Issues in Trace-Driven Simulation

David R. Kaeli

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, N.Y. 10598

Abstract

Considerable effort has been devoted to the development of accurate trace-driven simulation models of today's computer systems. Unfortunately many modelers do not carefully inspect the input to their models. The fact is that the output of any model is only as good as the input to that model.

This paper discusses the many issues associated with the input traces used in trace-driven simulation. A description of the different types of traces is provided, followed by survey and discussion of the following trace issues: trace generation techniques, trace-length reduction techniques, trace selection and representativeness, and common trace misuse.

The aim of this tutorial paper is to equip modelers with enough information about the different trace types and tracing methodologies, so that they can be more critical of the quality of the input traces used in their trace-driven simulations. Keywords: *instruction traces, address traces, trace-driven simulation, representativeness.*

1. Introduction

Trace-driven simulation is a popular technique used to evaluate future computer designs [1, 2]. Many times the modeler is so focused on the problem being studied that the content of the input trace used in the evaluation is overlooked. If the modeler is not critical of the input traces chosen as input to his or her model, the output of the model may be of little value.

Many different types of traces can be used as input to models. The content of the input trace is dictated by the particular elements of a computer that the modeler chooses to study, and also by the level of detail that is of interest. Next, a review of the different trace types is provided.

1.1 Address Traces

Address traces are probably the most commonly used kind of trace. An address trace is a record of memory reference activity at some level of the memory hierarchy. An address trace typically contains the following information:

- virtual/physical address
- stride (the number of types transferred)
- type or identification (e.g., instruction vs. data, fetch vs. store, etc.)

The trace can contain all or a subset of the above fields. Other fields frequently captured in an address trace are updates to translation lookaside buffers, process identifiers, and program state indicators.

Typical uses of address traces include memory hierarchy studies, program pathlength analysis, and page size sensitivity studies. More papers have been published on trace-driven memory hierarchy studies (using address traces as input), than on any other topic in computer architecture [3, 4, 5, 6].

Address traces contain a snapshot of the memory reference activity during a time interval. One problem encountered when attempting to capture address traces on current microprocessors is the inability to collect memory references that are resolved on the on-chip cache [7]. Either the on-chip cache must be disabled (which will introduce some perturbation into the trace) or the input address lines to the cache must be surfaced to the external world (i.e., to I/O pins). The ability to capture traces on these microprocessors must be included in the design process.

1.2 Instruction Traces

Instruction traces contain the actual instructions executed during a snapshot of time. These traces contain similar information to that found in address traces, with the addition of instruction opcodes, interrupts, and exceptions. These traces, while being substantially "wider" than address traces, are also commonly used in memory hierarchy studies. Other common uses of instruction traces are:

- processor pipeline studies,
- branch prediction studies,
- floating-point unit evaluation, and
- instruction profiling

Instruction traces are considerably more difficult to collect, since instruction opcodes must be captured as well as memory reference addresses. This can pose a technological challenge on current superscalar microprocessors, where multiple instructions can be executed on a single processor clock cycle [8].

1.3 I/O Traces

A third type of trace used in trace-driven simulations is an I/O trace [9, 10, 11, 12]. This type of trace is a record of I/O events, capturing a variety of disk I/O activity. Other activity captured in I/O traces includes transfers between devices on an external bus (e.g., LAN, video adapter, etc).

A typical I/O disk trace contains the following information:

- disk address (e.g., sector #, track #, etc.),
- # of blocks transferred, and
- memory address.

Typical uses for I/O traces are for tuning paging algorithms, studying disk caches, and analyzing I/O subsystems. Many times, queuing models are used in favor of trace-driven simulation for studying I/O performance issues [13].

Trace Type Summary

While address traces are the most common type of trace, all three of the trace types just presented are frequently used to evaluate design trade-offs. The remainder of this paper will discuss the many issues related to traces. The organization is as follows. Section 2 reviews the many trace generation methodologies. Section 3 discusses trace-length reduction strategies. Section 4 discusses trace selection and trace representativeness. Section 5 provides some examples of common trace misuse. Section 6 summarizes this work and provides some rules-of-thumb for the trace-driven simulation modeler.

2. Trace Generation Methodologies

Many approaches have been proposed to obtain traces on computer systems. These approaches can be divided into two class: 1) software-based, and 2) hardware-based.

2.1 Software-based Trace Generation

A variety of software-based tools have been made available for obtaining traces on current computer systems [14, 15, 16]. These tools modify the source program at different stages of the compilation process. There are two compiler-based modification methodologies: 1) compile-time modification, and 2) link-time modification.

Compile-time Code Modification

:hp1Compile-time code modification is a commonly used method of generating traces of program execution [14, 15, 17, 18]. This methodology takes as input the assembly code of a program, and produces a modified version of the assembly code. The modified version contains additional code that will call trace library routines. Another feature allows a program map of the code to be generated. This provides for the generation of instruction traces. The modified code is then linked with the standard libraries, as well as with the additional tracing library routines (as provided by the tracing tool). When the program is run, a trace is generated which consists of an encoded stream of events. The encoded trace is then expanded using the program map generated previously. Eggers etal. provide a description of such a tool for generating traces on a multiprocessor system [14].

Link-time Code Modification

Another methodology commonly used is called *link-time code modification* [18, 19]. Using this methodology, code is added at link time for each memory reference. When the modified code is executed, memory reference information is stored in the trace buffer. Code is added at the entry and exit of every basic block in the program. When the code is executed, the basic block information is also stored in the buffer.

The major advantage of link-time tracing over compile-time tracing is that the former captures trace information for all code, including the code in link libraries. Compile-time tracing does not trace this code. One example of link-time code modification for a RISC-based machine can be found in [16].

Microcode-based Trace Generation

Microcode-based trace generation collects traces by modifying the processor microcode on the target machine. A detailed description of this approach can be found in [20, 21].

No changes are made to the source code when using microcode-based trace generation. Instead, the microcode of the microprocessor is modified. Routines are added to the microcode which store address information in a reserved memory area on each memory request. The major advantage of this approach is that both application and operating system code can be traced. Since the operating system can produce a significant number of memory references (Flanagan etal. report, that for their MACH 2.6 single-process traces, 12-24% of all references are due to the operating system [22], LaMaire and White report that in MVS workloads, up to 70% of the references are due to the operating system [23]), it is very important to capture these references.

Software Emulation

Another approach used to generate traces is called *software emulation.* Using this methodology, a software program is developed that emulates the instruction set architecture (ISA) of the system of interest [24]. Code compiled for this system will run on the emulator.

A translation takes place between the target system ISA and the ISA of the host machine (i.e., the system upon which the emulator runs). When a program is run on the emulator, a trace is generated. The speed of the software emulation system typically depends upon the efficiency of the translation between the two ISA's and the overhead associated with saving the trace data.

2.2 Hardware-based Trace Generation

An alternative approach to modifying code/microcode or writing an emulator is to use a hardware-based trace generation methodology. There are two types of hardware tracing mechanisms: 1) trap-bit tracing, and 2) real-time tracing.

Trap-bit Tracing

Trap-bit tracing is a commonly used technique to generate instruction traces on microprocessor-based systems. A bit is provided by the ISA of the microprocessor which, when set, causes an interrupt to occur on the machine being traced. An interrupt service routine is entered which inspects the current instruction and captures any desired information (e.g., addresses, instructions, etc.). The Intel 80386 microprocessor family provides such a facility [25], as does the VAX architecture [26].

The interrupt service routine, which is called when the microprocessor traps out, can be customized to gather the particular information of interest. The trap bit is reset during the time when the interrupt service routine runs, and is set upon the exit from the interrupt service routine (otherwise the interrupt service routine would be traced). When the next instruction is executed, a trap will occur, and the procedure is repeated.

Real-time Tracing

Real-time tracing captures traces from the target machine by electronically monitoring signals on pins and/or busses [22, 27, 28, 29] Traces are gathered while the machine is running, so the hardware used to capture the traces must match the fastest trace generation speed. Real-time tracing has the main advantage of not perturbing the system being traced. The traces are complete and accurate.

The two main challenges when designing a real-time tracing system are: 1) matching peak data rates, and 2) capturing long traces. Next, the trade-offs associated with the different mechanisms presented are discussed.

2.3 Trace Methodology Comparison

Figure 1 lists the six different tracing methodologies just presented. The table compares the six methodologies based on: 1) the amount of time dilation introduced into the trace, 2) whether the operating system is captured, 3) the typical sample size gathered, and 4) the typical cost.

Time Dilation

Time Dilation occurs because the trace methodology introduces some type of overhead into the system. The reason why this is a concern is that by slowing the system down, events that used to occur in real time (e.g., input/output, interrupts, and timers) now occur with non-realistic timings. These events may timeout or may occur, artificially, too soon due to the overhead of the tracing mechanism. This will affect the correctness of the trace.

Link-time and compile-time code modification suffer substantially from time dilation in that considerable time is spent storing information into the trace buffer on each memory reference or basic block entry/exit. Published results indicate that a 10x slowdown (1/10 as fast as real-time) is experienced when using these methods [20].

Similarly, microcode modification experiences considerable time dilation. Again, the overhead is associated with saving the trace information. It has been reported that microcode-based tracing produces slowdowns comparable in magnitude to those found in the compiler-based modification methodologies [20].

Software emulation suffers from at least two sources of inaccuracy: 1) emulation of an ISA typically does not emulate the I/O subsystem, and 2) the time to translate between the target ISA and the host ISA can be on the order of 10x. Either of these issues can substantially affect the correctness of a trace.

	Time Dilation	O/S Coverage	Sample Size
compile-time code modification	10X	NO	1GS
link-time code modification	10X	NO	1GS
microcode-based code modification	10X	YES	1GS
software emulation	10X	YES	UNLIMITED
trap-bit tracing	100X	YES	UNLIMITED
real-time tracing	1X	YES	100MS

GS - 1 billion samples
MS - 1 million samples

Figure 1. Trace Methodology Comparison

The overhead introduced into the system with trap-bit tracing is due to trapping out to an interrupt service routine, executing the code necessary to save the desired trace information, and then returning to the next instruction. This sequence is performed for every instruction executed, and thus more overhead is associated with using this methodology than with the code modification techniques. System execution is dilated on the order of 100x using trap-bit tracing.

Real-time tracing does not introduce time dilation into the system. Some real-time tracing implementations have suggested slowing down the system clock. This should not be considered real-time tracing. Some events (e.g., I/O, timers, etc.) will still execute at full speed, thus corrupting the integrity of the trace. Other reported implementations suggest stopping the system to unload the trace buffer [22]. If this approach is employed, it should be clearly understood what perturbations are caused by halting the system. It must be stressed that stopping the system at all will usually produce some perturbation.

Operating System Coverage

It is quite important to capture accesses made by the operating system. There are two reasons why: 1) a large percentage of all references on the system are due to the operating system (as reported earlier), and 2) the behavior of operating system code is very different from application code (e.g., operating system code is notorious for causing poor cache performance [20]).

Summarizing the second column in Figure 1, only the two compiler-based code modification methodologies are incapable of capturing the operating system code (unless, of course, the entire operating system has been instrumented and recompiled). The other three methodologies capture both the operating system and the user programs.

Sample Size

The appropriate length of a trace will be dictated by the problem being studied when using the trace. It has been stated that traces longer than 5 billion references in length are necessary for modeling current memory hierarchy designs [16]. This point is debatable, but having longer traces is always more desirable (i.e., we do not have to use the entire trace, but if we have it in hand, we can then determine what length is appropriate).

Figure 1 lists the longest possible trace lengths for the six tracing methodologies. We see that code modification techniques can generate very long traces. The limiting factor here is the size of the trace buffer allocated on the machine. Software emulation can generate traces of unlimited length. Tracing is under software control, such that the trace can be unloaded from the system at any time, and tracing can pick up from where it left off. The trap-bit tracing approach can also capture traces of unlimited length. This is true because the traced system is under the control of a interrupt service routine. The interrupt service routine can detect when the buffer is full and take the appropriate action. While this is not true for the code modification methodologies, a separate detection routine could be invoked when an addressing exception occurs.

Real-time tracing is the most severely restricted is this category, being limited by the amount of memory (random access memory or disk) supplied on the trace system implementation. One option is to allow the tracing system to detect when the trace buffer is full. The system being traced can be halted, the trace memory unloaded, and then tracing resumed. Even though this approach is feasible, it is undesirable to stop the system under test since some artifacts may be introduced.

Cost

The last column in Figure 1 shows the estimated cost of each of these six methodologies. The cost for the software-based code modification methodologies is low. Either the compiler or the linker needs to be modified. Some systems already provide such tools [30].

Microcode-based modification is quite expensive, unless one is fortunate enough to be a microprocessor manufacturer and have access to the microcode.

Development of a software emulation system can be a substantial software coding project. The emulation program has to be able to execute every instruction in the target ISA. This can be over 300 instructions for some ISA's [25].

The cost of implementing a trap-bit tracer is low, since many of the current microprocessors provide such a feature. The only development effort necessary is associated with the coding of the interrupt service routine that will save the trace information.

The cost of custom hardware to perform real-time tracing is quite high. The system must be able to capture traces at high clock frequencies (when full instruction traces are desired). The memory used to store the trace must be able to accept samples at very fast rates (typically faster than 100 MHz.). The cost of static and dynamic random access memory in this clock frequency range is quite high (see Figure 2 for a range of current prices for static random access memory).

One approach suggested to reducing the cost of a real-time tracing system is to interleave between banks of slower memory, buffering the data in high-speed registers, and multiplexing in a round-robin pattern through the slower memory arrays [31]. This can reduce the cost of a real-time tracing system by an order of magnitude, since a majority of the cost of the system is tied to the cost of the trace memory.

2.4 Summary of Trace Generation Methodologies

Comparing the many tracing methodologies just presented, there exist disadvantages in each of the approaches. If the goal is to acquire accurate and complete traces (i.e., containing no time dilation and containing all operating system execution,) the only choice is to use a real-time tracing methodology. The main problem with real-time tracing is the cost of the storage necessary for capturing the trace. The cost of the tracing system is directly proportional to the amount of storage necessary to hold the trace. By using the simple multiplexing scheme described above, the speed requirements on the trace memory can be relaxed.

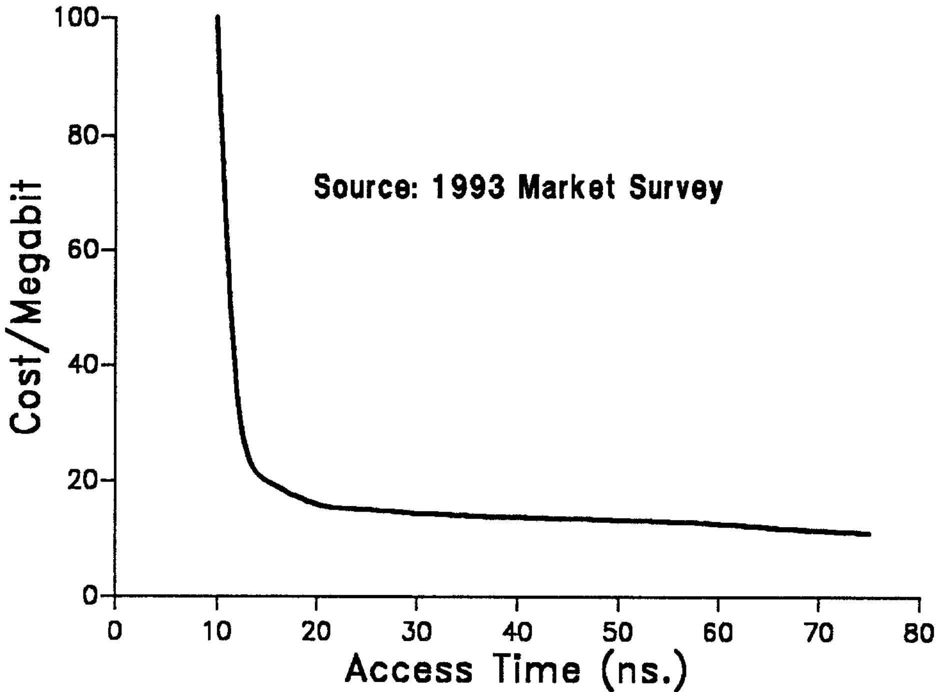

Figure 2. Static RAM cost vs. Access Time

3. Trace Length Reduction

Assuming that the required trace has been obtained, we now need to explore how we can store and use the trace efficiently. Traces take up a considerable amount of space. Trace-driven simulation execution time is directly proportional to the length of the trace being processed. The next two topics covered describe how to minimize the amount of space occupied by a trace, as well as how to reduce the amount of time needed to process a full trace in a trace-driven simulation run.

3.1 Compacted Traces

Typically, traces consume a considerable amount of space (some traces are many gigabytes in size). Methods have been devised to reduce the physical space consumed by a trace. Some of the desirable attributes of a trace size reduction methodology are:

- the size reduction factor should be significant,
- the reduction/expansion algorithm should run efficiently, and
- no information should be lost.

One strategy that has proven to be successful in reducing the length of address traces is called *Mache* [32]. This technique combines a type of cache filter, emitting either a miss or hit record, and then uses the Lempel-Ziv compression algorithm [33] to compress the miss/hit records. The algorithm reduced address traces (containing both instruction and data references) by 91-97%.

While this methodology does reduce the size of the trace, it does not reduce the overall trace-driven simulation time. The trace record still needs to be expanded in order to be input to the model. The trade-off is that less disk I/O is taking place, while extra processing is occurring due to expansion.

3.2 Simulation Time Reduction

If the goal is to reduce the overall execution time of the trace-driven simulation, then another strategy besides compaction must be employed. It has been noted that memory references tend to display the property of temporal locality [34]. By taking advantage of this characteristic in address traces, significant reductions in simulation time can be realized.

Smith proposed a method called *stack deletion* [35]. Using this approach, all references that hit to the top N levels of an LRU stack are discarded from the trace. The assumption is that most memory management systems will typically retain these references in memory, and thus, a similar number of misses will occur when discarding the hits to the top of the stack. While the results presented indicate a substantial reduction in trace length (25-95% shorter), the method has only been applied in paging studies. The large variance in the reduction factor is due to a large variance in the locality of the page references contained in the traces used.

Another methodology, that produces exact results when using the reduced address trace, is called *trace stripping* [36]. A direct-mapped cache is modeled, and only misses to the model are kept in the reduced trace. Exact results are obtained when modeling caches with the same or less number of

sets, provided that the cache line size remains the same. Traces are reduced by a factor of (90-95%) using trace stripping.

Other extensions to trace stripping have been proposed. Wang and Baer describe how to reduce traces using a modified version of trace stripping, that addresses simulation of write-back caches [37]. In addition to misses in a direct-mapped cache, first-time writes are also kept in the reduced trace.

Agarwal and Huffman propose a scheme called *trace blocking* [38], which takes advantage of both the temporal and spatial locality in programs to reduce the size of the trace. A cache filter with a block size of 1 is used to discard references within a temporal locality. Then a block filter is used to compact the trace to take advantage of spatial locality in the trace. Some errors are introduced using this method. The size of the resulting trace is reduced by 95-99% when using trace blocking.

Chame and Dubois introduce a new method for reducing the length of multiprocessor traces used in trace-driven simulation called *trace sampling* [39]. Their strategy first applies the Wang and Baer method, and then samples a number of processors. While this approach suffers from inaccuracies, the errors are typically small (less than 5%), while the overall simulation time is reduced by more than 97%.

While each of these methodologies reduces the overall simulation time, the accuracy of the methodology must be clearly understood. Errors as small as 5% can invalidate the modeling results.

4. Workload Selection and Representative Traces

In the preceding sections, the issues of how to capture traces and how to use them more efficiently were presented. The next question is: "What do we want to trace?" In this section we will discuss how to select an appropriate workload to trace and how to obtain representative traces.

4.1 Workload Selection

The selection of an appropriate workload to trace is typically driven by the particular problem under study (i.e., what type of work is typically performed on the machine we are designing). Workload types can be broken down into various categories:

- fixed-point vs. floating point,
- processor bound vs. memory bound vs. I/O bound,
- standardized benchmark vs. application vs. operating system, and
- scientific vs. transaction processing vs. database vs. general purpose.

The above list is not nearly complete, but it demonstrates the many different facets of selecting an appropriate workload.

To clearly understand the performance of the modeled system, the correct input must be selected. Besides tracing workload that is particular to the problem being studied, it is vital to use a range of workloads. Many of the new benchmark suites (e.g. SPEC 92 [40] and SDM [41]) attempt to provide this range of workloads. Gray provides a good reference covering the current state of benchmark programs [42].

While benchmarks are the most readily available, and easiest to trace, traces of real workload on customer machines are more interesting. In an attempt to create a more realistic transaction processing benchmark, the Transaction Processing Council was formed. Since its creation, the council has produced a number of benchmarks (TPCA, TPCB, TPCC). A very good description of each of these benchmarks can be found in [42].

Other workloads of interest include the SPLASH benchmark suite [43], commonly used in multiprocessor studies [44, 45], and the PERFECT Club benchmark set [46], used to study supercomputer performance issues [47].

4.2 Collecting Representative Traces

After the particular environment that needs to be traces has been selected, and after traces have been obtained, how do we know if our traces are of any value (i.e., did we capture the *important* part of the execution in our trace).

To help answer this question, we introduce the term *representativeness*, which describes how well the sample we have collected captures the certain characteristics (we are probably only focusing on a subset of the workload characteristics) of the entire execution that we are studying. To better judge the representativeness of a trace, workload characterization is commonly used [2].

Two approaches can be taken to perform characterization: 1) execution monitoring, and 2) trace sampling. Execution monitoring involves using either internal instrumentation provided with the system, or some external hardware to monitor particular events on the system. In [23], LaMaire and White provide a workload characterization study for the IBM System/370 system.

The second approach to characterizing the workload on the system is to obtain samples of execution. Two approaches can be taken here. One approach is to capture a set traces and then use statistics to evaluate the representativeness of any particular trace. A second approach captures short samples over the entire execution of the trace and then attempts to stitch them back together [48, 49]. The selection of the which method to use will

Instruction Opcode	% of all instr.
MOV rw,m	7.53
PUSH rw	5.77
POP rw	5.31
JNE disp	4.46
LOOP disp	4.40
JE disp	3.82
MOV m,rw	3.02
JMP disp	3.02
MOV rw,rw	2.81
CALL disp	2.76
RET	2.67
CMP rb,kk	2.23
Total	47.8

(Combined statistics for 96 traces)

Figure 3. Instruction Opcode Frequencies

depend upon the amount of transient behavior encountered in the entire execution. If the execution is predictable, then a single trace should prove to be sufficient. If the execution exhibits more random behavior, then the short sample technique may be more useful.

Figures 3, 4, and 5 show examples of workload characteristics commonly used to study representiveness in traces. Figure 3 shows the opcode frequencies contained in a set of 96 traces, taken from an Intel 80386-based personal computer workload [50]. Figure 4 shows the number of unique memory pages touched over the execution of a single trace. Figure 5 shows the frequency, over time, of the memory references across the memory address space (also called the basic block usage). Each of these characteristics helps the modeler to gain more insight into the contents of the captured trace.

5. Common Trace Misuse

Traces are commonly misused in trace-driven simulation studies. There are many reasons why. In this section we will discuss examples of trace misuse, and suggest how they can be avoided.

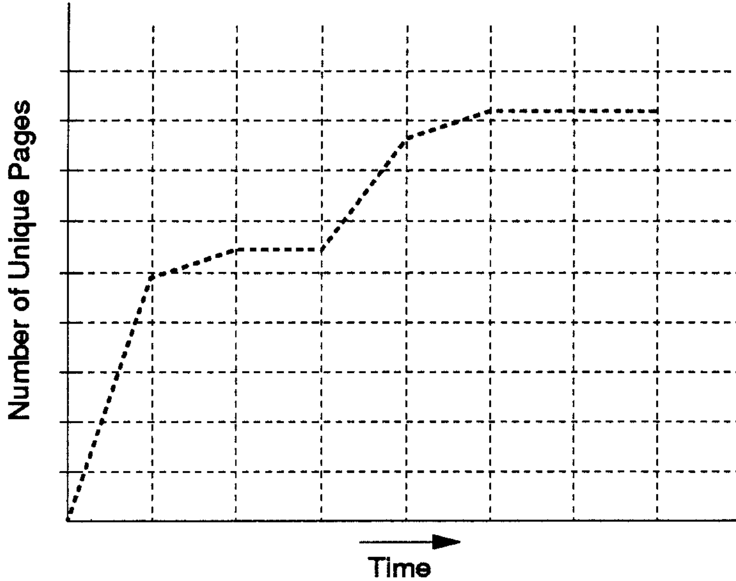

Figure 4. Cumulative Number of Unique Pages Touched

5.1 Appropriate Workload

Modelers often decide to use traces that are inappropriate for their purposes. One example of an inappropriate workload would be to use a compute-bound benchmark (e.g., matrix300 from SPEC '89 [48]) to study cache performance. The cache hit rate would be so high for any reasonably-sized cache, such that the results would be very misleading (i.e., this benchmark is compute-bound, and does not stress the memory subsystem).

To avoid this type of trace misuse, review the suggestions provided in section 4.1. Another important issue here is to learn as much about the benchmark or application that you can. Inspect source code if possible. Attempt to identify what parts of the application that you have captured in your trace (e.g., modules, functions, etc.).

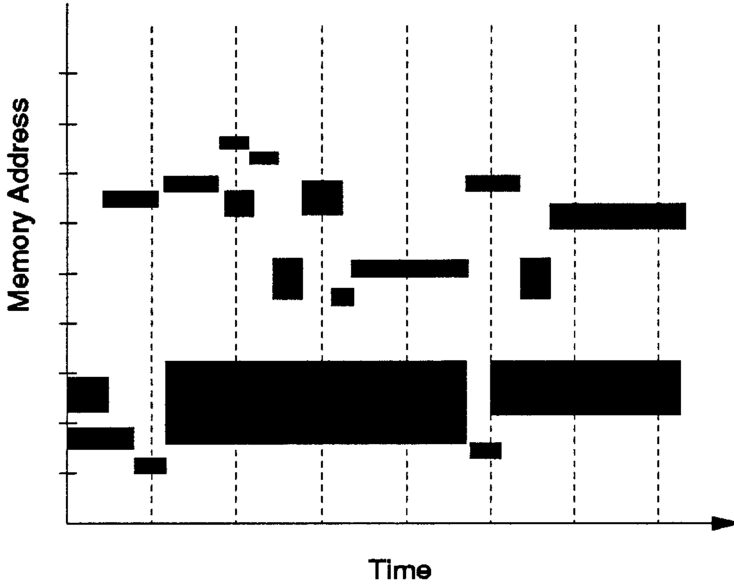

Figure 5. Address Space Traversal

5.2 Generality of Results

To be able to claim generality of a particular modeling result, a range of benchmarks or applications needs to be used. We often review papers that leave a particular application out of the results section. Sometimes this is done legitimately, but sometimes this is done in order to hide some less favorable results.

If a particular application does not perform as well as the rest of the benchmarks/applications in the set, find out why. Generality is a key quality that reviewers look for in papers. If the particular design issue has some shortcoming, as long as the reason why this occurs is understood and explained clearly, the merits of the design will still be evident.

5.3 Snapshot Selection

Besides selecting the appropriate workload to study, it is very important to understand, within the workload, what you have captured in the trace. One mistake that is often made is to begin tracing an application from the beginning of its execution. What happens is that the trace will contain a majority of the overhead of loading the code into memory, and the initialization of variables. If a majority of the execution is devoted to initialization, then capturing the startup of the execution is fine. But in most applications/benchmarks, execution is dominated by steady-state execution. Always attempt to determine the dominating behavior of the code being traced.

The next issue after deciding when to begin tracing is to decide when to stop tracing (i.e., what is the right trace length). One example of this is found when modeling caches. The miss rate in the cache will be quite high as the *working set* [34] is loaded. If the trace ends while the working set is still being loaded, unrealistic (i.e., very high) miss rates will be produced. For this reason, a study of the *cache footprint* [51] should be performed before deciding what the appropriate length of a trace should be.

5.4 Snapshot Length

Once the correct length is determined, a methodology should be selected that captures a representative sample. While there is not one methodology that can be applied to every trace, an evaluation using workload characteristics (as described in section 4.2) should be performed. One example of using the wrong trace sample is when a tight (small) timing loop is captured in a trace (timing loops are quite common in personal computer workloads). The performance evaluation will center around speeding up the timing loop. The net effect of increasing a timing loop's execution speed is to generate more iterations of the timing loop in the optimized design. Increasing the speed of timing loops does not generally improve the program execution time.

6. Conclusions

This paper has provided an overview of the issues related to the content of the traces used in trace-driven simulation. A review of the three types of traces was provided. The issues of trace generation, trace-length reductions, workload selection and representativeness, and trace misuse were covered. An extensive set of references is also provided, which should be used for further information on any particular trace-related issue.

The modeler should come away from this paper with a more critical view of input traces. Some important questions that should be raised when using traces are:

- What type of trace is required by the model?
- Is the trace complete, and does it contain any perturbations?
- What is an appropriate trace length?
- Can I reduce the size or length of my trace?
- Does the trace contain the appropriate workload?
- Does the trace contain the appropriate execution snapshot?
- Can I identify the portion of the application contained in the trace?

Increasing the emphasis on the quality of the input trace can only improve the quality of the modeling results.

References:

1. D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning Computer Systems,* Prentice Hall, 1983.
2. R. Jain, *The Art of Computer Systems Performance Analysis,* John Wiley and Sons, 1991.
3. A.J. Smith, "Cache Memories," *ACM Computing Surveys* , Vol. 14, No. 3, Sept. 1982, pp. 473-530.
4. J. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. of the 10th International Symposium on Computer Architecture,* June 1983, pp. 124-131.
5. J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computers,* Vol. 4, No. 4, Nov. 1986, pp. 273-298.
6. J. Tsai and A. Agarwal, "Analyzing Multiprocessor Cache Behavior Through Data Reference Modeling," *Proc. of Sigmetrics '93,* May 1993, pp. 236-247.
7. *i486 Microprocessor Programmer's Reference Manual,* Intel Corporation, Santa Clara, CA, 1990.
8. *Alpha Architecture Reference Manual,* DEC, Burlington, MA, 1992.
9. J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. of Sigmetrics '90,* May 1990, pp. 134-142.
10. A.J. Smith, "Disk Cache - Miss Ratio and Design Considerations, " *ACM Transactions on Computer Systems,* No. 3, Aug. 1985, pp. 161-203.
11. J. Ousterhout , H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proc. of the 10th Symposium on Operating System Principles,* December 1985, pp.35-50.
12. A.J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms," *IEEE Transactions on Software Engineering,* Vol. SE-7, No. 4, July 1981, pp. 403-417.

13. S.S. Lavenberg, *Computer Performance Modeling Handbook,* Academic Press, New York, N.Y., 1983.
14. S.J. Eggers, D.R. Keppel, E.J. Koldinger, and H.M. Levy, " Techniques For Efficient Inline Tracing on a Shared-memory Multiprocessor," *Proc. of Sigmetrics '90,* May 1990, pp. 37-46.
15. C. Stephens, B. Cogswell, J. Heinlein, and G. Palmer, " Instruction Level Profiling and Evaluation of the IBM RS/6000," *Proc. of the 18th International Symposium on Computer Architecture,* May 1990, pp. 180-189.
16. A Borg., R. Kessler, and D.E. Wall, "Generation and Analysis of Very "Generation and Analysis of Very Long Address Traces," *Proc. of the 17th International Symposium on Computer Architecture,* May 1990, pp. 270-279.
17. E.J. Koldinger, S.J. Eggers, and H.M. Levy, "On the Validity of Trace-Driven Simulation for Multiprocessors," *Proc. of the 18th International Symposium on Computer Architecture,* May 1991, pp. 244-253.
18. C.B. Stunkel and W.K. Fuchs, "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation," *Proc. of Sigmetrics '89,* May 1989, pp. 70-78.
19. D.W. Wall, "Experience with a Software-Defined Machine Architecture," *ACM Transactions on Programming Languages and System,* Vol. 14, No. 3, July 1992, pp. 299-338.
20. A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming,* Kluwer Academic Pub., Norwell, Mass., 1989.
21. A. Agarwal, R.L. Sites, and M. Horowitz, "ATUM: A Technique for Capturing Address Traces," *Proc. of the 17th International Symposium on Computer Architecture,* , May 1986, pp. 119-127.
22. J.K. Flanagan, B. Nelson, J. Archibald, and K. Drimsrud, "BACH: BYU Address Collection Hardware; The Collection of Complete Traces," *Proc. of the 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation,* Sept. 1992.
23. O.R. LaMaire and W.W. White, "The Contribution to Performance of Instruction Set Usage in System/370," *Proc. of the Fall Joint Computer Conference,* Dallas, TX., Nov. 1986, pp. 665-674.
24. H. Davis, S.R. Goldschmidt, and J. Hennessy, "Tango: A Multiprocessor Simulation and Tracing System, " *Proc. of International Conference on Parallel Processing,* Aug. 1991, pp. 99-107.
25. *Intel 80386 Programmer's Reference Manual,* Intel Corporation, Santa Clara, CA, 1986.
26. *VAX-11 Architecture Reference Manual,* Digital Equipment Corporation, Bedford, MA, 1982, Form EK-VARAR-RM-001.
27. D.W. Clark, "Cache Performance in the VAX-11/780," *ACM Transactions on Computer Systems,* Vol. 1, Feb. 1983, pp. 24-37.

28. D.R. Kaeli, O.R. LaMaire, P.P. Hennet, W.W. White, W. Starke, "Real-Time Trace Generation," submitted to the *International Journal of Computer Simulation,* July 1993.
29. T. Horikawa, "TOPAZ: Hardware-Tracer Based Computer Performance Measurement and Evaluation System," *NEC Research and Development* Vol. 33, No. 4, Oct. 1992, pp. 638-647.
30. *MIPS Languages and Programmer's Manual,* MIPS Computer Systems, Inc., 1986.
31. P.P. Hennet, O.R. LaMaire, P.J. Manning, and W.J. Starke, "Self-Clocking SRAM Sequential Memory System," *IBM Technical Disclosure Bulletin,* Vol. 32, No. 2, July 1991, pp. 40-42.
32. A.D. Samples, "Mache: No-Loss Trace Compaction," *Proc. of Sigmetrics '89*, May 1989, pp. 89-97.
33. J. Ziv, and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory,* Vol. 23, 1976, pp. 75-81.
34. P.J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM,* 11(5), May 1968, pp. 323-333.
35. A.J. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 1, January 1977, pp. 94-101.
36. T.R. Puzak, "Analysis of Cache Replacement-Algorithms," Doctoral Dissertation, Univ. of Massachusetts, Amherst, Mass., February 1985.
37. W.H. Wang, J.L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis," *Proc. of Sigmetrics '90,* May 1990, pp. 27-36
38. A. Agarwal and M. Huffman, "Blocking: Exploiting Spatial Locality for Trace Compaction," *Proc. of Sigmetrics '90,* May 1990, pp. 48-57.
39. J. Chame, and M. Dubois, "Cache Inclusion and Processor Sampling in Multiprocessor Simulations," *Proc. of Sigmetrics '93,* May 1993, pp. 36-47.
40. K.M. Dixit, "CINT92 and CFP92 Benchmark Descriptions," *SPEC Newsletter,* 3(4), Dec. 1991.
41. S.K. Dronamraju. S. Balan, and T. Morgan, "System Analysis and Comparison Using SPEC SDM 1," *SPEC Newsletter,* 3(4), Dec. 1991.
42. J. Gray, *The Benchmark Handbook,* Morgan Kaufmann Pub., San Mateo, CA., 1993.
43. J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Technical Report CSL-TR-91-469,* Stanford University, April 1991.
44. L.A. Barroso, and M. Dubois, "The Performance of Cache-Coherent Ring-based Multiprocessors," *Proc. of the 20th International Symposium on Computer Architecture,* May 1993, pp. 268-277.

45. A.L. Cox, and R.J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proc. of the 20th International Symposium on Computer Architecture,* May 1993, pp. 98-108.
46. G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer Performance Evaluation and the Perfect Benchmarks," *CSRD Report No. 965,* Univ. of Illinois, March 1990.
47. S. Vajapenyam, G.S. Sohi, an W.-C. Hsu, "An Empirical Study of the CRAY Y-MP Processor using the PERFECT Club Benchmarks," *Proc. of the 18th International Symposium on Computer Architecture,* May 1991, pp. 170-179.
48. M. Martonosi, and A. Gupta, "Effectiveness of Trace Sampling for Performance Debugging Tools, " *Proc. of Sigmetrics '93,* May 1993, pp. 248-259.
49. S. Laha, J.H. Patel, and R.K. Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Transactions on Computers,* Vol. 37, No. 11, Nov. 1988, pp. 1325-1336.
50. *SPEC Benchmark Suite, Release 1, Supercomputing Review,* 3(9), Sept. 1990, pp. 48-57.
51. H.S. Stone, and D. Thiebaut, "Footprints in the Cache," *Proceedings of Performance '86,* May 1986, pp. 1-4.