NStrace: A bus-driven instruction trace tool for PowerPC microprocessors

by P. A. Sandon
Y.-C. Liao
T. E. Cook
D. M. Schultz
P. Martin-de-Nicolas

NStrace is a bus-driven hardware trace facility developed for the PowerPC® family of superscalar RISC microprocessors. It uses a recording of activity on a target processor's bus to infer the sequence of instructions executed during that recording period. NStrace is distinguished from related approaches by its use of an architecture-level simulator to generate the instruction sequence from the bus recording. The generated trace represents the behavior of the processor as it executes at normal speed while interacting normally with its run-time environment. Furthermore, details of the processor state that are not generally available to other trace mechanisms can be provided by the architectural simulation. There are two main components to the process of generating bus-driven instruction traces: bus capture and trace generation. Bus capture is triggered by a call to a system program that puts a particular address on the bus, then establishes the initial state of the processor by a combination of writing out register values and invalidating caches. A logic analyzer records the bus activity, and from this a file of bus transactions is produced. Trace generation proceeds by driving a processor simulator with these bus transactions and recording the sequence of instructions that results. The processor simulator is an elaboration of that developed for the PowerPC Visual Simulator. We have successfully generated instruction traces for a mix of utility programs and real applications on several microprocessor platforms running several operating systems. The capacity of the bus recording hardware is two million transactions, yielding instruction traces with lengths of the order of one hundred million instructions. This trace facility has been used for a number of studies covering a range of performance issues involving software, hardware, and their interactions.

1. Introduction

An instruction trace is a listing of the instructions executed by a processor while an application or system program is running. Such traces can be used to drive performance models and to extract statistics on a variety

0018-8646/97/$5.00 © 1997 IBM

of measures associated with system behavior. Dynamic usage of instruction types and other fields, address translation and memory reference patterns, and branch behavior and basic block sizes are among the many subjects of trace analysis. The extracted statistics reflect the behavior of the particular combination of architecture, implementation, system software, and application program from which the trace was collected, and so can be used to guide design decisions by developers of all of these components.

Instruction traces may be obtained in various ways. While *simulated* traces can be generated by executing a target program on a software model of the processor, instruction traces from real systems are created in one of two ways. *Software* traces are typically generated by instrumenting the target program with traps or exceptions, so that the operating system can monitor the instruction sequence during program execution. *Hardware* traces are generated by connecting external circuitry to the system under test to record activity while the target program executes. We discuss the trade-offs involved in choosing a trace methodology in the next section.

We have developed a bus-based hardware trace facility for the PowerPC* family of superscalar reduced-instruction-set microprocessors. When the processor's bus activity is passively recorded for some period of time that includes execution of the application of interest, the program executes at speed while interacting normally with its run-time environment. The bus recording is then used to drive a processor simulator that generates the complete sequence of instructions (including kernel, library, and user code) executed during the recorded period.

The trace tool, called NStrace, was developed for processors using the PowerPC 60X system interface. We have traced several systems that use the PowerPC 604* processor [1], including both reference platforms and commercial systems, in both uniprocessor and multiprocessor configurations. We have traced systems running AIX*, Windows NT**, and MacOS** operating systems. We have generated traces with lengths of the order of one hundred million instructions on applications ranging from AIX utilities to multimedia programs to Java** benchmarks.

The process of generating bus-driven instruction traces comprises two operations, bus recording and processor simulation. After a brief review of related work in Section 2, we provide in Sections 3 and 4 an overview and selected details for the recording and simulation operations. In Section 5, we present some sample trace data, describe some tools we have developed for working with the data, and discuss the use of these tools and data in performing several performance studies. In the final section, we summarize the characteristics of the NStrace tool.

2. Related work

Trace data sets vary according to the particular system variables recorded, the type of software traced, and the trace environment. System variables that may be recorded include instructions, instruction addresses, data, data addresses, memory management state, bus transactions, and others. The software traced may be kernel, driver, library, or user code. The trace environment may include a variety of software and hardware instrumentation used to make the system variables of interest observable.

Depending on the characteristics of the trace data set, it can be used for one or more of a wide range of performance analysis and optimization studies of both hardware and software. Instruction traces, for example, are used for various processor performance studies such as degree of parallelism for superscalar [2] and multithreaded [3] processors, branch prediction strategies [4], and evaluation of novel [5] or alternative [6] implementation mechanisms that address latency, bandwidth, or throughput problems. Address traces are used for studies of the memory system, in particular for comparing the performance characteristics of alternative cache configurations (see for example Reference [7]). Bus traces have been used to study table-lookaside buffer (TLB) design trade-offs [8] and performance characteristics of the bus itself, including utilization, transaction-type run lengths, and inter-request timing statistics [9]. Other uses for trace data include program visualization for debugging [10] and optimization through code restructuring [11].

Some trace tools record system variables that are specific to a particular type of analysis. In a study of I/O prefetching and caching algorithms, for example, file block read requests were traced [12]. In another study, a set of page table events were traced to analyze the performance of a hardware page table manager [13].

One way to generate trace data is to use an instruction set simulator that loads the executable for the application to be traced, then records the sequence of instructions executed as simulation proceeds. Examples of instruction set simulators are Shade [14] for SPARC processors and PVS [15] for PowerPC processors. The drawback of simulated traces is that they trace only the application code and are insensitive to many of the details of the run-time processing environment, including interactions with the operating system, I/O, and other processes.

A major distinction among approaches to generating trace data from real systems is whether they are software-based or hardware-based. While bus tracing is always hardware-based, address and instruction tracing may be either hardware- or software-based. Software-based tracing involves instrumenting the code to be traced such that whenever a significant event occurs during execution, a

record of the event is stored. Significant events may include the execution of an instruction, a memory access, or a discontinuity in the program counter.

One recently developed software-based trace tool is IDtrace [16], which uses late code modification [17] to instrument the executable to be traced. IDtrace can generate instruction or address traces, and so is useful for a variety of tasks including studies of cache behavior, branch prediction, and code profiling. Another software trace tool specifically designed for capturing long address traces is described in [18]. Long address traces are particularly important to studies of secondary cache behavior, since such large caches reach steady state only slowly.

Software-based tracing has the advantage of low cost. Often the system under test performs the function of trace capture and storage itself. The disadvantage of software-based tracing is that the tracing mechanism interferes with normal operation, yielding trace data that may not accurately characterize the target system. For example, a program instrumented for a full execution trace using IDtrace is an order of magnitude larger and an order of magnitude slower to execute than the original [16]. In addition, such instrumentation is often performed only on application code, so that the characteristics of operating system code are not captured or analyzed. Both [8] and [19] provide data on how ignoring the characteristics of the operating system can bias performance analysis, and presumably the resulting design decisions.

Maintaining the normal run-time environment is particularly important for timing-driven applications. For example, display of video data encoded in the Motion Picture Experts Group (MPEG) format requires real-time response characteristics from the processor. If execution of an MPEG player is slowed down, the application will spend more of its time in the frame drop code, which is what will then be traced. Similarly, a soft modem application, when slowed down, will appear to spend a disproportionate amount of its time handling circuit drop. In multiprocessor environments, timing changes in one processor can completely change the characteristics of the interactions among processors. The software-based MPtrace tool [20] minimized the amount of data recorded during tracing for just this reason.

Hardware-based tracing involves the use of special hardware that is attached to the system under test to record significant events. Hardware-based tracing has the advantage of being less intrusive than software-based approaches. Once tracing begins, the system executes in its normal run-time environment for the entire trace, or for relatively long periods between interruptions for trace maintenance. For many applications, where interactions between the user code and system code, or between the processor and the I/O system, are significant, this nonintrusive aspect of hardware-based approaches will yield traces that more closely reflect normal system behavior than do software-based ones.

Several hardware-based tracing systems have been developed for capturing bus traces. In one case, these traces of bus transactions are used to characterize the cache behavior in a multiprocessor system [19]. In another, bus traces are used to compare the performance of alternative software-managed TLBs [8]. In this second case, the tracing approach is actually a hybrid, using software instrumentation to put marker instructions at selected points in the operating system code, and using a hardware monitor to detect and record these marker instructions. A different approach is that of MacDEBIT, which is a hardware trace tool that uses a target-host pair of identical systems to collect bus traces for bus performance studies [9].

Hardware tracing methods for capturing instruction traces are few because of the difficulty of inferring an instruction stream from variables that can be directly measured. One approach is to disable the on-chip L1 cache, monitor bus transactions, extract the instruction fetch stream, and then attempt to reconstruct the stream of instructions actually executed [21]. Not only does this represent a significantly invasive technique, but the reconstruction can be done only approximately. Furthermore, the amount of information that such a trace can provide is far less than that available in the NStrace approach.

The NStrace tool described in this paper produces instruction, data, address, and bus traces. It is a hardware-based approach that provides noninvasive capture of the system variables from which an instruction trace is generated. Since it records and then simulates the processor activity as it occurs over the course of the entire recording period, it traces user code as well as kernel and library code. The NStrace tool processes snooping activity on the bus, and so is capable of tracing the activity of one processor in a multiprocessing environment. The use of bus activity to generate instruction execution activity provides a synchronized view of the behavior of the system both inside and outside the processor. In addition, the use of the processor simulator to produce the trace provides access to a wide range of architectural details not generally available in other tracing approaches. For example, since address translation is simulated, statistics on different translation mechanisms and TLB efficiency are available, as are both effective and real addresses. The following sections describe the recording and simulation phases of NStrace in more detail.

3. Bus recording

The input to the NStrace simulator is the sequence of processor bus transactions that occur during execution of

the target application. The current bus recording facilities have been developed for the PowerPC 60X system interface used by the PowerPC 601*, 603*, and 604 families of processors. The goal of the bus recording process is to capture the bus signals for the longest possible period while achieving 100% signal accuracy and including all activity that is relevant to the subsequent simulation phase. In this section we briefly describe the 60X bus, how data are acquired, processor initialization at the time of trace initiation, and the contents of the resulting transaction trace.

- *The 60X bus*

The PowerPC 60X system interface consists of two decoupled synchronous buses—one for address operations and one for data operations. The address bus includes a 32-bit address, and the data bus includes 64 bits of data. The separate controls of the two buses allow address and data operations to be combined in a variety of ways, providing a range of complexity and performance design points for systems developers. In addition to bus arbitration, a set of bus control signals support cache and TLB coherency, atomic memory operations, direct-store I/O operations, and L2 cache. A more complete description of the bus can be found in [22].

To drive the NStrace simulator, bus transactions must be discriminated according to which master initiated the transfer, the transfer format, and the transaction type. Transactions initiated by the target processor (the one whose instruction sequence is to be generated) are recognized by the assertion of bus grant to the target processor during transaction initiation. Transaction format and type are determined by the set of attribute signals that accompany any valid address on the bus.

Bus transactions occur in one of three formats: address-only, single-beat, and burst. Address-only transactions do not use the data bus, transferring only control information using the address- and transfer-type signals. Address-only transactions are used to support memory coherency and synchrony between processor and I/O. Single-beat transactions are used to transfer from 1 to 8 bytes of data to and from I/O or noncacheable areas of memory. Burst transactions transfer 32 bytes of data in four beats between processor cache and memory.

- *Data acquisition*

While access to processor bus signals varies from system to system, a typical configuration for the NStrace hardware is the following. For a system in which the CPU is mounted on a daughtercard that fits into a connector on the motherboard, we use an interposer card between the motherboard connector and the daughtercard to bring out the bus signals. We have developed a data acquisition and compression (DAAC) card to take these bus signals as inputs, process them according to the 60X bus protocol, and produce as output a record for each significant transaction that appears on the bus. Significant transactions are those that affect the subsequent processor simulation. These include all transactions initiated by the target processor, as well as all other transactions that affect the target processor state, primarily snoop hits. These transaction records are then captured in a logic analyzer. The combination of a relatively large logic analyzer memory (100 MB in the fully configured Tektronix DAS 9200) and compression of signals by the DAAC card yields a typical recording time of about one second, corresponding to an instruction trace of around 100 million instructions.

Once bus capture is complete, the signal data are uploaded to a workstation. Since we have used several logic analyzers, and configured each in several different ways, it is necessary to convert these signal data to a standard format for the simulator. Our *txn* format contains one record per transaction. Each record contains a time stamp (the bus cycle relative to the start of recording at which the transfer start signal appeared), the address, and the transaction type. For single-beat and burst transactions, the record also contains the data. A sample sequence from a txn file is presented in Section 5.

- *Trace initiation*

The process of translating a bus recording into an instruction trace involves simulating the processor's execution of the application. For this simulation to be valid, the state of the simulated processor must match that of the real processor at the time the simulation begins. The initiation of bus recording, therefore, involves two steps: achieving a known processor state, and signaling the bus capture hardware to begin recording.

For the 604 simulator, described in the next section, the state of the processor that must be controlled to predict its subsequent behavior comprises the contents of the two caches, the contents of the two TLBs, and the values of the architected registers. Initiation of the trace process involves a combination of reading out the current state of the processor and changing that state to a known state. The procedure we use is first to disable and invalidate the caches, then to trigger the recording by placing a distinctive address on the bus, then to store all of the register values, then to invalidate the TLBs. Storing the register values forces them out onto the bus where they can be recorded, since the caches are disabled. At the end of this procedure, the caches are enabled so that normal processing resumes.

This procedure is implemented as a device driver that can be called asynchronously while the application to be traced is running. Alternatively, a call to the device driver

can be compiled into the application to initiate tracing at a particular point.

4. Trace generation

Once a bus recording is taken, the next step is to generate from it the sequence of instructions executed by the processor during this period. The approach we have taken is to simulate the processor at the architectural level, using the bus transactions to update the processor state when necessary, and otherwise letting instruction execution proceed according to the current state of the simulated processor. Our simulator is based on the PowerPC Visual Simulator (PVS) [15]. After a brief overview of that simulator, we describe the bus-driven simulation approach and then discuss some of the issues involved in making an architecture-level simulator handle implementation-level details in order to produce an accurate instruction trace.

- *PowerPC Visual Simulator*

PVS is an architecture-level simulator for the PowerPC family of processors. PVS simulates sequential execution of instructions in program order, ignoring the implementation-specific details involving issue and execution pipelines, caches, and bus latencies. PVS traces the execution of a single application by taking an executable file as input, building a memory image from that file, and then simulating the execution of the application.

The PVS implementation is fairly complete in its support for the major architectural features of the PowerPC. It also implements some of the processor-dependent features. Among the facilities supported are address translation, instruction execution, exception handling, caches, and reservations. The simulator also includes facilities for single-step instruction execution, for setting checkpoints and breakpoints, for displaying memory and register contents, for modifying memory and register values, for assembling and disassembling instructions, and for tracing instruction execution.

In order to use the PVS simulator for our bus-driven approach, we enhanced the simulator in a number of ways. The main deficiency of the simulator for our purposes was that since it used an "internal" model of memory, it largely lacked the concept of a bus interface unit (BIU). Therefore, our efforts in modifying the simulator were in two main areas. First, all functional units that could generate bus activity were modified to make appropriate calls to a bus interface unit. Second, a bus interface unit was implemented that provides access to the transaction file. We discuss below some of the issues involved in enhancing the simulator to perform correctly in the absence of complete processor state information.

- *Bus-driven processor simulation*

The bus-driven simulator is straightforward in concept. First, the processor state of the simulator is initialized, using the transactions recorded during trace initiation. Recall that during initiation the data cache is disabled while register values are written to some memory location. This causes a predictable stream of transactions to appear on the bus, from which the processor register values can be extracted. Since all details of the initiation procedure are known, the transactions corresponding to execution of this procedure can be accounted for without simulating them. The simulator state is initialized to the state of the processor at the end of initiation, and the stream of transactions that follow those corresponding to initiation drive the subsequent simulation. Once the processor is initialized, it begins to execute from that state.

The address of the first instruction is in the next instruction address (NIA) register. To execute the first instruction, the simulator first attempts to translate its address. Since the TLB starts out empty, a miss occurs, causing a read access to the cache. Since the data cache starts out empty, a miss occurs, causing a read transaction on the bus. The BIU model matches this simulated transaction with a recorded one from the transaction file and returns the corresponding data to the cache. The cache then satisfies the read from the TLB, and the requested page table entry (PTE) is available to the translator. The table walk continues until the required PTE is found, and then the instruction address is translated.

The real instruction address is now used to fetch the first instruction from the cache, but encounters a miss, generating an instruction fetch transaction. The BIU matches this fetch to one in the transaction file and returns the data to the instruction cache. The cache then returns the addressed instruction to be decoded. If the instruction requires a memory access, a similar sequence of events occurs to translate the data address and then load or store the data. For a load that misses in the cache, a read transaction is generated. For a store that misses in the cache, a read-with-intent-to-modify (RWITM) transaction is normally generated.

This general procedure continues until the BIU cannot match a simulated transaction with a recorded one. Normally this occurs when the stream of recorded transactions has been exhausted, and the simulation terminates. Of course, as the simulation proceeds, the caches and TLBs are warming up, so fewer accesses result in bus activity than occur initially. The output of the simulator is a trace containing the sequence of instructions executed during the recording period, along with corresponding instruction addresses, data, data addresses, bus timing, and translation information. A sample

sequence from an instruction trace file is presented in Section 5.

- *Handling asynchronous events*

On first consideration, it may appear that this simple concept is even trivial. After all, given the initial state of a finite-state machine and the sequence of inputs to that machine for some period of time, one can exactly reconstruct the state trajectory during that period. However, to simulate the entire processor state requires a gate-level simulator which can execute only thousands of instructions per hour. This is not sufficient for the millions of instructions needed to conduct performance analysis. The alternative is to simulate at the architecture level. This leaves much of the state of the target processor hidden, and turns the task of generating instruction traces from bus recordings into a significant challenge. One effect of this hidden state is that asynchronous events are more difficult to synchronize with instruction execution. We describe here a sampling of the problems resulting from asynchronous behavior and unmodeled processor state that require special consideration in the simulator.

An obvious asynchronous event is an external interrupt. Identification of when, with respect to instruction execution, the interrupt occurred is deterministic given the full processor state. However, it is problematic to precisely identify the last instruction executed before the interrupt was taken in the simulator. We address this problem by recording the external interrupt signal at the processor. The timing correspondence between simulator activity and bus activity established by the BIU transaction-matching process is then used to inject the external interrupt into the simulator at the time it occurred in the target processor.

Similarly, the decrementer exception, which is generated internally in the processor, occurs asynchronously to instruction execution. While in this case there is no external signal to indicate that the decrementer exception is pending, we can use the decrementer register modeled in the simulator to temporally localize the exception event. The trick in this case is to keep the decrementer register up to date using the bus cycle of recorded transactions as they are matched.

Two implementation mechanisms that cause the bus transaction sequence to vary from that expected by the simulator are instruction prefetching and out-of-order execution of instructions. When a branch occurs prior to execution of a line of prefetched instructions, the simulator will have no reason to call for the corresponding cache line. In order to keep the actual and simulated caches synchronized, including the LRU status of each line, these prefetched cache lines are forced into the simulator if they are not requested in a timely manner. A more subtle issue involves access to the TLB when translating instruction addresses for prefetching. Since the state of the TLB replacement bits is difficult to manage in this case, transactions associated with accessing the page table must be detected and specially managed so that PTEs persist in the simulator.

In the case of out-of-order processing of instructions (the simulator executes in strictly program order), there are instances when bus activity is affected. One example is that a load occurring after a store in program order may begin execution in the processor before the store. If both instructions reference the same missing cache line, the load miss in the target processor will generate a read transaction on the bus, while the store miss in the simulator will generate an RWITM transaction to be matched. In other cases, such differences between the processor and simulator lead to missing transactions in the recording. These various cases must be detected and accommodated in the BIU.

Yet another class of asynchronous activity is bus snooping. In order to maintain coherence among different copies of data in memory, the processor compares bus addresses to the addresses held in its cache, and modifies the coherency state or writes back data to main memory as needed to satisfy requests from other bus masters. This snooping behavior is asynchronous to instruction execution in the processor, and so must be handled explicitly in such a way that the simulator performs the same processing sequence as the target processor.

Among the cases where such special processing of snoop transactions is critical in obtaining the correct sequence of instructions is one involving synchronization. Consider the case in which two processors in a symmetric multiprocessor environment are using a semaphore to obtain exclusive access to some resource. The semaphore is implemented in the PowerPC architecture using the Load and Reserve (e.g., *lwarx*) and Store Conditional (e.g., *stwcx.*) pair of synchronization instructions. The *lwarx* instruction sets a reservation bit in the processor, and the *stwcx.* performs a store only if this bit is still set. A store access to the semaphore address by another processor, as would occur if it successfully executed a *stwcx.*, generates a bus transaction that is snooped by the target processor and causes the reservation bit to be reset. To correctly simulate the synchronization sequence, a snoop transaction on the bus must be processed in proper order with respect to the *stwcx.* instruction being executed in the processor. We have implemented an algorithm in the simulator to process this sequence, and have observed it working correctly in our multiprocessor recordings.

5. Trace analysis

In addition to the bus capture facility and the NStrace simulator, we have developed several tools and analysis programs to aid in processing our instruction traces for

performance studies. In this section we first describe in general terms several ways in which the trace data can be put to use, and then discuss three specific studies that we have performed using trace data generated by NStrace.

- *Trace-driven simulation*

Like traces obtained by other methods, our traces contain address and instruction streams that can be used to drive other tools. These include simulators for memory hierarchy studies, processor timers for microarchitectural analysis, and system models for performance studies on alternative platform configurations. Similarly, we can take advantage of existing trace analysis packages that have been developed to extract various statistics that are commonly used to characterize system behavior. Packages that we have used include statistics on frequency of execution and size of subroutines, first- and second-order instruction frequency, branch behavior, and memory references by segment and page. While there is an ongoing effort to standardize trace data formats, it is currently necessary to convert between trace formats to allow tools and data to be intermixed.

Unlike most other traces, ours also contain translation, timing, bus, and data information that can be used to analyze aspects of hardware and software behavior that are beyond the scope of many trace data sets. We describe some examples later in this section.

- *Qualitative analysis*

The binary transaction file for a bus recording contains 100 megabytes of data, while the corresponding instruction trace typically comprises around two gigabytes of data. We have developed two tools for looking at these data sets, one to get the big picture and one to see the details.

The *opstat* program extracts several basic statistics from the transaction and instruction trace files, sampling these statistics at regular intervals to produce a report of how these statistics change with time over the course of the entire trace (or some other period of interest). The statistics extracted include transaction frequencies by class [the classes are fetch (instruction), read (data), RWITM, write, snoop, and other], instruction frequencies by class (the classes are ALU, load, store, float, control, and branch), instructions per cycle (IPC), percentage of time in system mode, occurrences of exceptions, and reference counts for highly referenced instruction and data pages. **Figure 1(a)** shows a plot of transaction frequencies, and **Figure 1(b)** shows the corresponding plot of instruction frequencies extracted using opstat.

These plots have been quite useful in identifying the activity within and outside the processor at a millisecond level of detail. Note that since both the transaction and instruction traces contain bus cycle information, they can be aligned to show the corresponding activity on the bus and in the processor. Also note that this level of detail clearly shows the periodic nature of the application being traced—an MPEG player. Finally, note that the particular distribution of instruction or transaction class frequencies is characteristic of each processing phase of the application. In Figure 1(a) we can distinctly see two major phases, which turn out to be the computation and display components of processing for each frame of MPEG playback. In Figure 1(b) we see that the computation component has two distinct phases, one with nearly 70% ALU instructions, the other with only 50% ALU instructions. In **Figure 2**, which is a similar plot of instruction frequencies from a different recording, we can see the three phases of the MPEG player, interspersed with activity that is unlike any in the first recording. This second recording was made with two applications running, the MPEG player and a synthetic application that exercises the bus. This *signature* aspect of the distribution of these class frequencies has allowed us to determine when a particular application is in a particular phase of processing, as well as when a particular application is running at all.

Once the general structure of the recording has been established, a more detailed view of a particular period of interest can be had using the corresponding bus cycle number to index into the trace data itself using the *Browse* tool. This data browser displays a sequence of either instruction or transaction records. It supports an indexed search for a bus cycle number, and a sequential search for values in other record fields.

For example, around cycle two million in Figure 1, there is a brief dip in a number of the statistics, including the IPC value. **Figure 3(a)** shows a screen of transaction records from the browser for this time in the recording. For each bus transaction, the display shows, from left to right, the bus cycle number (number of bus cycles from the beginning of the recording until the transfer start for this transaction), the transaction type, the transaction address, and the transaction data. Note that there are three address-only transactions in this particular sequence (*eieio*, *sync*, and *lwarx*) and that all data transactions are bursts.

Figure 3(b) shows a corresponding screen of instruction records as displayed by the browser. From left to right, each record contains the transaction type(s) of any transactions associated with this instruction, the corresponding bus cycle number, the instruction address, the instruction machine code and mnemonic, the data address, and the data. The addresses are displayed somewhat cryptically, but contain the cacheability and protection bits, the effective page, the real page, and the page offset. Only the offset is displayed when the instruction addresses are sequential.

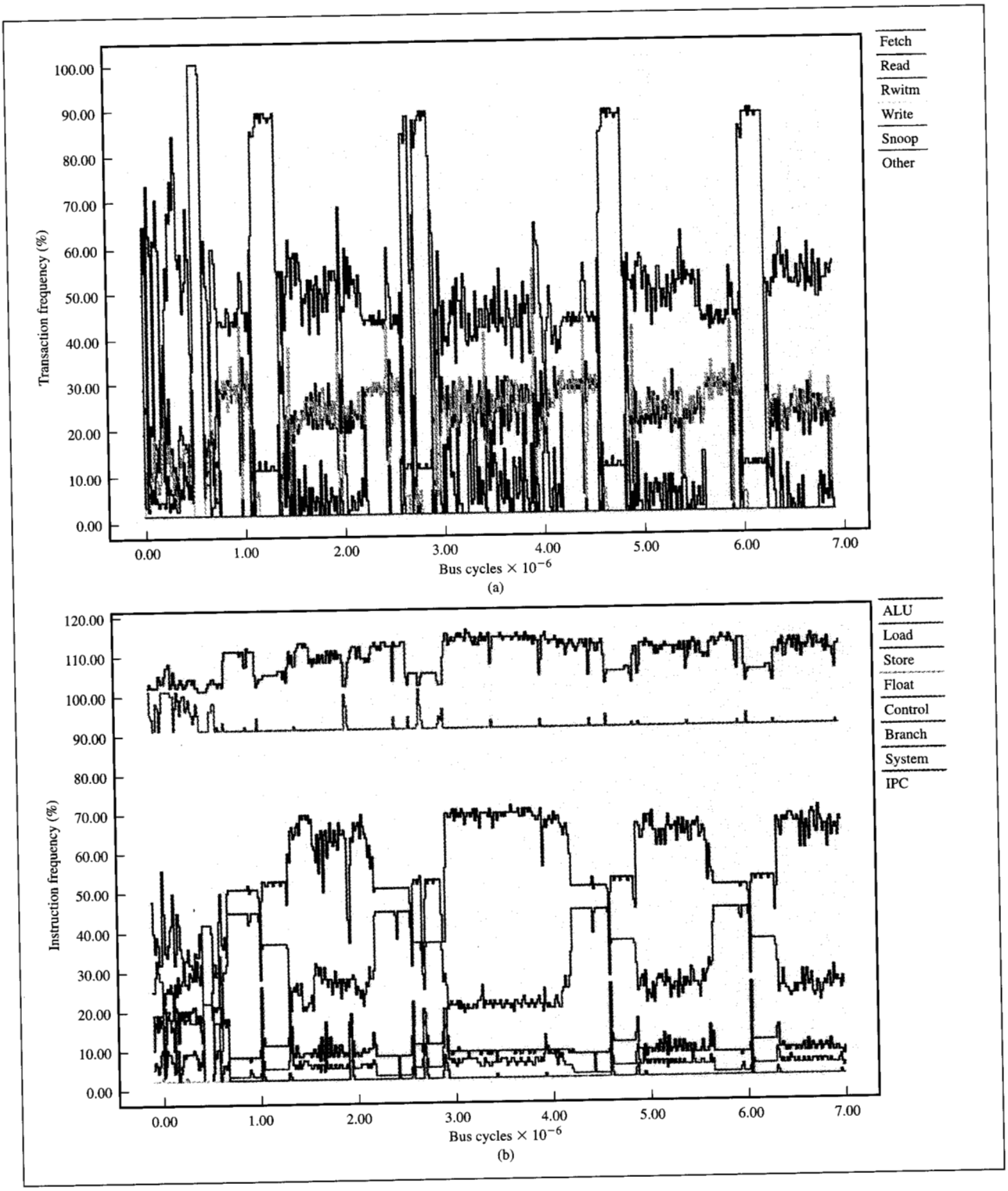

Figure 1

(a) Percent of bus transactions in each of six classes versus bus cycles for a recording period that includes several frames of an MPEG player. (b) Percent of instructions in each of six classes versus bus cycles for the same recording period as in (a). In addition, the top plot represents instructions per cycle (IPC), where 100 corresponds to $IPC = 0$ and 110 corresponds to $IPC = 1$; the second plot from the top represents amount of time spent in system mode, where 90 represents exclusively user mode execution and 100 represents exclusively system mode execution.

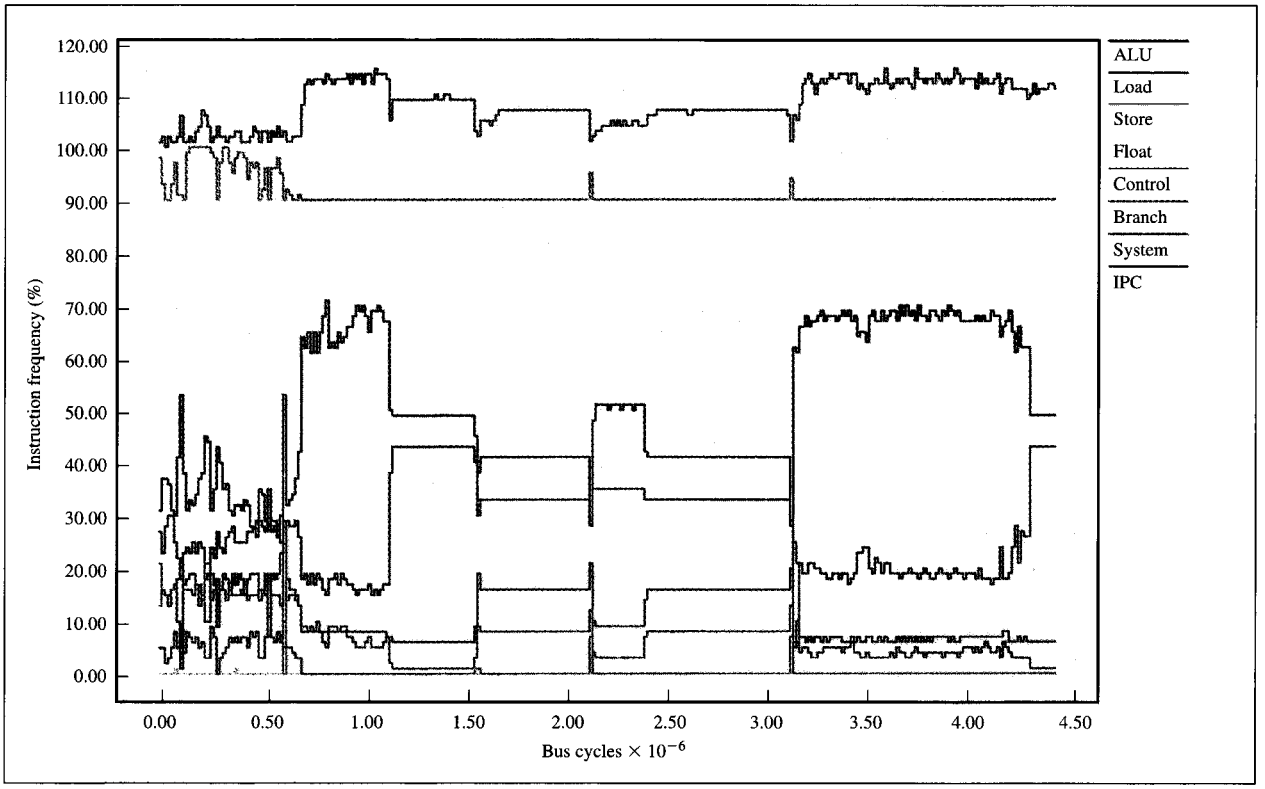

Figure 2

Percent of instructions in each of six classes versus bus cycles when both an MPEG player and a utility application are executing. IPC and system mode plots are included at the top, as in Figure 1(b).

It is clear from Figure 3 that the processor has suspended the application and is in system mode. By scanning through more of the trace in this vicinity, one finds that a timer interrupt is being serviced here, and the application resumes after some system maintenance functions are performed.

- *Quantitative analysis*

We have used the opstat and Browse tools to help in verifying that we have captured what we expected to capture in the trace, and to narrow the scope of analysis to a particular region of interest. Within this region, we can extract a variety of features and measures to identify performance problems or otherwise characterize system behavior. Among the features we have examined are frequencies of occurrence of specific addresses or instruction types, memory segment usage, page table reads, bus data rates, address patterns associated with peripheral component interconnect (PCI) devices, and occurrences of system calls by type. We briefly describe three studies in which combinations of these and similar features were used to analyze the overall behavior of the system.

We measured the frame rates of an MPEG playback application for several combinations of operating system version, graphics device, and display mode. Some variations in frame rates, such as that across 8-bit, 16-bit, and full-color modes, were expected. However, for three different configurations of 8-bit display mode, the frame rates were 30, 22, and 8 frames per second. We traced the application in these three configurations and found that the variations were due to the way in which the different graphics device drivers managed the frame buffer on the adapter. In the first case, the values for 4 pixels at a time are packed into a 4-byte word and written to the PCI card. In the second case, this packing is not done. Since the PCI address range is defined as noncacheable, each byte is sent individually over the bus to the adapter. In the third case, the value of the previous frame, which is used in the computation of the current frame, is obtained from the frame buffer on the adapter. It is read back one byte

cycle	txntype	address	data						
1999303	Read_B	0017BDE8	EF063EA0	EF063EA0	EF04C7B0	EF04C7B0	8017BDF8	8017BDF8	...
1999323	Fetch_B	00106920	80DF0014	40980014	7F861840	80630014	80A40000	40BD0020	...
1999345	Read_B	000D1B40	8000073C	0088F394	00000000	00B2B394	00000000	010FD394	...
1999367	Fetch_B	00106940	93850004	90BF0018	48000090	93840000	7F042800	80840004	...
1999389	Wrt_Kill_B	0023CE80	00000020	EC95BEA8	80106754	8017B802	EC95BEA4	8023CED0	...
1999424	Read_B	0088FE98	00000007	84390360	00000000	00000000	00000000	00280007	...
1999445	Fetch_B	001069D8	80010068	887F0024	809F001C	807F0018	90640000	9BBF0024	...
1999472	Read_B	0088FEA0	8017BDE8	8017BDE8	00000001	00000000	00000000	00000000	...
1999502	Fetch_B	00106960	907F0018	90BF001C	93850000	93830004	907B0004	807DE00C	...
1999534	Fetch_B	00106980	809DE010	807B0004	409EFFE4	7F832000	809B0004	807F0014	...
1999561	Fetch_B	001069A0	80810038	807F0010	409D0014	7F832040	809B0004	807F0014	...
1999590	Fetch_B	001069E0	83810070	8361006C	83C10078	83A10074	7C0803A6	83E1007C	...
1999617	Fetch_B	00106A00	93C1FFF8	7C0802A6	9001FFF4	93E1FFFC	90410040	9421FFB0	...
1999626	Fetch_B	00133D20	88610040	60000000	60000000	480067D5	80010060	7FC3F378	...
1999655	Eieio	000A0007							
1999657	Fetch_B	0011BD80	987F00B1	7FC3F378	38630080	807F0050	480168A1	889F0069	...
1999686	Sync	01C0FD66							
1999693	Fetch_B	0011BDA0	41BA0010	2F030000	4BFEAF7D	807F0050	887F00B1	60000000	...
1999720	Fetch_B	0011BDC0	4BFEAF65	807F0050	807F0094	60000000	419A0014	2F030000	...
1999743	Lwarx	0089AE6C							
1999755	Fetch_B	0011BDE0	805F00BC	98640000	48000008	807F00A4	801F00C0	805F00BC	...
1999783	Fetch_B	0011BE00	83ECFFFC	83CCFFF8	382C0000	7C0803A6	7C0802A6	4E800020	...

(a)

txn	cycle	instruction address	opcode	instruction	data address	data
F	1999720	dc0	807f0050	lwz 3,80(31)	23-eef39.01c0f d00	ef06ee80
	1999721	dc4	4bfeaf65	bl -86172		
	1999721	23-80106.00106 d28	7c0802a6	mfspr 0,8		8011bdc8
	1999723	d2c	93a1fff4	stw 29,-12(1)	23-eef39.01c0f ca4	00000000
	1999724	d30	93c1fff8	stw 30,-8(1)	23-eef39.01c0f ca8	00000000
	1999726	d34	93e1fffc	stw 31,-4(1)	23-eef39.01c0f cac	eef39cb0
	1999727	d38	9001fff0	stw 0,-16(1)	23-eef39.01c0f ca0	8011bdc8
	1999729	d3c	9421ffb0	stwu 1,-80(1)	23-eef39.01c0f c60	eef39cb0
	1999730	d40	7c7d1b78	or 29,3,3		ef06ee80
	1999731	d44	3bddffe8	addi 30,29,-24		ef06ee68
	1999731	d48	7fc3f378	or 3,30,30		ef06ee68
	1999732	d4c	48013fad	bl 81836		
L	1999743	23-8011a.0011a cf8	7ca01828	lwarx 5,0,3	23-ef06e.0089a e68	00000002
	1999744	cfc	3885ffff	addi 4,5,-1		00000001
	1999745	d00	7c80192d	stwcx. 4,0,3	23-ef06e.0089a e68	00000001
	1999746	d04	40a2fff4	bc 5,2,-12		
	1999747	d08	7c832378	or 3,4,4		00000001
	1999747	d0c	4e800020	bclr 20,0		
	1999748	23-80106.00106 d50	60000000	ori 0,0,0		8011bdc8
	1999748	d54	7c631b79	or. 3,3,3		00000001
	1999749	d58	408200d4	bc 4,2,212		
	1999749	23-80106.00106 e2c	80010040	lwz 0,64(1)	23-eef39.01c0f ca0	8011bdc8

(b)

Figure 3

(a) Sample transaction records from the same txn file used in Figure 1(a). (b) Sample instruction records corresponding to the transaction records shown in (a).

at a time, which again yields single-byte bus transactions and degraded performance.

In another study, we looked at the characteristics of general-purpose register (GPR) usage for two different operating systems. The main purpose of the study was to characterize the sequences of data patterns seen by each register. At the same time, we measured the frequency of access of each GPR. **Figure 4** shows plots of the relative

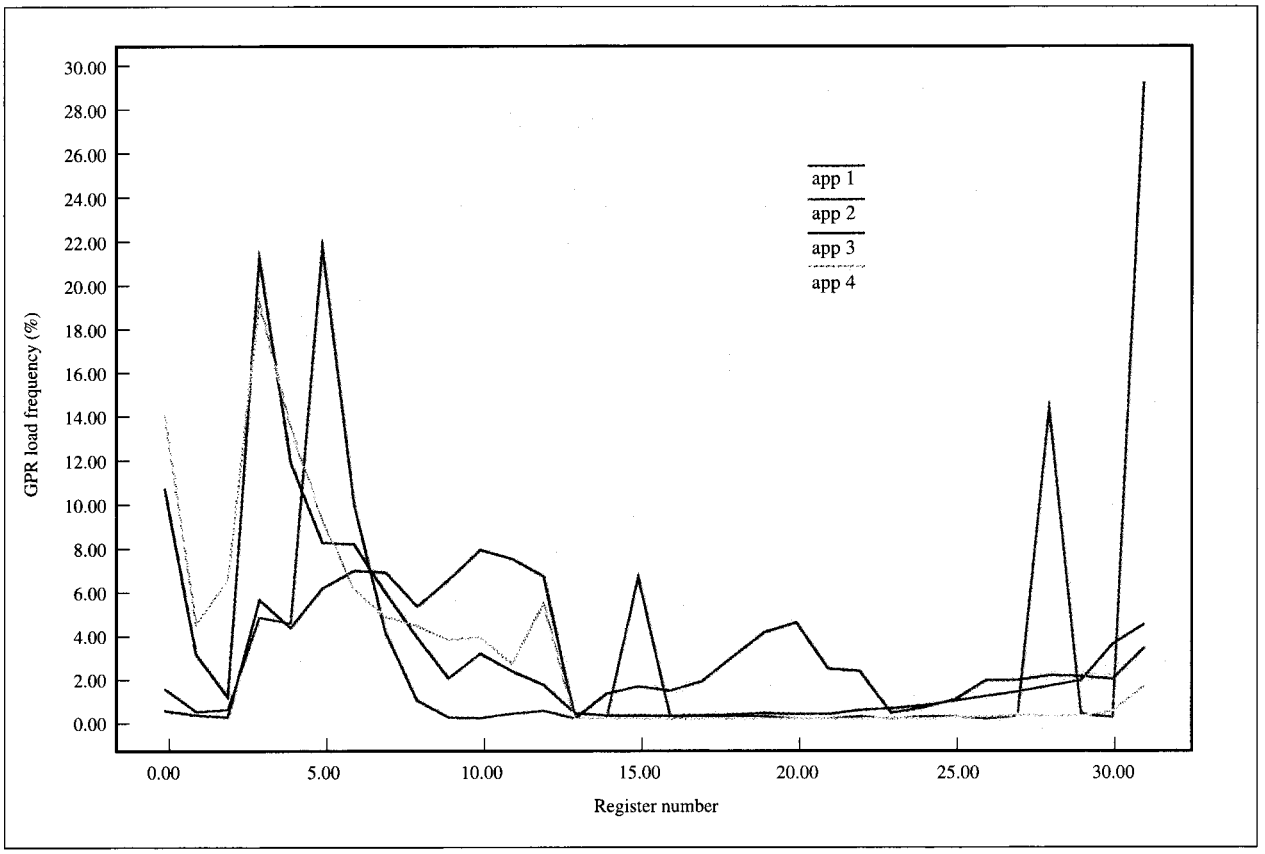

Figure 4

Percentage of GPR load instructions for individual registers versus register number.

frequency of usage of the 32 GPRs for four different applications running on three different operating systems. Among the interesting aspects of these data is the fact that the statistics are similar for the first and fourth applications, despite the fact that they correspond to quite different applications on two different operating systems.

We can quantify the register usage in these four cases by measuring how much they vary from the case in which all registers are equally used. The entropy of the register frequencies is an appropriate indicator of the flatness of the distribution. For 32 registers, equal usage will yield an entropy of 5. The entropies of the four applications shown in Figure 4 appear in **Table 1**. The corresponding powers of two, also shown in Table 1, represent the effective number of registers in use by the application. Low register usage, such as that in application 2, indicates that better register management is a potential source for performance improvement.

Table 1 GPR entropy and effective register set size.

Application	*Entropy*	*Effective number of registers*
1	3.92	15
2	2.95	8
3	4.52	23
4	3.57	12

Another performance issue that we have examined is that of the Java virtual machine (JVM) running certain benchmarks. There are a number of potential bottlenecks in the process of interpreting Java programs, one of them being the execution of the byte codes (that is, the virtual machine instructions) themselves. While analysis of the interpretation of individual byte codes can be done statically, dynamic analysis is needed to identify which byte codes are used most frequently. Trace analysis of byte codes is somewhat more problematic than analysis of

Table 2 Percentage of most frequent byte codes in ray trace.

Byte code	*Percentage*
dload	23.096
dmul	12.693
dadd	7.879
i2d	6.666
dstore	6.454
dsub	2.843
aaload	2.407
iaload	2.406
ldc2_w_quick	1.533
aload_0	1.322
ior	1.320
if_icmplt	1.318
iinc	1.314
isub	1.310
ldc_quick	1.207
getfield_quick	0.990
iconst_0	0.874
dconst_0	0.767
invokestatic_quick	0.660
ifge	0.660
dcmpg	0.660
ishl	0.660
ddiv	0.660
bipush	0.660
dconst_1	0.660

Table 3 Percentage of most frequent byte-code pairs in ray trace.

First byte code	*Second byte code*	*Frequency*
dload	dmul	11.265
dload	dload	6.783
dmul	dadd	5.249
i2d	dload	3.610
dmul	dload	3.397
dadd	dload	3.386
dstore	dload	2.736
dload	dadd	2.300
dsub	dstore	2.183
i2d	dstore	1.746
dadd	dstore	1.427
i2d	dsub	1.310
isub	i2d	1.310
iaload	isub	1.310
getfield_quick	i2d	0.990
aload_0	getfield_quick	0.990
if_icmplt	ldc_quick	0.877
ldc2_w_quick	dload	0.873
dmul	dsub	0.873
dload	dconst_0	0.767
ldc2_w_quick	dmul	0.660
dcmpg	ifge	0.660
ddiv	dstore	0.660
dsub	ldc2_w_quick	0.660
dadd	aload_0	0.660
bipush	ishl	0.660

Table 4 Percentage of byte codes at various JVM stack depths in ray trace.

Stack depth	*Percentage*	*Cumulative*
0	15.348	15.348
1	5.275	20.623
2	36.217	56.841
3	7.880	64.721
4	19.817	84.538
5	2.311	86.849
6	9.967	96.815
7	0.660	97.476
8	1.864	99.340
9	0.000	99.340
10	0.660	100.000

native instructions, because the byte codes themselves are data. Since our trace contains data, we can easily extract byte-code information from the trace. **Table 2** presents the frequencies of the most commonly occurring byte codes in a particular segment of the public-domain ray-trace benchmark. **Table 3** presents the frequencies of the most commonly occurring pairs of byte codes in that same segment.

Another aspect of the JVM that is of interest is its data handling. The JVM is a stack-based architecture. It is useful in dealing with a stack to characterize the depth of the stack over time. This will have implications both for real hardware implementations and for virtual machine implementations of the architecture. **Table 4** shows the percent of time that the stack is a particular depth for the same segment of the ray-trace benchmark as in previous tables. Note the relatively larger numbers for the even-numbered depths. This is due to the use of double-precision arithmetic throughout the application, each variable using up two positions when stored in the stack.

While we have touched here only on the analysis that has been done with these particular trace data sets, these examples highlight several aspects of the overall functionality of the NStrace toolset.

Concluding remarks

Hardware-based approaches to trace generation are generally acknowledged as yielding high-quality traces, but are commonly criticized for their cost. The NStrace tool described in this paper fits both of these characterizations. NStrace requires a logic analyzer and supporting hardware to access the processor pin signals. In order to get bus recordings that yield sufficiently long traces (up to 100 million instructions), we use additional data acquisition and compression hardware along with deep logic analyzer memory. The result of this investment is an ability to generate traces having the following characteristics.

The trace data very accurately reflect the characteristics of the system under normal operation. This is due both to the noninvasive nature of the recording and to the use of the bus-driven simulation. By passively recording the bus signals, normal timing relations among the processor and other system components are maintained. Since the simulator continually synchronizes with the bus recording, any divergence between the behavior of the simulated processor and that of the target processor quickly becomes apparent.

The trace data reflect the behavior of the target processor throughout the recording period. This includes execution of kernel code, of driver and library code, and of user code. These traces can therefore be used to characterize and analyze the behavior of both system and application code, and their interactions.

The trace data contain bus timing information that allows the internal state information, such as the instruction sequence, to be aligned with external state information, such as bus transactions. This is useful in analyzing potential bottlenecks in uniprocessor systems and provides the mechanism for generating traces in multiprocessor systems, or more generally whenever snooping may occur.

Finally, the trace is rich, in the sense that it contains more information than other trace generation approaches. This richness is due to the use of the architectural simulator used to generate the trace from the bus recording. Since the simulator models most details of the architecture, it has access to a wide range of system variables. For example, while a typical instruction trace might include the instruction and the effective instruction address, the NStrace trace data set includes both effective and real addresses of both instructions and data, along with the instruction, the data (for both memory accesses and for register operations), cacheability and protection information, and bus timing.

NStrace generates rich, accurate traces of both user and system code and has been used with several operating systems and system environments. As we described in the previous section, these traces have proven useful in performing a variety of performance and functional studies, some of which would be difficult or impossible to do with other tracing methods.

Acknowledgments

The authors wish to thank the other members of the NStrace team for their contributions to the project: Marina Davidovich, Christy Johnson, Kathy McGroddy, Ethan Rowe, and Dave Russell. We would particularly like to thank Dr. Ron Black and Tom Wilson for their support in making this effort possible.

References

1. *PowerPC 604 User's Manual*, Order No. MPR604UMU-01, IBM Corporation, 1994; available through IBM branch offices.
2. R. A. Kamin, G. B. Adams, and P. K. Dubey, "Dynamic Trace Analysis for Analytic Modeling of Superscalar Performance," *Performance Eval.* **19,** No. 2–3, 259–276 (March 1994).
3. R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu, "Evaluation of Multithreaded Uniprocessors for Commercial Application Environments," *Proceedings of the 23rd International Symposium on Computer Architecture*, Philadelphia, PA, May 1996, pp. 203–212.
4. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer* **17,** No. 1, 6–22 (January 1984).
5. C.-P. Wen, "Improving Instruction Supply Efficiency in Superscalar Architectures Using Instruction Trace Buffers," *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, Kansas City, MO, March 1992, pp. 28–36.
6. A. Poursepanj, "The PowerPC Performance Modelling Methodology," *Commun. ACM* **37,** No. 6, 47–55 (June 1994).
7. A. J. Smith, "Cache Memories," *Computing Surv.* **14,** No. 3, 473–529 (September 1982).
8. D. Nagle, R. Uhlig, T. Stanley, T. Mudge, S. Sechrest, and R. Brown, "Design Tradeoffs for Software-Managed TLBs," *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, May 1993, pp. 27–38.
9. T. Adams, "A Measurement Study of Memory Transaction Characteristics on a PowerPC-Based Macintosh," *Proceedings of COMPCON '96*, San Francisco, 1996, pp. 100–110.
10. D. Kimelman, B. Rosenburg, and T. Roth, "Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior," *Proceedings of IEEE Visualization '94*, October 1994, pp. 172–178.
11. R. R. Heisch, "Trace-Directed Program Restructuring for AIX Executables," *IBM J. Res. Develop.* **38,** No. 5, 595–603 (September 1994).
12. T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li, "A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching," *Technical Report UWCSE 96-09-01*, University of Washington, Seattle, 1996.
13. M. VandenBrink, "Performance Implications of the PowerPC Architecture's Hashed Page Table," *Proceedings of the IEEE International Performance, Computing and Communications Conference*, Phoenix, AZ, February 1997, pp. 315–320.
14. R. F. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Technical Report UWCSE 93-06-06*, University of Washington, Seattle, 1993.
15. *PowerPC Visual Simulator Users Guide*, Order No. 87GA0201, IBM Corporation, 1993; available through IBM branch offices.
16. J. Pierce and T. Mudge, "IDtrace—A Tracing Tool for i486 Simulation," *Technical Report CSE-TR-203-94*, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 1994.

17. D. W. Wall, "Systems for Late Code Modification," *Research Report 92/3*, Digital Western Research Laboratory, Palo Alto, CA, *wrl-techreports@pa.dec.com* (May 1992).
18. J. B. Chen, D. W. Wall, and A. Borg, "Software Methods for System Address Tracing: Implementation and Validation," *Research Report 94/6*, Digital Western Research Laboratory, Palo Alto, CA, *wrl-techreports@pa.dec.com* (September 1994).
19. J. Torrellas, A. Gupta, and J. Hennessy, "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System," *Proceedings of ASPLOS V*, Cambridge, MA, October 1992, pp. 162–174.
20. S. Eggers, D. Keppel, E. Koldinger, and H. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor," *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990, pp. 37–47.
21. S. McMahon, "The Capture, Characterization and Performance Analysis of Macintosh Traces," *Proceedings of COMPCON '96*, San Francisco, 1996, pp. 94–99.
22. M. S. Allen, M. Alexander, C. Wright, and J. Chang, "Designing the PowerPC 60X Bus," *IEEE Micro* **14,** No. 5, 42–51 (October 1994).

Received August 8, 1996; accepted for publication April 30, 1997

Peter A. Sandon *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (sandon@btv.ibm.com).* Dr. Sandon is a Senior Engineer in the PowerPC Performance Engineering Department. He received the B.S. degree in electrical engineering from Cornell University, the M.S. degree in electrical engineering from the University of California at Berkeley, and the Ph.D. degree in computer science from the University of Wisconsin. His current interests are in microprocessor performance measurement and analysis.

Yu-Chung Liao *IBM Microelectronics Division, 11400 Burnet Road, Austin, Texas 78758 (ycl@vnet.ibm.com).* Dr. Liao is Manager of the PowerPC Performance Engineering Department. He received the Ph.D. degree in mathematics from Brown University in 1982. The focus of his current work is on performance analysis and software technology for PowerPC microprocessors.

Thomas E. Cook *IBM Microelectronics Division, Burlington facility, Essex Junction, Vermont 05452 (tomcook@btv.ibm.com).* Mr. Cook is an Advisory Programmer in the PowerPC Performance Engineering Department. He received the B.S. degree in computer science from Clarkson University. For the past ten years, Mr. Cook has worked for IBM developing operating system and networking software. Currently, he works on improving the performance of software and hardware for PowerPC platforms.

David M. Schultz *IBM AS/400 Division, 3605 Highway 52 N., Rochester, Minnesota 55901 (dschultz@vnet.ibm.com).* Mr. Schultz is a Staff Engineer in the AS/400 Processor Verification Department. He received the B.S. degree in computer science from the University of Wisconsin at Stout. Mr. Schultz has worked in tool development, processor performance, and processor verification groups since joining IBM. His interests are in processor, operating system, and compiler development.

Pedro Martin-de-Nicolas *IBM Microelectronics Division, 11400 Burnet Road, Austin, Texas 78758 (pedrom@austin.ibm.com).* Mr. Martin-de-Nicolas is an Advisory Programmer in the PowerPC Performance Engineering Department. He received the B.S.E.E. degree from Rice University in 1987. Prior to joining IBM in 1990, Mr. Martin-de-Nicolas worked for NCR designing computer peripheral chips. His interests are in designing and analyzing both hardware and software for high performance.