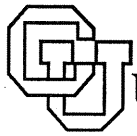
Execution Architecture Independent Program Tracing

Dirk Grunwald Gary Nutt Anthony Sloane
David Wagner William Waite Benjamin Zorn

CU-CS-525-91 April 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Execution Architecture Independent Program Tracing

Dirk Grunwald Gary Nutt Anthony Sloane
David Wagner William Waite Benjamin Zorn

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-525-91 April 1991

University of Colorado at Boulder

Technical Report CU-CS-525-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Execution Architecture Independent Program Tracing*

Dirk Grunwald Gary Nutt Anthony Sloane
David Wagner William Waite Benjamin Zorn

Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

April 1991

Abstract

Due to dramatic increases in microprocessor performance, *medium-grain ensemble multiprocessors* have become an economical hardware platform on which to solve compute-intensive problems. Unfortunately, the use of these systems to solve such problems is hampered by a lack of understanding about the behavior of parallel programs at all levels of execution — hardware, operating system, and runtime system. The goal of the Parallel Execution Evaluation Testbed project at the University of Colorado is to improve the general understanding about the performance of parallel programs and systems at these levels using trace-driven simulation.

In this paper, we discuss the validity of trace-driven simulation of parallel programs, the difficulties of applying this approach to evaluating parallel programs, and a new technique to abstract the *logical* behavior of the program and capture it in the traces we collect. We describe how this abstract trace information can be used to understand the behavior of parallel systems.

*This work was supported in part by NSF Cooperative Agreement No. DCR-8420944, NSF Grant No. CCR-9010624, and NSF Grant No. CCR-9010672.

1 Introduction

The increase in performance of microprocessors in recent years has far outpaced that of supercomputers, and this trend shows no signs of abating (prompting cries of the "attack of the killer micros"). Recently announced microprocessors, such as the MIPS R4000, IBM RIOS, Intel i860, and National Semiconductor Swordfish have claimed performance in the range of 20-100 MIPS. More importantly, many of these microprocessors represent that latest in a line of architectural families. Thus, computers that take advantage of these new micros can be designed very quickly, and manufactured very economically.

Some manufacturers, such as BBN, Intel and Alliant have introduced medium-scale multiprocessors based on these "killer micros". The performance of these architectures scales in two dimensions: by adding processors, or by increasing the speed of each processor. This has resulted in multiprocessor systems rivaling the computational performance of more expensive supercomputers.

In spite of this rapidly emerging hardware technology, there are several remaining obstacles to the effective use of multiprocessors. There is inadequate architectural and operating systems support for parallel programming constructs; the tools to measure the efficacy of extant architectural and operating systems constructs are not up to the task. In today's systems it is difficult to determine what mechanisms, (such as synchronization hardware, scheduling policies or multiple processor contexts), contribute most significantly to system performance. This is due to a lack of data concerning parallel program behavior at the hardware, operating system, and runtime system levels; furthermore, we know of no inexpensive, effective way to compare design alternatives.

We call the combination of machine architecture, operating system, and runtime system the *execution architecture* of a system. A commonly-used technique for evaluating alternative execution architectures is *trace-driven simulation*, or TDS [14]. Traditionally, address traces have been used to evaluate cache organizations [15, 2, 7, 17, 5], page reference traces have been used to evaluate paging policies [3, 12], and file system-call traces have been used to evaluate file system designs [13].

Increasingly, TDS has been applied to parallel systems, using traces of parallel programs. Despite questions about the validity and usefulness of parallel traces (see Section 2.2), there are few tools for quantitatively comparing architectures. Traces of parallel programs have been used

effectively to compare distributed cache algorithms [1] and to investigate cache performance on distributed computers [16].

The complications arise from the fact that tracing systems typically provide *per-processor* traces, and thus the number of simulated processors must be equal to the number on which the traced program was run. As a consequence, one cannot easily use the traces to compare the effect of varying numbers of processors. Methods for simulating arbitrary parallelism exist for special cases [18], but there do not appear to be any generally applicable techniques.

SPAE[1] is intended to be a general purpose tracing facility for parallel systems. SPAE is used to create an instrumented parallel program that will produce an *abstract trace* of its execution. The abstraction eliminates many details of the execution architecture such as the number of processors, the synchronization mechanisms, scheduling policies, memory allocation policies and operating system behavior. When the abstract trace is used, the resulting abstractions are instantiated to a particular target execution architecture. Thus, we call such abstract traces *execution architecture independent* traces.

A program is an implementation of a software design driven by some specification. While the program's requirements are typically independent of an execution architecture, the design depends on the execution architecture to some degree. Any program implementation depends to an even larger extent on certain features on the execution architecture, e.g., the instruction set, the number of registers or the compiler technology used. Hence a trace of a program has an inherent set of assumptions about the execution architecture. Moreover, many programs are written to optimize their performance on a particular execution architecture. For example, an algorithm may be implemented very differently for shared-memory architectures than for distributed-memory architectures. Despite the prevalence of these architecture-specific algorithms, we feel that tracing parallel programs will enable quantitative system comparison within an "architectural family."

The important point is that each of these problems is an inherent limitation of program tracing itself, rather than an artifact of parallel execution, or our approach. Our objective is to make traces as independent of the execution architecture as possible, subject to these inherent limitations. Eventually, we expect to be able to identify a number of *architectural equivalence classes* within which traces can be generalized. This has already been applied in very restricted cases, such as

[1] *Spae* (spī) is a fourteenth century Scottish term, meaning to spy or foretell.

using traces from an execution architecture with one cache organization to simulate one with a different cache organization.

The remainder of this paper is organized as follows: we discuss the TDS technique in general, including its uses, the gathering of traces, and its limitations. We particularly examine the shortcomings of current trace gathering technology when applied to a parallel system. We then describe our trace gathering technique, its advantages over current approaches, and some implementation details. A simplified example illustrates the technique, and is followed by a more complex example illustrating novel aspects of SPAE.

2 Trace-Driven Simulation

This section discusses the motivation for evaluation techniques based on trace-driven simulation and addresses some of the problems associated with this approach when it is used to evaluate parallel systems.

2.1 Motivation

Solving a problem on a parallel computer involves several design stages:

Specification: A problem is identified and the designer begins to formulate a solution. The solution is largely independent of any particular execution architecture (except at the most gross levels, e.g., shared vs. distributed memory environments); the approach focuses on algorithms and general constraints.

Design: During design, the algorithm is molded to fit some more specific execution architecture constraints (e.g., synchronous vs. asynchronous execution, static vs. dynamic parallelism, etc), but has not yet been committed to a particular implementation of the algorithm.

Implementation: A programming language is selected to implement the algorithm, resulting in a concrete program for a concrete execution architecture. Ideally, specifics of the execution architecture would not enter into this stage of the solution, but any concrete instance must be specific to the execution architecture.

Execution: The program is executed on a particular execution architecture, resulting in a single solution to the problem.

Evaluation and Tuning: The program or algorithm is modified in response to observed behavior of the program on the execution architecture to obtain a more efficient problem solution on subsequent executions *on that execution architecture.*

When a parallel program exhibits unacceptable performance, the blame may lie with its specification, design, implementation, or the execution architecture. One might choose to achieve a better match of specification to execution architecture by changing the design or implementation (program tuning), or by changing the execution architecture (system tuning).

The *Parallel Execution Evaluation Testbed (PEET)* project is intended to support the change of execution architectures — hardware, operating systems, and runtime systems – so that they are well-matched to a body of well-behaved parallel application programs.

Because of the relative success of TDS in cache studies, PEET uses the technique as a fundamental methodology. Performance estimates of new architectures are determined by tracing the behavior of a set of programs on the original execution architecture and using the traces to drive simulations of new execution architectures. We prefer simulation in general, and TDS in particular, as the paradigm for evaluating the impact of changing an execution architecture in PEET.

Other approaches to evaluating performance exist, but have drawbacks. Analytic models simplify many details of an execution architecture. Many phenomena that are crucial to the performance of parallel programs, such as cache behavior and process synchronization, are very difficult to model analytically. Furthermore, simulation is more cost effective than prototyping when exploring a design space that contains a wide variety of execution architectures. Synthetic traces, or statistical profiles of program behavior, are often used due to the complexity of gathering actual traces. However, a detailed simulation is only as accurate as its inputs; thus, using actual traces increases the credibility of simulation results.

The goal of the PEET SPAE tool is to provide abstract trace data of parallel programs that rival the accuracy of actual traces.

2.2 Issues in Trace Driven Simulation

While trace-driven simulation appears straightforward, the sheer volume of data may make it a daunting undertaking (depending on the level at which the traces are taken and the length of the program execution). Some issues that arise are:

System Name	Instrumentation Tool	Dilation Factor
ATUM [2]	Microcode	20
TRAPEDS [16]	Instrumented executable code	10–30
MPtrace [6]	Instrumented executable code	2–3
Titan Trace [4]	Compiler, kernel	8–12
AE [9]	Compiler	1–4

Table 1: Current trace collection systems.

What to trace? The trace might contain a record of high-level events (e.g., operating system calls), medium-level events (e.g., synchronization primitives), and/or low-level events (e.g., address references). We believe that each of these event levels are important for understanding some aspect of the execution architecture, and that *correlating* the events across levels can provide a more complete picture of the behavior of the program on the execution architecture. However, address level tracing presents the most serious challenge because the large volume of data slows program execution and makes heavy demands on storage space.

How to trace? The difficulty of gathering address traces has led to a spectrum of collection techniques; see Table 1. In the table, the *dilation factor* refers to the ratio of execution time of a traced program to that of an identical, untraced program. The tools shown have dilation factors ranging from 1 to 30, depending upon the program behavior and on the tool. Although most systems trace only user-level execution, some, such as Titan Trace, also trace the operating system. Hardware instrumentation can also be used; see [17] for a comprehensive survey.

How to manage the data? Tracing parallel production programs generates more data than can be stored on many systems. Furthermore, long traces identify program behavior that is not captured by shorter traces, as observed by Borg et al. [4]. This problem has two aspects: execution time is excessively dilated by I/O activity, and an enormous amount of secondary storage is required to store the traces. These problems are exacerbated in a parallel environment, since there are many streams of execution to trace simultaneously.

Program Name	Untraced Execution Time (seconds)	Traced Execution Time (seconds)	Raw Trace Size (Mbytes)	Compression Factor
Shallow	0.45	0.62	12	197
ARC/2D	3214.2	5484.9	109,408	22.5

Table 2: Trace Measurements of Two Scientific Programs

Gathering and effectively using large amounts of trace data suggests *on-the-fly* simulation [4] or *execution-driven* simulation [16], in which the output of a traced program is processed directly by the simulator. This implies that the measured application is executed each time a simulation is run. Some tracing systems, such as MPtrace and AE, generate *compressed* traces; unfortunately, Table 2 shows that compression is insufficient for large programs. **Shallow** is a finite difference program modeling shallow-water flow, while **ARC/2D** solves the Euler equations using an implicit finite difference method. Both programs were run on a Solbourne Series 5 system, and generate $\approx$ 20 megabytes of raw trace information each second, or $\approx$ 5 million events per second. Although, trace compression techniques reduce the storage for the **ARC/2D** trace 23 fold, to 4.7Gbytes, it is still impractical to store the trace data. This single trace contains over 19 billion instruction references and 8 billion data references. Thus, on-the-fly simulation is currently the only *practical* solution to tracing such large programs. We feel that it is reasonable to trade the extra processor time of reexecuting the application in exchange for the savings of large amounts of secondary storage; however, a small execution dilation is critical, because it drastically affects the trace generation time.

2.3 Tracing Multiprocessor Programs

A number of new issues arise when using TDS techniques to evaluate parallel programs and architectures. First, parallel programs are more sensitive to changes in execution architecture than their sequential counterparts. Second, most parallel program execution architectures are nondeterministic to some extent. While there is widespread agreement that these properties are characteristic of parallel programming environments, there is some controversy over whether or not they present true obstacles to the use of TDS.

Koldinger *et al.* [8] showed that, unlike sequential programs, most parallel programs will yield different process or task instruction interleavings each time they are executed. This interleaving is likely to be dependent not only on characteristics of the execution architecture, but also on subtle timing perturbations introduced by the tracing software [11, 10], other users on the system, page faults, and so on. The effect on the overall execution of the program depends on its structure. Traces of statically scheduled parallel programs can be affected in subtle ways, such as the time to rendezvous at a barrier. Dynamically scheduled programs can be affected more dramatically. Timing variations can result in different assignments of threads to processors, affecting cache behavior, lock acquisition, bus traffic and so on. Thus, executing a parallel program on a parallel execution architecture yields one of many possible instruction interleavings. A consequence of nondeterminism is that no single trace is completely representative of a program's behavior; one can gain confidence in observation by tracing a program many times, then computing an "average behavior" from the traces.

Certainly, if a program trace is not repeatable on the *same* execution architecture, there is little reason to rely on it to predict performance on *different* execution architectures! This leads to a related issue about the validity of using such traces to model new execution architectures: since the trace contains information about the tracing execution architecture, how will it affect the results of the simulation? For example, is it possible to model different cache behavior using parallel traces if the original cache misses and bus contention affected the original program execution and the resultant trace?

A more subtle consequence of nondeterminism is that tracing may make it impossible to observe *representative* program behavior. Tracing dilation slows the execution of a program; some tracing techniques have a larger dilation than others, but none are entirely free from it. Since dilation is not perfectly uniform across all processors, it perturbs the program instruction interleaving. Just as minor events such as cache misses can perturb program behavior, this tracing-induced perturbation can affect the program dynamics. For example, suppose tracing slows one thread of execution enough that it loses a race for a shared lock. This can have a significant effect on the remaining execution. If such a scenario occurs frequently in traced execution, but only rarely in untraced execution, the traced observations of the program will not be representative of the true average behavior of the program. Thus, taking a large number of observations and averaging the

results is not guaranteed to produce performance measures that are indicative of the real execution architecture; confidence intervals for a performance measure may not contain its true mean.

Furthermore, dilation may mask effects that occur on a fine time scale. It is unlikely that one could capture the effect of a particular bus contention protocol in the execution architecture, because the dilation of the program's execution may considerably reduce bus contention.

It is difficult to avoid the problem of nonrepresentative program behavior measurements within a trace, or that of masking small-scale effects by dilation; using current tracing technology, there does not appear to be any way to prevent these effects. However, traces are an important tool in system design, and more extensive and flexible trace collection tools are needed.[2] The best that can be done is to validate the traced data against untraced program execution.

Even if we were confident that a traced program exhibited representative behavior "on the average," the necessity for multiple experiments increases the computation and storage requirements of the methodology. The requisite secondary storage is significant — prohibitively so to most researchers — and thwarts the distribution of parallel traces. Fortunately, increases in CPU performance means that on-the-fly simulation is practical. Unfortunately, nondeterministic behavior makes it virtually impossible to provide identical input to two different on-the-fly simulations, unless they are run simultaneously.

Most tracing systems record *per-processor* traces. Thus, the number of processors that can be simulated using such a trace is the same as the number of physical processors used to produce that trace. At best, this means that the program must be traced on every number of processors that one wishes to simulate, involving substational cpu time and mass storage. In the worst case, it means that it may not be possible to simulate future execution architectures with more processors than the one used to capture the traces. There are certain exceptions to this for limited domains: for example, Fortrace [18] traces FORTRAN programs containing `doall` and `doacross` loops, simulating the parallel execution on an arbitrary number of processors.

In the next section we present a tracing mechanism that provides trace data of parallel programs rivaling the accuracy of traditional traces, while attacking the problems mentioned here. It remains to be determined if our method suffers to some extent from the identified problems; however, we know of no *practical* method that does not.

[2] Prompting one to recall the joke, "The food here is terrible, and the portions are too small."

3 Execution Architecture Independent Traces

In this section we describe *execution architecture independent (EAI) tracing*, designed to address the problems with tracing multiprocessor programs.

Conceptually, an EAI trace is an address-level trace of the execution of a program on an unbounded number of processors, with a separate trace per thread. Since there are conceptually as many processors available as are required by the program, there is no scheduling information in the trace other than precedence constraints between the threads. Furthermore, the traces contain no absolute timing information. Finally, selected events, such as synchronization primitives (e.g., fork, join, lock, or barrier), operating system calls, and runtime system operations (e.g., memory allocation and garbage collection), are represented in the traces as *abstract events* rather than as sequences of lower-level events (e.g., address references).

The lack of embedded scheduling information is a key advantage to an EAI trace. The simulation is responsible for determining the nature of the interleaving among thread traces using a strategy determined by the effective architecture chosen by the simulator (rather than by the execution architecture on which the traces were taken). The simulator also assigns threads to the processors present in the simulated execution architecture, permitting the simulator to model execution architectures with more or fewer processors than the tracing architecture. In fact, we generate EAI traces on a uniprocessor, greatly simplifying the data gathering and increasing portability of the system. Moreover, EAI traces allow simulation of scheduling policies other than the policy used in the system being traced.

Abstract events permit the simulation to explicitly represent events in a way that is representative of the target execution architecture, rather than the tracing architecture. Thus, one can model different methods for obtaining a lock or reaching a barrier, different virtual memory or garbage collection algorithms, and even different operating system organizations. In particular, traces with abstract events enable one to consider large-scale reorganization of the division of function between operating and run-time systems.

In addition to the advantages of abstracting details of the trace-capturing execution architecture, EAI tracing solves, or at least mitigates, the problem of perturbation of program execution. The simulation program becomes responsible for representing the execution architecture's implementation of these events, in particular, interleaving of thread execution. On the other hand, this is a

powerful tool that threatens to complicate the construction of simulations, since they must now represent more detail in the effective architecture. This complexity can be moderated by three techniques:

- Data from very low-dilation instrumentation (i.e., a record of thread scheduling order) can be used by the simulation to guide the fully instrumented execution.
- The analyst can use "expert intuition" to decide to skew the distribution of executions one way or another. Thus, if the analyst discovers that a particular, but unlikely, instruction interleaving or thread schedule leading to superior performance, this information may produce a more efficient design.
- The simulation can be written to ensure a random selection of possible execution traces are measured. This random selection may not have the same distribution as the execution of the actual program, but it might reduce the probability of the measurements being overwhelmingly skewed by some artifact of the instrumentation.

Obfuscation of small-scale events is not a concern, because the traces are recorded on a virtual time scale completely unrelated to real time; effects at any scale can be explicitly simulated. Unimportant events can be intentionally excluded; contrast this with current measurement technology, where these effects are always present, even those not representative of the execution architecture being simulated. We also intend to provide library routines that simulate detailed behaviors of common execution architectures, such as particular synchronization methods and scheduling policies. These routines can be used as common facilities within the simulation application program.

Finally, although the EAI software can generate events at a rate of approximately 5 million per second, program execution speed is not critical to ensuring trace fidelity. Thus, EAI tracing allows efficient on-the-fly simulation, reducing I/O and storage costs and expanding the range of programs that can be traced.

3.1 AE

Our implementation of EAI tracing is based on the AE (Abstract Execution) tool [9]. Figure 1 is a schematic outline of the stages of tracing a *measured application*, denoted "foo.c." AE provides

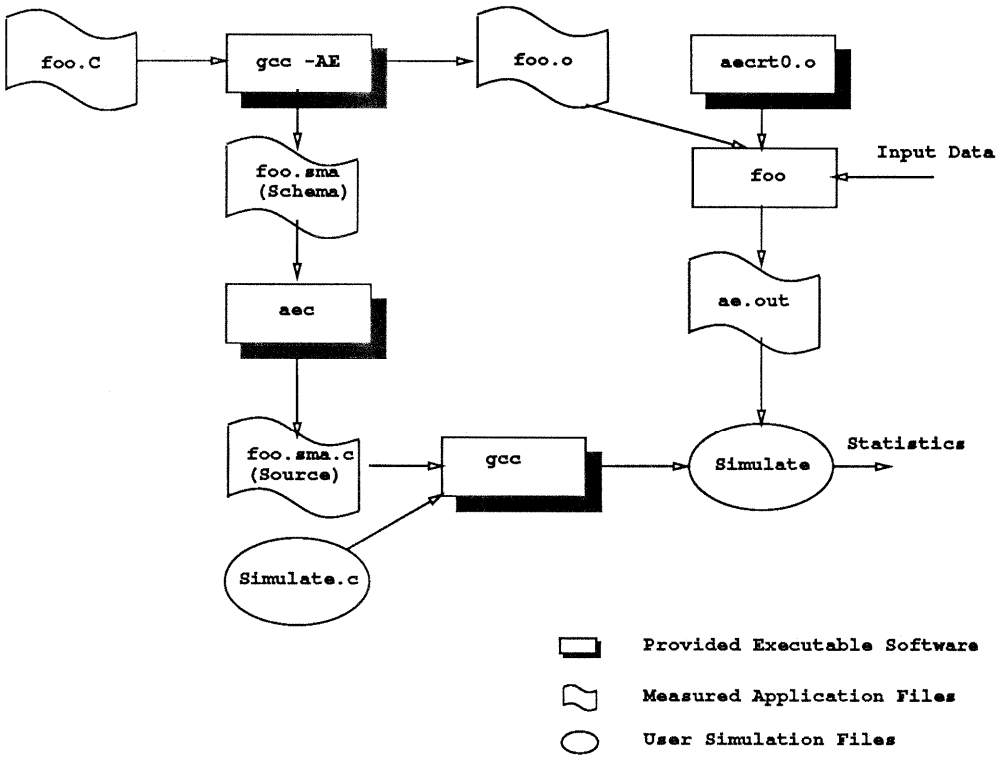


Figure 1: Information Flow In AE [9]

a modified C compiler and associated runtime library. The measured application is compiled using this modified C compiler, producing a static *schema file* and an *object file*. The schema file is an abstract version of the assembly code of each basic block in the measured application, while the object file is the executable file resulting from the compilation. The object file is linked with various libraries producing an *instrumented application program*. This instrumented application performs the same function as the original application. However, it also generates *dynamic trace information* in the file `ae.out`, used to reconstruct the program trace. This dynamic information is generated each time the instrumented application is run; the data can also be sent directly to another program.

The schema file contains information used to reconstruct instruction and data references. This static information reduces the execution time of the instrumented application and the total volume of dynamic data. The schema file is compiled into a C program, the schema driver, by the `aec` utility. The schema driver, when linked with the simulator, reads data from the dynamic trace

#	Schema File
1	`line_number 120 "shallow.c"`
2	`uneventful_inst 2 2`
3	`compute_defn_2 R8 #I32 + #S_cons_`
4	`load_inst #R8 + #I0 %struct_ref`
5	`load_inst #R8 + #I4 %struct_ref`

Figure 2: Sample AE Schema for statement"`mnmin = min(cons_1.m,cons_1.n);`"

information file (`ae.out`), reconstructs the full reference stream, and passes individual references to the simulator.

Consider the schema file excerpt shown in Figure 2. Static information, such as the current line number, can be generated by knowing the current basic block of the application, without cluttering the dynamic trace file with such data. *Uneventful instructions*, i.e., instructions that do not reference memory or modify registers used for addressing, are condensed into a single event in the static schema file; no information need be sent from the executing application to the schema driver. Knowing the number of executed instructions means the program counter can be directly updated and instruction references generated for those instructions.

Many memory references are relative to registers and compile-time constants. When an AE trace is reconstructed, the contents of registers are also tracked by the schema driver. The AE schema file records the computations of register values. Line three in Figure 2 illustrates one such computation, setting register eight to 32 plus the constant address of the variable `_cons_`. Lines four and five use register eight and constant offsets to generate data load addresses. Note that, with no dynamic information from the executing program, the schema sample shown here encodes five instruction references and two data references.

Not all addresses can be encoded this way; some (e.g., arbitrary memory references), require information from the executing program. In this case, the executing program writes data to the file `ae.out`. The data in `ae.out` also encodes the *dynamic behavior* of the program, such as the branches taken and subroutines called. Each execution of the instrumented application results in a new `ae.out` file; this information must be processed by the schema driver to reconstruct a complete trace of the program. By tracking the value of registers and knowing the addresses of global variables, AE greatly reduces the amount of data emitted by the instrumented application,

reducing the storage needed for a trace *and* the execution dilation of the instrumented program. This greatly increases the quantity and variety of events that can be traced.

This mechanism is not unique to AE; both MPTrace and TRAPEDS use a similar technique. However, AE uses dataflow information produced by the compiler to limit the amount of dynamic information needed; furthermore, AE has been ported to a variety of processor architectures.

3.2 SPAE: Symbolic Parallel AE

We have modified AE to provide *symbolic parallel traces*, that is, parallel program traces in which certain architectural constructs are symbolized by abstract events. We first describe the problems germane to all variants of SPAE, then show an example for simple `doall` programs, and follow with extensions for thread-based applications.

The first modification provides multiple *program contexts* within the `ae.out` file. In AE, the activity of a single program context is traced. The data in `ae.out` is not tagged in any way, and can only be interpreted by the correct schema routine. In SPAE, we interleave the execution of each program context, requiring the data in `ae.out` to be associated with its context.

When data is delivered to the simulator it is tagged with its context identifier; logically each context has its own `ae.out` file. A full set of information specific to a context, such as the contents of registers and the program counter, is maintained for each active context. The SPAE analog of `aec`, `spaec`, compiles the schema file to a set of C "decoding functions," one for each basic block in the measured application. When a thread in the application enters a basic block, the appropriate decoding function reads data specific to that context from `ae.out`, returning the events generated by that basic block. These functions reconstruct data and instruction references for an arbitrary context, provided we have enough dynamic information (i.e., data from `ae.out`). Moreover, we can reconstruct references for *multiple* contexts, simulating an interleaving of the contexts active in the measured application; this will be explained in more detail shortly.

Contexts can represent independent iterations of a `doall` loop or arbitrary parallel threads. We must also record any interaction between contexts. In particular, we are interested in providing abstract events for specific interactions. For example, assume that barriers are implemented using hardware-assisted locks in a certain computer. If we were interested in simulating the behavior of that specific computer, we could simply record the interaction of contexts with the lock primitives.

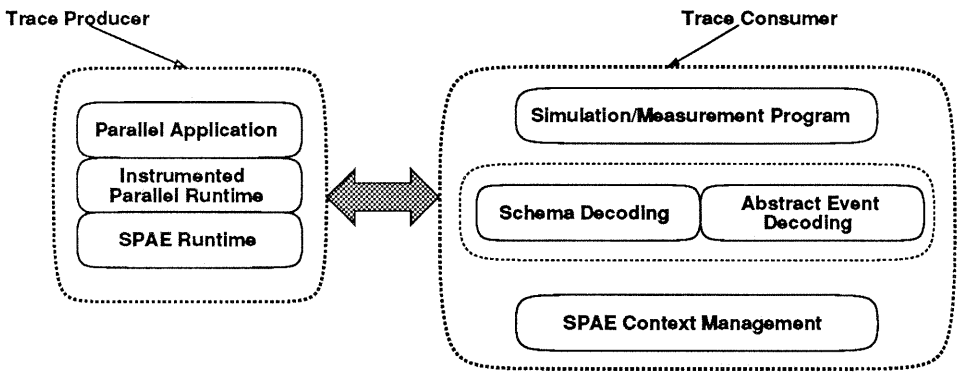


Figure 3: Interaction of SPAE components

However, if we wanted to simulate alternative implementations of barriers, we would record a single abstract event indicating a barrier join rather than a sequence of load, store, and hardware locking operations. This abstract event could be expanded by the simulator to represent the behavior of an existing architecture, or it can emulate novel architectural approaches.

There are other events, such as spinlock acquisition or context creation and destruction, that are best treated as abstract events. In SPAE, such events can be explicitly entered in the the static (`foo.sma`) and dynamic (`ae.out`) trace description. Figure 3 illustrates the interaction of different SPAE components. We modify the parallel tasking library used by an application to create SPAE contexts and inject specific events into the dynamic event stream (`ae.out`). The SPAE data transport interacts with the simulation to provide correct dynamic information for each context. A decoding (event interpreting) interface, specific to the parallel tasking library used by the application, reconstructs the abstract events of individual contexts in the measured application. These decoded events are used by the simulation program to drive a specific simulation. The figure also indicates information flow from the trace consumer back to the trace producer; this flow is explained in Section 3.4.

3.3 SPAE: Constrained Synchronization

This section uses a simple parallel tasking library to describe how SPAE is used. The library traces programs using `doall` and `doacross` loops, providing a "simulated parallelism" similar to Fortrace [18].

Recall that SPAE traces parallel applications on uniprocessor computers. The semantics of `doall` and `doacross` are satisfied by sequential execution. Consider the following `doall` loop:

```
DOALL 10 I=1,4
    A[I] = I
10 CONTINUE
```

At a gross level, four instructions are issued:

Time	Instruction
1	A[1] = 1
2	A[2] = 2
3	A[3] = 3
4	A[4] = 4

Simulating parallel `doall` on P processors involves storing the references of P iterations, then interleaving those references at simulation time. For example, the following is a possible interleaving for two processors:

Relative Time	Processor 0	Processor 1
0	A[1] = 1	A[2] = 2
1	A[3] = 3	A[4] = 4

There are a multitude of possible interleavings of the iterations corresponding to the scheduling policy, the underlying simulated execution architecture and so on. The program simulating a parallel computer architecture must be able to select the interleaving applicable to the simulated hardware or scheduling policy. Each `doall` iteration is a context, or distinct parallel entity, in SPAE. The simulation program can map different contexts to specific simulated processors. In this example, contexts one and three were assigned to simulated processor zero, while the remainder were assigned to processor one.

When simulating the parallel execution of a `doall`, the *order* of the instructions issued on a processor is specified because it is intrinsic to the execution of the program. However, the *total time order* of the executed instructions is not known, because it depends on memory latency or contention for resources in the simulated parallel architecture.

This example simplifies several issues, but it illustrates the important points of tracing `doall` programs: sequential execution satisfies the `doall` semantics; each iteration is a distinct *context*, allowing the simulator to arbitrarily order contexts; and instruction issue within a context is intrinsic to the program, but ordering between contexts is controlled by the simulation program.

```
C$DOALL
       DO P=1,S
         M[P] = 0.0
         DO I = P*N,(P+1)*N
           IF ( M[p] < A[I] ) M[P] = A[I]
         END DO
        END DO
```

Figure 4: Sample `doall` Code

```
ae_special_event (FORK_DOALL, 2);
for (p = 0; p < S; p++) {
  int i;
  ae_special_event (START_DOALL_ITER, p);
  M[p] = 0;
  for (i = p * N; i < (p + 1) * N; i++)
    if (M[p] < A[i]) M[p] = A[i];
    ae_special_event (FINISH_DOALL_ITER, p);
}
ae_special_event (JOIN_DOALL, 0);
```

Figure 5: Converted `DOALL` Code

We now expand the example to include the actual program instrumentation needed by SPAE. (This example was extracted from programs used to measure the efficacy of hierarchical distributed cache algorithms for scientific applications.) The original FORTRAN program locates the maximum element of an array. The program in Figure 4 has `NSECTIONS` independent iterations, and has been mapped to a two-processor system.

We convert the code fragment to the C program, shown in Figure 5, using a FORTRAN to C translator, and hand-instrument it to indicate the `doall` loop. This corresponds to the instrumented parallel library depicted in Figure 3; although we could automate this process, we have not found that it hampers our work.

The subroutine calls to `ae_special_event` are recognized by the modified C compiler (no actual subroutine call is performed). The first argument (e.g., `FORK_DOALL`) is inserted into the static schema file and code is generated to inject successive arguments into the dynamic trace file (ae.out). This information is used when decoding the events specific to the parallel library, i.e.,

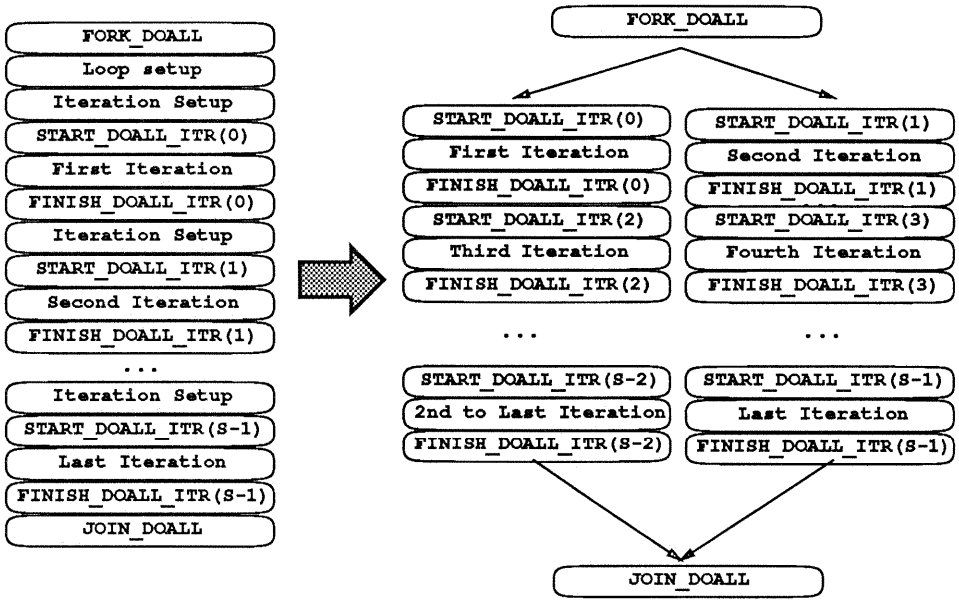


Figure 6: Schematic Mapping of Trace

the `doall` loops in this example. The raw trace contains unwanted information, including the outer loop setup, increment and termination. Figure 6 shows a schematic view of this information mapped to two processors, labeling each set of events with the applicable context.

The dynamic data for contexts one and two are buffered by the SPAE context management interface. Extraneous events, such as those following the `FINISH_DOALL_ITER` are removed from the trace by the library-specific interface. Similarly, this interface informs the simulator of the `doall` loop and the implicit barrier synchronization. The cache simulator translates the `START` and `FINISH` events into more elaborate scheduling activity; likewise, the barrier synchronization is translated into activity specific to the simulated architecture.

Figure 7 shows an event interleaving for the first two `doall` contexts. Each line encodes events for a single CPU. For example, CPU0 executed an abstract event (DIS) at address 0x22e8, read a data value from address 0xfffffd4 and then executed four more instructions. Initially, both contexts behave identically, but diverge because of data dependent behavior. SPAE has been implemented on the Sun SPARC and MIPS R3000 processors; this example used a SPARC processor.

We can also generate similar traces for `doacross` loops; there, additional events corresponding to POST and WAIT synchronization events are added to the event stream. Furthermore, we can rearrange memory addresses, performing a virtual-memory to virtual-memory remapping; this is

```
cpu 0:  DIS: #0, 0x22e8, start iter 0
cpu 1:  DIS: #0, 0x22e8, start iter 1
cpu 0:  I x 6:  22e8
cpu 1:  I x 6:  22e8
cpu 0:  R: ffffffd4
cpu 1:  R: ffffffd4
cpu 0:  I x 1:  2300
     ....
cpu 1:  I x 2:  2398
cpu 0:  R: ffffffcc
cpu 1:  I x 1:  23a0
cpu 0:  I x 2:  23dc
cpu 1:  I x 1:  23a4
cpu 0:  W: ffffffcc
cpu 1:  R: ffffffcc
```

Figure 7: An Event Interleaving for Sample Code

used in the distributed cache simulator to examine the effect of array layout on distributed cache behavior.

Our modifications slow the instrumented application by less than 1% over a regular AE trace; this in turn is about one to four times slower than uninstrumented application execution time. Our initial measurements have shown that for large traces (e.g., of ARC2D) it is faster to simply rerun the application than to read the trace from disk. Currently, SPAE slows event decoding two-fold; we expect to reduce this overhead, but the time spent decoding events is typically dwarfed by the simulation overhead, so optimization will have little overall effect.

SPAE is unique in that it provides both data *and* instruction traces while maintaining high efficiency. Futhermore, the mechanisms used in this simple example allow us to trace programs that use *thread* or *task* execution models as well as programs that employ arbitrary synchronization primitives.

3.4 SPAE: Arbitrary Synchronization

Explicit, independent streams of control (threads) are a general parallel program mechanism. A thread library, such as Sun LWP or Mach C-Threads, consists of routines to create and destroy threads, acquire and release locks, enter and leave monitors, signal condition variables, join at

a barrier, etc. There are also transparent operations that schedule the threads on the physical processors. While `doall` and `doacross` programs have limited dependence relations, thread-based computations can have arbitrary precedence constraints between the individual threads. This complicates program tracing, because thread behavior is time-dependent; for example, acquiring a lock in differing orders can present different program behaviors. Although many thread libraries exist, most provide similar abstract functionality; SPAE is designed to integrate with different thread libraries. To create EAI traces of a thread-based computation, we modify the run-time thread library.

Some changes to the thread library simply insert events into the trace, using `ae_special_event` as in §3.3. Consider thread creation: when a thread is created in the traced program, the thread library calls the SPAE context management routines to create a new context for the thread and inserts an event in the `ae.out` file indicating to the simulator the existence of a new context. This is similar to the `DOALL_FORK` operation in the previous example. Likewise, thread library routines that schedule threads must inform the simulator of their activity, and must indicate the current context to the SPAE context management routines.

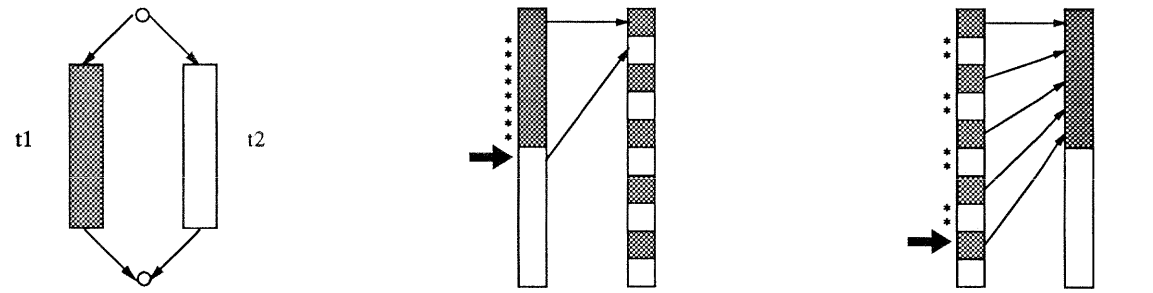
If threads did not interact, this process would be similar to the `doall` loops of §3.3; however, consider acquiring a spinlock. Spinlocks can be implemented in a variety of ways, and this is an important variable in simulations of cache and system design. We want an execution architecture independent representation of the spinlock acquisition. In other words, we do not want an attempt to acquire a spinlock to generate a multitude of low-level events; instead, a single event, indicating the attempt to acquire the lock, should be generated. The parallel library decoding interface (Figure 3) synthesizes a sequence of address references appropriate to the architecture being simulated, and based on the length of time that the *simulated* thread is forced to wait.

There are several instances, such as lock acquisition, in which the simulator must direct thread scheduling. For example, if two threads attempt to acquire a single lock, one succeeds and the other fails; the success or failure depends on the simulated execution architecture. Thus, whenever threads interact, or any scheduling decision is made, the simulator controls the scheduling. Primitives such as lock acquisition are modified to pause and wait for direction from the simulator, allowing the simulator to determine which thread obtains a lock. This enables the simulation of significantly different execution architectures. This is similar to the iteration scheduling decisions possible in

`doall` programs; however, rather than selecting the next context from the remaining iterations, the thread approach selects a context from the *thread ready queue* maintained by the thread library.

With this model (Figure 3), the simulator can select a thread from the ready queue, trace it until it reaches a scheduling event such as a lock or system call, select another thread and so on. This creates two other problems. First, not all program are *perfectly synchronous*; that is, a program may examine data without locking that data, in an effort to reduce lock contention. This produces one result on a real multiprocessor, where multiple threads execute concurrently, and quite a different result when simulating parallelism by running a single thread as described. When necessary, SPAE can circumvent this by interleaving the execution of the measured program's threads. Currently, a small amount of code is executed at the beginning of each basic block to check for sufficient buffer space for dynamic (ae.out) information in that block. We augmented this to count the number of instructions executed by the current context, and switch to another context when a threshold is reached. The expense of this *thread interleaving* depends on the processor architecture, because the complete processor context must be switched. The simulation program can specify the set of runnable contexts, but is not informed of these context switches, because the interleaving does not affect thread synchronization.

For many measured applications, thread interleaving is unnecessary and can be suppressed, speeding trace generation. However, simulator-directed thread interleaving is still useful for reducing the amount of data from the measured program that needs to be buffered by the simulator. On-the-fly simulation uses data directly from the instrumented program; however, if that data is not immediately needed, it must be buffered. Consider as an example a measured program having a fork-join structure with two parallel threads, `t1` and `t2` (Figure 8(a)). Suppose that the run-time system schedules the threads in a run-to-completion fashion, that is, with no thread interleaving, but the simulator interleaves the two threads on a single processor due to some constraint of the simulated architecture. In this case, the mapping from the trace file to the simulation will be as shown in Figure 8(b). Thus, all of the data from `t1` marked by asterisks (*) would need to be buffered before the simulator could simulate the first reference from `t2`. An alternate scenario, in which the run-time system interleaves the two threads but the simulator simulates run-to-completion scheduling, produces a similar situation, shown in Figure 8(c). In this case, nearly



(a) Precedence graph (b) Sequential trace generation driving time-sliced simulation (c) Time-sliced trace generation driving sequential simulation

Figure 8: An example of the buffering problem. Asterisks denote data that must be buffered before the block marked with the bold arrow can be simulated.

all of the references from `t2` would need to be buffered before the simulator can finish simulation of `t1`.

Using thread interleaving, we can limit the buffering requirements to at most one basic block per concurrently executing thread in the program. The SPAE runtime system is directed by the executing program to execute "interesting" threads first. With this facility, the simulator can control thread scheduling without an onerous amount of data buffering.

4 Status & Conclusions

The PEET project is driven by the desire to simulate a variety of execution architectures with detailed trace data. To achieve generality, we have developed a tool, SPAE, to generate abstract traces that are consumed by on-the-fly simulations.

We have shown that on-the-fly simulation is a practical method for simulating large, long-running programs, where gigabytes of trace data are common. Furthermore, we have found that trace data can be regenerated faster than it can be read from disk.

SPAE generates an *execution architecture independent* (EAI) trace, allowing investigation of a broader design space of execution architectures. An EAI trace decouples the trace generation architecture from the evaluated architecture, allowing simulated multiprocessor traces to be generated on a uniprocessor. EAI tracing requires intervention by the simulation program to resolve

architecture-specific events, such as lock acquisition by threads in the instrumented application. Simulator-directed tracing also reduces buffering of the trace data.

The initial version of SPAE traces `doall` and `doacross` loops as described in §3.3. Data from these traces is being used in a study of distributed cache architectures for scientific applications. The volume of data and detail of the traces would be impossible without a tool like SPAE.

The design work has been completed for tracing general thread-based applications, and the implementation is nearing completion. We have accumulated a number of parallel applications. As part of a broader endevour, we will eventually distribute SPAE, the instrumented program libraries and sample applications.

The most serious concern is validating the traces generated by SPAE; as the PEET project progresses, validation studies will be conducted, and the tools and traces will be distributed to other researchers.

5 Acknowledgments

Anthony Sloane was supported by NSF Cooperative Agreement No. DCR-8420944. Phillip Farber participated in the design, and has used the evolving environment for distributed cache simulations. Dirk Grunwald received partial support for the work under NSF Grant No. CCR-9010624. David Wagner received partial support for the work under NSF Grant No. CCR-9010672.

References

[1] AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. Cache performance of operating system and multiprocessing workloads. *ACM Transactions on Computer Systems 6*, 4 (November 1988), 393–431.

[2] AGARWAL, A., SITES, R. L., AND HOROWITZ, M. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture* (June 1986), pp. 119–127.

[3] BELADY, L. A. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal 5*, 2 (1966), 78–101.

[4] BORG, A., KESSLER, R. E., LAZANA, G., AND WALL, D. W. Long address traces from RISC machines: Generation and analysis. Tech. Rep. 89/14, Digital Equipment Corporation Western Research Laboratory, Palo Alto, CA, Sept. 1989.

[5] CHAD. L. MITCHELL AND MICHAEL. J. FLYNN. The effects of processor architecture on instruction memory traffic. *ACM Transactions on Computer Systems 8*, 3 (August 1990), 230–250.

[6] EGGERS, S., KEPPEL, D., KOLDINGER, E., AND LEVY, H. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Boulder, CO, May 1990).

[7] KATZ, R. H., EGGERS, S. J., WOOD, D. A., PERKINS, C. L., AND SHELDON, R. G. Implementing a cache consistency protocol. In *Proceedings of the Twelfth Symposium on Computer Architecture* (Boston, Massachusetts, June 1985).

[8] KOLDINGER, E., EGGERS, S., AND LEVY, H. On the validity of trace-driven simulation for multiprocessors. In *Proc. 18th Annual Symposium on Computer Architecture* (May 1991), IEEE Computer Society Press. to appear.

[9] LARUS, J. R. Abstract execution: A technique for efficiently tracing programs. Tech. Rep. 912, Computer Sciences Dept., Univ. of Wisconsin—Madison, Madison, WI, Feb. 1990.

[10] MALONY, A. D., LARSON, J. L., AND REED, D. A. Tracing application program execution on the Cray X-MP and Cray-2. In *Proceedings SuperComputing '90* (1990), IEEE.

[11] MALONY, A. D., REED, D. A., AND WIJSHOFF, H. Performance Measurement Intrusion and Perturbation Analysis. Tech. Rep. CSRD No. 923, Center for Supercomputing Research and Development, Oct. 1989.

[12] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9*, 2 (1970), 78–117.

[13] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the sprite network file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 134–154.

[14] SHERMAN, S. W. Trace driven modeling: An update. In *Proceedings of Symposium on Simulation of Computer Systems* (1976), pp. 87–91.

[15] SMITH, A. J. Cache memories. *ACM Computing Surveys 14*, 3 (September 1982), 473–530.

[16] STUNKEL, C. B., AND FUCHS, W. K. TRAPEDS: Producing traces for multicomputers via execution driven simulation. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1989), pp. 70–78.

[17] STUNKEL, C. B., JANSSENS, B., AND FUCHS, W. K. Address tracing for parallel machines. *IEEE Computer 24*, 1 (January 1991), 31–38.

[18] TOTTY, B. *ForTrace Users Manual.* Univ. of Illinois, 1304 W. Springfield, Urbana, Il, 1990.