A Parallel Execution Evaluation Testbed

Dirk Grunwald Gary J. Nutt
David Wagner Benjamin Zorn

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

CU-CS-560-91 November 1991

University of Colorado at Boulder

Technical Report CU-CS-560-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

A Parallel Execution Evaluation Testbed

Dirk Grunwald Gary J. Nutt
David Wagner Benjamin Zorn

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

November 1991

Abstract

It is difficult to design and use high-performance parallel architectures because little data is available about their performance.

The goal of the *Parallel Execution Evaluation Testbed* (PEET) project is to develop and validate a set of tools that allow for very detailed analyses of the performance of parallel programs on parallel architectures. The PEET tools can be thought of as an "architectural microscope" to study the execution of parallel programs and can be used to simulate the execution of a parallel program on a variety of execution architectures with different hardware, operating systems or runtime system structures. The data can be used directly, to understand the application, or by architectural simulators, to understand complete systems. The tools work on uniprocessor computers, simplifying the study of novel or proposed architectures. The tools are also efficient, making it possible to measure long-running applications. This work will allow architectural researchers and developers to examine a wider design space, and to have more confidence before implementing a specific design.

Building such tools is also a challenging research area. If their efficiency is sacrificed, they become expensive to use, limiting their use as a research tool. If their accuracy is questionable, they become useless. The PEET project is producing specific tools for evaluating parallel program performance and general knowledge about effective means for constructing such tools.

1 Project Description

The increase in performance of microprocessors in recent years has far outpaced that of supercomputers, and this trend shows no signs of abating: recent 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 are the 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 Intel and Alliant, have introduced medium-scale multiprocessors based on these high-performance microprocessors. The performance of these architectures scales in two dimensions: by increasing the speed of each processor, or by adding processors. This has resulted in multiprocessor systems rivaling the computational performance of traditionally more expensive supercomputers.

In spite of this trend in hardware technology, there are several obstacles to the effective use of such multiprocessors. There is inadequate architectural and operating systems support for parallel programming constructs; the tools to observe the performance of architectural and operating systems constructs are not up to the task. In today's systems it is difficult to determine which mechanisms (e.g., synchronization hardware, scheduling policies or multiple processor contexts), contribute most significantly to system performance. This is due in part to a lack of data concerning parallel program behavior at the hardware, operating system, and runtime system levels. 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. System designers commonly use *trace-driven simulation*, or TDS [32], to evaluate execution architectures. Traditionally, traces have been used to evaluate cache organizations [3, 28, 16, 33, 35], page reference traces have been used to evaluate paging policies [4, 26], and file system-call traces have been used to evaluate file system designs [29].

Traces can be used to measure applications as well as systems. For example, Zorn [37] used traces to understand the impact of cache designs on the performance of different garbage collection algorithms. Zorn and Grunwald are currently investigating the effect of memory locality and lock contention in memory allocation for parallel programs. Data from traces provides more detail on the dynamics of the interaction between the system and the application than is possible through standard program instrumentation. This information is particularly critical when designing applications for non-uniform memory architectures (NUMA), where reference locality is critical to performance.

Other approaches to evaluating performance exist, but have drawbacks. Analytic models simplify many details of an execution architecture: phenomena that may have large impacts on the performance of parallel programs (e.g., cache behavior and process synchronization) are difficult to model analytically. Prototyping can faithfully represent any level of detail, but is much less cost effective than simulation when exploring a wide design space.

Due to the complexity of gathering actual traces, synthetic traces, or statistical profiles of program behavior, are often used to drive a simulation. However, a detailed simulation is only as accurate as its inputs; thus, using actual traces increases the credibility of simulation results.

Increasingly, TDS has been used to study parallel systems; traces have been used to represent memory references in parallel programs (e.g., to compare distributed cache algorithms [2] and to investigate cache performance on distributed computers [34]). Despite the need to understand the issues surrounding parallel program design and effective execution architecture organization, there are few tools for quantitatively comparing execution architectures using parallel traces. Part of the problem can be related to questions about the validity and usefulness of parallel traces (see §2.1), and part is due to the lack of research in performance tools for such programs and systems.

Complications arise from the fact that a trace is typically specific to one particular execution of the traced program. For example, if one extracts a trace stream for each processor on which the traced program is executed, then the data are not generally useful for simulations of an execution architecture with a different number of processors.

As a more subtle example, even different executions on the same number of processors may produce markedly different traces, due to nondeterministic aspects of execution behavior exhibited by nearly all parallel programs.

The *Parallel Execution Evaluation Testbed* (PEET) is envisioned as an environment to study the change of an execution architecture — hardware, operating systems, and runtime systems —

so that it is 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.

The heart of PEET is the *Symbolic Parallel Abstract Execution* tool, SPAE,[1] which has been designed to be a general purpose tracing tool for parallel systems. It is used to create an instrumented parallel program that will produce an *abstract trace* of its execution. The abstraction eliminates selected details of the specific instance of an execution architecture (such as the number of processors, the synchronization mechanisms, scheduling policies, memory allocation policies and operating system behavior) on which the traces are gathered. 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.*

An early version of SPAE has been constructed [13], and has been used for the analysis of a shared-bus multiprocessor memory system [9]. One of us (Nutt) is using SPAE to generate predicate/transition net models of a program on a particular execution architecture so that tools can be used to observe the performance of the program on related execution architectures (here, SPAE is used to generate the model, and to specify a high-level description of the load as represented by token flow in the net).

We do not claim that execution architecture independent tracing is a panacea: any trace of a program contains an implicit set of assumptions about the execution architecture for which the program was written. While the program's specification typically is independent of the execution architecture, the design depends on the execution architecture to some degree. The programs's 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. Moreover, many programs are specifically written to optimize their performance on a particular execution architecture. For example, an algorithm may be designed for a particular cache structure to enhance performance [10, 19, 37].

The important point is that many of these problems are inherent limitations of program tracing itself, rather than an artifact of parallel execution, or of our approach. The objective of PEET/SPAE is to make traces as independent of the execution architecture as possible, subject to these inherent limitations. We are confident that we will be able to characterize a number of *architectural equivalence classes* within which traces can be generalized. This has already been applied in very restricted cases, such as using traces from an execution architecture with one cache organization to simulate one with a different cache organization.

2 Background

In this section we provide more background on the TDS technique in general (including its uses, its limitations, and experience of other researchers), we describe how execution architecture independent tracing addresses several of these shortcomings, and we include a simplified example illustrating the use of SPAE.

2.1 Trace-Driven Simulation

When a parallel program exhibits unacceptable performance, the blame may lie with its specification, design, implementation, or the execution architecture on which it runs. One might

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

System Name	Instrumentation Point	Execution Dilation	Comments
AE [21]	Compiler	1–4	sequential; compressed traces
CARA [27]	Compiler	unknown	sequential; intermediate code representation
Titan Trace [5]	Compiler, kernel	8–12	sequential; traces OS activity
ATUM [3]	Microcode	20	parallel; not portable
MPtrace [8]	Instrumented assembly code	2–3	parallel; execution-driven; compressed traces
RPPT [6]	assembly language, runtime	1.3–20	parallel; simulation-driven[a]
Tango [7]	compiler, runtime	1–18000[b]	parallel; simulation-driven; variable level of detail
TRAPEDS [34]	assembly code	10–30	parallel; execution-driven
SPAE [13]	compiler, runtime	1–4	parallel; simulation-driven; variable level of detail

[a] In a limited sense; RPPT does not support dynamic mapping of simulated execution streams to simulated processors.

[b] It is difficult to distinguish the program execution dilation from the simulation overhead in Tango using available references. The dilation factor for Tango includes the time for simulation, which is not reflected in the other values.

Table 1: Current trace collection systems.

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). Although trace-driven simulation is a powerful tool for accomplishing either of these objectives, it has traditionally been used only for system tuning.

While trace-driven simulation appears to be straightforward, the sheer volume of data involved can slow program execution by many orders of magnitude and results in excessive storage requirements. Tracing parallel programs suffers from the additional difficulty that the program may exhibit different instruction interleavings each time it is run. The difficulty of gathering traces has led to a variety of collection techniques (Table 1). In the table, *execution dilation* 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 over four orders of magnitude, depending upon the program behavior or the tool.

This has led us to consider several issues: What should be traced? How should trace data be collected? How should the data be managed? How do we address parallelism in the trace data? How should events be represented?

What events should be represented? 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). Although many instrumentation systems trace only user-level execution, some (e.g., Titan Trace) also trace the operating system activity. We believe that each of these event levels are important for understanding different aspects 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. 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 systems listed in Table 1 represent a variety of software techniques.[2] The ATUM (Address Tracing Using Microcode) system instruments the microcode of the machine on which the traces are captured; this obviously limits flexibility severely. The remaining systems all instrument the traced program itself, either by modifying the compiler (AE (Abstract Execution), CARA (Compiler-Aided Research on Architectures), Titan Trace, Tango, SPAE) or by post-processing the output of the compiler (MPtrace, RPPT (Rice Parallel Processing Testbed)). This is much more flexible than microcode modification and need be no more costly, as can be seen from the execution dilation factors in the table.

A key idea of the instrumented-executable approach is called *execution-driven simulation* (EDS). The idea behind EDS is that emulation of the majority of instructions in the traced program can be avoided by executing them directly on the architecture that is being used to generate the trace. (This is only realistic if the instruction set architectures of the tracing and target architectures are similar.) The only instructions that are explicitly emulated by the simulation code are those that are not properly represented by any instructions of the tracing architecture. The instrumentation step involves breaking the code into basic blocks[3] and calculating the expected execution time of all of the instructions in each basic block. Then, at runtime, the only action needed to track the execution of many basic blocks is to update a virtual clock.

This scheme works especially well for sequential programs. However, a number of new issues arise when trying to trace parallel programs that make this technique applicable in only very limited circumstances. These issues, and a technique for addressing them, will be discussed shortly.

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. [5]. 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.

Some systems, such as AE, CARA, and MPtrace, generate compressed traces which are later expanded using information obtained from a static dataflow analysis of the program.[4] Unfortunately, compression is often insufficient: our measurements show that some interesting applications still produce many gigabytes of data [13].

The problem of storing huge traces can be avoided by consuming traces as they are generated. This implies that the measured application is re-executed each time a simulation is run. We feel it is reasonable to trade the extra processor time of re-executing the application in exchange for the savings of large amounts of secondary storage; we recognize that a small execution dilation is critical, since it dramatically affects the trace generation time.

[2] Hardware instrumentation can also be used; see [35] for a comprehensive survey of both software and hardware techniques.

[3] A basic block is a sequence of instructions with a single entry and exit.

[4] CARA also translates the program to be traced into an intermediate representation (U-code) before simulating. A U-code trace can then be translated into a trace from one of many supported instruction set architectures, thus eliminating dependence of the trace on instruction set architecture and compiler technology.

How to deal with parallelism? A number of new issues arise when using TDS techniques to evaluate parallel programs and architectures. Unlike sequential programs, most parallel programs will yield different process or task instruction interleavings each time they are executed [17]. 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 [24, 25].

As an example of the dependence of instruction interleaving on the execution architecture, consider the effect of different implementations of synchronization operations. Since differences in the outcome of synchronization operations in dynamically scheduled programs can result in different assignments of tasks to processors over time, the end result can be dramatic variations in cache behavior, bus traffic, load balance, and so on. Quite possibly this could lead to totally erroneous conclusions about the system being simulated.

As another example, very few parallel execution architectures exhibit *sequentially-consistent* memory semantics [20, 11]. The ramification of this is that the value returned from a read of a shared variable can be affected by slight variations in program timing, which may in turn affect program dynamics. If the effects of the memory consistency model of the target architecture are not taken into account, the validity of the trace for simulating that architecture is questionable.

An obvious consequence of nondeterminism is that no single trace is completely representative of a program's behavior; one can gain confidence in observation only by tracing a program many times, then computing an "average behavior" from the traces. A more subtle consequence 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.

As a practical matter, nondeterminism also makes it essentially impossible to reproduce a particular trace of a program — an obvious problem if traces are not saved.

Thus, the probability of obtaining meaningful results from a trace-driven simulation of a parallel program could be increased substantially if certain properties of the target execution architecture could actually change the execution path of the program being traced. One can conclude that at certain points in the traced program the program must stop and wait for the outcome of a "micro-simulation" of the target architecture. This requires a tight integration between the simulator, the program instrumentation, and the runtime system.

This strategy is used by RPPT, Tango, and SPAE. We call this strategy *simulation-driven execution* (SDE) to distinguish it from execution-driven simulation, in which the simulation cannot affect the dynamic behavior of the traced program. Put another way, the difference between execution-driven simulation and simulation-driven execution is that, in the former, information flows strictly in one direction: from the traced program to the simulator. In the latter, there is a two-way flow of information. It should be noted that both RPPT and Tango are considered by their authors to be instances of execution-driven simulation; however, we believe that the distinction described here is important enough to justify the additional nomenclature. Using SDE, a parallel program can be simulated on a uniprocessor, using a specially modified runtime system that multiplexes simulated threads of execution onto the cpu in response to directives from the simulator.

Holliday and Ellis [15] also have pointed out that the operations and control paths that are executed in a parallel program depend on the target execution architecture rather than the execution

architecture used to generate the traces. They call this the "global trace problem." They propose identifying a program's "address change points," which are places where differences in the control paths of the program can occur. Having identified these points, they then construct an abstract version of a process trace (the *intrinsic trace*) that allows them to generate the correct global trace for a particular execution architecture. Their approach has several drawbacks. The proposed framework does not appear to have been fully developed: the authors have not built a simulation system using their proposed solution nor have they used it to characterize a body of parallel programs. By comparison, the SDE technique allows the simulator to directly control the behavior of the program generating the trace, allowing it to create any global trace required.

A fair criticism of the SDE technique is that it has not as yet been systematically validated against untraced program execution (but note that this criticism applies just as well to *any* TDS technique). Usually, the system being simulated does not actually exist, making this impossible; however, proper performance analysis methodology suggests that a tracing tool should be validated against a set of existing architectures before being applied to nonexistent ones. As we elaborate in §4.5, this is much more difficult than it might appear.

In the next section we discuss SPAE, a simulation-driven execution mechanism that provides trace data rivaling the accuracy of traditional traces for a limited class of parallel programs. The overall goal of our research is to create a very flexible, modular, and efficient simulation-driven execution environment by completing the PEET architecture, of which SPAE is only one component. Our strategy for this will be discussed in detail in §4.

How to represent events? Conceptually, an execution architecture independent 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). We will use simulation-driven execution to generate execution architecture independent traces on a uniprocessor.

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 runtime systems.

Note that obfuscation of small-scale events is no longer a concern, because the traces are recorded on a virtual time scale unrelated to real time; effects at any scale can be explicitly simulated. The power of execution architecture independent traces threatens to complicate the construction of simulations, since they must now represent more detail in the target architecture. However, unimportant events can be intentionally excluded; contrast this with current measurement technology, where these effects are always present, even when not representative of the execution architecture being simulated.

3 Previous Experience: SPAE

Our implementation of simulation-driven execution tracing, SPAE[13], is an extended version of the AE (Abstract Execution) tool [21]. Using SPAE, a *subject program* is compiled by a modified compiler that generates an executable and a *schema file*. The schema file is an abstract version of the assembly code of each basic block in the subject application. The combination of executable and schema file produce a full trace with little execution dilation; see [21] for more information.[5]

We modified AE to support a separate *program context* for each independent execution stream in the program, such as an iteration of a `doall` loop or a parallel thread. When data is delivered to the simulator it is tagged with its context identifier; logically each context has its own trace. Data for each context is "decoded" by functions that reconstruct data and instruction references using the schema file. We can reconstruct references for multiple contexts, simulating an interleaving of the contexts active in the subject application.

In addition to recording events specific to each context, we also must record interactions between contexts. In particular, we provide *abstract events* for process synchronization. At the present time, the only abstract events supported by SPAE besides context creation and destruction are those having to do with `doall` and `doacross` (e.g., POST/WAIT) synchronization. In order to extend SPAE to handle more general types of synchronization, it will be necessary to integrate SPAE more tightly with the runtime system. We discuss simulation-driven execution techniques in detail in §4.

3.1 Using SPAE

This section employs 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 [36].

Recall that SPAE traces parallel applications on uniprocessor computers. The semantics of `doall` and `doacross` are satisfied by sequential execution. Simulating parallel `doall` on P processors involves storing the references of P iterations, then interleaving those references at simulation time. There are a multitude of possible interleavings of the iterations corresponding to the scheduling policy, the underlying simulated execution architecture and so on. The simulator of a parallel computer architecture must be able to select the interleaving applicable to the simulated hardware or scheduling policy. Each `doall` iteration is a separate context in SPAE. The simulation program can map different contexts to specific simulated processors.

When simulating the parallel execution of a `doall`, the order of the instructions issued within a particular context is specified because it is intrinsic to the execution of the program. However, the *global trace*[15] of the executed instructions is not known, because it depends on memory latency or contention for resources in the simulated parallel architecture.

The following example demonstrates the actual program instrumentation needed by SPAE. The original FORTRAN program locates the maximum element of an array. The program in Figure 1(a) has `S` independent iterations, and has been mapped to a two-processor system. We convert the

[5] This strategy is not unique to AE; both MPTrace and TRAPEDS use a similar technique. However, AE, like CARA[27], 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.

```
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
```

(a) Original `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);
```

(b) Converted `DOALL` Code

Figure 1: `doall` Example

code fragment to the C program, shown in Figure 1(b), using a FORTRAN to C translator, and hand-instrument it to indicate the `doall` loop.[6]

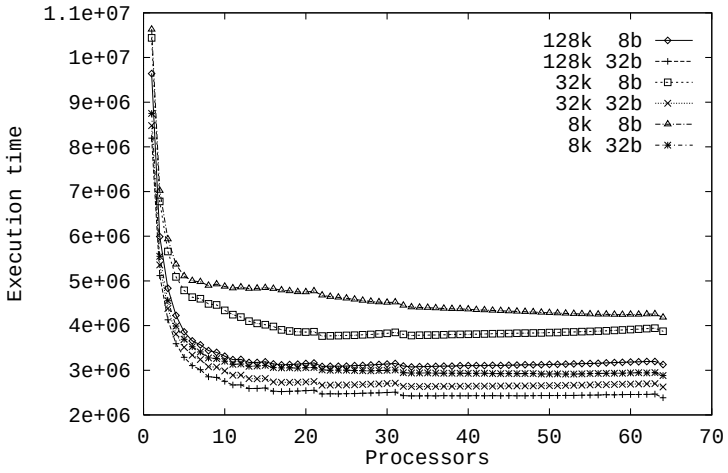
The subroutine calls to `ae_special_event` are recognized by a modified compiler (no actual subroutine call is performed). The first argument (e.g., `FORK_DOALL`) is used to decode the events specific to the parallel library, i.e., the `doall` loops in this example. The raw trace contains unwanted information, including the outer loop setup, increment and termination. In the example, the dynamic data for both contexts 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 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; see [14].

We have used SPAE to trace a FORTRAN program from the National Center for Atmospheric Research based on models of the shallow-water equation [30]. The traces were used by a multiprocessor cache simulator [9] to predict the program's performance on a cache-coherent multiprocessor using the Berkeley cache consistency protocol. The program generates over seven million instruction references and over two million data references; simulating the system for a particular configuration takes from two to six minutes, depending on the number of processors and the cache size simulated.

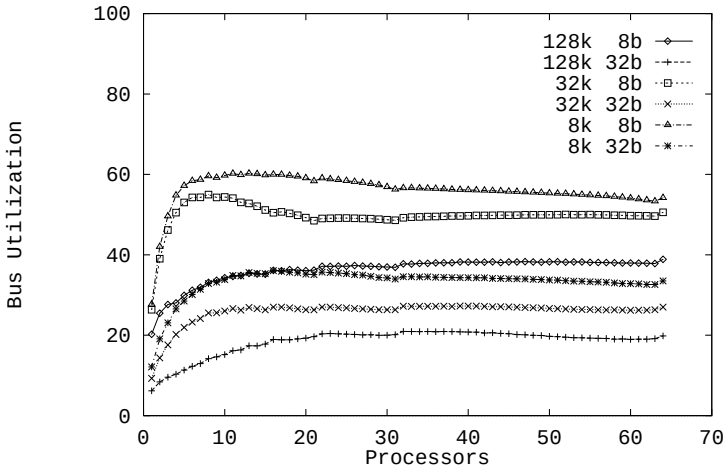
Figure 2(a) shows the total execution time as a function of the number of processors; different data sets show the peformance of varying the cache and line size. Figure 2(b) shows the bus utilization over the entire execution of the program. Initially, we were puzzled by the low bus utilization, but Figure 2(c) illustrates the reason; bus utilization is only high during the four `doall` loops in the program, and significantly lower during the sequential portion of the program. This example shows the scope of the data we can currently collect; we are also using this example to verify the accuracy of the generated traces.

We can generate similar traces for more general `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

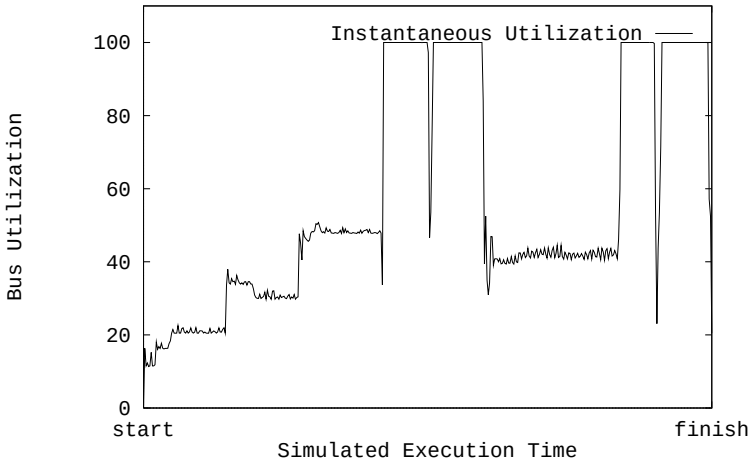
[6] When SPAE is extended to trace more general models of parallelism, the associated runtime library will be instrumented to generate abstract events automatically. (Refer to Figure 3 in §4.)



(a) Execution time for SHALLOW



(b) Average Bus Utilization for SHALLOW



(c) Time Varying Bus Utilization for 128k 8b Simulation

Figure 2: Data Generated by Simulation of SHALLOW Program

remapping; this is used in the multiprocessor cache simulator to examine the effect of array layout on cache behavior.

Our modifications affect both the instrumented application (trace producer) and the simulation (trace consumer). The instrumented application is slowed 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., portions of the Perfect Club suite) it is faster to simply rerun the application than to read the trace from disk. Currently, SPAE slows event decoding two-fold over AE. We expect to reduce this overhead, but in this example, SPAE generates more than five million references per second, and the time spent decoding events is only a part of the total simulation time.

SPAE is unique in that it provides both data and instruction traces while maintaining high efficiency. This use of SPAE suggests the value of PEET: it is possible to design simulation programs that interact with SPAE as it executes (rather than only using trace files generated by SPAE). Further, PEET will include libraries to model various implementations of abstract events that can be used at the interface between the trace consumer (the simulation program in the example) and SPAE.

4 Current Research

The goal of our research is to provide PEET – a modular, extensible and easy to use environment for studying changes to execution architectures. To accomplish this, we will

- Extend the SPAE tool to manage more complex control and synchronization mechanisms, such as threads, and different memory models, such as weakly consistent memory.
- Build an architectural toolkit to simplify the construction of architectural models.
- Develop techniques to validate the data provided by SPAE and other tools.

It is worth noting that the nature of PEET is such that much of the research can be conducted on a uniprocessor computer, although we are modeling and studying computers with radical, emerging execution architectures. The validation work will require that we compare SPAE predictions with observed results on available multiprocessor systems.

4.1 The Architecture of PEET

A major component of PEET is the SPAE tool; Figure 3 shows the overall architecture of PEET. SPAE is represented by the boxes labeled "compiler" and "event decode/context demux." The program to be traced (the *subject*) is at the top of the diagram. The compiler produces an instrumented executable file as well as the static schema file. When the subject executes, the instrumentation causes it to produce a compact data stream that can be used to reconstruct a full address trace of the subject's execution.

The subject is linked to a specially instrumented runtime system that supports the model of parallel programming in use by the subject (e.g., PARMACS, FORCE, C-THREADS). The runtime system instrumentation inserts abstract events, such as the start of a new `doall` iteration, the creation or destruction of a thread of execution, or the use of a synchronization primitive, into the

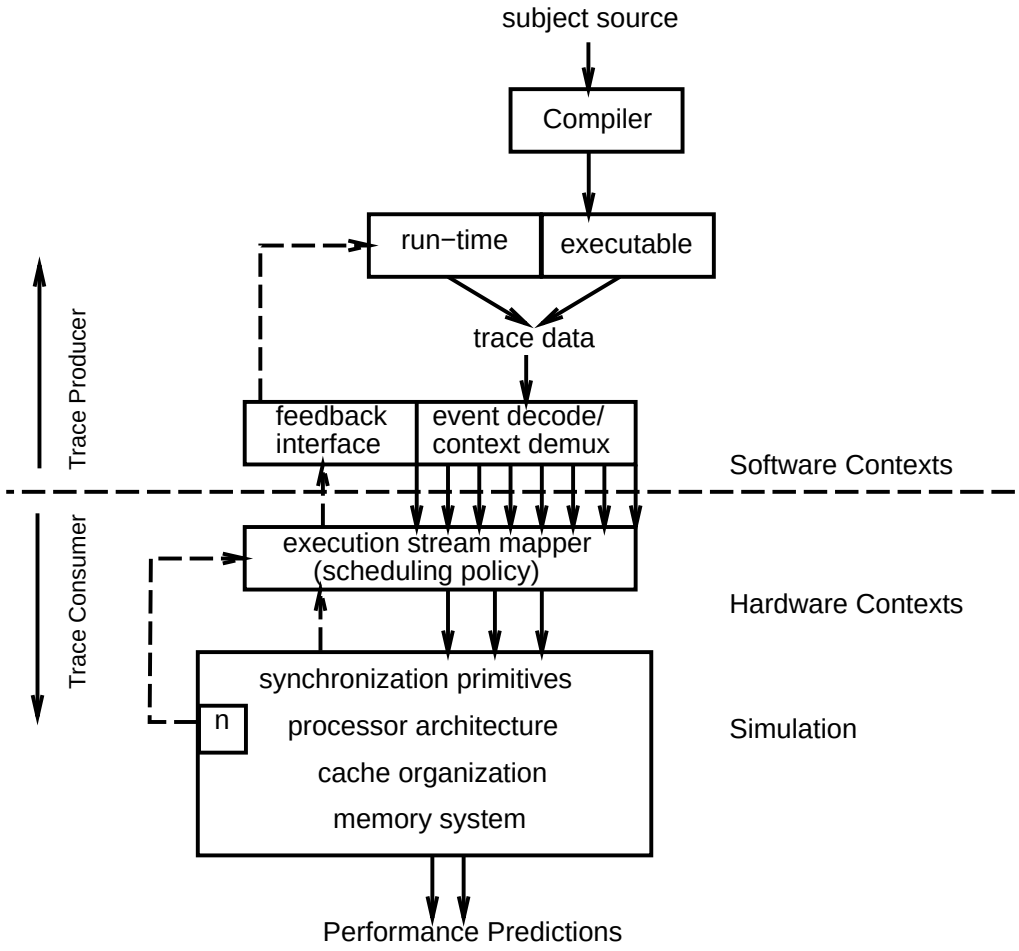


Figure 3: The PEET Architecture

trace. The SPAE event decoder keeps track of each context (thread of execution) in the subject. The combination of the subject, runtime system and decoding interface is termed the *trace producer*.

Since the subject is being executed on a single cpu, a single context is active at any one time, regardless of how many contexts are *logically* executing simultaneously. The instrumentation inserted by the compiler causes the active context to relinquish control to the runtime system at the end of the current basic block or generation of an abstract event, depending on the event granularity requested.[7] At that point, the runtime system waits for instructions from the *execution stream mapper* (described below) telling it which context to run next. This protocol is used to simulate scheduling policies as well as to resolve contention for synchronization primitives. It also reduces the amount of data that needs to be buffered by the context manager module.

The bottom of the diagram describes the *trace consumer*, including the simulator itself, written by a user of PEET. In our study, the function of the simulator is unimportant, save that it conform to certain interfacing conventions. Software contexts are mapped to a number of *hardware contexts* specified by the simulator. Each hardware context represents an independent instruction stream at

[7] Assuming a sequential memory consistency model; see §4.3 for a discussion of different consistency models.

the hardware level; for example, it might represent the number of simulated cpus in a conventional architecture, or it might be larger than the number of cpus for a multi-stream architecture.

The execution stream mapper provides the interface between the simulator and SPAE, multiplexing the N software contexts it is receiving from the trace producers onto the n hardware contexts expected by the simulator. The simulator repeatedly requests the next event (which may be a simple address reference or an abstract event) from the execution stream mapper for all hardware context(s) that are currently ready. "Readiness" is determined by the simulator, and may depend on simulated cache and memory delays, hardware multiplexing of contexts (e.g., on a multi-stream architecture), and whatever else the user decides is germane to the experiment.

4.2 Tracing General Thread Applications

Explicit, independent streams of control (threads) are a general parallel programming mechanism. A thread library (e.g., 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 execution architecture independent traces of a thread-based computation, we modify the runtime thread library.

Some changes to the thread library simply insert events into the trace, using `ae_special_event` as in §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 dynamic data indicating the creation of a new context to the simulator. 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; 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 simulator interprets this abstract event in a mechanism appropriate for the simulated architecture. Interpretation of abstract events may require the simulator to modify the control flow of the application. Figure 4 illustrates the information flow in this more general environment. In this example, several threads are contending for a mutual exclusion lock; the simulator selects the specific hardware context that successfully acquired the lock and uses the stream mapper to transform the hardware context to a software context (thread). The runtime system waits for the thread to be identified, and then executes that thread until an appropriate stopping condition has been reached. The resulting trace data for the thread is sent to the simulator.

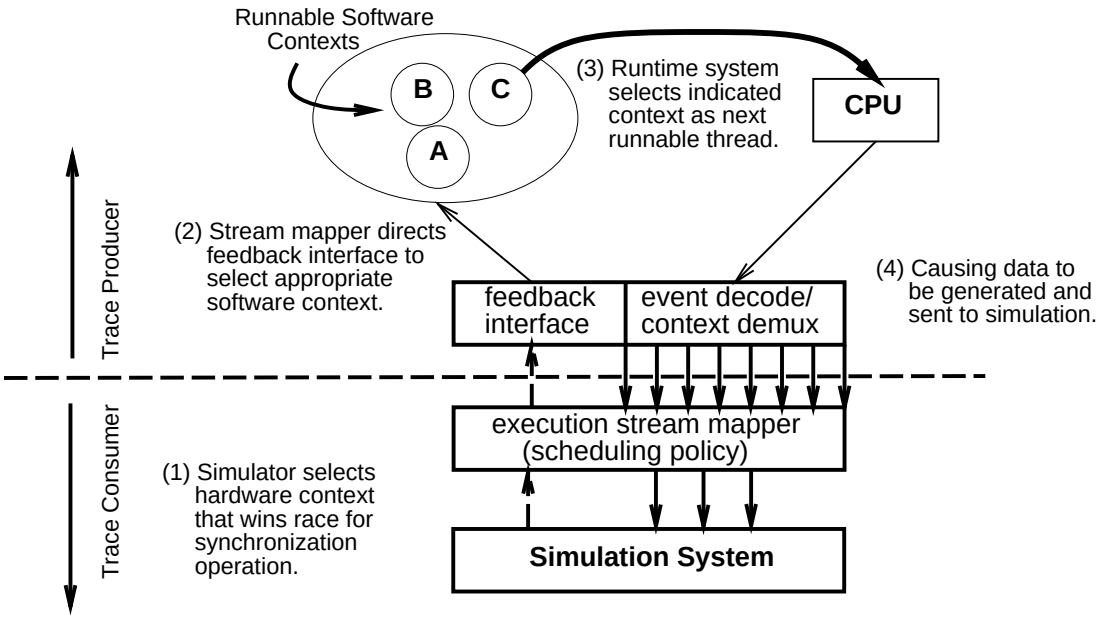


Figure 4: Interface Between Runtime System and Simulation System

There are several instances, such as lock acquisition, in which the simulator must direct thread scheduling. Other examples include interleaving thread execution to expose unsynchronized behavior, reducing the amount of data buffered for executing threads; see [13] for more details.

These modifications require extensive coordination between the simulation and thread libraries, via the feedback interface depicted in Figure 3. This work will define standardized interfaces between the event decoding and context demultiplexing interfaces of Figure 3 and the actual simulation. Our goal is to provide an environment in which trace producers and trace consumers can be shared. The interface with the thread scheduling system is the most difficult, because it makes tacit assumptions about the capabilities of the thread library.

We will initially integrate SPAE with the AWESIME thread library [12]. AWESIME is an object oriented parallel programming environment written in C++. It runs on several commercial multiprocessors and uniprocessors. We have considerable experience with the library, reducing the complexity of developing the feedback interface. Because few programs using the native AWESIME interface are available, we have built compatibility libraries to support more common parallel programming libraries, including C-THREADS, PARMACS and a draft version of POSIX threads.

4.3 Modeling Weak Memory Consistency Models

SPAE is a central component of the PEET system, and is designed to provide efficient trace generation without excessive communication between the trace producer and the trace consumer. Part of this efficiency is gained by having a single address space, implemented by a single Unix process, for all user contexts. However, this enforces a *sequentially consistent memory model* [20, 11] in which each store operation is *atomic*, i.e., immediately following a store to a location, all contexts "see" the same value when loading from that location.

Increasingly, computer architects are investigating systems with different memory consistency models [1, 22] because the semantics of strong consistency are difficult to implement efficiently for a large number of processors. These models require the *issuance* and *performance* of store operations to be separate; in effect, each store operation must be controlled by the simulation consuming the trace. Some simulators manage this by placing each context in a distinct address space, resulting in considerable overhead[8].

One advantage of the PEET architecture is that it provides an efficient simulation environment when sequential consistency suffices. We plan to extend SPAE to allow PEET to efficiently support *multiple consistency domains*, or multiple views of memory that are not mutually consistent. The state of each consistency domain is controlled by the memory system simulation. Each context is associated with a specific consistency domain; consistency domains represent the boundaries between different views of a common memory space. The challenge is to model this efficiently, greatly expanding the number of simulations that can be run; furthermore, less accurate models should take less time to process.

Since SPAE traces the execution of a program by instrumenting the compiler output, each load and store is identified and may cause information to be written to a trace consumer. Notice that the *reported* load or store address can differ from the *actual* address, as long as control-flow is not disturbed; that is, the instrumented program can reference one memory location but report the use of another. We use this observation to implement multiple consistency domains.

Our initial design assumes each consistency domain is allocated a copy of the shared region of memory. Rather than modify the "actual" shared area, the copy is modified. For example, if a variable is mapped to a specific memory address in the shared data, each consistency domain allocates storage for the variable. When the program is executed, it reports the address of the variable to the trace consumer as the primary location of that variable; however, loads and stores actually access the shadow copies in the different consistency domains.

The simulator must provide modules responsible for maintaining the various domains in accordance with the consistency model of the simulated architecture. The simulator must also indicate *when* it needs to make the consistency domains consistent. This is usually done for each shared load or store, or when synchronization events are encountered. By using the latter, the simulation overhead is reduced, and the results may be accurate enough for detailed simulations [7]. When tighter control is needed, overhead can be reduced slightly by triggering interaction with the simulator only on *shared* references. The interaction points are indicated by the compiler, where it is possible to distinguish between local references (say, those relative to the stack) and shared references. This greatly reduces the total amount of interaction between the tracer consumer and trace producer.

Although our approach uses the compiler to reduce the number of operating system processes needed to manage multiple consistency domains, synchronization between the simulator and the traced application will introduce delay. We expect such detailed simulations to be run on small-scale shared memory multiprocessors. Because a limited number of operating processes are used, the interaction can be mediated via a shared memory segment: the simulator and application will execute simultaneously, and the simulator will directly manipulate the memory images of the different consistency domains. If a multiprocessor is not available, a uniprocessor can provide an

[8] E.g., [7] cites a slow-down of between one-hundred and eighteen-thousand fold depending on the level of detail needed.

equivalent, albeit slower, interface. With two processors devoted to the tracing and simulation processes, we expected PEET to be significantly faster than comparable systems.

4.4 Simplifying Simulation Construction

SPAE provides considerable detail about program execution. Some simulation methodologies require only a fraction of the available information; furthermore, there are many applications for the raw data produced by the traced application. By disregarding the underlying execution architecture, we can construct precedence graphs of the parallel activity, measure maximal parallelism and determine the volume of data shared between tasks.

Beyond architecture-independent uses of the data, simulators can model cache and page references, simulate complex architectures or record the mobility of data in a network. Building such simulators requires communication with the feedback interface and the thread libraries. For example, the simulator may assume threads are scheduled in first-come-first-served order; this is such a common assumption that a standardized module providing an interface to the runtime system would simplify simulator construction. Likewise, a simulator may wish to run data through a cache simulator to determine miss rates, and thus, the data exposed to an interconnection network; again, the use of a few particular cache models is so common that a set of standardized modules seems useful.

Both the architecture independent and dependent information is represented by a series of modules in the simulation (see Figure 3). We have constructed some of these modules [9]. The current simulation environment is built using C++ modules running under the AWESIME environment. We plan to construct additional modules; since our uses of the trace data are varied, the modules are designed with reuse in mind.

Many details of architectural simulation are beyond the scope of such a library. PEET will not provide a completely general architecture simulator; rather, it will provide several modules that simplify the construction of such simulators, freeing the architect to concentrate on the novel aspects of an architecture. The bulk of these modules, derived from our own studies, will serve as templates for others to follow.

Other filters assist in *data interpretation*. For example, in our studies of parallel memory allocation we require the detailed information enabled by the general version of SPAE; we can only measure execution time and locking contention in actual systems, and deduce the reasons allocation strategies exhibit their behavior. However, with SPAE we are awash in data; we need tools to manage and present the data, allowing us to interpret it in a meaningful manner. Our goal is not a generalized data visualization system. As before, we plan to build tools that are primarily of interest to ourselves, and have others use these as guidelines when implementing their own tools. For example, the presentation module for the memory allocation study simply displays spatial and temporal memory references using standard graphics; more complex presentation is certainly possible.

4.5 Validation of Multiprocessor Traces

Given that we are constructing a testbed for system performance evaluation, it is important that we understand the accuracy and the limits of our evaluations. This section describes the issues related to validating results from our simulation system and points out the research that still needs to be performed to make validation a more scientific process. This section does not attempt to

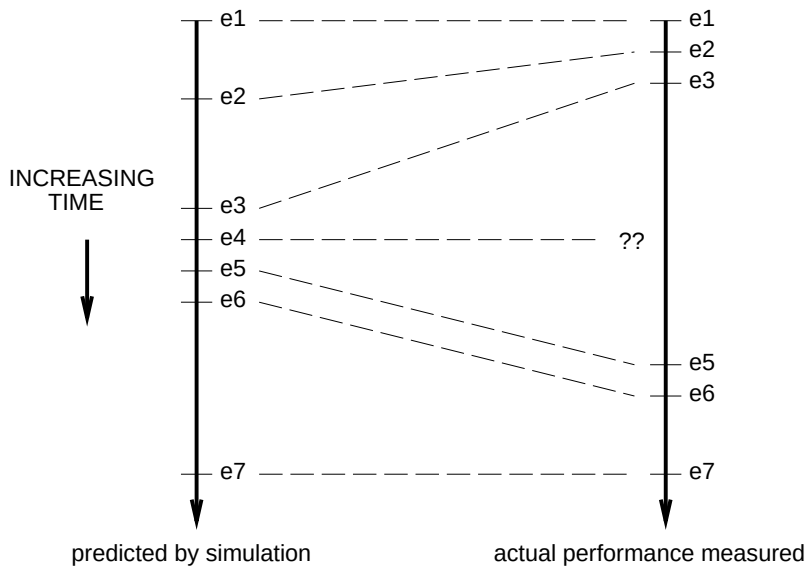


Figure 5: Example of Comparing Predicted and Actual Event Streams

describe the potential sources of error in our simulations. Instead, given that we have performed a particular simulation, this section describes how we evaluate how close that simulation was to the observed behavior of an actual system. For an informative overview of the issues associated with trace validation and the problem of trace distortion, see Stunkel, Janssens and Fuchs [35].

The Data Reduction Problem To understand what we are validating, it is important to understand the structure of the output of our simulation system. Conceptually, the output of a particular PEET simulation is a collection of event streams, each of which corresponds to a sequence of events performed by a simulated hardware processor (see Figure 3). For each event in each stream there is an associated time (as measured by some global clock).

Naturally, this representation of a simulation is particularly cumbersome and uninformative. For this reason, system evaluators prefer to reduce this large volume of information to a performance measure that is intuitive and informative to a wide audience, such as time-to-completion or cache miss ratio. While this data reduction results in measures that are easy to understand, a large amount of data is lost in the process. When attempting to validate the result of a simulation against measured performance, data reduction can result in incorrect conclusions. Consider the trace of events from a single thread as indicated in Figure 5.

The figure illustrates a stream of events (e1 through e7) that occurred in a program both as predicted through some simulation and as actually measured. Assume for now that the events in the observed event stream represent an unperturbed execution of the program. Suppose that one of the events (in this case e7) represents the end of the program. This example illustrates that by reducing the event stream to a single number (like time-to-completion), we lose significant information about the validity of the simulation. In particular, in this example we would conclude that the simulation was completely accurate, when in fact there are large differences between the behavior predicted by simulation and the observed behavior.

The fact that data reduction can void any attempt at validation is well known; consider attempting to compare two probability distributions having the same mean. We are unaware, however, of any attempts to validate simulations at any level other than at the level of the reduced data. In particular, we see the problem of comparing event streams as an interesting and promising area of open research. Needless to say, we also plan to validate our methodology at the level of the reduced data.

Comparing Event Streams If we want to understand how accurate a particular simulation is, we need to compare the event stream from the simulation with an unperturbed event stream of an uninstrumented program. Such an unperturbed stream is difficult to obtain because the perturbation of information collection may significantly change the behavior being observed. We plan to reduce the problem of perturbation in two ways. First, we will use low-dilation instrumentation to record the observed event streams from programs. We envision collecting a small set of events, such as thread creation, destruction, and synchronization using a fast mechanism such as a hardware microsecond timer. Second, we plan to apply proven techniques for recovering actual performance from perturbed performance as described by Malony [23].

Assuming that we have a sufficiently accurate observed event stream, the problem of finding the distance between two long sequences is not a new one: Sankoff and Kruskal provide an excellent summary of the field [31]. A similar problem arises in both DNA sequence analysis and in speech recognition. We plan to evaluate the appropriateness of the existing algorithms for this problem and adapt them as necessary.

The two broad categories of techniques for finding the distance between sequences depend on whether the sequence is discrete or continuous. Because our event streams include both discrete events and the times at which they occurred, either of these approaches or some combination might prove fruitful.

Discrete sequences are commonly studied in the comparison of DNA sequences. Levenshtein introduced two widely used metrics for distance between two sequences based on the smallest number of substitutions, insertions, and deletions needed to make the sequences the same. Efficient dynamic programming algorithms to compute the Levenshtein and other distance metrics exist and are widely used [31]. In DNA sequence analysis, differences in sequences are expected to arise through mutation. In our case, we expect most events to be present but for the times of the events to differ, perhaps significantly. Discrete sequence comparison methods can be used effectively by viewing each (event, time) as a discrete unit. In determining the distance between any two such pairs, we expect the events most commonly to coincide, leaving us to determine a proper distance function for differing time values.

Because the time values are continuous, comparisons of sequences using continuous techniques may also be valuable. Of particular interest is the concept of time-warping, which defines a transformation in the time domain that minimizes the distance between two sequences. Time warping techniques are discussed at length in [18].

Given that we have a measure of distance between two event streams, we can compare a number of event streams from actual executions to determine how much variation exists in the actual program. We can also use sequence comparison techniques to create an "average" stream from a group of such streams. Moreover, we can compare the behavior predicted by the simulator against the observed event stream. Having determined the variation among different executions of the same program, we can see how close or far the predicted execution is from the observed execution.

Given that we have a metric of closeness to the observed behavior, we can used this metric to attempt to improve the closeness of fit between the predicted and observed behavior by changing the simulation parameters.

All of these efforts will provide a much more detailed understanding of the behavior of a program as represented by an event stream.

5 Summary

Our previous experience with SPAE has shown it to be a valuable tool for gathering data about an architecture or application. The construction of PEET will provide similar data for more general programs and a wider variety of architectures. We believe the PEET system will increase the quantity and quality of information about parallel systems. We also believe it will be efficient enough to measure actual programs rather than application kernels or small benchmarks. The tools will be flexible and modular; we intend to make the tools available to other researchers.

Data at this resolution, akin to an architectural microscope, is unavailable to most researchers, limiting the scope of possible research. Accurate information on program dynamics will aid architectural researchers and developers. SPAE is based on a portable compiler that executes on a variety of architectures. The actual collection of data from parallel programs can be accomplished on a uniprocessor. This makes the technique widely applicable.

Our tools use uniprocessors to simulate parallel programs, but validation of these tools requires use of a parallel system. We plan to compare data from simulated and actual architectures, using conventional shared-memory multiprocessors and novel architectures.

We are currently using the data from our first tool, SPAE, for a variety of projects. Research projects now under way or planned include the simulation of shared-bus cache architectures, distributed cache architectures, memory management policies, and simulation of latency-tolerant architectures, and performance visualization. The abstract traces are also being used to characterize and understand parallel program behavior. These concurrent projects will provide us with valuable feedback into the design of the tools and the interfaces between them, and PEET is an enabling technology for these studies.

6 Acknowledgements

Tony Sloane did the initial implementation of SPAE. Bill Waite contributed greatly to the overall direction of this research through his frequent discussions with us. Gary Stormo and Andrzej Ehrenfeucht helped us to understand the issues involved in sequence comparison.

References

[1] Adve, S. V., and Hill, M. D. Weak ordering - a new definition. In *Proc. Seventeenth International Symposium on Computer Architecture* (1990), ACM.

[2] 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.

[3] 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.

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

[5] 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.

[6] Covington, R., Madala, S., Mehta, V., Jump, J., and Sinclair, J. The Rice parallel processing testbed. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1988), ACM, ACM.

[7] Davis, H., Goldschmidt, S., and Hennessy, J. Multiprocessor simulation and tracing using Tango. In *Proc. Int. Conf. on Parallel Processing* (May 1991), ACM, ACM.

[8] 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).

[9] Färber, P. G. Analysis of a shared bus multiprocessor memory system using trace drive simulation. Master's thesis, University of Colorado, Boulder, 1991.

[10] Gannon, D., Jalby, W., and Gallivan, K. Strtegies for cache and locality memory managment by global program transformations. In *Proc. First ACM Intl. on Supercomputing* (1987).

[11] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable schared memory multiprocessors. In *Proc. Seventeenth International Symposium on Computer Architecture* (1990), ACM.

[12] Grunwald, D. Awesime: An object oriented parallel programming and simulation system. Tech. Rep. CU-CS-552-91, University of Colorado, Boulder, 1991.

[13] Grunwald, D., Nutt, G., Sloane, A., Wagner, D., Waite, W., and Zorn, B. Execution architecture independent program tracing. Tech. Rep. CU-CS-525-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, April 1991.

[14] Grunwald, D., Nutt, G., Wagner, D., Waite, W., and Zorn, B. A testbed for improving the performance of parallel programs and systems. Tech. Rep. CU-CS-512-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, Feb. 1991.

[15] Holliday, M., and Ellis, C. Accuracy of memory reference traces of parallel computations in trace-driven simulation. *IEEE Transactions on Parallel and Distributed Systems* (1991). (to appear).

[16] 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).

[17] 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.

[18] Kruskal, J. B., and Liberman, M. *The Symmetric Time-Warping Problem: From Continuous to Discrete.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1983, ch. Chapter Four, pp. 125–161.

[19] Lam, M., Rothberg, E., and Wolf, M. The cache performance and optimization of blocked algorithms. In *Proc. of the Fourth Intl. Conference on Architectural Support for Programming Languages and Operating Systems* (1991).

[20] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers C-28*, 9 (Sept. 1979), 241–248.

[21] 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.

[22] Lenoski, D., Laudon, J., Gharachorloo, K., Gibbons, P., Gupta, A., and Hennessy, J. Memory consistency and event ordering in scalable schared memory multiprocessors. In *Proc. Seventeenth International Symposium on Computer Architecture* (1990), ACM.

[23] Malony, A. Even-based performance perturbation: A case study. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Williamsburg, VA, Apr. 1991), pp. 201–212.

[24] 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.

[25] 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.

[26] 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.

[27] Mitchell, C., and Flynn, M. A workbench for computer architects. *IEEE Design & Test of Computers 5*, 1 (Feb. 1988), 19–29.

[28] Mitchell, C., and Flynn, M. The effects of processor architecture on instruction memory traffic. *ACM Transactions on Computer Systems 8*, 3 (August 1990), 230–250.

[29] 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.

[30] Sadourny, R. The dynamics of finite-difference models of the shallow-water equations. *Journal Of Atmospheric Sciences 32*, 4 (April 1975).

[31] Sankoff, D., and Kruskal, J. B., Eds. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

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

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

[34] 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.

[35] Stunkel, C. B., Janssens, B., and Fuchs, W. K. Address tracing for parallel machines. *IEEE Computer 24*, 1 (January 1991), 31–38.

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

[37] Zorn, B. G. The effect of garbage collection on cache performance. Tech. Rep. CU-CS-528-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, May 1991.