# CONSTRUCTING INSTRUCTION TRACES *from* CACHE-FILTERED ADDRESS TRACES (CITCAT)

*Charlton D. Rose*
*sharky@byu.edu*

*J. Kelly Flanagan*
*kelly@cs.byu.edu*

*Performance Evaluation Laboratory*
*http://pel.cs.byu.edu*
*Department of Computer Science*
*Brigham Young University*

**Abstract**

Instruction traces are useful tools for studying many aspects of computer systems, but they are difficult to gather without perturbing the systems being traced. In the past, researchers have collected instruction traces through various techniques, including single-stepping, instruction inlining, hardware monitoring, and processor simulation. These approaches, however, fail to produce accurate traces because they interfere with the processor's normal execution.

Because processors are deterministic machines, their behavior can be predicted if their initial states and external inputs are known. We have developed a technique, called "CITCAT," which exploits this fact to generate nearly perfect instruction traces through trace-driven simulation. CITCAT combines the best features of instruction inlining, hardware monitoring, and processor simulation to produce long, accurate instruction traces without perturbing the system being traced. Because CITCAT instruction traces are computed, rather than stored, this hybrid technique delivers not just accurate traces, but also an extremely efficient trace compression algorithm.

## 1. introduction

When studying a component of a computer system in order to see how it can be improved, it is often useful to make the system perform a specific task while recording the activities of the component. If the recorded data, called a "trace," contains enough information to reproduce the component's behavior in a software-based simulator, then that same data can also be used to simulate the activities of components which are functionally identical but implemented differently.

Using trace data to subject a simulated component to the same sequence of demands experienced by a real component is called "**trace-driven simulation**." Because trace-driven simulation makes it possible to subject hypothetical components to real-world demands, it is an extremely useful tool for evaluating changes to systems before they are implemented.

Computer systems have many traceable components, but one of the most interesting and worthwhile components to trace is the CPU. An instruction trace enables researchers to model and study many aspects of computer systems. For example, processor-internal caches can be simulated, instruction-level parallelism measured, branch prediction algorithms tested, and dynamic instruction counts generated. Complete instruction traces also contain enough data to construct additional traces, such as memory activity and disk I/O traces.

In this paper, we will review several traditional methods that researchers have used to collect instruction traces. We will describe several problems associated with these methods and explain why they have failed, in our opinion, to produce perfect instruction traces.

We will then describe CITCAT, a technique through which three of these traditional approaches, instruction inlining, hardware monitoring, and processor simulation, can be combined to construct nearly perfect instruction traces. After discussing the pros and cons of our hybrid approach, we will conclude with a statement about past success, work in progress, and future research goals related to this technique.

## 2. traditional instruction trace-gathering techniques

Ironically, one of the most *useful* traces of a computer system to have is also one of the most *difficult* to collect. Many researchers are frustrated by the fact that if you disturb a computer system in order to measure it, the validity of the measurements becomes uncertain because they are taken from a perturbed system.

This problem is especially prominent when it comes to gathering perfect instruction traces. We define a **perfect instruction trace** as a continuous record of CPU instruction execution, including operating system code and context switches, that has been gathered from an unperturbed system and is long enough to be statistically useful. Although many techniques have been attempted, including single-stepping, instruction inlining, hardware monitoring, and processor simulation, all of these approaches have fallen short of producing *perfect* instruction traces. They were either too difficult to implement correctly or caused unacceptable perturbations in the system being measured.

### 2.1. single-stepping and breakpoints

Some researchers have gathered instruction traces by exploiting debugging features built into the processor. By enabling the processor's **single-stepping** mode or by setting instruction **breakpoints** at each basic block, debugging interrupts can be used to follow — and hence record — the processor's instruction execution. When these interrupts occur, the interrupt handler records information about the instructions being executed and then returns processor control to the point where the exception was raised.

This strategy is useful for generating user-code instruction traces, but it has several significant limitations. First of all, the overhead of processing an interrupt every instruction or basic block is expensive, and system performance will be dramatically degraded as a result. On many systems, this can mean a slowdown of 100 times or more.

Even if we are willing to wait, the instruction traces gathered by this technique will reflect the activities of a degraded system, not the original system, because instruction-level behavior often depends on processor speed. The main problem with single-stepping is that it requires a precious amount of the processor's most valuable resource — time.

Furthermore, because the debug interrupt handler is so frequently executed, it will always consume more resources than just processor time. Repeated execution of the debugging interrupt handler code will impact many of the processor's performance-enhancing caches. For example, in a virtual memory system, the page containing the interrupt handler will rarely be swapped out. Corresponding TLB entries will probably not be replaced, and first- and second-level memory caches may contain much of the code. True, some caches and buffers can be disabled while the interrupt is being processed, but this results in the interrupt handler consuming an *even greater* share of the CPU's time.

One of single-stepping's biggest challenges is tracing privileged code, such as the operating system and kernel, because privileged code sometimes performs time-critical operations. In addition, single-stepping cannot be used in critical sections because the interrupts must be disabled.

Unfortunately, there is no easy way to get around *all* of the problems associated with single-stepping. In itself, single-stepping is inherently self-limiting; it requires the CPU's direct involvement in the measurement of its own activities, resulting in a different sequence of instructions to trace.

### 2.2. instruction-inlining

To circumvent the problems of interrupt-based single-stepping, other researchers [1, 2, 3, 4] have tried **instruction-inlining**, a practice in which machine code designed to record instruction trace data is inserted into each basic block of the code. Although this technique successfully avoids the overhead of frequent interrupt processing, it is still subject to many of the same shortcomings that plagued single-stepping approaches. Again, because the CPU is directly involved in its own measurement, its workload increases, and the resulting measurements are of a degraded system.

### 2.3. hardware monitoring

Many processor chips and motherboards can be monitored by logic analyzers and other hardware monitors [5, 6, 7, 8, 9, 10, 11]. If a fast-enough logic analyzer is used, and a large-enough recording buffer is available, it might be possible to gather perfect instruction traces without disturbing the processor's instruction execution.

Unfortunately, most processors do not provide information on the pins that directly identify the instructions being executed. Monitoring the system bus for instruction fetch activity won't work, either, since most of the fetches will be satisfied in the first- or second-level cache. Of course, we can force all instruction fetches to appear on the bus by disabling the caches, but this will inevitably alter the original system's behavior. Furthermore, it might be difficult to distinguish normal instruction fetches from speculative execution, prefetching, and data reads.

Another issue that must be considered is buffer size. Although there have been many publications of trace-driven studies based on only a few millisecond's worth of instruction execution, some researchers are not comfortable drawing conclusions from such small amounts of data. Yet an instruction trace long enough to be considered "statistically significant" requires an enormous amount of storage space. Large storage devices are generally not fast enough to record live instruction traces without periodically losing data or halting the system being traced.

### 2.4. processor simulation

Single-stepping, instruction inlining, and hardware monitoring all fail to provide perfect instruction traces because they perturb the system. If the processor being traced is simulated in software, however, the system's activities, including instruction execution, can be freely monitored without affecting the simulation's outcome.

Although processor simulation shows promise as a method for generating perfect instruction traces [12], the procedure can be overwhelmingly difficult. In order to guarantee that the simulation is accurate, several requirements must be fulfilled:

(1) The simulator must execute each instruction correctly.
(2) The simulator must also consider the behavior of hardware components interacting with the processor, including peripheral devices such as SCSI controllers, network cards, etc.
(3) The simulator must take into account the *timing* of each instruction. This is necessary so that interrupts caused by external agents and timer ticks can be processed at precisely

the right moments.

(4) If the software being simulated requires real-time, continuous user input, such as an interactive application, special considerations must be made to provide the simulator with user-generated input. A real user cannot be used, however, because the speed of a simulator is extremely slow compared to that of a real machine, and people respond differently to 50 Hz machines than to 200 MHz machines.

Obtaining or writing accurate, instruction-level simulators is relatively easy. Software that is capable of meeting the above requirements, however, is both difficult to obtain and even more difficult to implement — especially if proprietary information about the processor and connected hardware is unavailable.

## 3. CITCAT — a hybrid technique for generating instruction traces

We have found a way to combine software-based processor simulation, instruction inlining, and hardware-based bus monitoring to build accurate instruction traces, or traces that represent *real* machines operating under normal, *unperturbed* conditions. Our technique is affectionately named "CITCAT," an acronym for "**C**onstructing **I**nstruction **T**races from **C**ache-filtered **A**ddress **T**races." CITCAT combines the best features of instruction inlining, hardware monitoring, and processor simulation to produce a hybrid instruction tracing technique that is capable of generating extremely long, statistically accurate instruction traces.

The first step in the CITCAT method is to pass processor control to a device driver that outputs the machine's state to the bus, where it is received by a hardware monitor. The processor is then set into full motion as the hardware monitor continues recording processor-external events, including cache-filtered bus activity and timing data. This information is later processed by an instruction-level simulator to produce a nearly perfect instruction trace.

### 3.1. required data

Because processors are deterministic, finite state machines, instruction-level simulators are capable of generating accurate instruction traces if the initial conditions and external influences are known. Thus, in order to generate a perfect instruction trace through instruction-level simulation, at least three records must be available:

(1) The **initial processor state**, including registers, flags, TLB entries, etc. must be known so that the processor can be properly initialized.

(2) The **initial memory state**, including the states of the first- and second-level caches, must be known in order to properly initialize the simulated memory subsystem.

(3) A list of **processor-external events**, including interrupts, DMA, and timing information, is needed to simulate external devices connected to the processor.

Each of these records can be derived from a specially instrumented, cache-filtered address trace. We now describe in detail how each of these records is created.

#### 3.1.1. the initial processor state

The initial processor state can be extracted from the beginning of an address trace if, at the beginning of the trace period, that information is written to the bus. This can be accomplished through a device driver that is activated at the very beginning of the trace. When activated, the driver first disables the caches, or at least sets them to write-through, and then writes the initial processor state directly to main memory. A hardware monitor connected to the bus receives this information and records it in the trace.

Because in many systems memory writes do not appear on the bus in the order they are issued, elements of the processor's initial state must be written to carefully chosen, unique addresses so that they can later be identified in the trace. It is best if the processor's state is written to addresses that are not used during the rest of the trace.

### 3.1.2. the initial memory image

In order for the instruction simulator to begin fetching and executing instructions, the machine's initial memory state must be known. A complete memory image is not required, however, because it is unlikely that every address in main memory will be used. Furthermore, only those memory locations which are read (or fetched) before written need to be defined in the initial memory image.

By invalidating the memory caches at the beginning of the trace, the first reference to each location will appear on the bus and be recorded in the address trace. Afterwards, the trace data can be scanned by a software tool that finds the first occurrence of each address. Of these first occurrences, those which are writes can safely be ignored, and those which are reads will contain all the data necessary to build the initial memory image.

### 3.1.3. the timed event list

Although a processor's behavior is deterministic, it can be influenced by many different kinds of external events, such as interrupts, I/O, and DMA. A record of these events, including timing information, must be available to the simulator so that the processor's reaction can be accurately simulated.

During address trace collection, most processor-external events, such as interrupt signals and I/O activity, can be monitored and recorded by a logic analyzer without affecting the system. DMA events can be extracted from the trace by scanning for values read from memory which are inconsistent with previous reads and writes to the same locations.

As mentioned earlier, external events must be recorded with timing information, or the simulator will not know when to replay them. This timing information should be expressed in units that are meaningful to the simulator: namely, number of instructions executed, number of basic blocks entered, or number of branch instructions processed. Many processors have pins that provide this information, so this data can also be recorded by the logic analyzer.

For peripheral DMA events, timing does not need to be precise. This is because DMA events have a wide window of time in which they can occur without changing the way they affect the processor. This window begins immediately after the processor's last access to memory locations associated with the DMA event, and ends just before the processor's first access to those same locations after the event.

In cases where asynchronous interrupts are generated internally by the processor, a small amount of code that signals the event to the hardware monitor must be incorporated into the interrupt handler. For example, because the MIPS R4000 chip generates internal timer ticks, we have had to include code in the R4000 operating system's timer routine that sends a special signal to the bus. This made it possible to include the timer ticks in the cache-filtered address trace.

## 3.2. putting it all together — the CITCAT instruction simulation algorithm

Once the initial processor state, initial memory image, and external events schedule has been obtained, the following algorithm can be used in an instruction-level processor simulator to generate very accurate instruction traces:

(1) **Initialize the processor** with the initial processor state, including the program counter, registers, flags, TLB entries, etc.
(2) **Initialize main memory** with the initial memory image.
(3) **Execute the instruction** at the program counter. (This includes incrementing the program counter or setting it to the target of a taken branch instruction.)
(4) **Output trace information** for the instruction just executed.
(5) **Check the events schedule** to see if it is time for an external event.
(6) **If it's time for an external event, simulate it** by signaling an interrupt, modifying main memory, or performing any other appropriate action.
(7) **Go to step 3.**

Simulation is the final step in the CITCAT instruction trace generating technique. Because this algorithm makes it possible for a simulator to execute the same instruction sequence as a real, unperturbed processor, the simulator can produce a nearly perfect instruction trace.

## 4. evaluation of the CITCAT technique

Given an instrumented, cache-filtered address trace taken from a real computer system, we have shown how an instruction-level simulator can be used to reconstruct the original processor's behavior. As we have stated earlier, the result is a *nearly* perfect instruction trace. We use the word "nearly" because there are a few aspects of our hybrid approach which admittedly introduce small changes into the processor's natural behavior. We believe, however, that the overall impact of these changes is insignificant and that instruction traces generated by the CITCAT technique have a very high degree of accuracy.

### 4.1. weaknesses

In the CITCAT method, caches must be invalidated at the beginning of the address trace so that the first reference to each memory location will appear on the bus. Filling the cache takes a small amount of time, and this unarguably affects the behavior of the processor. However, after a very long run — particularly after the caches have been refilled — the impact of this procedure on the processor's overall behavior is negligible. If necessary, the affected portion of the instruction trace can simply be thrown out.

The other reason we have said our instruction traces are "*nearly* perfect" is that, for some processors, a very small amount of code must be added to the operating system to accommodate internally generated, asynchronous interrupts, such as the MIPS R4000's timer tick. Again, we believe that the impact of this is very small.

### 4.2. strengths

On the positive side, the CITCAT procedure has two very prominent features. Its first and most important strength is that it produces accurate instruction traces. The second benefit is the fact that since the trace is computed, not stored, an unprecedented measure of trace data compression can be realized.

#### 4.2.1. accuracy

In the past, it was nearly impossible to evaluate the integrity of trace-driven simulation studies because the traces used in the studies were collected from perturbed systems. Some of these studies also excluded operating system code and the effects of task switching. With the hybrid method we have described in this paper, the validity of these theories can now be assessed. The implications of this may be enormous.

The value of CITCAT instruction traces stems from the fact that they are based on the activities of an unperturbed system. These instruction traces include not only user code, but operating system code as well. Instruction execution resulting from interrupt handling, context switching, and user-input processing is also included in the trace. Our resounding cry is, *"Every instruction executed by the processor, as the processor runs at full speed and under normal, unperturbed conditions, appears in CITCAT traces!"* To the best of our knowledge, this has not been accomplished with traditional trace collection techniques.

As more and more people use the CITCAT technique, nearly perfect instruction traces will make it possible for researchers to conduct accurate, trace driven simulation studies in many areas:

- measuring instruction-level parallelism of commercial workloads and operating systems
- observing the effects of context switching on performance-enhancing processor components
- testing the effectiveness of various cache configurations
- analyzing dynamic instruction frequency for different benchmarks
- testing various branch prediction algorithms
- evaluating power consumption in processor components
- debugging and validating systems

With nearly perfect instruction traces, researchers can concentrate on creating accurate simulation models, rather than worrying about the validity of their trace data.

#### 4.2.2. trace compression

An unintentional side-effect of the CITCAT procedure results from the fact that instead of *storing* instruction traces, we *compute* them. This makes it possible for very long traces to be generated from relatively small amounts of data. As we outlined earlier, successful implementation of CITCAT requires three records: two of them contain the initial state of the system, and the third contains the event list.

Because the maximum combined size of the initial state records is independent of the instruction trace's length, the storage requirement for these records is negligible. The length of the event list, however, directly affects the maximum length of the instruction trace because the simulator cannot continue after the list of external events has been exhausted. Fortunately, the ratio of external events to instructions executed is extremely low. Thus, CITCAT doubles as a remarkably effective algorithm for instruction trace compression. Just imagine using this technique to fit a 100 gigabyte instruction trace on a single CD-ROM! This may eventually be possible.

## 5. work completed and work in progress

We have implemented a prototype of CITCAT on an Intel i486 based system. A hardware monitor was connected to the processor address, data, and control lines to collect all bus transactions. From the collected trace data we were able to generate an initial memory image and an event list. For event timing we used the instruction count signals available on our special ICE (In-Circuit Emulator) i486 component. To obtain the initial state of the processor, we wrote a device driver that invalidates the internal cache and TLB contents at the beginning of the trace and writes the contents of each register to reserved memory addresses. We used a proprietary, instruction level simulator to generate an instruction trace, and then confirmed our results by comparing the final states of memory generated by the real processor and the simulator. We found that they were identical.

Due to the proprietary nature of the collected trace data and instruction level simulator, further results from our prototype cannot be published. To remedy this situation, we are currently implementing CITCAT on a MIPS R4400-based computer system. Future plans include an implementation of CITCAT on the PowerPC 604. We will make the resulting traces and tools available to the research community.

## 6. References

[1] A. Borg, R. E. Kessler and D. W. Wall, "Generation and Analysis of Very Long Address Traces." *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pp. 270-279. ACM 1990.

[2] S.J. Eggers, D. R. Keppel, E. J. Koldinger, and H. M. Levy, "Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor." *Proceedings of 1990 ACM Sigmetrics*, pp. 37-45.

[3] C. Stephens, B. Cogswell, J. Heinlein, and G. Palmer, "Instruction Level Profiling and Evaluation of the IBM RS/6000." *Proceedings of the Eighteenth International Symposium on Computer Architecture*, pp. 180-189. ACM 1990.

[4] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*. MIPS 1988.

[5] Douglas W. Clark, "Cache Performance in the VAX-11/780." *ACM Transactions on Computer Systems*, February 1983, vol. 1 no. 1, pp. 24-37.

[6] Douglas W. Clark and Joel S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement." *ACM Transactions on Computer Systems*, February 1985, vol. 3 no. 1, pp. 31-62.

[7] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and R. Brown, "Design Tradeoffs for Software-Managed TLBs." *Proceedings of the Twentieth International Symposium on Computer Architecture*, pp. 27-38. ACM 1993.

[8] Josep Torrellas, Anoop Gupta, and John Hennessy, "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System." *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 162-174. ACM 1992.

[9] J. Kelly Flanagan, Brent E. Nelson, James K Archibald, and Knut Grimsrud, "BACH: BYU Address Collection Hardware, The Collection of Complete Traces." *Proceedings of the Sixth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, pp. 128-137. 1992.

[10] J. Kelly Flanagan, Brent E. Nelson, James K Archibald, and Knut Grimsrud, "Incomplete Trace Data and Trace Driven Simulation." *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS*, pp. 203-209. SCS 1993.

[11] K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "BACH: A Hardware Monitor for Tracing Microprocessor-Based Systems." *Microprocessors and Microsystems*, October 1993, vol. 17 no. 6.

[12] Gurindar S. Sohi and Manoj Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors." *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 53-62. ACM 1991.