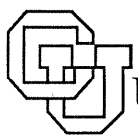
A Parallel Program Tuning Environment

Gary J. Nutt

CU-CS-664-93 August 1993



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Parallel Program Tuning Environment

Gary J. Nutt
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430
(303) 492-7581
(303) 492-2844 (FAX)
nutt@cs.colorado.edu

CU-CS-664-93 August 1993

University of Colorado at Boulder

Technical Report CU-CS-664-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

A Parallel Program Tuning Environment[†]

Gary J. Nutt
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430
(303) 492-7581
(303) 492-2844 (FAX)
nutt@cs.colorado.edu

August 1993

Abstract

Programming shared memory parallel machines introduces a new dimension to an already complex problem, namely the challenge of managing parallel execution effectively. In this paper, we describe a tool environment to assist the parallel application programmer in *tuning* a parallel program once it has been implemented. Many of the tools are based on interactive visual presentations, although they rely on a spectrum of underlying support and analysis tools. The value of this work is perceived to be the incorporation of a broad spectrum of tuning tools within a consistent environment that provides fundamental measurement services. The basis of all of the tools is a flexible facility for generating abstract causal traces of the program's execution, and which allows the abstract trace to be bound to a particular execution architecture as either a specific causal trace or a specific timestamped trace. In this paper we describe the general architecture of the system, and then briefly characterize components within the environment.

[†]An abbreviated version of this paper appears in the *Proceedings of the 1993 International Conference on Parallel Processing*

1 Introduction

Designing and implementing *effective* parallel programs is proving to be a difficult task. The difficulties result not only from the development of effective algorithms that take advantage of data placement and computing cycles, but also from the number of variables in the underlying computing platform. An efficient program must have an effective algorithm that matches the platform configuration. While contemporary abstract machines (e.g., the hardware, operating system, and runtime system) provide access to a relatively uniform set of facilities, the performance characteristics of the implementations of the facilities must generally also be known by the application programmer. For example, the programmer may be able to construct a more efficient matrix multiplication implementation if he knows the size of the cache memory in a shared memory system than if he is oblivious to its size.

Thus the parallel application programmer must be able to observe almost arbitrary performance characteristics of the execution of his program from a number of different perspectives as determined by his specific needs at any time. This suggests that performance tools to support parallel application development must either be defined explicitly for each application domain (or worse, for each programmer), or that the tools must be comprehensive, flexible, and configurable to support specific domains and programmers. The distinction is essentially one of expert systems versus toolkits.

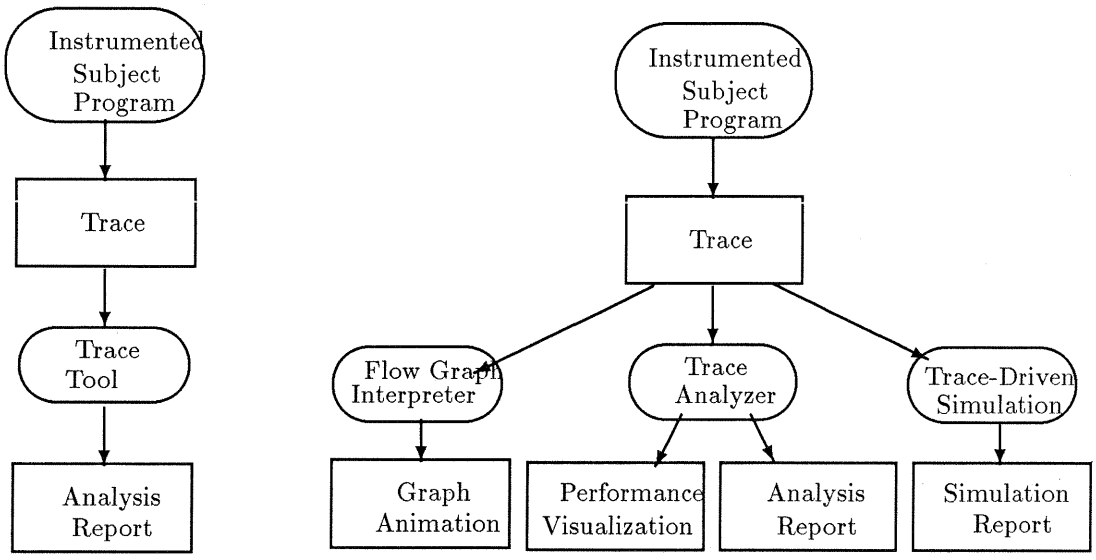
Our experience with individual tools also strongly supports the utility of visualization when addressing parallelism. Irregular concurrency is difficult to comprehend because it inherently has a "thread" for each part of the operation; the observer must be able to synthesize the collective actions of the "threads", e.g., by finding an intellectual lever for abstracting away the details so that one can see the more general pattern of behavior. We have found that this kind of knowledge sometimes requires that the designer be able to obtain a *qualitative* understanding of the behavior of the program before focusing on the *quantitative* details. For example, animation introduces a new dimension to the perception of the performance of a program – the resulting observations tend to be qualitative rather than quantitative. While the animation is useful for identifying general troublespots in the computation, it is less useful for determining the specific cause. We rely on visualization to gain a qualitative understanding of performance and on traditional measures for observing the performance of the program quantitatively.

A parallel program tuning environment is most useful if it provides a comprehensive set of performance tools which provide a broad (but complementary and consistent) set of views of the execution. Like all other tools, the tuning tools must be easy for the application programmer to use. This is the goal of the tuning environment described in this paper.

The parallel program tuning environment framework supports either expert systems or toolkits, and tools for both qualtitative (visual) and quantitative observations of the performance for shared memory machines. We have been building components for several years, and we are now integrating them into a consistent environment that could be used to design expert system performance tools or tailored environments from generic toolkits. In this paper we first provide a high level description of the parallel program tuning environment, then we briefly describe some of the tools; we also attempt to identify key issues on which we have focused (and continue to focus) in our research efforts.

2 Background

There is a widely held belief that the paucity of software tools is a substantial barrier to the use of parallel machines to solve general problems in science, e.g., see [27]. Researchers that address this area tend to base their approach around *traces* of the programs execution that can be analyzed after the subject program has executed; the tool is built to support post mortem analysis (see Figure 1(a)). The



(a) A Trace Tool (b) A Tracing Environment

Figure 1: Trace Driven Performance Analysis

critical components of the tool are the instrumentation component and the analysis tool itself — some researchers concentrate on the instrumentation and others on the tool.

Our goal is to provide a tool environment which incorporates common tracing and analysis tools and which accommodates the development of extended tools. Since we believe that most performance tuning tools are based on traces, we incorporate a comprehensive trace collection tool into the environment, then provide for means for a spectrum of different performance analysis tools to use the common tracing facility (see Figure 1(b)). Our tracing facility – the PEET parallel execution evaluation testbed – does not depend on the existence of the target computing platfrom, but rather it executes in a uniprocessor abstract execution environment that can be bound to a family of specific shared memory multiprocessor execution environments. Varous tools can (and have) been built to use the bound traces to drive other simulations and to analyze the performance of the program on the hypothetical computing environment. This allows us to obtain visual feedback on the performance from a control flow perspective, a resource utilization perspective, and from a customized perspective.

The tuning environment has been influenced by a wide spectrum of research, ranging from multiprocessor trace collection, to trace analysis and simulation, to performance visualization. In the remainder of this section we relate our work with some of the other work in the literature.

2.1 Using Traces

Traces have been used to study computer performance for over two decades, e.g., see [32]. In some cases traces have been used to represent the load due to real programs (as opposed to hypothetical loads generated by synthetic programs and mixes). In other studies, the trace is analyzed to obtain an understanding of various characteristics of the program and computer that produced the trace.

In the 1970s, traces were most heavily used for evaluating the performance of paging algorithms, e.g., see [31]. Now traces have again become an important tool for studying the behavior of cache

systems in shared memory multiprocessors e.g., see [1, 22, 35, 6, 8, 14, 30]. A set of subtle, new problems are introduced when one attempts to use traces to represent the load provided by a parallel program [5] (we summarize below).

Trace Variations Based on Utilization. A trace can represent a highly detailed list of event occurences such as memory references. The execution of a simulation can significantly change the order in which events occur in the global trace when the trace is used to drive a different execution architecture from the one on which the trace was gathered. Holliday and Ellis have addressed this problem by identifying *address change points* where the memory trace on an extrapolated architecture may differ from the given trace because of different actions at *address affecting points* with respect to the address change points [20].

Nondeterminacy. An important aspect of shared memory multiprocessors is that programs written for the hardware are often nondetermistic – sometimes inadvertently. This occurs whenever independent (logically concurrent) blocks of computation read and write a common block of memory (the problem studied by Holliday and Ellis is a variant of this problem – the nondeterminacy may result from races to address affecting points). There is also nondeterminacy in resource competition managed by the hardware and the operating system. For example, if N processes are blocked on a spinlock when another process releases it, the selection of the next process to obtain the lock is usually nondeterministic (it depends upon the cache coherence policy among other things).

How can one talk about "the" trace of a nondeterministic program and execution architecture? Often, the analyses that use a particular trace assume that it is representative of related traces, and that the behavior implied by the representative trace is acceptable. Emrath, Ghosh and Padua describe a set of tools used to detect nondeterminacy in the execution of a program [28]; while the work does not relate directly to traces, the results and the tools are related to the tools we use. Netzer and Miller have considered the relationship of nondeterminacy and race conditions, and has shown that the computational complexity of detecting such problems is NP-complete [24]; Eckert has addressed the complexity of managing classes of nondeterministic computation in generating or migrating traces [7].

Execution dilation. Tracing depends on instrumentation, and software instrumentation techniques introduce overhead – called *execution dilation* in the tracing literature. While most trace studies go to considerable effort to reduce the execution dilation, Koldinger et al. [9] argue that *any* dilation can cause the trace to be an invalid representation of the behavior of the parallel program, essentially because of the potential change in the order of event occurences whereever races may occur (cf. Holliday and Ellis and the discussion of nondeterminacy above).

2.2 Performance Visualization

We have been influenced by the success of Paragraph [16], and others as tools for tuning a parallel program e.g., see [2, 34]. Heath has illustrated that even modest development efforts at building such a tool can have tremendous impact on the domain programmer, provided that the tool has been designed by/for the programmer rather than in a vacuum [15]. Therefore, the perspective from which we study performance visualization is the domain programmer's perspective (as opposed to the system perspective).

Performance Visualization Toolkits Because of the need for the visualization tools to be customized to the needs of the domain programmer, there is a trend toward providing a toolkit from which the programmer can construct their own performance visualization, e.g., see [17]. There is little

doubt that some variant to this approach will ultimately have to be part of any effective performance visualization system. The parallel program tuning environment is intended to support such efforts.

Visualization and Tracing Malony et al. have done considerable work in areas related to trace analysis and visualization [19]. There work had addressed to problem of presenting the results of trace analysis to the programmer.

Control and Data Flow Graphs Our own past work has focused on representing performance characteristics of a parallel program in terms of control flow and data and message flow [25, 11, 4]. There are many other studies that emphasize this graph-oriented view for visualizing the program's behavior, e.g., see [33], particularly for animation of the behavior.

3 The Parallel Program Tuning Environment Framework

The goal of our parallel program tuning environment (ppte) is to allow application programmers to tune an application for a shared memory multiprocessor on various configurations of the machine using only a uniprocessor system for the study. Further, the ppte should enable the application programmer to view the performance from a very broad range of perspectives, including quantitative and qualitative views, visual and numeric reports, metric-based and flow graph views, etc.

The ppte presumes the existence of a parallel program written using a sequential programming language with a parallel programming package (e.g., C programs that use a threads package), a specification for the target execution architecture (i.e., combination of hardware, operating system, and runtime system), and a uniprocessor with a similar operating system and a processor that executes the target instruction set. [1] An instrumented version of the program is traced on the uniprocessor, producing a trace with abstract events; the resulting trace is bound to a specific execution architecture, then is used to drive various tuning tools in the ppte.

A *causal event trace* lists the sequence of certain events in the order in which they occurred on the instrumented specimen; the precise nature of the events is determined by the nature of the instrumentation. A *timestamped event trace* incorporates the causal order, and also adds the virtual time at which each event occurred during the execution. Causal traces can be used to define the load due to the program for some model of the execution architecture; timestamped traces are most useful for direct analysis, since they already incorporate the resultant performance behavior. Both causal and timestamped traces are used in the ppte.

Figure 3 is a block diagram of the components in the parallel program tuning environment. The environment includes tools to instrument the target program, to execute it to produce an abstract causal trace, to bind the trace to a causal or timestamped trace, and to provide a wide set of views of the performance of the program on a specified execution architecture.

The SPAE trace generator for parallel programs [5, 12] is based on Larus's AE tracing facility [18]; AE is used to collect trace data from sequential C programs while SPAE uses the same mechanism to collect abstract trace data from parallel programs written in C and using the C threads library. SPAE automatically instruments the target program so that it will issue normal events such as memory references, and *abstract events* relating to concurrent operation, e.g., that the program is spawning a new thread, that it is attempting to obtain a lock, etc. The instrumented program can be executed serially on a uniprocessor, causing normal events and abstract events to be saved as an *abstract causal trace*. The AE technique separates information that can be determined about the trace at compile

[1] We have implemented our ppte for the C/C threads tools but have also used it on Fortran/Parmacs programs by using a Fortran-to-C conversion package, and by mapping Parmacs calls to C threads calls.

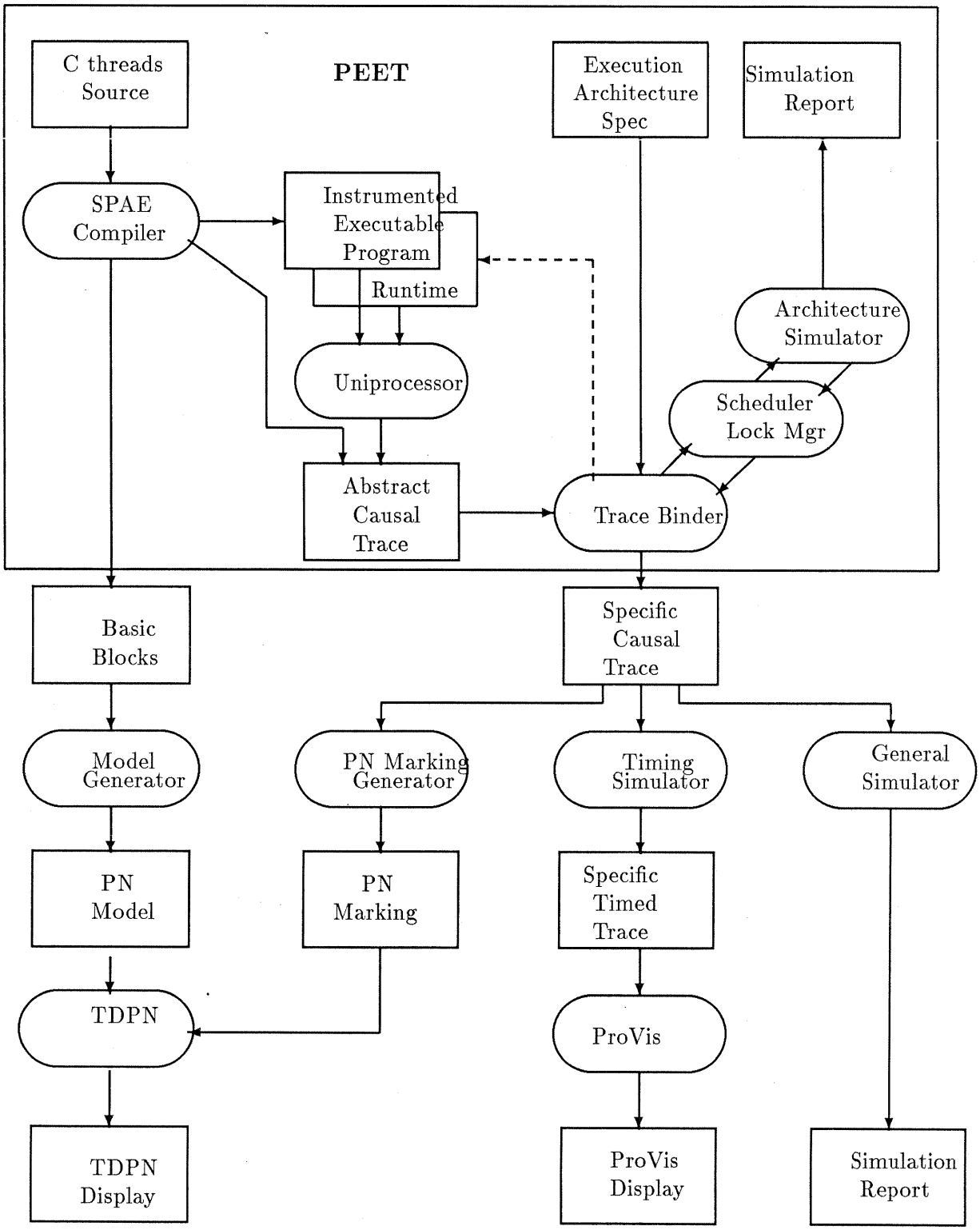


Figure 2: The PEET Tracing Facility

time from information that can only be obtained at runtime; the figure illustrates this by indicating that part of the abstract causal trace is derived directly from the SPAE compiler and part of it from the execution of the program – see Subsection 4.1 for more details.

The abstract causal trace is a sequential causal trace from the uniprocessor execution, with abstract events inserted whenever logically concurrent operations were encountered. The abstract causal trace could be bound to a *specific causal trace* by simulating the behavior of a particular parallel execution architecture on each abstract event, translating its effect into a set of parallel event traces. Notice that this simulation essentially performs two tasks: it converts abstract events into specific events, then establishes an order on these events with the specific events that exist in the abstract causal trace. After such a simulation, there will be as many specific causal traces as there were processors in the target execution architecture, and no specific causal trace will contain any abstract events.

Translation from abstract to specific causal traces implies that the *trace binder* in the figure simulates the behavior of the runtime thread systems on the target execution architecture, i.e., the trace binder simulates the thread scheduler, lock contention, barriers, and condition variables. In the figure we indicate this by showing how the PEET performance testbed uses SPAE traces to bind abstract traces to specific traces, i.e., PEET is itself an environment for using the trace data produced by SPAE. The trace binder reads the abstract trace, identifies abstract events, then invokes the scheduler/lock manager simulation to have it model the activity. The scheduler may be simulate the architecture itself, or it may invoke a more detailed architectural simulation, e.g., one that models cache behavior. When the behavior for an event has been defined, it is fed back to the trace binder. (This technique is defined in more detail in Subsection 4.2.) (PEET also employes feedback between the trace binder and the C threads runtime system to literally replace the C threads scheduler and lock management implementations by the simulated scheduler and lock manager in the trace binder. This allows (requires) PEET to gather traces on the fly by directing the execution of the program.) This requires that the program be executed each time it is analyzed, but eliminates the need for storing massive trace files.

Specific causal traces are used to drive three classes of performance visualization tools: timestamped trace analyzers, trace-driven flow graph animator/interpreters, and arbitrary high level trace-driven simulations.

A specific timestamped trace can be derived from the specific causal trace by a trace-driven simulation that introduces resource utilization metrics. The simulator uses the causal trace to define the load on the target system, then simulates resource utilization of the resulting execution architecture; e.g., a cache simulator would introduce delays related to cache misses and the implied data movement in the memory hierarchy. The resulting timestamped trace is a serialized audit trail of the performance of the program that can be analyzed to present the performance data using numeric reports or performance visualization tools. Notice that the simulator not only introduces time to the trace, it also filters event occurrences so that only the appropriate events are passed to the presentation tools. ProVis – a prototype tool for performance visualization [3] – is one example of such a tool. We elaborate on performance visualization frontends in Subsection 4.3.

It is sometimes difficult to infer characteristics of the flow of control from conventional performance visualization displays. Instead, a flow graph or precedence graph model may be a more useful view of the parallel program's execution. Petri nets (and Petri net variants) are often used to represent control flow aspects of a parallel program, e.g., see [29, 23]. The tuning environment uses the Olympus systems [11, 25] to provide a visualization tool based on trace-driven Petri nets. Causal traces are used to provide additional constraints on the flow of tokens in a timed Petri net; this results in a view that illustrates the parallelism inherent in the parallel program correlated with the execution of parallel segments constrained by available resources. Subsection 4.4 provides more discussion of this part of the tuning environment, including the tool for generating parallel Petri net representations of the target program.

Performance visualization tools and Petri net models may not provide precisely the view that the analyst needs to tune the program; the final class of visualization tools uses specific simulators to produce performance reports for specific aspects of the program and platform. In this case, the causal trace provides a load (as in the case of the generic timing simulator used for the performance visualization toolset), but the simulator and its performance reports are arbitrary. Subsection 4.5 relates more information about this class of tools.

The underlying assumption in the design of the ppte is that the tool designer cannot always predict the appropriate view of performance data that will be of the most use to the application programmer. The ppte provides a framework for gathering essential performance data on a uniprocessor, then processing that data with the tools that are most useful to the programmer. It is essential that these tools be easily invoked, so that there are few barriers to their use; that is part of the challenge in building a useful tuning environment. Our approach has focused on providing a useful set of built-in tools rather than in providing a mechanism for customizing tools. However, we believe that the environment is a prerequisite to such visulization toolkits; we will continue our investigation in parallel program tuning with such toolkits once the environment is sufficiently easy to use.

4 The Tools

The ppte provides different views of the computation by supporting different tools driven by a common specific causal trace and other information regarding the structure of the program. The PEET portion of the environment produces a specific causal trace for a target execution architecture; the trace can then be filtered again to influence the control flow model view of the computation, filtered by a timing simulator to produce a specific timed trace, or passed to an arbitrary trace-driven simulator. In this section we elaborate on these components of the tuning environment.

4.1 PEET and SPAE

The Parallel Execution Evaluation Testbed (PEET) includes the Symbolic Program Abstract Execution (SPAE) tool to generate abstract causal traces, and various other tools to simulate execution architectures and to bind abstract causal traces into specific traces.

As mentioned earlier, SPAE is based on Larus's AE tracing facility; the goal of AE is to *efficiently* collect detailed trace data on the execution of a sequential program [18]. The targeted efficiency is in the time overhead (dilation) due to instrumentation and in the space required to save the resulting trace.

The AE technique is to modify the compiler so that it statically analyzes the target program to detect which parts of its behavior are invariant to different executions, and which parts depend on information that is only known at runtime. The invariant parts are saved as abstract code in a *schema* file and the compiler emits instrumented object code to collect and write events relating to the variant part of the program when it is executed. On execution, the program produces an *ae.out* file containing the trace information from the variant parts of the program. The information in the ae.out file and the information in the schema file are combined by translating the schema into a C program that incorporates statements to read information from ae.out at appropriate times; the specific trace is then generated by executing the C version of the schema file on the ae.out file.

AE provides tools that allow arbitrary events to be introduced in the ae.out file. SPAE takes advantage of this feature to recognize events related to parallel activity and to cause arbitrary events to be written to ae.out – in particular, SPAE instruments the C threads library so that when any library routine is called, it emits the appropriate abstract event(s) to ae.out. Second, SPAE keeps track of execution contexts based on calls to the C threads library; whenever a new thread is created, then

the instrumented library code emits an abstract event to identify that occurence. Similar abstract events are emitted whenever a thread call might cause a context switch or a thread to be destroyed. The result is that whenever an uninstrumented program would create or change a context, it does so by calling the thread library; in the instrumented version the thread library has been changed to generate an abstract event to that effect in the trace. Next, the trace information is processed dynamically by the trace binder. The trace binder buffers contexts as it prefers, since it can cause the execution to advance in the uniprocessor instrumented execution by scheduling a particular thread to execute, i.e., the thread call results in action by the trace binder (possibly backed up by architectural simulation) to schedule the thread. Within PEET, the C thread implemenatation is effectively the trace binder, scheduler/lock manager, and architecture simulator.

Issues

1. Time and space efficiency
2. Is it possible to generate robust abstract causal traces
3. Are abstract causal traces unique for each program
4. How much dependency should there be on the architecture simulator

4.2 Binding the Abstract trace

The trace binder is a specialized implementation of the thread library to support feedback execution. As explained in the previous subsection, the execution on the uniprocessor is a single process that supports many threads (to represent the parallelism in the subject application). PEET begins execution by starting the subject program and the trace binder, with the abstract causal trace piped into the trace binder. When the subject intends to create a thread, it calls the thread fork library routine; this causes an abstract event to be written to the abstract causal trace. The trace binder detects the fork abstract event and creates a new context to represent the thread; the subject continues executing the original thread. The computation will only switch execution to another thread as the result of some thread library call; each such call (e.g., lock) results in the generation of an abstract event that causes the trace binder to multiplex across contexts.

How does the trace binder choose the new thread context to execute? It will have built a ready list of threads that are competing for the processor, so it must select from those threads; this is accomplished by running the scheduler/lock manager module. This module can be a stochastic implementation, or it can be a full thread library implementation.

In some cases the scheduler/lock manager can only determine which thread to schedule next on the basis of the behavior of the underlying execution architecture. Again, a simulator can select the thread through a stochastic mechanism or by running a architecture simulator. In the PEET study Grunwald and Farber have built different architecture simulators to study cache memory designs for shared memory processors (without using the rest of the ppte) [10]. Grunwald has also built his architecture simulation package so that it can easily interact with the SPAE abstract trace facilities [13].

Issues

1. Representativeness of the specific trace when there is nondeterminacy
2. Effect of nondeterminacy in the execution architecture

3. Effect of nondeterminacy in the program

4. Correctness of execution architecture model (timing aspect)

5. Tools for constructing architectural simulations

4.3 The Performance Metric View

This part of the ppte is concerned with ways that arbitrary analysis tools can be made to produce different views of the program behavior, illustrating the results numerically or visually from a specific causal trace produced by PEET. The ppte currently provides no specific facilities for constructing analysis tools, although we have prototyped various performance observation tools within the ppte.

Performance tools should be able to report the behavior of a program on a specific execution architecture (or closely related execution architectures) in terms that are familiar and useful to the programmer. Examples of such tools are those that report CPU utilization, synchronization, send-receive behavior, process architecture, working set characteristics (e.g., relating to caching and to paging), and input-output behavior.

The factors that influence parallel program performance are complex, ranging from algorithm selection and implementation, to programming language and runtime system, to the operating system, to the hardware architecture. We distinguish among performance issues that are strongly related to each of these aspects of the total system environment, and will attempt to study factors specific to each. For example, one might consider the problem of modifying an existing sequential program to be a parallel program on a specific execution architecture; or that of tuning a parallel program for a specific execution architecture configuration (number of processors, cache size, etc.).

We take the position that it is unlikely that we will be able to design a set of specific tools *a priori*, that will be useful to a wide range of domain programmers. Thus, we direct our efforts at supporting *meta tools* [17] which the domain programmer can use to construct specific tools for observing the performance of a particular parallel program on a particular execution architecture.

The extension to our current tool prototyping effort is to develop a meta tool that can be used by a domain programmer to instantiate and configure a set of tools specific to his program. Our preliminary experience with these tools in our ppte is limited to the ProVis prototype [3], where the specific causal trace is translated into a specific timed trace using a variant of the architecture simulator. (In our prototyping efforts, the architecture simulator and the trace binder were modified to introduce time so that the trace binder actually produced specific timed traces directly.) In the future we expect to use McWhirter's visual frontend generation package to rapidly build tools that provide different views of the performance data [21]. This implies that the meta tool may influence the specific configuration of SPAE as well as the number and nature of the visualizers.

Issues

1. What are reasonable performance views based on cognitive issues

2. The design of application programmer toolkits

3. Languages for programming the tools

4.4 The Control Flow View of Performance

Control flow graphs are also a useful representation of a program's performance, e.g., representing the best parallelism that could be obtained with an unbounded number of processors e.g., see [4]. In the

case of Petri net control flow models, one can obtain traces containing events to drive the execution – dictate the flow of tokens – of the Petri net for the instrumented configuration. Each path that a token traverses through the Petri net represents the control flow of a sequential unit of computation; thus the dynamic behavior of the Petri net model represents the behavior of an execution architecture for the particular program.

We have also built a control flow interpreter that operates within our ppte by adapting our Olympus modeling system [11, 25, 26]

We can outline the general tasks in the approach as follows (and represented pictorially in Figure 3). Suppose that we wish to study the behavior of a parallel program written in C using a threads library (suitable for processing by SPAE.

1. Use the SPAE compiler to translate the program into object code modules with embedded instrumentation. SPAE also identifies basic blocks of computation in the program.

2. The instrumented program is executed on a uniprocessor configuration with an instruction set corresponding to the target execution architecture to generate a trace of the logical parallelism among the basic blocks.

3. Each basic block is mapped into a Petri net place; edges and transitions are added to reflect the control flow among basic blocks, generating a Petri net model of the program. (If the degree of parallelism is determined at runtime, then part of the trace data is used to determine the degree of parallelism in the model.)

4. The trace data is correlated with basic blocks and is translated into a history of markings in the Petri net.

5. Use Olympus to generate a tailored version of the model such that places and transitions remain consistent with those identified in the trace.

6. The markings of transition firings drive the Petri net simulation model.

Issues

1. Automatically generating a flow graph model from the subject program

2. Melding trace data with token flow with timing data

3. Transformations on the flow graph and transformations on traces

4. Generating the maximally parallel graph from the trace

4.5 High Level Trace Driven Simulation

Contemporary shared memory multiprocessor simulation studies tend to focus on the design of the memory hierarchy and the movement of date within the hierarchy, as that is a key element of the performance. However, trace-driven simulation has also traditionally been used to study higher level issues, particularly for system configuration studies. In this case, the fine details of the trace are not so important as the more gross measures of resource utilization requirements of realistic programs. The trace(s) define the number and type of threads that are competing for the CPU, the amount of CPU processing required by each thread, message traffic, etc.

Application programmers are interested in this type of simulation once they have tuned their program, and are intend to match the program to an idealized system configuration. Such studies are

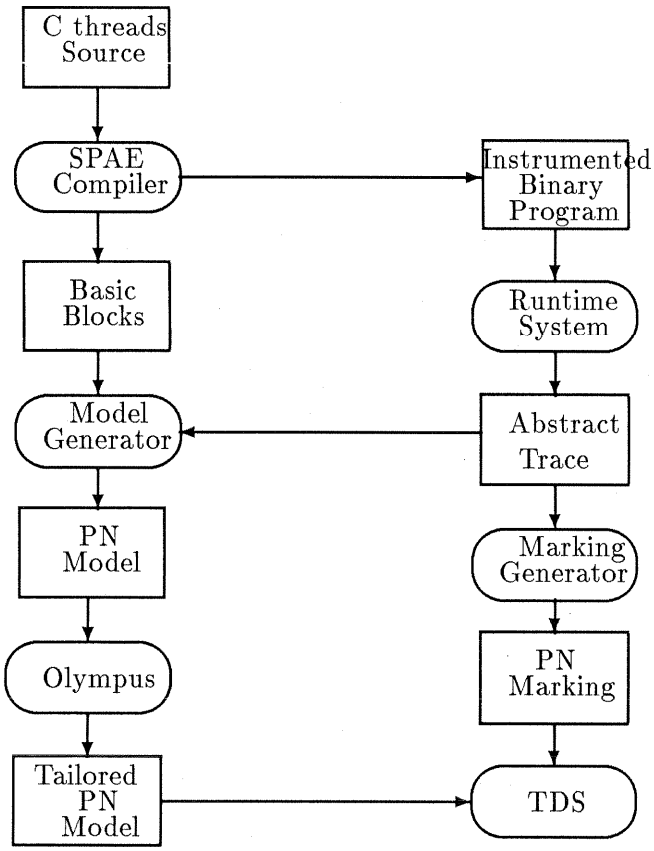


Figure 3: Trace Driven Petri Net Modeling

typically conducted as a part of system purchasing decisions; they can help determine the number of processors to configure into the machine, the amount of cache memory, the amount of disk space, the number and speed of input/output devices, etc.

Issues

1. What is the nature of this class of tools to make their use easy?
2. What is the overlap between these tools and the architecture simulation tools?
3. How do these specific causal traces differ from low level traces?

5 Conclusions

The ppte represents the convergence on a particular set of base facilities (PEET) for obtaining trace data, and on using various parts of PEET to support different trace analyses. PEET is a particularly valuable part of the ppte because of its ability to produce specific causal traces for multiprocessors while executing the ppte in a uniprocessor environment The measurement and modeling tools that we have been developing for many years have been adapted to use PEET specific causal traces, illustrating how one can define a uniform mechanism for composing diverse performance tools into a single ppte.

We believe that the utility of performance tuning tools will depend heavily on the breadth of views that can be offered, and the ease with which the tools can be used. The ppte does not provide any of the tools per se, but it provides the infrastructure for which those tools can be developed with far less effort than if they were to developed without it. In some cases we have built tools that represent this philosophy, (Olympus tools, the ProVis tool) while in other cases our position is speculative, but based on other experiences (high level trace-driven simulators).

Our intent is to continue to use the existing ppte facilities by refining our work in trace-driven control flow (Petri net) simulation, performance visualization tools and toolkits, and in selective high level simulation tools. At this point we believe that performance tuning can only take a major step forward by using a ppte like the one we have built, and then by applying visualization toolkit technology to the environment.

6 Acknowledgements

The various projects that are parts of the parallel program tuning environment have been built by the author and many other people. First, PEET is due to work with Tony Sloane, Dirk Grunwald, Dave Wagner, and Ben Zorn, otherwise known as the Parallel Program Measurement Group. Grunwald and Phillip Farber built the architectural simulators. Olympus and related tools were built with many other people over several years, notably Bruce Sanders, John Hauser, Steve Elliott, Adam Beguelin, Isabelle Demeure, Jeff McWhirter, and Mohammad Amin. ProVis was built by Casey Boyd, Mike Jones, and Mike Thielen. Various parts of this work have been supported by NSF, U S West Advanced Technologies, Bull System Automatique, and others.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprocessing workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] Ann H. Hayes, Margaret L. Simmons, and Daniel A. Reed. *Workshop Summary Parallel Computer Systems: Software Performance Tools.* NSF and Department of Energy, 1991.

[3] Casey Boyd, Mike Jones, Joe Thielen. Visualizing the performance of parallel programs: Interface design using task-centered walkthroughs. Department of Computer Science, University of Colorado, June 1992.

[4] Isabelle M. Demeure and Gary J. Nutt. Collected papers on visa and paradigm. Technical Report CU-CS-488-90, University of Colorado, Department of Computer Science, CB 430, August 1990.

[5] Dirk Grunwald, Gary Nutt, Anthony Sloane, David Wagner, William Waite, and Benjamin Zorn. A testbed for improving the performance of parallel programs and systems. Technical Report CU-CS-512-91, University of Colorado, Department of Computer Science, CB 430, January 1991.

[6] E. A. Brewer, C. N. Dellarocas, C. N. Colbrook, and W. E. Weihl. Proteus: A high performance parallel architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[7] Zulah Karen F. Eckert. A proposal for phd research in complexity of the trace migration problem for parallel programs. PhD Dissertation Proposal, December 1992.

[8] Susan Eggers, David Keppel, Eric Koldinger, and Henry Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.

[9] Susan J. Eggers Eric J. Koldinger and Henry M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Proceedings of the 18th Symposium on Computer Architecture*, 1991.

[10] P. G. Farber. Analysis of a shared bus multiprocessor memory system using trace driven simulation. Master's thesis, University of Colorado, 1991.

[11] Gary J. Nutt, Adam Beguelin, Isabelle Demeure, Stephen Elliott, Jeff McWhirter, and Bruce Sanders. Olympus: An interactive simulation system. In *1989 Winter Simulation Conference Proceedings*, pages 601–611, 1989.

[12] Gary Nutt, Dirk Grunwald, Anthony Sloane, David Wagner, and Benjamin Zorn. A testbed for studying parallel programs and parallel execution architecture. *Proceedings of MASCOTS 93*, April 1992.

[13] Dirk Grunwald. Awesime: An object oriented parallel programming and simulation systems. Technical Report CU-CS-552-91, University of Colorado, Department of Computer Science, CB 430, 1991.

[14] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[15] Michael T. Heath. Performance visualization with paragraph, October 1991.

[16] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[17] James Arthur Kohl and Thomas L. Casavant. Use of paradise: A meta-tool for visualizing parallel systems. In *Proceedings of the Fifth International Parallel Symposium*, pages 561–567, May 1991.

[18] James R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software – Practice and Experience*, 20(12):1241–1258, December 1990.

[19] Allen D. Malony, David H. Hammerslag, and David J. Jablonows ki. Traceview: A trace visualization tool. *IEEE Software*, 8(5):19–28, September 1991.

[20] Mark A. Holliday and Carla S. Ellis. Accuracy of memory reference traces of parallel computations in trace-driven simulation. Technical Report CS-1990-8, Duke University, Department of Computer Science, July 1990.

[21] Jeffrey D. McWhirter and Gary J. Nutt. A characterization framework for visual languages. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 246–248, 1992.

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

[23] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[24] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1992. to appear.

[25] Gary J. Nutt. A simulation system architecture for graph models. *Advances in Petri Nets 1990*, 1990.

[26] Gary J. Nutt. Trace driven simulation of petri nets. Technical Report CU-CS-5xx-91, University of Colorado, Department of Computer Science, CB 430, December 1991.

[27] Cherri M. Pancake. Where are we headed? *Communications of the ACM*, 34(11):53–64, November 1991.

[28] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, 9(1):69–77, January 1992.

[29] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., 1981.

[30] R. Covington, S. Madala, V. Mehta, J. Jump, and R. Sinclair. The rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1988.

[31] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[32] S. Sherman and J. C. Browne. Trace driven modeling: Review and overview. In *Proceedings of Symposium on Simulation of Computer Systems*, pages 201–207, 1973.

[33] Steven Tanimoto, editor. *Proceedings of the 1992 IEEE Workshop on Visual Languages.* IEEE, September 1992.

[34] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan Chung, and Charles F. Fineman. Visualizing performance debugging. *IEEE Computer*, 22(10):38–51, October 1989.

[35] Wen-Hann Wang and Jean-Loup Baer. Efficient trace-driven simulation methods for cache performance analysis. *ACM Transactions on Computer Systems*, 9(3):222–241, August 1991.