACH. This can be used to gather information on system calls, interrupts, exceptions, and context switch points.

i486 Specifics:

On the i486 this is done by executing OUT instructions to special I/O addresses. Although the addresses written to are non-existent ports, the operations appear at the CPU pins as I/O writes. The port number and the data value written are both useful. For example, a study we recently completed required identifying system calls. An OUT to port 0x2A0 was used to mark this in the trace, the 32 bit data value written identified the particular system call as given in “*syscall.h*”. In a similar way we were able to identify all exceptions and interrupts. Section 6 presents the results of a study summarizing the data collected in this way and the conclusions we were able to draw from it.

4 The Trace Format

In general the only restriction on the trace format is that it fit into 96 bits. The format of the trace record currently being collected on the i486 is given in Table 1. As shown, most signals are taken directly from the CPU pins and are named as in the *i486 Hardware Reference Manual*. There are three fields, however, that are different: (1) the DELTA field indicates the number of cycles elapsed since the last trace record, (2) the SYS field indicates whether the record is from system or user code, and (3) the ST field contains the interface board’s state.

Size	Name	Description	How Generated
32	ADDR	Address referenced	i486 pin
32	DATA	Data associated with reference	i486 pin
4	BE	Byte Enables	i486 pin
1	W/R_	Write or read	i486 pin
1	M/IO_	Memory or I/O operation	i486 pin
1	D/C_	Data access or control transaction	i486 pin
1	BS8_	8 bit bus transfer	i486 pin
1	BS16_	16 bit bus transfer	i486 pin
1	LOCK_	Locked transaction	i486 pin
1	HLDA_	Bus hold	i486 pin
1	PCD_	Page cache disable	i486 pin
1	PWT_	Page write through	i486 pin
8	DELTA	Elapsed cycles since last bus transaction	By BACH's hardware
2	ST	State of the interface board's FSM	By BACH's hardware
1	SYS	System or user reference	By instrumentation software

Table 1: Trace Record Contents

This raw trace is similar to that which would be acquired by a logic analyzer - it contains a time-accurate record of the CPU pin values. Unlike a logic analyzer which collects data at regular intervals, BACH records data related to bus cycles (which occur at irregular intervals).

5 Evaluation Of The BACH System

The advantages and disadvantages of other tracing methodologies were discussed in Section 2. This section describes those of BACH.

5.1 Long, Contiguous, and Complete Traces

The traces from the BACH system are contiguous since the traced processor can be halted during the unloading of the trace buffer. After the buffer is emptied the traced machine resumes execution at the point where it was interrupted and tracing resumes. All such buffers are concatenated resulting in a complete trace.

The process of collecting and concatenating buffers can continue indefinitely resulting in traces of arbitrary length. Several traces have been collected which exceed 200 million CPU references. In contrast many published studies have used traces orders of magnitude shorter than BACH's.

Another important characteristic of BACH is that the tracing procedure can be synchronized using software. Tracing starts when TRACEE opens */dev/bach* and ends when it closes the same file. Thus, single applications, single routines, or complete job mixes can be traced.

Many of the tracing methods discussed in Section 2 collect only user references. Other references are generated by the CPU that may be significant when determining system

performance. Prefetched references are an example [10]. The BACH system records all important CPU bus cycles.

Finally, to adapt BACH to trace a new system (i) the interface board which monitors the signals of interest in TRACEE must be designed and (ii) the software described in Section 3.3.1 must be written. Neither of these tasks is unusually difficult.

5.2 Disadvantages

Although the BACH system has overcome many of the limitations and disadvantages of other tracing methodologies several problems remain. Time and space dilation affects have been minimized, but do exist and are quantified below for the i486 system running MACH 2.6.

5.2.1 Dilation Effects

The addition of lightweight instrumentation for trace annotation increases both the size of the resulting trace and the execution time of the traced application. These dilations add one memory reference and several CPU cycles for each extra trace record added by the instrumentation software.

In addition, dilation occurs due to the halting of the system. There are several devices that make requests via interrupts. These include the system timer, keyboard, hard drive controller, network interface, and serial ports. While the system buffer is being emptied these devices are capable of queueing up one interrupt each which will be serviced immediately after the trace buffer is emptied. The resulting distortion could be severe if the number of interrupting devices were large, but in our traces the only interrupts that are seen at these boundaries are the timer and network routines. Analysis of the trace data shows both are quite short, measuring 1000 and 3500 memory references respectively. In the worst case, interrupts for both are queued at each trace buffer boundary, resulting in an I/O dilation of 4500 references per buffer of 400,000 references, or 1.125%.

There are several other sources of trace perturbations due to BACH:

1. To acquire full traces TRACEE's on-chip cache is disabled. This slows execution by a factor of about 2 and may alter the interleaving of processes in a multiprogramming workload.
2. The interrupt routine which halts TRACEE is itself traced for 55 references at the beginning of each buffer and an additional 195 references at the end of each buffer. This code can be removed from the trace if desired.
3. The 250 references generated by the execution of the interrupt routine which suspends the traced machine could displace otherwise useful data in the second level cache, slightly altering normal system operation.

In spite of these perturbations, we believe BACH traces are significantly more accurate than traces gathered using other techniques. Although there is I/O dilation, it has been quantified and the extra references can, in general, be removed from the trace if desired.

Name	References	% OS	% User	% Ifetches	% Reads	% Writes
cnet	5,989,097	14.62%	85.38%	73.97%	15.45%	10.39%
grep	12,557,619	20.34%	79.66%	81.78%	11.77%	6.25%
nroff	14,613,781	17.73%	82.27%	80.85%	12.44%	6.51%
sed	13,628,111	15.12%	84.88%	81.20%	14.31%	4.24%
jigsaw	15,214,820	14.93%	85.07%	76.14%	13.62%	10.14%
det	8,142,566	14.25%	85.75%	69.35%	20.06%	10.51%
fft	12,091,984	12.30%	87.70%	66.95%	18.56%	14.41%
gauss	6,834,097	13.41%	86.59%	75.12%	15.35%	9.45%
int	6,080,158	16.13%	83.87%	58.68%	25.78%	15.44%
sort	5,682,538	22.28%	77.72%	75.65%	18.27%	5.84%
eqntott	7,299,706	26.30%	73.70%	76.44%	14.52%	8.95%
queen	19,376,617	13.19%	86.81%	74.70%	15.72%	9.51%
esim	13,239,488	28.00%	72.00%	74.76%	17.07%	7.99%

Table 2: Single process trace characteristics

6 BACH Traces: A Case Study

A true measure of the need for BACH is the impact the resulting traces have on the accuracy of trace-driven simulations. In the cache simulation case study presented below, we show that the estimated miss rates obtained using BACH traces are as much as 50 times higher than those obtained from inline traces. However, the real gauge of memory hierarchy performance is effective memory access time rather than miss rate. Differences in memory access time estimates of nearly a factor of two are demonstrated between traces produced by BACH and other methods.

6.1 Trace Characteristics

Thirteen traces were collected and used for this case study. The applications chosen were taken from several areas of computing:

- System utilities such as grep, sort, sed, and nroff
- Numerical codes including Gaussian elimination, fft, integration, and determinant
- CAD tools such as esim, eqntott, and cnet
- A LISP routine solving the 4 queens problem and a simple jigsaw puzzle benchmark.

Initially each application was traced without other user processes running. This resulted in 13 traces containing single applications and operating system references. All of the traces were collected on an i486 running MACH 2.6. Their characteristics are summarized in Table 2. Each application was traced from the beginning of its execution until its normal termination.

The total number of exceptions, interrupts, system calls, and contexts for each trace are presented in Table 3. The presence of multiple contexts in 3 of the traces may

Name	Except.	Ints	Sys Calls	Con
det	32	162	24	1
fft	42	281	19	1
gauss	29	134	27	2
int	21	193	15	1
queen	25	370	17	1
jigsaw	84	292	97	1
cnet	27	139	19	1
grep	5	268	246	2
nroff	32	330	82	1
sed	8	329	75	1
sort	11	141	41	1
eqntott	103	137	84	4
esim	44	267	794	1

Table 3: Single process trace exceptions, interrupts, system calls, and contexts

seem unusual - further analysis revealed that the *eqntott* trace had 4 contexts because it repeatedly forks the C preprocessor, the *grep* process had 2 contexts because the *update* program 'synced' the disk, and the *gauss* trace had 2 contexts due to the execution of *cron*. We chose to use these last two runs which contained system tasks since they represent real program execution.

In addition, a set of multiprogram workload traces were captured. These consist of combinations of the single applications from Table 2. They were generated by using a shell script to run the desired mix of applications concurrently while tracing was enabled. Characteristics of these multiprogram traces are given in Table 4.

6.2 Operating System References

The single-process traces have from 12 to 28 percent system references while the fraction of system references in the multiprogramming workloads varies from 16 to 20 percent. This system activity is substantial and is not distributed evenly throughout the trace data. One program, *nroff*, yields a trace in which the percentage of supervisor references, measured across 400,000 reference windows, varies from 6 to 92 percent.

In addition to single-bit trace annotation, full trace annotation was used to provide information on system calls, exceptions, interrupts, and context switches. This data is also presented in Table 3. The table provides only the total number of each type of system activity — however, the data is available in the trace to determine which interrupt, system call, or exception occurred.

6.3 Cache Simulation Results

BACH traces can be used for various studies, but a demonstration using a 64 K byte direct-mapped cache with 16 byte lines will be used to demonstrate the accuracy of BACH traces. The estimated miss rate using BACH traces varies from one portion of the trace to another in agreement with the results of [2]. Figure 1 illustrates the various

Name	References	% OS	% User	% Ifetches	% Reads	% Writes
dfgiq	54,952,010	16.93%	83.07%	70.12%	18.26%	11.54%
cgns	51,652,388	17.90%	82.10%	79.49%	13.82%	6.57%
sifgc	37,640,917	16.84%	83.16%	69.32%	18.69%	11.88%
ecif	32,225,152	18.63%	81.37%	68.82%	18.50%	12.60%
gism	32,408,898	18.89%	81.11%	74.04%	16.73%	9.08%
csn	18,242,205	20.62%	79.38%	76.53%	15.38%	7.90%
ggn	30,197,703	17.72%	82.28%	79.66%	12.90%	7.31%
ge	20,778,604	20.46%	79.54%	78.94%	13.67%	7.27%

dfgiq	det	fft	gauss	int	queen
cgns	cnet	grep	nroff	sort	sed
sifgc	sort	int	fft	gauss	cnet
ecif	eqntott	cnet	int	fft	
gism	gauss	int	sort	nroff	
csn	cnet	sort	nroff		
ecif	eqntott	cnet	int	fft	
ggn	grep	gauss	nroff		
ge	grep	esim			

Table 4: Multiprogram trace characteristics

contributors to this miss rate for the *nroff* trace. It should be noted that for most of the trace the major contributor to the miss rate is the operating system. In particular the contributions made by the exception handlers and interrupt routines are significant.

A number of studies have used traces which lack these operating system references [2, 6, 7]. To compare these traces with BACH’s, traces similar to those used in [2, 6, 7] were created by removing operating system references from BACH traces. The *nroff* trace was processed in this manner and used in a set of cache simulations. The resulting miss rates were compared to those obtained using the complete *nroff* trace, from BACH, and the error computed as

$$EF = \frac{MR_{BACH}}{MR_{userOnly}} \quad (1)$$

where EF is the error factor, MR_{BACH} is the miss rate predicted using the full BACH trace and $MR_{userOnly}$ is the miss rate predicted using the user-only trace. This was computed for each cache configuration and the results are presented in Table 5. It can be seen in this table that a user-only trace is not an accurate predictor of overall miss rate.

As mentioned above, Figure 1 shows that different components of the trace have varying impact on miss rates. By processing complete traces to remove selected components such as interrupts, exceptions, user code, or system code, controlled experiments were performed to determine to what extent each component affects the accuracy of simulation results. Each trace was processed to yield 8 new subtraces:

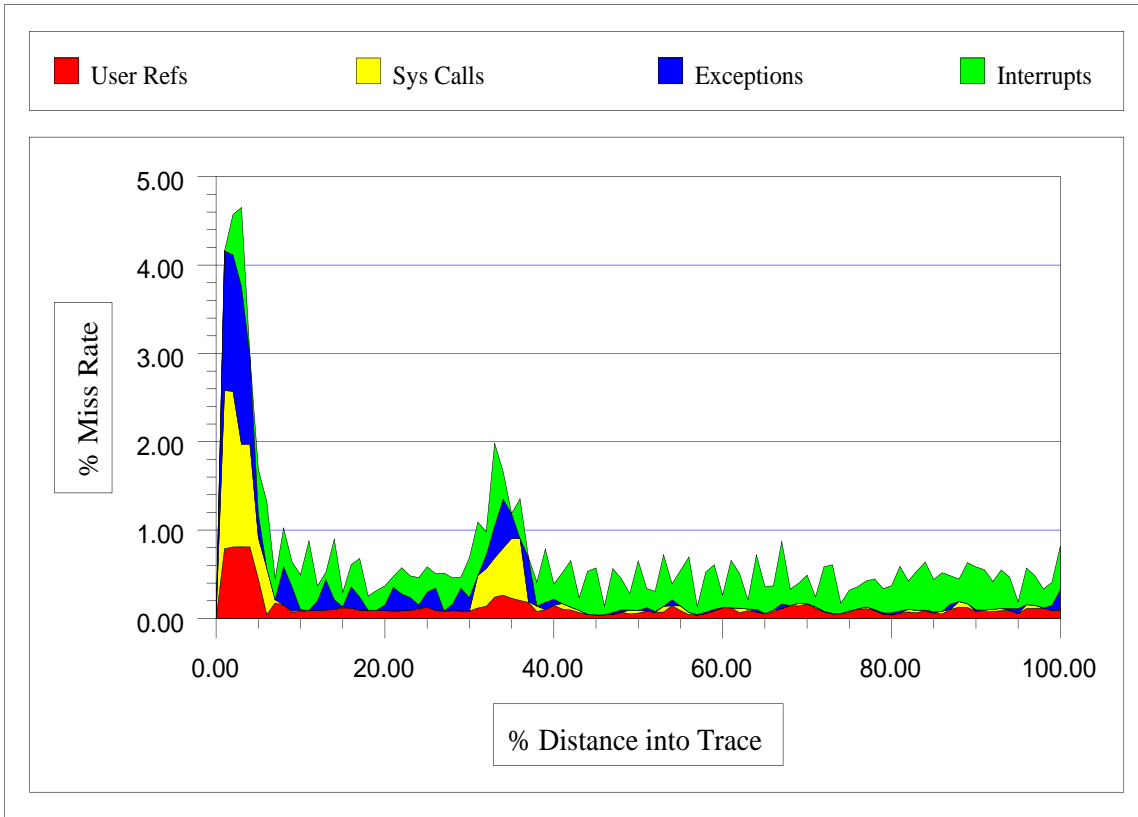


Figure 1: Detailed view of miss rate contributions using the *nroff* trace and 64 K cache

- u* User references only
- ui* User references and interrupt routines
- ue* User references and exception handlers
- us* User references and system calls
- uie* User references, interrupts, and exceptions
- uis* User references, interrupts, and system calls
- ues* User references, exceptions, and system calls
- eis* Supervisor references only

In addition, we traced the 13 applications described above using an inlining technique similar to [2, 6, 7]. The 117 subtraces were run through a 64 Kbyte direct-mapped unified data and instruction cache. The resulting error factors are presented in Figure 2 and are as high as 58. Also note that given two subtraces, one is not consistently better in predicting miss rate. Finally, it can be noted that all trace components are required to achieve an error factor less than 2 and that all subtraces yield optimistic miss rate estimations.

6.3.1 Trace Accuracy and System Performance

Miss rates are not the best estimate of system performance but they do allow easy calculation of the access time of a memory hierarchy:

$$t_{eff}(T) = t_{cache} + m(T) \times t_{memory} \quad (2)$$

Size=	4K	8K	16K	32K	64K	128K	256K
Assoc=1	6.89%	3.95%	2.31%	1.28%	0.69%	0.44%	0.28%
Assoc=2	4.21%	2.48%	1.53%	0.88%	0.48%	0.26%	0.16%
Assoc=4	3.43%	2.00%	1.29%	0.74%	0.37%	0.19%	0.13%
Assoc=8	3.17%	1.91%	1.19%	0.69%	0.29%	0.15%	0.13%
Assoc=16	3.07%	1.89%	1.17%	0.69%	0.25%	0.15%	0.13%

a) Complete Trace Miss Rates

Size=	4K	8K	16K	32K	64K	128K	256K
Assoc=1	6.11%	2.84%	1.31%	0.52%	0.12%	0.05%	0.04%
Assoc=2	3.19%	1.45%	0.56%	0.17%	0.07%	0.04%	0.04%
Assoc=4	2.32%	0.95%	0.35%	0.08%	0.04%	0.04%	0.04%
Assoc=8	2.08%	0.88%	0.25%	0.06%	0.04%	0.04%	0.04%
Assoc=16	2.02%	0.88%	0.23%	0.05%	0.04%	0.04%	0.04%

b) User-Only Trace Miss Rates

Size=	4K	8K	16K	32K	64K	128K	256K
Assoc=1	1.1	1.4	1.8	2.5	6.0	9.1	6.7
Assoc=2	1.3	1.7	2.7	5.1	6.7	6.5	4.0
Assoc=4	1.5	2.1	3.7	9.2	8.6	5.0	3.5
Assoc=8	1.5	2.2	4.8	10.8	7.2	3.9	3.3
Assoc=16	1.5	2.2	5.1	13.1	6.3	3.8	3.3

c) Error Factors ($EF = \frac{MR_{actual}}{MR_{subtrace}}$)

Table 5: Complete and User-Only *nroff* Trace Results

where t_{cache} , $m(T)$, and t_{memory} are the cache hit time, cache miss rate for a trace T , and memory access time respectively.

To evaluate this, we took one of the multiprogram workload traces from Table 4 and removed all system references from it, resulting in a trace similar to that described in [2]. We then passed it and the original trace through a 64 K byte direct-mapped cache model. Assuming that $t_{memory} = 30$ and $t_{cache} = 1$, the estimated effective memory access times are:

$$\begin{aligned}
 t_{eff}(userOnly) &= 1.0 + m(userOnly) \times 30 \\
 &= 1.0 + 0.2\% \times 30 \\
 &= 1.06
 \end{aligned}$$

and

$$\begin{aligned}
 t_{eff}(BACH) &= 1.0 + m(BACH) \times 30 \\
 &= 1.0 + 3.3\% \times 30 \\
 &= 1.99
 \end{aligned}$$

The estimate of average memory access time is thus off by a factor of 1.88. Since

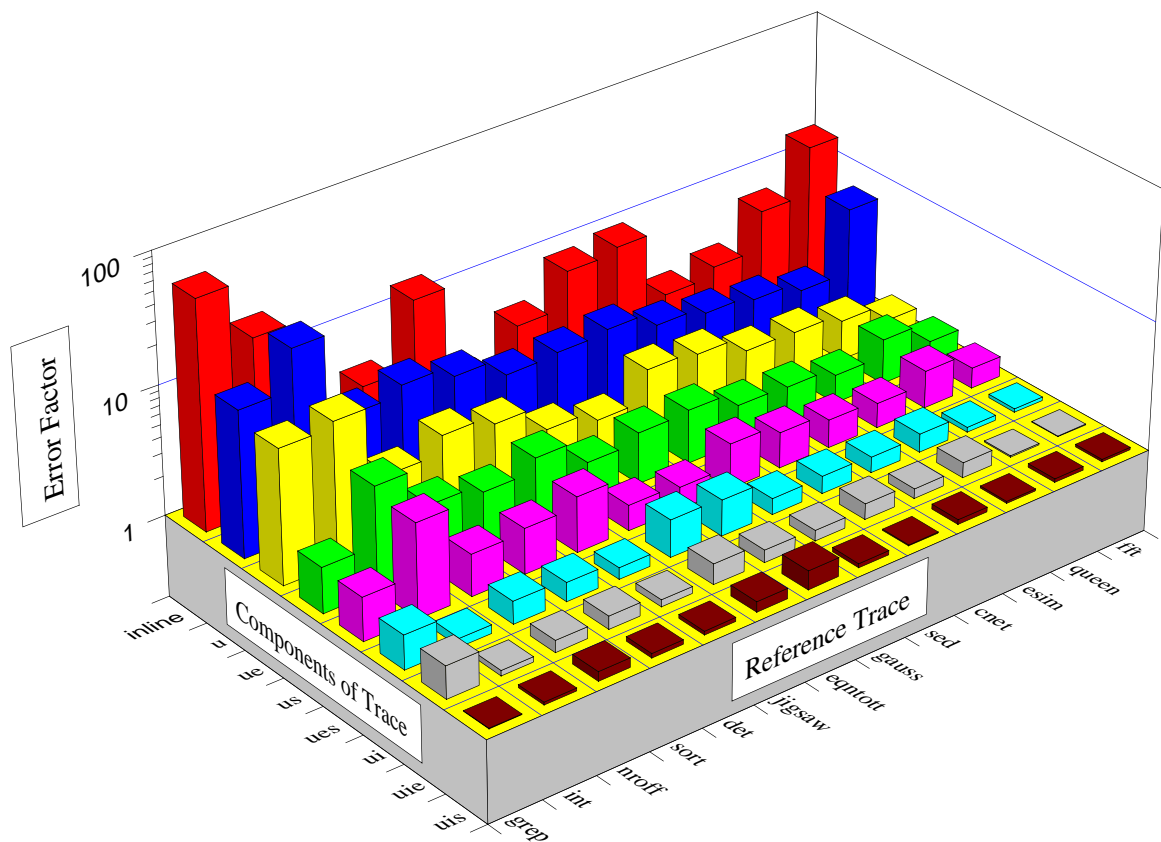


Figure 2: Contribution to simulation error for each trace component

memory access time is often the limiting factor in the performance of a system, an effective doubling of memory access time is significant.

7 Conclusions and Future Work

In this paper we have described BACH, a tool for computer systems performance evaluation which enables the collection of long, accurate, and complete traces. We have shown, through cache simulation, that the miss rate results obtained from BACH traces are different from those obtained from other, less complete, traces by as much as a factor of 58 and that the resulting effective memory access time can be off by almost a factor of two.

One of BACH's main strengths is its flexibility, both to trace different hardware platforms and to collect extra trace information when used in conjunction with lightweight instrumentation. Accordingly, in the future we intend to use BACH to trace MC68030 systems and SPARC systems. In addition, we have begun instrumenting MACH 3.0 to compare its behavior with MACH 2.6 and UNIX SysVR4.

References

- [1] J. C. Mogul and A. Borg, The Effect of Context Switches on Cache Performance, In *Proc. of 4th Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 75–84. ACM, 1991.
- [2] A. Borg, R. E. Kessler, and D. W. Wall, Generation and Analysis of Very Long Address Traces, In *Proc. of 17th Int. Symp. on Computer Architecture*, pages 270–279. ACM, 1990.
- [3] O. A. Olukotun, T. N. Mudge, and R. B. Brown, Implementing a Cache for a High-Performance GaAs Microprocessor, In *Proc. of 18th Int. Symp. on Computer Architecture*, pages 138–147. ACM, 1991.
- [4] D. A. Wood, M. D. Hill, and R. E. Kessler, A Model for Estimating Trace-Sample Miss Ratios, In *Proc. of 1991 ACM Sigmetrics*, pages 79–89. ACM, 1991.
- [5] S. Przybylski, M. Horowitz, and J. Hennessy, Characteristics of Performance-Optimal Multi-Level Cache Hierarchies, In *Proc. of 16th Annual Int. Symp. on Computer Architecture*. IEEE, 1989.
- [6] S.J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy, Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor, In *Proc. of 1990 ACM Sigmetrics*, pages 37–45. ACM, 1990.
- [7] C. Stephens, B. Cogswell, J. Heinlein, and G. Palmer, Instruction Level Profiling and Evaluation of the IBM RS/6000, In *Proc. of 18th Int. Symp. on Computer Architecture*, pages 180–189. ACM, 1990.
- [8] A. Agarwal, R. L. Sites, and M. Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, In *Proc. of 13th Int. Symp. on Computer Architecture*, pages 119–127. IEEE, 1986.
- [9] Gurindar S. Sohi and Manoj Franklin, High-Bandwidth Data Memory Systems for Superscalar Processors, In *Proc. of 4th Int. Conf. on Arch. Support for Prog. Lang. and Operating Systems*, pages 53–62. ACM, 1991.
- [10] Douglas W. Clark, Cache Performance in the VAX-11/780, Technical report, Systems Architecture Group, Digital Equipment Corp., March 1982.
- [11] Douglas W. Clark and Joel S. Emer, Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement, *ACM Transactions on Computer Systems*, **3**(1):31–62, February 1985.