A Testbed for Improving the Performance of Parallel Programs and Systems

Dirk Grunwald, Gary Nutt, David Wagner,
William Waite and Benjamin Zorn

CU-CS-512-91 February 1991

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Testbed for Improving the Performance of Parallel Programs and Systems

Dirk Grunwald Gary Nutt David Wagner
William Waite Benjamin Zorn

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

February 1991

Abstract

This report is an expanded version of a proposal submitted to the NSF in November 1990 to be funded starting in June of 1991. In the proposed project, we will study parallel computing systems and the parallel programs that use those systems. To support this study, we need various kinds of trace data from a representative sample of programs running on parallel hardware, tools to collect this data, and a suite of programs characterizing a variety of parallel computations. Ultimately the purpose of the work is to improve the performance of real programs running on real hardware and to allow meaningful predictions about proposed techniques.

We currently have access to an interesting and diverse set of scientific parallel programs and are porting this collection to our testbed environment. We will continue to collect a variety of reference programs during the grant period. As useful trace data, tracing tools, and analysis tools become available, we will distribute the data, tools, and significant reference programs to interested researchers at other sites.

The alternative to a single, concentrated effort in this area is for each researcher to develop a suite of reference programs and data collection tools, and then to collect the appropriate data. That alternative is not only inefficient, but also leads to difficulties in comparing results across research groups.

Contents

1 Introduction

This report describes the planning for a project to capture data about the behavior of parallel programs at a number of levels, making that data and the means of collecting it available to the research community.

1.1 Goals

Our goals are to provide trace data at a variety of levels from a representative sample of programs running on parallel hardware, tools by which such data can be collected, and a suite of programs characterizing a variety of parallel computations. This material is needed as a basis for a wide range of measurement and evaluation tasks, whose ultimate purposes are to improve the performance of real programs running on real hardware and to allow meaningful predictions about proposed techniques.

We believe that a single, concentrated effort should be made in this area. The alternative is for every researcher to develop their own benchmark program suite and data collection tools, and then to collect the appropriate data. That alternative is not only inefficient, but also leads to difficulties in comparing results across research groups. Our own group is an example of this problem. Each of us is either currently conducting work based on the kind of material we are proposing to collect, or is contemplating such work. This proposal consolidates the effort needed to support our projects.

In the remainder of this section we briefly justify each of the specific products mentioned above:

- A suite of representative parallel programs.
- Tools for generating and analyzing traces.
- Traces of program behavior at several levels.

Memory system researchers have relied heavily on traces, first to study interleaved memories, e.g., see [68]; and then virtual memory systems, e.g., see [12, 58]. Today, traces are used heavily for comparing the performance of different memory hierarchy systems, especially those involving memory caches, e.g., see [66, 6, 2, 15, 67, 23].

OS file systems, database management systems and I/O systems have all benefited from trace-driven evaluation. Traditionally, event traces have been used for a variety of such performance comparisons cite cheng:traces,sherman:cacm72,sherman:sscs73,sherman:sscs76. Our own group has used tracing for system performance studies in the past [49], and we currently use this approach in evaluating programming language runtime systems (Zorn [81]) and multiprocessor scheduling algorithms (Grunwald [26]). If a collection of large, detailed event traces from significant parallel applications is made widely available to researchers in diverse fields, the overall benefit of a common set of observations of parallel program behaviors should be considerable.

A particular event trace represents a view of a single execution of a single program. It may or may not be similar to other executions of the same program or to executions of other programs. Statistical analysis of a number of such traces can be used to establish *signatures* characterizing various kinds of behavior. Because event traces are very large, it is not possible to store many of them for this kind of analysis. Users of event traces must be able to determine whether the appropriate signatures differ significantly from those of their local workload. Thus they need to be able to generate and analyze traces of programs in their workload for validation purposes.

Standard benchmarks are important because they allow meaningful comparison across a variety of experiments. Without such a program suite, it is impossible to know whether different results are significant or are simply artifacts of the experimental data. Benchmarks suite can also be validated over many hardware and software architectures, guaranteeing that all interesting signatures are covered without excessive redundancy.

1.2 Summary of the Research Plan

The project will concentrate on the following tasks:

- Acquire and retarget a variety of parallel applications of general interest.
- Instrument the applications and the underlying system so that appropriate traces can be collected and correlated.
- Package the traces for distribution.
- Generalize the collection and analysis software for a variety of machines.
- Make the application programs portable via preprocessors and/or configuration tools.

These subtasks have been investigated independently by others, and we will use as many of the existing tools and techniques as possible.

Because of the unique nature of the Department of Computer Science at the University of Colorado [59] we currently have access to an interesting and diverse set of scientific parallel programs. Our first major task will be to port this collection, augmented with other publicly-available programs from different application areas, to our testbed environment (initially a 20-processor Encore Multimax system).

The preliminary tracing work will employ only user-level programming techniques including interpretive execution [26, 81], source code instrumentation, and other generally-available tracing tools such as gprof. As a first use of the data, these traces will be evaluated for utility by using them in an ongoing study of process placement on distributed memory architectures (see Section B.1). These projects will also be used to develop appropriate signatures.

As useful trace data continues to become available, it will be distributed to other interested parties. This distribution process will require that we develop standard methods for compressing trace data and making it available to a wide audience via appropriate media.

Trace generation and analysis software will also be distributed as they prove their worth. These programs will almost certainly *not* be parallel, and therefore their portability problems are reasonably well-understood. Nevertheless, most will require more effort than the trace data to distribute in a useful form.

Finally, we will undertake the distribution of significant benchmark programs. While our programs and data are specific to particular environments, it is our intent to construct our benchmarks so that they operate in a relatively platform-independent manner. We have little experience in this area, but we believe that widely-available packages like C threads and standard approaches to portability will allow us to attain this goal.

2 Project Description

The project is part of an ongoing effort to improve the performance of large-scale parallel programs running in a logical shared memory multiprocessor environment. We are looking at the system consisting of the program, compiler and runtime support as a whole. It is an empirical study, in which we must develop a representative set of parallel programs, analyze the execution of those programs in existing environments, and then predict the effects of proposed changes of the environments.

2.1 Objectives

The first task is to obtain appropriate test data. We believe that a single, concentrated effort should be made in this area and the results shared among all interested parties. This will avoid redundant effort and simplify the comparison of results across research groups.

We will use our data to evaluate current operating system designs in the context of high-performance parallel computing. This study will not only shed light on operating system support of parallel computation, but also motivate and validate the data collection process. Our objectives for the project are therefore to:

- Collect representative parallel programs
- Collect trace data describing the execution of those programs
- Construct appropriate tools and methodologies for instrumentation
- Distribute the programs, data, and tools
- Evaluate multiprocessor operating system designs

Each of these objectives is expanded in the following subsections. Our intent in these subsections is to indicate the scope of the work associated with each objective. Section 2.3 explains how the objectives will be achieved.

2.1.1 Representative Parallel Programs

We are focus our attention on programs suitable for *medium grain ensemble systems* [31, 32] – systems composed of numerous processors that may communicate through shared memory or via an interconnection network. SIMD data-parallel programs, systolic algorithms or massively parallel image processing applications will not be as well-represented as medium grain ensemble programs largely for one pragmatic reason: collecting data from such massively data-parallel programs is intrinsically more difficult than for medium grain ensemble systems. We may address such programs after gaining more experience with the current reference programs.

We have classified the medium grain ensemble systems along the following dimensions:

- Application Domain
- Structure of Parallelism
- Communication Mechanism

Application Domain The fundamental driving force behind this work is to be able to achieve high performance of real programs using real operating systems and real parallel hardware. An implicit assumption is that the underlying computational support facilities are a limiting factor in execution of several real programs. We distinguish four different categories of such demanding parallel application programs.

Scientific programs are those used in physics, atmospheric sciences, mechanics and similar areas. We assume that most of these programs are written in FORTRAN (although this is not required), were initially written as sequential algorithms, and were parallelized either by hand or using parallelization tools. We assume that these programs are limited by floating-point computation, input/output and available memory.

Engineering programs are those used in engineering, and in particular, electrical engineering. We differentiate these programs because many are concerned with optimization, and tend to be integer intensive rather than floating-point intensive. The majority of these sample programs are written in C, and are explicitly coded for parallelism.

Computer Science programs are similar to engineering programs, but address a different spectrum of domains including parallel discrete event simulation, optimization problems, and applications in artificial intelligence. We expect that most of the programs will be written in C and are explicitly coded for parallelism.

Research programs are those that are too small to be complete applications, but are still valuable indications of parallel program activity. Examples of these programs are sample implementations of new parallel algorithms or other small "toy" problems demonstrating a particular technique.

Structure of Parallelism We are also concerned the *kind* of parallelism available in each program. In general, scientific programs involving physical models are parallelized by transforming `DO` loops to some parallel form, either `DOALL` or `DOACROSS`. The resulting program can be represented by a *fork-join* graph, showing the dependence of each task on the others. Such programs are said to have *static parallelism* because repeated execution of the same program using the same data produce the same number of tasks and the same fork-join graph. They can be *statically scheduled* by assigning a processor for each task prior to execution, or *dynamically scheduled* by assigning processors at execution time.

Other programs are *adaptively parallel*, changing the available parallelism at execution time. For example, a game-tree search program may choose to decompose the problem differently depending on the number of processors available, their relative speeds, and contention for various memory locations.

Communication Mechanism The two prominent communication models are the *shared memory* and *message passing* models. Applications which are written to employ one mechanism can be supported in a hardware system that naturally caters to the opposite mechanism, e.g., the software in a distributed memory system can mimic shared memory [40], and a shared memory system can employ message passing [1]. The important performance factors relating to communication are the way programs are structured and the way data can be recorded. In certain cases, programs can use either model [10]; these are particularly valuable, because they will allow researchers to compare the effect of different communication models.

Trace Data	Uses
Memory Addresses	Cache studies, process placement studies
Symbolic Memory References	Distributed cache studies, process placement studies, memory hierarchy design
Page References	Virtual memory policies (local and networked), smart runtime systems, database systems
Synchronization Events	Scheduling policy studies, OS design studies
System Calls	OS design studies

Table 1: Summary of data collection and analysis objectives.

2.1.2 Data to be Collected

We intend to record program activity at many different levels of abstraction, including address references (both absolute and symbolic), virtual memory activity, system calls, and synchronization activity. Table 1 summarizes the data collection and analysis objectives of this study. The trace data will provide information on general performance, paging behavior, cache behavior, functional program behavior for parallel applications, function calls, process/thread spawning and destruction, dynamic precedence relationships among processes, resource contention, and program behavior in terms of abstract events. This complete set of measurements is necessary for the study of many system software issues; the remainder of this section briefly characterizes each measurement and its uses.

Memory Address Traces Address traces simply record the instruction and data access addresses developed during program execution. Sequential address traces have long been used to validate cache memory designs, e.g., see [66]. A small number of parallel address traces exist, and have been used to validate *distributed* cache memory designs, e.g., see [2].

Symbolic Memory Reference Traces Symbolic traces provide more information than simple address traces by providing a *symbolic* memory reference as well as a physical memory address. Thus, accesses to array elements in a FORTRAN program would be recorded by the array name and the appropriate subscripts. For example, memory address `0xffff0056` may be a reference to `A(4,47)`.

Currently, symbolic memory reference traces are not widely used because they are difficult to construct. However, we expect them to be very useful for examining distributed cache or network virtual memory, because one can specify the mapping from individual arrays to physical addresses. Automatically partitioning data to compensate for cache line size or page sizes may have a dramatic effect on cache coherence protocols.

Page Reference Traces Page traces are similar to address traces, but at a more coarse level: Only references to individual memory pages are recorded. In practice this is a filtered version of the address traces, but this form is a more convenient and compact representation suitable for certain studies.

We expect these traces to be used in conjunction with operating system traces and synchronization traces, to study the tradeoffs of system paging policies on application behavior.

Synchronization Event Traces These event traces record the interactions of processes in a multiprocess application. It is simple to record the birth and death of each process. Recording additional synchronization events is more difficult because a "high level" synchronization event may be composed of many small synchronization events. For example, when several processes meet at a barrier, each process may first acquire a semaphore that guards the barrier. These latter synchronizations are an artifact of the barrier implementation. Our traces will record only the higher level actions, such as the barrier join in this example.

These traces can be used to determine the effectiveness and appropriateness of extant synchronization primitives. Certain synchronization patterns may emerge that would benefit from specialized operating system or architectural support. In addition, these traces will furnish the correct level of behavioral detail for studies of multiprocessor scheduling policies.

System Call Traces These traces record the interaction of processes with the operating system. For many application domains, particularly engineering and scientific applications, these traces are compact because such applications do not interact extensively with the operating system. System call traces provide information on the I/O activity of a program and indicate what operating system facilities limit the performance of particular applications.

In conjunction with synchronization traces and page reference traces, system call traces can be used to study the effectiveness of operating system policies on I/O allocation, asynchronous I/O, and scheduling operations. More importantly, these traces can be used to study the interaction of processes in new high-performance computing environments.

Profile Data Program profile information is also very helpful in characterizing the behavior of parallel programs. Where trace collection extracts the linear sequence of events that make up a program execution, profiling counts numbers of events and correlates the information with a structured view of the program's execution (such as the dynamic call graph). Profiling can collect CPU usage, synchronization statistics, processor usage and memory usage. Profiling has the advantage of being even less intrusive and generating less data.

2.1.3 Tools

Three requirements on the tools developed for this project are:

- The data must be collected without perturbing the program being monitored.
- The large volume of data generated must be compactly stored or processed on-the-fly.
- Information from different levels of execution must be collected and correlated.

In order to meet these requirements, we plan to utilize existing methods and make modifications or develop new methods if those are not adequate.

We will also cooperate with colleagues to develop a standard format for trace information, thus making it easier for researchers to write programs that evaluate the traces. Dennis Gannon of Indiana University and Sue Utter of Cornell are currently working toward establishing standard trace formats (they organized a session about this subject at the Supercomputing '90 Conference). We plan to actively participate with these researchers in the design and use of standard trace formats.

2.1.4 Distribution

As useful trace data becomes available, we will distribute it to interested parties at other sites. Trace generation and analysis software will also be distributed as it proves its worth. These programs will almost certainly *not* be parallel, and therefore their portability problems are reasonably well-understood. Nevertheless, most will require more effort than the trace data to distribute in a useful form. Finally, we will undertake the distribution of significant reference programs. While our programs and data are specific to particular environments, it is our intent to construct our reference programs to operate in a relatively platform-independent manner. We believe that widely-available packages, such as Mach C threads, and standard approaches to portability will allow us to attain this goal.

2.1.5 Operating System Analysis

We will evaluate current operating system designs with the trace data and propose changes with the specific goal of supporting high-performance parallel computing.

This work will provide validation of the usefulness of our traces. It would be naive to expect that our first attempt at measuring program behavior would provide all that is required for actual performance analyses. By having a concurrent subproject that uses our data, tools and reference programs, we will have a valuable source of feedback to guide the main effort of this project.

In addition, a study of operating system design in this context has considerable intrinsic and extrinsic merit. Not only does it touch upon many issues that are of interest to the computer science operating system community, but also it has the possibility of identifying mismatches between operating systems and programs from the target application domains. These mismatches might be corrected by changing the operating system design or the compilation strategy for the applications.

2.2 Related Work

2.2.1 Address-Level Tracing

Address traces, whether virtual or physical, are the most challenging form of trace data to collect. Because programs generate at least one address reference for every instruction executed, the cost of collecting address reference information from executing programs is very high. While the sheer volume of data complicates the task, the act of tracing itself perturbs the program execution. If this perturbation is significant, it may greatly alter program synchronization or scheduling and render the trace unrepresentative of any actual execution of the program.

Historically, *hardware instrumentation* introduces the least program perturbation [50]. Agarwal et al. used the ATUM system (Address Tracing Under Microcode) to record the address traces of four VAX processors [2]. The machine microcode of a four-processor VAX was modified by Digital Equipment engineers to generate address reference strings, slowing the execution speed by a factor of twenty. This method sufficed for a single, shared bus architecture. It is impractical for hierarchical memory architecture, where no single interrupt can halt all processors simultaneously. In this case, a *hardware monitor* is needed. Malony et al. directly recorded the address references of two clusters in the CEDAR system using hardware probes [43]. This did not slow program execution, but collecting and storing the traces presented a daunting task in data acquisition.

Alternate approaches instrument the executing program itself. As the program executes, it performs its normal tasks and also records address references. Borg et al. describe a method for modifying the executable code to extract address traces from programs executing on the Titan processor [15]. The Titan work is significant because the traces collected include a "seamless" mix of references from different programs and the kernel. A major aspect of the Titan approach was to minimize the interference of reference collection on program execution. As a result, the overhead of trace collection was further reduced—program execution was dilated by a factor of eight to ten. The Titan group also used *on-the-fly simulation*, in which the address trace is consumed directly by a simulation. This results in cache analyses based on billions of addresses, rather than the tens of millions common at that time. This work showed that very long address traces provide significant information that cannot be determined in any other way.

Stunkel and Fuchs describe the more general approach taken by the TRAPEDS system, in which the executable code was modified to produce address traces during execution [67]. While the TRAPEDS overhead is as large as the ATUM overhead (again a factor of twenty), it forms the basis of more efficient address collection methods for parallel programs. The TRAPEDS approach is also an example of on-the-fly simulation. Because a number of processes in a hypercube were being traced, the simulations using the data were executed concurrently with the applications generating the trace.

Two recent research projects modify the approach taken by Borg et al. to further reduce data collection overhead. Both of these approaches take the position that traces from a single user program are of interest, and they do not trace execution within the kernel. In the MPtrace project the executable file is pre-processed, adding trace generation code only where necessary [23]. This technique greatly reduced the overhead of trace collection—program execution was slowed by a factor of only three. MPtrace is also significant because multiprocessor traces were collected using this technique.

Larus further reduces trace collection overhead using a technique called AE (Abstract Execution) [37]. With AE, the compiler is modified to produce two object files. The first is the program, with small modifications. The second is an abstract version of the program that, when executed, will produce exactly the same reference stream as the original program (hence the term abstract execution). When the original program runs, it produces a small data file that contains address references that could not be predicted in the abstract execution (e.g. certain data references). The resulting file is directed into the abstract version of the program which can then duplicate the reference behavior exactly.

The AE approach to trace collection is significant in two ways. First, the amount of trace data collected is reduced significantly (by a factor of 50–500). Second, the execution overhead imposed by AE is quite small, dilating the execution time by a factor of only 2–3. We plan to use the abstract execution model of trace collection to collect traces at various levels of execution.

While traces of virtual address references can be used to measure the performance of cache and virtual memory implementations, higher-level traces are also needed. Brewer, Dongarra and Sorenson developed the MAP tool [16] as a visual aid for programmers attempting to increase memory locality in **Fortran** programs designed for hierarchical cache architectures. This tool annotated a **Fortran** program to automatically emit a trace of array references. **ForTrace** performs a similar source to source translation on **Fortran** programs [69]. The annotated program directly records every symbolic address reference in the program. **ForTrace** extends **Map**, because it also simulates the parallelism in `DOALL` and `DOACROSS` loops.

Symbolic data reference information is used to study possible reorganizations of the program objects in memory. For example, Zorn's thesis investigated the performance of different garbage collection algorithms using symbolic data references [81]. Grunwald's thesis uses symbolic data references to investigate the communication costs in multiprocessors [26].

2.2.2 Higher-Level Profiling and Tracing

Gprof is a program execution profiler that samples the program counter to provide information about where a program spends its time [25]. Statistical sampling is one technique that we will consider to reduce the amount of data collected.

Mprof is tool that relates program memory allocation to the dynamic call graph, indicating what subroutines were truly responsible for allocation [82]. Mprof is interesting because it maps allocation information exactly onto the dynamic call graph, whereas gprof does only a probabilistic mapping.

Parasight is a low-overhead tool that can be used to dynamically instrument parallel programs on the Encore Multimax [5]. Parasight can be used to gather coarse-grained execution information, much as the gprof does for sequential Unix programs. It can also be used to gather much finer-grained profiles using a dynamically linked function profiler.

Quartz is a profiling tool for parallel program performance debugging developed at the University of Washington [4]. Using a combination of existing compiler support (for gprof) and an instrumented thread library, it keeps track of *normalized processor time*, which is the time spent executing a routine normalized to account for the level of parallelism displayed by the program during execution of that routine. The claim is that normalized processor time is a useful metric for identifying bottlenecks in parallel program performance.

IPS-2 is a second generation of a performance measurement system written by Miller et al. at the University of Wisconsin [44]. Miller uses a combination of function call counting (as in gprof) and an instrumented runtime system to collect function-call level profiles of parallel programs. This data is made available to the user through a sophisticated user interface that provides views of the performance at different levels of granularity.

IPS-2's fundamental goal, which is to provide information to the programmer so that he can improve the performance of his program, is complementary to our own objectives, which are to provide information to operating system, compiler, and runtime system designers to better match those systems to the applications that use them. Because of this difference in goals, however, the data collected by IPS-2 are a subset of the data we are proposing to collect. This difference also means that there is no benchmarking component of the IPS-2 project.

The *Crystal* package from the University of Illinois records all communication in a message passing application running on an Intel iPSC/2 [62, 63].

Finally, CASPER is an extremely general-purpose trace collection mechanism for UNIX System V [8]. CASPER is most notable for its ability to intermingle trace data from the user and kernel levels; this is accomplished by implementing it as a UNIX device driver.

2.2.3 Operating System Design

Needless to say, a great many operating system design decisions have been motivated by measurement studies of real programs in execution; unfortunately, in most cases these measurement studies have not been given as much attention in the literature as the systems that were spawned by them. In this section we mention a representative sample of such systems.

The Accent system's *copy on write* mechanism, which was later adopted by Mach, was motivated by measurement studies of interprocess communication patterns [7, 24].

The design of the Sprite network file system protocol was based on a study of UNIX file system call traces [54].

Bershad's LRPC (lightweight remote procedure call) service was motivated by a study of RPC call patterns, which revealed that an overwhelming fraction of all remote procedure calls contain a very small number of simple data types as parameters [13].

Finally, the CASPER tool mentioned in the previous section has been used to compare the performance of the *VM* and *region* approaches to virtual memory management in UNIX System V [19]. This study was used to guide the design of the virtual memory management implementation of UNIX System V Release 4.0.

2.3 Research Plan

We have conceived of a four-year plan to meet the specific objectives discussed in §2.1. The three fundamental parts of the research are:

- collecting and standardizing reference programs.
- using traces derived from these programs to analyze system behavior.
- distributing trace data, analysis tools and reference programs to the research community.

We stated five specific objectives for the work in Section 2.1. In this section, we present our research program. Each subsection provides a general discussion of the approach to be used to address objectives, the resources that we intend to devote to realizing that objective, and the time period in which the work is to be carried out.

2.3.1 Collecting Representative Parallel Programs

Because of the unique nature of the Department of Computer Science at the University of Colorado [59], we currently have access to an interesting and diverse set of scientific parallel programs. Our first major task will be to port this collection, augmented with other publicly-available programs from different application areas, to our testbed environment.

In §2.1.1 we described how we classify programs according to Application Domain, Structure of Parallelism and Communication Mechanism. Table 2 gives examples of parallel programs in each Application Domain that we intend to use. *Note that Table 2 is not intended to be exhaustive.* Each reference program is described in more detail in Appendix A.

We are acquiring these programs and porting them to our Encore Multimax (20 NSC 32032 processors) as a testbed for establishing the suite of reference programs. We will also begin our user level trace data gathering, and analyze that data to the extent possible. This work is currently being pursued as an unfunded project.

Program Name	Language	Communication Model	Source
Scientific Applications			
Perfect Club	FORTRAN	Shared Memory	University of Illinois, Center for Supercomputer Research and Development
Shallow	FORTRAN	Shared Memory, Distrib. Memory	National Center for Atmospheric Research
FORCE Suite	FORCE	Shared Memory	University of Colorado
Finite Element	FORCE	Shared Memory	University of Colorado, Aerospace
Weather & Pollution	FORTRAN	Shared Memory	Environmental Protection Agency
Engineering Applications			
Pthor	C	Shared Memory	Stanford University
LocusRoute	C	Shared Memory	Stanford University
Puppy	C	Shared Memory	University of California, Berkeley.
MaxFlow	C	Shared Memory	Stanford University
MinCut	C	Shared Memory	Stanford University
Computer Science Applications			
Mul-T	C	Shared Memory	MIT and DEC Cambridge Research Lab
Synapse	C++	Shared Memory	University of Washington
VM_pRAY	C	Distrib. Memory	IRISA, Rennes, France
Research Applications			
Presto Suite	C++	Shared Memory	Rice University

Table 2: Summary of Reference Programs with Static Parallelism

2.3.2 Tools

The objective of §2.1.2 requires collecting data from many different levels of program execution. We plan to use existing techniques in cases where they are suitable, and enhance or rework these approaches if they are not adequate for our purposes. In the remainder of this section we indicate the approach we plan to take when collecting each of the kinds of data described in §2.1.2.

Memory Address Traces Because of the generality and storage compactness of the Abstract Execution approach described in §2.2, our initial tool development will extend the AE package. Our variant, SPAE[1] will use Symbolic and Parallel Abstract Execution. Our goals are to:

- Record direct address traces of parallel applications, executing in a known reference environment on a fixed number of processors.
- Be able to generate symbolic address traces for parallel programs.
- Measure *architecture-independent* parallelism when possible.

The first component, *parallel abstract execution*, will function similarly to the existing AE system, but allow the tracing of parallel applications. We expect that the instrumentation interference introduced by SPAE will be small because the existing AE has a small execution dilation (2–3 times normal execution). The data currently collected by AE includes the instruction reads and data read/writes. Our modifications will also record global timestamps on basic block entry so that we can synchronize data from different processors, and thread context switch events so that we can trace the execution of logical threads rather than the execution of larger-grained processes. For certain long-running programs, the trace compression of AE will probably not suffice. To solve this problem, we will modify SPAE to support on-the-fly execution of a simulation while the program trace is being gathered. This involves executing the program schema whenever data is available, rather than collecting the data for later execution.

Symbolic Memory Reference Traces Symbolic data references can be collected just as virtual memory references are, except that the program source must be analyzed when the tracing code is being inserted to correctly attribute each reference to the proper program variable. Small modifications to the current AE system will make this possible. Other tools collect symbolic data references, including the FORTRACE system [69] for FORTRAN programs and Zorn's MARS system [81] for Lisp programs.

SPAE also will use *symbolic abstract execution* to address the needs of symbolic execution. SPAE must allow remapping of program data, to reflect repartitioning of program arrays, and we will pursue two implementations:

- The first strategy converts the data reference stream to actual symbolic addresses using the program symbol table associated with the instrumented program. This can be done each time the trace is "replayed," obviating the large storage associated with a symbolic trace. This method will not capture all symbolic references (such as references to dynamic arrays), but will capture symbolic references to all identifiers available in the symbol table of the traced programs.

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

- The second strategy is less flexible, but significantly more efficient. In the first method, we implement remapping by converting data references to symbolic form and then mapping those symbolic forms to new virtual addresses. Instead, we can directly map the raw data references to new virtual addresses, with a user specified mapping function. The mapping function would be pre-computed using program symbol table information. This method will suffice for studies interested in remapping data structures, while not precisely identifying each reference to a particular data structure. This remapping process should be relatively efficient.

Both methods require information in excess of that provided by most symbol tables, such as array dimensions and information on automatic variables. This data, available from the program source code, can be gathered either automatically, through extensions to the compiler, or by hand, since it needs to be collected only once: at the time the program is added to the reference suite. Neither symbolic tracing technique will be adequate for all programs; some programs use *unnamed* storage, such as that allocated dynamically. However, we expect symbolic tracing to be of little interest for those programs.

Another use of SPAE will be to remove architectural parallelism constraints for certain FORTRAN programs. The semantics of two common parallel loop constructs, `DOALL` and `DOACROSS` are satisfied by sequential iteration execution. We can simulate unbounded loop-level parallelism for `DOALL` loops by recording the time at the beginning of the entire loop. The time of beginning of each iteration would be reset to this recorded time, and the references for *all iterations* would be collected and sorted by the time stamp. A similar method would be used for `DOACROSS` loops, but the times for each iteration would be affected by the time of the corresponding *post* and *wait* events that synchronize dependent iterations. (A similar technique is used in the FORTRACE package.) Because SPAE is based on AE, which accepts only C programs, we expect to modify a version of **F2C**, a FORTRAN to C translator, to recognize the `DOALL` and `DOACROSS` loops, emitting a C program suitable for processing by SPAE.

Synchronization Event Traces We plan to record all process creation, destruction, and synchronization events. The most difficult aspect of collecting this information is that different reference programs use different routines for multiprocessing and synchronization—some programs use operating system-provided routines while others use library packages implemented on top of the operating system. We plan to instrument the most commonly used thread libraries, such as the Argonne parallel library [31]. This approach has the advantage that anyone using these libraries can benefit from our instrumentation, if they choose to. Later, as part of instrumenting the kernel, we will collect information from kernel-provided multiprocessing routines.

One concern is that certain parallel programs will be tailored to execute on a fixed number of processors. Since we would like to be able to generate traces that are independent of the number of processors actually used, we would like to eliminate such dependencies, or at least note when they occur. To accomplish this task, we plan to hand-instrument these reference programs. Such hand-instrumentation, in combination with the instrumentation of the parallel libraries, will allow us to collect very detailed traces of the parallel behavior of the reference programs.

For message passing applications, we hope to use extant tools, such as the University of Illinois' Crystal package [62, 63]. In some cases, where synchronization or communication does not depend on the number of physical processes, we may be able to simulate the execution of the Intel iPSC/2, providing more detailed information than is provided by Crystal.

System Call Traces The easiest way to collect traces of the interaction between the user program and operating system would be to modify the user library routines to add information to a trace file each time a system call is about to be performed. For example, the C *open* function could be modified, appending a record to the trace buffer each time it is called. The method is simple and has the advantage that no modifications to the kernel are required. A second alternative is to instrument the kernel routines directly. This approach has numerous advantages. Kernel behavior, such as page faults, which are not part of the system call interface, could also be measured. Furthermore, we could record the interactions between processes, allowing us to study issues affecting resource sharing. For either library instrumentation or system instrumentation, traces from long-running programs may be problematic.

A third alternative would be to process the object file to produce an abstract representation of its system call behavior, much as AE currently abstracts its memory reference behavior. The abstracted program would be compact and capable of generating very long system call traces. By recording time stamps and cross correlating the traces, we can also approximate a record of process interactions. This method is also of interest because we want to evaluate the effectiveness of applying the basic idea of abstract execution to many different levels of trace collection.

Profile Data There have been several notable approaches to the gathering of profile information, such as *gprof* [25], *mprof* [82], *Quartz* [4], and *Parasight* [5], and we intend to use these tools unchanged. These tools are briefly described under related work (§2.2).

2.3.3 Distribution

Three problems must be solved in order to effectively distribute trace data, analysis programs and reference programs:

- The amount of data represented by many traces is physically large
- Analysis programs must run on a wide variety of hardware
- Reference programs must run on a wide variety of parallel hardware

A "small trace" (e.g. a raw address trace representing 1-5 seconds of execution) will typically occupy about 5 megabytes, while larger traces could require more than 2 gigabytes. Abstract execution might reduce these sizes by a factor of 100. This range of sizes implies a range of distribution media. We plan to use anonymous FTP over the Internet to distribute traces up to about 10 megabytes in size, and magnetic tape for larger data sets. Industry-standard 9-track tape at 6250bpi can store about 150 megabytes on a 2400' reel, and this capacity can also be reached by a DC6150 cartridge using QIC150 format. We are currently capable of writing both of these kinds of tapes, and most installations that would be interested in our data can read one or the other. Many installations, ourselves included, now use Exabyte tape drives for normal disk backup. These drives write cartridges that hold 2.2 gigabytes and therefore could be used for the large traces. As discussed in §2.1.3, we intend to cooperate with colleagues to develop a standard representation for trace data.

As the resource collection grows, we see the need for using the NSFNET to distribute traces, tools, and reference programs. Part of our proposal is to obtain funding to be able to gain access to NSFNET. In this case, we would again use anonymous FTP (or similar tools) to accomplish distribution.

Standard techniques for increasing program portability [46] should suffice to make the analysis programs available on a wide variety of hardware. The reference programs present a more challenging problem because many of them depend on non-standard support packages. To make such programs portable, we must either make the support packages portable or make the programs adaptable [56] with respect to support packages. Both of these approaches are available to us, and we will decide which to use on a case-by-case basis.

Various forms of trace data have begun to be collected. Initially, we have used relatively unsophisticated tools, since we have not yet begun the tool development research. As the tracing tools are developed the nature of the trace data will be more comprehensive, as described earlier in the report.

The efforts for data collection and distribution will become production-oriented after the first year of the project. The research in later years will result in new programs, tools, and data but the mechanism for distribution will be relatively standardized.

The porting effort for reference programs will begin at the beginning of the second year of the project. Some standardization will be attempted as the reference programs are collected, but during the second and third years we will expend considerable effort to convert selected reference programs to a portable computational platform.

2.3.4 Operating System Analysis

Our measurements will provide an enabling technology for studies of new operating system design methodologies, and the choices necessary to target an operating system towards a specific application class. In this particular case, we intend to use simulation models, trace-driven or derived from our measurement data, to answer questions such as:

- Where is extra operating system implementation effort most likely to result in increased application performance? Some examples of services that have shown promise of performance benefits from being tailored to a particular application or class of applications include remote procedure call [13], scheduling [64], and virtual memory management [1].

- Would the movement of some services typically associated with the operating system kernel into user space improve performance? Alternatively, how could the interface between operating system mechanism modules and user-level policy modules be improved? This desirability of doing this has been argued before, most notably by the Hydra [76], Eden[3], and Mach [1] systems, but their motivation was flexibility, not performance.

- Do certain parts of the operating system need to interact more than they typically do now? As an example, it will be possible to characterize the page reference behavior of a process immediately before and after synchronization points, perhaps suggesting ways in which the virtual memory and synchronization subsystems of the operating system should cooperate. (If synchronization were performed outside the kernel, then this would point out what parts of the virtual memory subsystem needed to be exposed outside the kernel as well.)

- Would it be advantageous for certain *hardware* exceptions to be handled directly by user code? The idea is to give user policy modules direct control, although for security, some of the mechanism needed to implement those policies might still be in the operating system. Nevertheless, this would be more efficient than vectoring to the operating system, transferring control to the user policy module, and then transferring control back to the operating system. Plus, in some instances it might not be necessary to involve the operating system at all, leading to significant overhead reduction. For example, dynamically-typed languages such as Lisp can use an efficient trap mechanism to implement features such as generic arithmetic [33]. Our measurements would show which kinds of trap exceptions common enough to merit this treatment.

Such questions take on more than merely academic significance with the advent of highly configurable operating systems such as Choices[18], where policies and mechanisms can be chosen at the time the kernel is configured.

The operating system analysis develops results based on the tracing work, and is an underlying motivator for the entire project. This research provides considerable feedback to the earlier phases, so it begins at a relatively low level of effort early in the project and grows to a major effort in the latter part of the work.

The measurements obtained in this study will also be used by the individual researchers to to further there own specific research interests. In Appendix B, the related research projects of several of the investigators are briefly reviewed.

3 Impact of the Proposed Research

The proposed research will have a large impact on a number of areas in computer science and high-performance computation. The measurement data generated by this study will drive research projects in memory cache, distributed system, operating system, compiler, and runtime system design. This research will pave the way for making inexpensive, medium-grained ensemble systems a competitive alternative to special-purpose supercomputers for scientific computation.

The distribution of reference programs and measurement data will obviate the need for every researcher to establish and measure his or her own benchmarks. More importantly, the "science" in Computer Science has sometimes been criticized for its lack of carefully controlled experimental conditions and repeatability of results by different researchers. By establishing a standard collection of portable parallel programs and traces, we will take an important step toward remedying this situation.

Despite a large number of earlier studies of program behavior, we have found no study that has done all of the following:

- Traced a significant number of programs.
- Traced applications at a variety of levels.
- Cross-correlated events between traces of distinct entities.
- Emphasized the distribution of program traces.

Furthermore, there exist few studies that address *any* of these points with regard to parallel and scientific programs.

The parallel program testbed project is an ambitious undertaking supported by five principal investigators (authors of this technical report) and their students and colleagues. The project is expected to extend for four years, ultimately culminating in a continuum of applied studies of system performance. At the time of the writing, the investigators are actively involved in the research and in seeking funding to support the program.

4 Bibliography

References

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. In *Proc. 1986 Usenix Summer Conference* (1986), Usenix Foundation, pp. 93–112.

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

[3] Almes, G., Black, A., Lazowska, E., and Noe, J. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering SE-11*, 1 (Jan. 1985), 43–58.

[4] Anderson, T., and Lazowska, E. Quartz: A Tools for Tuning Parallel Program Performance. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Boulder, CO, May 1990), ACM, pp. 115–125.

[5] Aral, Z., and Gertner, I. Non-Intrusive and Interactice Profiling in Parasight. In *Proc. ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems* (July 1988), ACM, pp. 21–30.

[6] Archibald, J., and Baer, J. L. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems 4*, 4 (November 1986), 273–298.

[7] Ball, J., Burke, E., Gertner, I., Lantz, K., and Rashid, R. Perspectives on Message-based Distributed Computing. In *Proc. 1979 Networking Symposium* (New York, 1979), IEEE.

[8] Barkley, R., and D.Chen. CASPER the Friendly Daemon. In *Proc.Ṡummer 1988 USENIX Conference* (San Francisco, CA, June 1988), USENIX Association, pp. 251–261.

[9] Beguelin, A. *Deterministic Parallel Programming in Phred.* PhD thesis, University of Colorado, Department of Computer Science, Boulder, Colorado 80309-0430, 1990.

[10] Beguelin, A., and Nutt, G. A visual parallel programming language. Tech. rep., Department of Computer Science - University of Colorado, Boulder, October 1990. Submitted for publication.

[11] Beguelin, A. L., and Nutt, G. J. Collected papers on phred. Tech. rep., Department of Computer Science - University of Colorado, Boulder, January 1991. Technical Report No. CU-CS-511-91.

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

[13] Bershad, B. *High Performance Cross-Address Space Communication.* PhD thesis, University of Washington, Seattle, WA, June 1990. Available as Department of Computer Science and Engineering Technical Report No. 90-06-02.

[14] Bershad, B., Lazowska, E., and Levy, H. PRESTO: A System for Object-Oriented Parallel Programming. *Software Practice and Experience 18*, 8 (Aug. 1988), 713–732.

[15] 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 Labortory, Palo Alto, CA, Sept. 1989.

[16] Brewer, O., Dongarra, J., and Sorensen, D. Tools to aid in the analysis of memory access patterns for fortran programs. Technical Memorandum ??, Argonne National Labs, Mathematics and Computer Science Division, June 1988. (available via netlib).

[17] CALLAHAN, D., AND KENNEDY, K. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing 2*, 2 (1988).

[18] CAMPBELL, R., RUSSO, V., AND JOHNSTON, G. The Design of a Multiprocessor Operating System. In *Proc. USENIX C++ Workshop* (1987).

[19] CHEN, D., BARKLEY, R., AND LEE, T. P. Insuring Improved VM Performance: Some No-Fault Policies. In *Proc. Winter 1990 USENIX Conference* (Washington, D.C., Jan. 1990), USENIX Association, pp. 11–22.

[20] DEMERS, A., WEISER, M., HAYES, B., BOEHM, H., BOBROW, D., AND SHENKER, S. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages* (January 1990), pp. 261–269.

[21] DEMEURE, I. M. *A Model, ParaDiGM, and a Software Tool, VISA, for the Representation, Design and Simulation of Parallel, Distributed Computations.* PhD thesis, University of Colorado, Boulder, August 1989.

[22] DEMEURE, I. M., AND NUTT, G. J. Collected papers on visa and paradigm. Tech. rep., Department of Computer Science - University of Colorado, Boulder, August 1990. Technical Report No. CU-CS-488-90.

[23] 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).

[24] FITZGERALD, R., AND RASHID, R. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems 4*, 2 (May 1986), 147–177.

[25] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. An execution profiler for modular programs. *Software—Practice and Experience 13* (1983), 671–685.

[26] GRUNWALD, D. C. *Heuristic Load Distribution in Circuit Switched Multicomputer Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1989.

[27] GRUNWALD, D. C., NAZIEF, B. A. A., AND REED, D. A. Empirical comparison of heuristic load distribution in point-to-point multicomputer networks. In *Proceedings of the Fifth Distributed Memory Computing Conference* (1990).

[28] GRUNWALD, D. C., AND REED, D. A. Benchmarking hypercube hardware and software. In *Hypercube Multiprocessors* (1987), M. Heath, Ed., Society for Industrial and Applied Mathematics, pp. 169–177.

[29] HALSTEAD, JR., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 501–538.

[30] HUDAK, P. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming* (Cambridge, MA, June 1986), pp. 351–363.

[31] JAMIESON, L. H., GANNON, D. B., AND DOUGLAS, R. J., Eds. *Characterizing Parallel Algorithms.* MIT Press Series in Scientific Computation. MIT Press, 1987, ch. 4.

[32] JAMIESON, L. H., GANNON, D. B., AND DOUGLAS, R. J., Eds. *Programming Paradigms for Nonshared Memory Parallel Computers.* MIT Press Series in Scientific Computation. MIT Press, 1987, ch. 1.

[33] JOHNSON, D. Trap architectures for Lisp systems. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France, June 1990), pp. 79–86.

[34] KALÉ, L. V., AND SHU, W. The Chare-Kernel language for parallel programming: A perspective. Tech. Rep. UIUCDCS-R-89-1451, University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 W. Springfield, Urbana, Il, May 1989.

[35] KESSLER, R., AND LIVNEY, M. An analysis of distributed shared memory algorithms. Tech. Rep. TR-825, University of Wisconsin, Dept. of Computer Science, Feburary 1989.

[36] KUCK, D. J., AND PADUA, D. A. High-speed multiprocessors and their compilers. In *Intl. Conference on Parallel Processing* (August 1979), IEEE, pp. 5–16.

[37] LARUS, J. Abstract execution: A technique for efficiently tracing programs. Tech. Rep. 912, University of Wisconsin, Dept. of Computer Science, Feburary 1990.

[38] LEUTENEGGER, S., AND VERNON, M. The Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1990), pp. 226–236.

[39] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (1986), pp. 229–239.

[40] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*, 4 (November 1989), 321–359.

[41] LI, K., AND SCHAEFER, R. A hypercube shared virtual memory system. In *The 1989 Proceedings of the Intl. Conference on Parallel Processing* (1989), pp. I–125 – I–132.

[42] MAJUMDAR, S., EAGER, D., AND BUNT, R. Scheduling in Multiprogrammed Parallel Systems. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1988).

[43] 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, October 1989.

[44] MILLER, B., CLARK, M., HOLLINGSWORTH, J., ET AL. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), 206–217.

[45] MOHR, E., KRANZ, D., AND ROBERT HALSTEAD, J. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France, June 1990), pp. 185–197.

[46] MUXWORTHY, D. T., Ed. *Programming for Software Sharing.* D. Reidel, Dordrecht, Holland, 1983.

[47] NELSON, R. A Performance Evaluation of a General Parallel Processing Model. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1990), pp. 13–26.

[48] NELSON, R., AND TANTAWI, A. Approximate Analysis of Fork/Join Synchronization in Parallel Queues. *IEEE Transactions on Computers 37* (1988), 739–743.

[49] NOE, J. D., AND NUTT, G. J. Validation of a trace-driven cdc 6400 simulation. In *AFIPS Proceedings of the Spring Joint Computer Conference* (1972), vol. 40, pp. 749–757.

[50] NUTT, G. J. Computer systems monitors. *IEEE Computer 8*, 11 (November 1975), 51–61.

[51] NUTT, G. J. A Simulation System Architecture for Graph Models. In *Advances in Petri Nets '90*, G. Rozenburg, Ed. Springer Verlag, 1990. To appear.

[52] NUTT, G. J., BEGUELIN, A., DEMEURE, I., ELLIOTT, S., MCWHIRTER, J., AND SANDERS, B. Olympus: An Interactive Simulation System. In *1989 Winter Simulation Conference* (Washington, D.C., December 1989), pp. 601–611.

[53] OUSTERHOUT, J. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd Intl. Conf. on Distributed Computing Systems* (Oct. 1982), pp. 22–30.

[54] Ousterhout, J., Costa, H. D., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles* (Orcas Island, WA, December 1985), ACM, pp. 15–24.

[55] Polychronopoulos, C. Multiprocessing versus Multiprogramming. In *Proc. Int. Conf. on Parallel Processing, Volume II* (1989), pp. 223–230.

[56] Poole, P. C., and Waite, W. M. *Portability and Adaptability*, vol. 30 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1973, pp. 183–277.

[57] Powell, M., and Miller, B. Process migration in DEMOS/MP. In *Proc. 9th ACM Symp on Op. Sys. Principles* (1983), pp. 110–110.

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

[59] R. Schnabel, e. a. Effective Use of Parallel and Distributed Computing. Tech. rep., Department of Computer Science - University of Colorado, Boulder, 1989. NSF CISE II Proposal.

[60] Ramkumar, B., and Kalé, L. V. Compiled execution of the reduce-or process model on multiprocessors. Tech. Rep. UIUCDCS-R-89-1513, University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 W. Springfield, Urbana, Il, May 1989.

[61] Rosing, M., and Schnabel, R. An overview of DINO—a new langauge for numerical computation on distributed memory multiprocessors. Tech. Rep. CU-CS-385-88, University of Colorado, Department of Computer Science, Boulder, CO, March 1988.

[62] Rudolph, D. C. Performance Instrumentation for the Intel iPSC/2. Master's thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, July 1989.

[63] Rudolph, D. C., and Reed, D. A. CRYSTAL: Operating System Instrumentation for the Intel iPSC/2. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, March 1989).

[64] Scott, M., LeBlanc, T., and Marsh, B. Multi-Model Parallel Programming in Psyche. In *Proc. ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Mar. 1990), ACM, pp. 70–78.

[65] Sevcik, K. Characterizations of Parallelism in Applications and Their Use in Scheduling. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1989), pp. 171–180.

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

[67] 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.

[68] Terman, F. W. A study of interleaved memory systems by trace driven simulation. In *Proceedings of Symposium on Simulation of Computer Systems* (1976), pp. 3–9.

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

[70] Towsley, D., Rommel, C., and Stankovic, J. Analysis of Fork-Join Program Response Times on Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 3 (July 1990), 286–303.

[71] Tucker, A., and Gupta, A. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proc. Twelfth ACM Symposium on Operating Systems Principles* (Dec. 1989), ACM, pp. 159–166.

[72] WAGNER, D. *Conservative Parallel Simulation: Principles and Practice.* PhD thesis, University of Washington, Seattle, WA, 1989. Available as Department of Computer Science and Engineering Technical Report 89-09-03.

[73] WAGNER, D. A Methodology for the Evalution of Multiprocessor Scheduling Policies. NSF Proposal, 1990.

[74] WAGNER, D. Discussion of interaction between operating systems and runtime systems for parallel applications. Private communication, August 1990.

[75] WOLFE, M. Semi-automatic domain decomposition. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers And ApplicationsVolume I* (1989), ACM.

[76] WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C., AND POLLACK, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM 17*, 6 (June 1974), 337–345.

[77] ZAHORJAN, J., LAZOWSKA, E., AND EAGER, D. Spinning Versus Blocking in Parallel Systems with Uncertainty. In *Proc. International Symposium on Performance of Distributed and Parallel Systems* (Dec. 1988).

[78] ZAHORJAN, J., LAZOWSKA, E., AND EAGER, D. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. Tech. Rep. 89-07-03, University of Washington, Seattle, WA, July 1989.

[79] ZAHORJAN, J., AND MCCANN, C. Processor Scheduling in Shared Memory Multiprocessors. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1990), pp. 214–225.

[80] ZAYAS, E. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (Austin, TX, November 1987), ACM, pp. 13–24.

[81] ZORN, B. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.

[82] ZORN, B., AND HILFINGER, P. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer 1988 USENIX Conference* (San Francisco, CA, June 1988).

A Reference Program Descriptions

Perfect Club is a suite of FORTRAN programs intended to be representative of actual commercial, scientific or engineering programs. As distributed, the programs are written in standard, sequential FORTRAN. We will use publicly available parallelization tools such as ParaScope and Parafrase-II, or commercial tools such as KAP, to produce parallel versions of the Perfect Club suite.

Shallow is a FORTRAN program that models shallow water equations using finite difference methods. The shallow water code was originally written by P. Swartztrauber of NCAR; we will use a parallel version due to R. Sato, also of NCAR. The program has appeared in the book, "Multiprocessing in Meteorological Models," eds. G. R. Hoffman, and D. F. Snelling, Springer-Verlag 1988, pp. 125-196.

FORCE Suite is a collection of programs collected by Harry Jordan at the University of Colorado. They represent a broad range of kernels of scientific programs which have been modified for parallelism. The FORCE language was developed by Dr. Jordan to provide a portable parallel programming language for scientific applications; it is a superset of FORTRAN.

Finite Element Solver is a large (100,000+ lines) FORCE program written by Dr. Charbel Farhat at the University of Colorado at Boulder. We will use the version developed for the Cray supercomputer. Other versions exist for the Intel iPSC/2 and Connection Machine.

Weather & Pollution are a set of FORTRAN programs that model weather conditions and pollution dispersal. These models are used by the Environmental Protection Agency to predict effects of acid rain and dispersion of other pollutants. These are a suite of sequental FORTRAN programs that will be parallelized.

Pthor is a parallel, distributed time event-driven logic simulator written in the C language by Larry Soule at Stanford. It has frequently been used as a reference parallel program in studies of distributed cache simulations. The program uses a widely ported set of tasking primitives.

LocusRoute is a parallel wire routing program written in the C language by Jay Rose, formerly of Stanford, and currently at Toronto. This program has also been used to study distributed cache systems, is relatively portable and widely used. Numerous data sets describing layouts are available.

Puppy is a parallel program for placement of macro-cell circuits which uses a parallel version of simulated annealing, developed by Andrea Casotto at Berkeley. This program has been used in other studies at the Univ. of Washington.

MaxFlow is a parallel program to solve the maxflow problem in flow networks. The program is a parallel version of the algorithm proposed by Goldberg and Tarjan. It was written in the C language by Francisco Javier Carrasco at Stanford University.

MinCut finds the minimum cutset of a graph using simulated annealing. It was written in the C language by Todd Mowry and John Dykstal of Stanford.

Mul-T is an implementation of a parallel Lisp language system. The Mul-T system extends the T language (a dialect of Scheme) with multiprocessing features, including *futures*. Mul-T is the work of Robert Halstead at DEC Cambridge Research Lab, who has cooperated with researchers from the MIT ALEWIFE project. Mul-T currently runs a handful of parallel Lisp programs, mostly solving computer science problems.

Synapse is a parallel discrete-event simulation environment that uses conservative synchronization mechanisms. Synapse was implemented by one of the PIs (Wagner) when he was a graduate student at the University of Washington [72].

VM_pRAY is a distributed ray-tracing application for the Intel iPSC/2 that uses software virtual memory to cache objects being ray-traced in local memories.

Presto Suite is a collection of programs that demonstrates the features of the Presto parallel run-time system [14]. The Presto Suite was written in C++ by John Bennett of Rice University.

B Research Interests of Individual Investigators

B.1 Process Placement (Grunwald)

The classical model of uniformly accessible shared memory is a useful, albeit fallacious, abstraction for the construction of parallel programs. Bus-based systems are limited in their parallelism and memory bandwidth. Network-based computers, or multicomputers, are more scalable, yet suffer from a complex programming model.

My previous research [26, 27] focused on the initial placement of processes in applications explicitly tailored for multicomputers. I am currently expanding this research in three areas:

1. Placement of explicit process-level programs.
2. Loop-level scheduling on multicomputers.
3. Network Virtual Memory polices for scientific applications.

Placement of explicit process-level programs. In my thesis research, I simulated the execution of eight program traces using a simulated modern multicomputer system with a variety of distributed load distribution strategies. Some programs traces were captured from actual programs while others were synthetic workloads. The parameters for the synthetic workloads were derived from a benchmark study [28] of existing multicomputer systems.

My study was complicated because there are few workload datasets representing the execution of actual multicomputer programs. Hence, I used both *captured* and *synthesized* workloads. The Chare Kernel [34] is a portable environment for distributed computation using *chares*, or small, very lightweight process with restricted control flow. The total number of processes and the execution time of each process are unknown at compile time. I used four captured workloads from Chare Kernel programs; two were C programs and two were compiled ROPM Prolog [60] programs. The externally observable behavior of each Chare Kernel process was recorded in a log by executing the programs on a shared memory multiprocessor. Computation time was measured using a microsecond timer and scaled to simulate a processor executing approximately ten million instructions per second. The synthesized workloads were generated from abstract descriptions of program behavior. These workloads provide a range of process behavior absent in the captured workloads.

Although the range of process activities (total computation time, total number of messages sent and average message size) spans two orders of magnitude, the model programs may not be truely representative of multicomputer programs. As part of my NSF Research Initiation Grant, I am currently collecting additional multicomputers programs and program traces; support from grant would facilitate gathering additional traces and sharing those traces with others.

Loop-level scheduling on multicomputers. Multicomputers are commonly used for scientific applications. The process dependence structure of such computationly intensive programs must be clarified for studies of process distribution and redistribution. Programs with small processes, such as those traced in my thesis, benefit from process distribution, yet can not benefit from process redistribution; the time to redistribute processes will outweigh the time needed to finish the computation for most processes. Although multicomputer operating systems provide less functionality than traditional distributed operating systems, the time to create and destroy processes may be similar. Li [41] measured the operating system overhead of context switching and communication on the Intel iPSC/2. His measurements do not differ dramatically from those of traditional operating systems, where process migration was found to be expensive [57, 80].

In part, the problem is with the programs being traced. The goal of my original distribution study was to distribute highly parallel languages, such as parallel functional or logic languages. Recent research [17, 75] has examined the automatic translation of FORTRAN programs to dynamic process models for multicomputers. Process distribution mechanisms should be applicable to these numerically intensive programs using runtime scheduling of loop iterations.

Thus, I propose to collect program traces from a wider range of program models and characterize the computation and communication behavior of those programs. Due to the paucity of programming environments supporting dynamic process creation for multicomputers, the measurements will come from both multicomputer environments and shared-memory multiprocessor environments. In particular, I am constructing tools to analyse FORTRAN programs that have been transformed by PARAFRASE [36] to execute on parallel systems; these programs express parallelism using `DOALL` and `DOACROSS` loops.

Network Virtual Memory polices for scientific applications. Several researchers [39, 41, 35] have investigated *network virtual memory* or *distributed shared memory* and have proposed to use such mechanisms to simplify the use of multicomputers. Employing such software smoke and architectural mirrors, programs can be made to run on architectures with a highly non-uniform memory hierarchy. However, multicomputers are commonly used for scientific applications, which are typically performance driven programs using large amounts of memory and I/O. For these applications, the utility of network virtual memory must be balanced against any commensurate performance degradation.

We have constructed a generalized distributed cache simulator for networks supporting relatively large (page sized) messages. We will use SPAE traces to drive the simulations, varying invalidation policies, transport mechanisms and mappings of arrays to memory to examine their effects on total application performance.

B.2 Interactions Between the OS and Runtime System (Zorn)

It is generally acknowledged that writing parallel programs is a difficult task made more difficult by the abundance and variety of parallel hardware. I believe that more sophisticated program language runtime systems can reduce the effort of writing parallel programs while at the same time improving the performance of these programs on a variety of parallel computers.

One approach to providing parallelism is to add language features the require support from the runtime system. *Futures*, as suggested by Halstead [29], allow the programmer to explicitly fork a computation whose value is implicitly joined by the runtime system at a later time. Lazy task creation has been suggested as an implementation technique for futures [45]. While these techniques take advantage of a complex runtime system to improve the performance of parallel programs, they also require changes to existing programming languages to implement.

Another approach to improving performance is to make the runtime system more sophisticated without changing the language definition. Performance tools such as gprof [25] and mprof [82] take this approach. My research will focus on improving the performance of parallel programs using sophisticated runtime systems without requiring changes to the definition of application implementation language. In particular, three aspects of performance improvement will be considered.

Predicting Task Lifespans. Excessive parallelism can significantly reduce performance of a parallel application if there are far more tasks than processors. A more desirable situation arises when there are as many large tasks executing as there are processors. If the creation of short-lived tasks can be predicted, they can also be eliminated. By identifying creation *sites*, the runtime system can attempt to correlate task lifespan with creation sites. Various researchers have used different techniques to identify dynamic program event sites, including using the static code location [30], the current stack pointer [20], a single dynamic caller [25], and the entire caller chain [82]. My research will use a truncated caller chain in a method I call backtrace site identification (BSI), which is a general technique for identifying the locations of dynamic program events.

The system call traces collected in this study, including task creation information and function call information, will allow me to study the correlation between task creation site and lifespan, and determine if this information can be used to reduce the creation of small tasks.

Adaptive User-Managed Virtual Memory Policies. Using the same technique of backtrace site identification, I can identify the program sites that account for the most page faults for a given memory size. These sites are likely to be places in the program where memory use assumptions break down. For example, operating systems generally assume that recently referenced pages are the most likely to be referenced again, and implement a pseudo-LRU replacement policy. However, in a particular program a function may sequentially scan a large array data structure. The best replacement policy in this case is last-in, first-out. By identifying sites where the standard policy fails, the runtime system can provide the information to make an adaptive replacement policy more effective than the default policy.

To accomplish this research, address reference traces will need to be correlated with program function call information. Larus' Abstract Execution (AE) tool provides such information [37]. While traces from parallel programs are not essential for this investigation, parallel programs present substantial challenges to conventional memory management policies because their behavior often differs from the time-shared program workload that traditional policies were designed to handle.

Negotiating with the Operating System. A final component of the research will be to consider possible interactions between the operating system and the program via the language runtime system. Current parallel languages, such as Schnabel's DINO [61], require that the compiler determine the degree of parallelism expected at runtime. This approach is effective when parallel hardware is used by a single user and the compiler knows how many processors will be available to the program. A more realistic scenario is that the hardware is being time-shared, and that the number of processors available to the program will depend on the workload of the multiprocessor. In this scenario, the degree of parallelism should not be decided until runtime. Furthermore, a dialogue should be carried out between the operating system and runtime system.

Zahorajan and McCann investigate the performance of a system where the number of processors available to an application changes during its execution [79]. Their results indicate that dynamic scheduling of processors to applications is superior to static scheduling. Wagner is considering an even more interactive algorithm, where the operating system provides the application with an option to accept a particular processor allocation, or to wait for possibly more processors later [74]. This model of the operating system, where applications have a role in determining resource allocation, differs from the traditional one where the operating system is completely responsible. The traditional role is effective when runtime systems are simple, because it reduces the work required of the programmer, who does not necessarily want to make resource allocation decisions. With more complex runtime systems, the programmers job is still easy because the runtime system can interact with the operating system in deciding how the schedule significant resources.

In this phase of my research, I will investigate performance improvements associated with dynamically determining the degree of parallelism an application needs (instead of having it done statically by the compiler). Also, in cooperation with other researchers on this grant, I will investigate the potential benefits of different operating system/runtime system dialogues.

B.3 Workloads for Multiprocessor Scheduling Investigation (Wagner)

As shared-memory multiprocessors become less expensive, they become attractive alternatives to conventional computer architectures for many applications. Presently, shared-memory multiprocessors are used chiefly in two fundamentally different ways: as dedicated batch servers for parallel computations, or as high-throughput timesharing machines for sequential computations. Most significantly, shared-memory multiprocessors are seldom used to timeshare parallel computations, because there are few existing scheduling policies that are adequate for this type of use. Timesharing parallel computations on a multiprocessor is desirable for exactly the same reason that timesharing sequential computations on a uniprocessor (or multiprocessor) is desirable: to increase processor utilization.

However, timeshared scheduling of parallel computations is more difficult than scheduling sequential computations because there are so many more parameters involved. In addition to the percent of a given processor's time to allocate to a particular job, there are also these: the *number* of processors to allocate to a job; whether or not the allocation can be changed during the lifetime of the job; whether or not to co-schedule the tasks that make up a job [53]; whether an application is using spinning or blocking for synchronization purposes [77, 78]; whether or not lightweight "threads" are being multiplexed on top of operating system tasks by the user level [64]; etc. Because of these complications, there may be a substantial benefit to widening the interface between the user level and the operating system level, to allow the application level to interact (negotiate) with the scheduler – to request resources, provide hints, etc.

Some of the research issues in this area include: how to trade off the number of processors allocated with the amount of time allocated; how to encourage the use of parallelism; how to prevent greedy users from acquiring an unfair share of the processing resources, and how to reconcile this goal with the immediately preceding one; how to adapt scheduling to a changing load; how to identify process thrashing; and how to widen the interface between the application level and the operating system level.

This situation has fostered a recent spate of research into multiprocessor scheduling policies [42, 38, 47, 55, 65, 70, 71, 79]. Unfortunately, while proposals for new scheduling policies are plentiful, there does not seem to be a well-developed methodology for evaluating their performance. I identify two aspects of current methodology that are flawed: unrealistic workloads and evaluation metrics that are not appropriate for the multiprocessor environment. The consequences of this are threefold:

1. Synthetic workloads that are not realistic may be driving the investigation of scheduling policies whose benefits are illusory.

2. The lack of standard workloads for policy evaluation makes the meaningful comparison of results from different research groups difficult if not impossible.

3. Policies that have been identified as good performers may actually be undesirable for some multiprocessor environments, because they strongly *discourage* parallel computation.

The first two of these problems can be effectively addressed using the data and resources of the Testbed project described previously in this report. The third point is discussed in more detail in [73].

In the discussion that follows, I define a *task* to be the smallest schedulable unit of computation, and I define a *job* as a computation that consists of some number of tasks, possibly with precedence constraints between the tasks. I assume that there are no precedence constraints between jobs.

Workload Parameterization Most previous work in this area has analyzed the performance of scheduling algorithms on synthetic workloads. A notable exception to this is the study by Tucker and Gupta, which utilized real benchmark programs on a production multiprocessor and operating system [71]. The use of benchmarking to evaluate system-level software such as a scheduler is fairly uncommon because of the large amount of implementation effort required for a comparative study. Furthermore, the results of a benchmarking study are difficult to extrapolate to other hardware and software platforms. For these reasons, most performance studies of multiprocessor scheduling policies have used analytic or simulation techniques, which leads to consideration of synthetic workloads.

For analytic studies, synthetic workloads are clearly necessary, and certain assumptions must be made for the sake of mathematical tractability. Examples of such assumptions are exponentially distributed task times, geometrically distributed number of phases in a fork-join job, etc. For simulation studies, however, there is much greater latitude in the parameterization of the workload, and one should endeavor to make the workload as realistic as possible. In fact, it would be feasible to drive a simulation from traces of actual parallel programs, as is commonly done in cache and virtual memory performance studies,[2] although this has not been done to my knowledge.

Proceeding under the assumption that synthetic workloads are a "necessary evil", it is then necessary to choose the types of jobs in the workload and the characteristics of each of those job types. Examples of job types are fork-join [42, 47, 48, 70, 79], static task graph (of which fork-join is a special case) [79], and randomly synchronizing tasks [38]. The parameterization of each of these job types is no simple matter. Parameters that must be chosen may include: the distribution of the number of tasks in a job; the distribution of total job demand; the correlation between number of tasks and job demand, if any; the distribution of the number of phases in a fork-join job, and the number of tasks in each phase; the distribution of the the inter-synchronization time for randomly synchronizing tasks; the overhead of context switching; and so forth. The problem is that while each of these job types seems realistic enough, there is no assurance that a particular parameterization of one of them bears any resemblance to those encountered in a real workload. I feel very strongly that synthetic workloads that are not realistic may be driving the investigation of scheduling policies whose benefits are largely illusory.

My goal for this portion of the project is to formulate synthetic workloads that I can be confident are representative of real workloads. My strategy will be to statistically analyze the trace data obtained from the Testbed to produce parallel program *profiles* that can be used to parameterize stochastic simulations. The measurement data provided by the Testbed will also allow me to drive simulations directly from program traces to validate the fidelity of the profiles.

Workload Standardization Since nearly every study uses a different workload, its results are difficult, if not impossible, to compare to others. For example, Leutenegger and Vernon found that

[2]I do not propose the use of memory address traces for this purpose, which contain far too much information. A much less detailed trace, containing only the amount of computation between synchronization points, is all that would be required.

round-robin policies outperformed first-come-first-served/run-to-completion, whereas Zahorjan and McCann found exactly the opposite [38, 79].

I intend to make the traces and profiles produced by this research available on request to other researchers in the performance analysis community. The media distribution infrastructure developed by the Testbed project will greatly facilitate this task. It is hoped that eventually these traces and profiles will will become commonly-enough used to allow comparisons of results across different studies.

B.4 Graph-Driven Program Performance (Nutt)

The Olympus projects have addressed several facets of parallel programming, assuming that the computation has been specified as a formal graph model. After briefly describing the spectrum of individual projects, we identify the relationship between this testbed project and the Olympus projects.

Olympus Architecture. Olympus systems are distributed, interactive, visual systems for creating, manipulating, storing, and exercising formal graph models [51]. An architecture for such systems has been developed to support various graph models. The architecture has been shown to be useful in that it addresses several specific issues in such systems:

1. Olympus systems are modeless. All of the system's facilities are available to the user at all times.
2. Parts of the system that are concerned with the graph model syntax and semantics are implemented as distinct modules (processes) in the architecture. This allows the user interface to the system to be largely independent of the model semantics.
3. Olympus systems support multiple users simultaneously. If one user modifies the graph model, then all users detect the modification at the same time.
4. Graph models can have sophisticated interpretations in a wide variety of languages. Olympus systems will execute those interpretations upon the command of the user.
5. Olympus system are easily extensible so that modules (processes) can be added to perform model-specific analysis.

Current work on the architecture is focused on facilities to reuse the architecture for radically different formal graph models.

BPG-Olympus Modeling System. The "charter" instance of the Olympus architecture is the BPG-Olympus modeling system [52]. This system is an interactive, distributed modeling system based on bilogic precedence graphs – a formal graph model similar in representative power to predicate-transition (interpreted Petri) nets. The system has been used to explore concepts of such systems, and to support simulation studies of other researchers.

PN-Olympus Modeling System. This instance of the Olympus architecture is an adaptation of the BPG-Olympus system to interpreted Petri nets. It is being used as a support tool for constructing various performance models.

ParaDiGM Modeling System. ParaDiGM is a two-level modeling system intended to describe computations for distributed memory systems [21, 22]. The goal of this study was to provide computer support to the distributed programmer that would allow him or her to experiment with various process partitions, and to be able to understand the impact of each strategy on the ultimate performance of the resulting implementation. The work culminated in a prototype modeling system, loosely based on the Olympus architecture.

Phred Parallel Programming System. Phred is a visual parallel programming language directed at application programmers [9, 11]. The goal of the Phred project was to derive a parallel programming language, based on a formal graph model, that could be analyzed for possible nondeterministic operation. The supporting system is based on the Olympus architecture, and allows the user to develop a Phred program as an annotated graph model. The model is parsed and analyzed in parallel with its construction, using a nonintrusive *critic* that is distinct from the graph editor.

Adaptive Load Balancing. This aspect of the study is in its formative stages, and is the most-closely related to the testbed effort. The basic idea is to assume that a computation has been specified as a directed graph model of the general form of an annotated precedence graph or Petri net. The graph model generally determines the order in which subcomputations are scheduled – based on control or data flow. The annotations associated with each node in a graph define the detailed subcomputation.

Given that a computation has been defined by an annotated graph model, then one could distribute the execution of the corresponding program by assigning different nodes to different processors. (This is analagous to Grunwald's process placement, but is based on different assumptions about the program specification than he uses.) The assignment might be static, dynamic, or dynamic-adaptive. Static assignment is straight-forward: the strategy for distribution is based on observable properties of the graph. Dynamic assignment suggests that runtime properties are used to adjust the assignment as the program executes; we emphasize the the importance of detecting the balance of work on the assigned processors by concentrating on dynamic-adaptive techniques for balancing the work.

The strategy that will be followed in this study is to ensure that the entire annotated graph model (program) is easily-accessible to each processing node, e.g., by providing each with a copy of the model. The state of the computation is represented by the distribution of tokens across the graph model; this distribution is partitioned across the processors in the system so that the processor that has been assigned node i also maintains the status of the computation at node i. In this case, process migration is accomplished by migrating the status of a node from one processor to another.

Synchronization among processors is determined by the flow of tokens among processors as dictated by the graph model. Whenever a processor is idled, then there are no tokens in the nodes assigned to the processor. The idle processor can only receive tokens – and more work – by receiving tokens from the "upstream" processors. Thus an idle processor might be able to adapt to the existing load conditions by assuming tokens that are queued on nodes in an upstream processor.

This is the general idea that we will pursue for the adaptive load balancing study. The study will make considerable use of the Olympus architecture for modeling the specific strategies, and possibly as the basis of a implementation.

Olympus and the Testbed Study. The testbed study can be used to observe properties of program behavior at various levels of detail. In partcular, the traces can be made at abstract levels corresponding to basic blocks of computation. These are natural "subcomputations" that can be mapped to nodes in a directed graph. Our intent is to use the trace date to define directed graph models of observed computations, then to use those models to drive the load balancing study.

The continuing architectural work is aimed at suporting a wide class of graph models; the load balancing work requires that a graph model be used to define the computation. The testbed traces can be used to derive a class of directed graph models. Thus the steps in this project are to:

1. Tailor the tracing tools so that they produce trace information from which an appropriate subcomputation graph can be inferred.
2. Define the formal graph model that can be inferred from the trace and which can be used for the load balancing study.
3. Create an Olympus instance that supports the derived graph model.
4. Explore adaptive load balancing strategies in the Olympus instance.

As a practical matter, the trace information can be abstracted to synthetic programs, based on the signatures of different classes of programs. Thus, a generated set of event traces of synthetic programs can be used in lieu of actual program traces, provided that the synthetic programs are representative of the actual programs.