Proceedings from the Second Workshop on Large-Grained Parallelism

October 11-14, 1987
Hidden Valley, Pennsylvania

Jeannette Wing, Maurice Herlihy, Mario Barbacci (eds.)
CMU-CS-88-112

These are the proceedings of the Second Workshop on Large-Grained Parallelism held October 11-14, 1987, in Hidden Valley, Pennsylvania. The workshop was organized by the Software Engineering Institute and the Department of Computer Science, Carnegie Mellon University, with the cooperation of the IEEE Computer Society.

The purpose of the workshop was to bring together people whose interests lie in the areas of operating systems, programming languages, and formal models for parallel and distributed computing. The emphasis of the workshop was on large-grained parallelism or parallelism between concurrent programs running on networks of possibly heterogeneous computers rather than parallelism within a single process or thread of control. Aspects of large-grained parallelism that were common to most participants' interests were fault-tolerance, heterogeneity, and real-time applications.

Ninety abstracts were submitted for review by the program committee and the authors of thirtyeight of these abstracts were sent acceptance letters and invitations to attend the workshop. To provide more time for discussion and audience participation, only sixteen authors were asked to give twenty-five minute talks based on their abstracts. The rest of the abstracts were summarized by discussion leaders. The workshop was divided into five sessions of talks and two parallel sessions of discussion. The five general areas covered by the talks were: scheduling, distributed languages, real-time languages and models, operating system support, and applications. There were parallel discussions on scheduling and distributed languages, and on real-time and operating system support.

There was a reasonable balance among the participants with regard to efficiency concerns on the one hand, e.g., by the software and hardware systems and application builders, and correctness concerns on the other, e.g., by the real-time modelers and language designers. We identified a number of key challenges:

- Distributed systems, languages, environments
  - Make transactions efficient. Integrate them into the operating system.
  - Implement applications that demonstrate how to use transactions at both the programming language and operating system levels.
  - Identify applications other than databases to motivate the need for multi-site transaction-based systems.
- Real-time systems, models, scheduling
  - Devise and test analytical models for distributed scheduling of tasks that range in degrees of computational complexity.
  - Show the correspondence between physical time and logical time using a formal modeling approach.
  - Identify a set of programming and specification language primitives that capture and abstract from real-time events of interest.

In the year that elapsed since the first workshop on large-grained parallelism that took place in Providence, Rhode Island, a number of the issues related to large-grained parallelism became more focused, as evidenced by the topics and the quality of the abstracts submitted. Considering the wide range of interests and background of the participants, the success of this workshop is a good omen for future meetings.

Jeannette M. Wing
Program Chair
Department of
Computer Science

Maurice P. Herlihy
General Chair
Department of
Computer Science

Mario R. Barbacci
Arrangements Chair
Software Engineering
Institute

Final Program

Time	Sunday, October 11	Moderator
4:00 pm	Registration desk opens	
6:00 pm	Dinner followed by informal discussions	
	Monday, October 12	
7:30 am	Breakfast	
8:30 am	Session 1 -- Scheduling Talks by Jack Stankovic and Jean-Luc Gaudiot	Barbacci
10:00 am	Break	
10:30 am	Session 2 -- Distributed Languages and Environments Talks by David Notkin, William Weihl, and Maurice Herlihy	Wing
12:00 am	Lunch	
1:30 pm	Parallel Discussions: Scheduling	Stankovic
	Parallel Discussions: Distributed Languages	Weihl
3:00 pm	Break	
3:30 pm	Session 3 -- Real-Time Languages and Models Talks by Janice Glasgow, Debra Lane, and Mario Barbacci	Herlihy
5:00 pm	Break	
6:00 pm	Dinner followed by informal discussions	
	Tuesday, October 13	
7:30 am	Breakfast	
8:30 am	Session 4 -- Operating System Support Talks by Mark Sullivan and Rick Bubenick	Satya
10:00 am	Break	
10:30 am	Parallel Discussions: Real-Time Languages and Models	Bryan
	Parallel Discussions: Operating System Support	Satya
12:00 am	Lunch	
1:30 pm	Free afternoon (unstructured meetings)	
6:00 pm	Dinner followed by informal discussions	
	Wednesday, October 14	
7:30 am	Breakfast	
8:30 am	Session 5 -- Applications Talks by Martin McKendry, Sid Ahuja, and Carl Diegert	Stankovic
10:00 am	Break	
10:30 am	Session 5 (Continuation) Talks by Liuba Shrira, Richard LeBlanc, and Hanno Wuppert	Wing
12:00 am	Lunch	

Attendee List

S. R. Ahuja
Systems Architectures Research Department
AT&T Bell Laboratories
Crawford Corner Rd.
Holmdel, NJ 07733
ucbvax!vax135!sra

Mario Barbacci
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
barbacci@sei.cmu.edu

Albert Benveniste
IRISA
Campus de Beaulieu
35042 Rennes Cedex
France
RANDRE@irisa.irisa.fr

Doug Bryan
ERL 456, Computer Systems Laboratory
Stanford University
Stanford, CA 94305
bryan@sierra.stanford.edu

Rick Bubenik
Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251
rick@rice.edu

Eric C. Cooper
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
ecc@cs.cmu.edu

Luis Cova
Department of Computer Science
Princeton University
Princeton, NJ 08544
allegra!princeton!cova
cova@princeton.pu.edu

Carl Diegert
Computer Science and Mathematics - Division 1412
Sandia National Laboratories
Albuquerque, NM 87185
diegert@sandia-2.arpa

Dave Detlefs
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dld@f.gp.cs.cmu.edu

Alan Downing
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
downing@spam.istc.sri.com

Alessandro Forin
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
af@speech2.cs.cmu.edu

Stuart A. Friedberg
Computer Science Department
University of Rochester
Rochester, NY 14627
stuart@cs.rochester.edu
{ames,cmcl2,rutgers}!rochester!stuart

Jean-Luc Gaudiot
EE-Systems Department/SAL-300
University of Southern California
Los Angeles, CA 90089-0781
gaudiot@usc-cse.usc.edu

Thomas B. Gendreau
Deparment of Computer Science
Vanderbilt University
Box 1679 Station B
Nashville, TN 37235
gendreau@vanderbilt.csnet

Janice Glasgow
Department of Computing & Information Science
Queen's University
Kingston, Ontario
Canada K7L 3N6
janice%qucis@wiscvm.wisc.edu

Andrew Grimshaw
Department of Computer Science
Universtiy of Illinois
1304 West Springfield
Urbana, IL 61801
grimshaw@p.cs.uiuc.edu

Maurice Herlihy
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
herlihy@cs.cmu.edu

Norman Hutchinson
Computer Science Department
University of Arizona
Tucson, AZ 85721
norm@arizona.edu

Michael B. Jones
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
michael.jones@spice.cs.cmu.edu

Debra S. Lane
Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92717
dlane@ics.uci.edu

Richard LeBlanc
Georgia Tech
School of ICS
Atlanta, GA 30332-0280
rich@gatech.edu

Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
lee@cis.upenn.edu

Paul LeGuernic
IRISA
Campus de Beaulieu
35042 Rennes Cedex
France
leguernic@irisa.irisa.fr

Glenn H. MacEwen
Computing and Information Science Department
Queen's University
Kingston, Ontario
Canada K7L 3N6
macewen%qucis@wiscvm.wisc.edu

Martin McKendry
FileNet Corporation
3530 Hyland Avenue
Costa Mesa, CA 92626
{hplabs/trwrb}!felix!martin

Michael Molloy
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
mkm@k.gp.cs.cmu.edu

K.T. Narayana
Department of Computer Science
Whitmore Laboratory
The Pennsylvania State University
University Park, PA 16802

Dave Nichols
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
nichols+@andrew.cmu.edu

David Notkin
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
notkin@cs.washington.edu

Calton Pu
Department of Computer Science
Columbia University
New York, NY 10027
calton@cs.columbia.edu

Mahadev Satyanarayanan
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
satya@andrew.cmu.edu

Liuba Shrira
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
liuba@xx.lcs.mit.edu

Jack Stankovic
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
stankovic@cs.umass.edu

Mark Sullivan
Computer Science Division, EECS Department
571 Evans Hall
University of California
Berkeley, CA 94618
sullivan@ucbarpa.berkeley.edu

William E. Weihl
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139
weihl@xx.lcs.mit.edu

Chuck Weinstock
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
weinstock@sei.cmu.edu

Tom Wilkes
Department of Computer Science
University of Lowell
1 University Avenue
Lowell, MA 01854
wilkes@hawk.cs.ulowell.edu

Jeannette Wing
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
wing@k.gp.cs.cmu.edu

Hanno Wupper
Katholieke Universiteit Nijmegen, Informatica V
Toernooiveld
6525 ED Nijmegen
The Netherlands

Table of Contents

USING A NETWORK OF COMPUTER WORKSTATIONS AS A LOOSELY-COUPLED MULTIPROCESSOR

RAKESH AGRAWAL

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

(201) 582-2250
rakesh%allegra.att.com@csnet-relay

ABSTRACT

A major trend in computing in recent times has been the creation of large networks of computer workstations. It has been speculated that the number of computing cycles installed in computer workstations is an order of magnitude greater than the number installed in mainframes. However, most of these cycles are idle most of the time. There are many applications amenable to large grain parallel processing that can profitably use these idle computing cycles by treating these networks as loosely-coupled multiprocessors. There seem to be two essential requirements for this approach to become feasible:

- We must provide simple to use system facilities to access computing cycles from an idle workstation.
- We must develop tools for partitioning the problem into pieces that may be executed in parallel.

In NEST, we have extended System V Unix with a remote execution facility that allows creation of transparent remote processes [1,3]. Developing applications that run in parallel on multiple machines is particularly simple using this remote execution facility. If there is a program involving multiple processes written in C that runs on a uniprocessor, it can be made to run on multiple machines by simply changing the *exec* system call to *rexec*.

We also have developed a model for optimally partitioning a class of problems in the workstations environment [2]. Our model recognizes that workstations are usually connected with a rather slow communication medium, and explicitly takes into account the communication costs in addition to the computation costs. The optimal partition can be determined for a given number of processors and, if required, the optimal number of processors to use can also be derived. We also have performed experiments that verify and demonstrate the effectiveness of our model using matrix multiplication as an example.

REFERENCES

1. R. Agrawal and A. K. Ezzat, Processor Sharing in NEST: A Network of Computer Workstations, *Proc. IEEE 1st Int'l Conf. Computer Workstations*, San Jose, California, Nov. 1985, 198-208.

2. R. Agrawal and H. V. Jagadish, Parallel Computation on Loosely-Coupled Workstations, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986.

3. R. Agrawal and A. K. Ezzat, Location Independent Remote Execution in NEST, *IEEE Trans. Software Eng. 13*, 8 (Aug. 1987), 905-912.

Parallelism in the Rapport Multimedia Conferencing System

S. R. Ahuja
J. R. Ensor
D. N. Horn

AT&T Bell Laboratories
Holmdel, New Jersey 07733

Rapport is a multimedia conferencing system which executes on a collection of network-connected workstations. This system provides communication protocols and user interfaces that effect a natural conferencing environment in which users conduct remote, interactive conferences by talking with each other and producing and editing common displays on their workstations. Rapport coordinates the transmission and use of shared information in several media, including voice, graphics, images, and text. Thus Rapport is a distributed system with a collection of simultaneously active agents accessing shared data and producing new data which must be broadcast in real time. Underlying mechanisms for global name service, data storage, and window management are used by Rapport to produce its conferencing aids.

Our current implementation of Rapport executes on a collection of Sun workstations which are connected by an Ethernet. A specialized processor we have built to handle voice (and eventually video) transmissions is attached to each Sun through its VME bus. The NFS file service provides common names and storage for programs and data used in conferences. The X window system is used to provide a common means of producing displays on the various workstations. Rapport provides each conferee with protocols for controlling a conference. Our system also allows user-level application programs to be associated with a conference. These programs manipulate shared data and produce common displays on the screens of the conferees' workstations.

Coordinating the input and output of application programs is a principal responsibility of Rapport. We are presently comparing the behavior of two approaches to the execution of application programs. In the first approach, a single workstation executes an application program and broadcasts its output commands to the other conferees' workstations. The major advantage of this approach is that it allows the various conferees to see results of programs without executing them. The corresponding disadvantage is that broadcasting all the window level commands and arguments for display generation usually generates significant network traffic. In the second approach, each workstation executes all application programs of a conference under some constraints of synchronization and input control. This technique tends to generate less network traffic since only the application program input commands are transmitted among the conference workstations. The major drawback of this technique is that each conferee must execute the same software in a consistent environment. Some programs are written to utilize local state and are not suitable for this technique. For example, a bitblit program might receive as an argument a pointer into its local machine's memory. Giving this command and its argument to each conferee would not preserve the consistency of the conference.

Though the basic tradeoffs between the two approaches are readily identified, the importance of these tradeoffs are not obvious. The first Rapport implementation requires that each workstation execute each application program locally. We are now building a version in which each application program is executed by only one workstation. The two versions of Rapport give us the opportunity to examine some parallel execution issues. We can determine the amount of network traffic generated by each approach, and hence determine whether the differences in network load are significant in various situations. We can also investigate whether synchronization among the application programs at program command input is notably different from synchronization both at command input and program output. The single site

execution of each application program allows different conferees to work on different displays simultaneously. We are going to investigate the usefulness of this parallelism between the synchronization points imposed by the conference management.

After performing these initial experiments with Rapport, we plan to create a modified system in which conferences can take place over wide area networks. This extension poses major difficulties. In the local area network environment we are using standard tools, NFS and X, to reduce the apparent heterogeneity of the workstations. Further, conferencing inherently involves the sharing and multicasting of information, which require a naming mechanism and efficiency of transmission. NFS gives us a global name service and a convenient storage for common programs and data. X allows us to conveniently coordinate the displays on the conferees' workstations. In the wide area environment these tools are not available, so we will be required to provide their services for ourselves. The implications for the real time characteristics of the system are even more dramatic. The delays in producing displays on remote workstations must be kept under control in spite of the larger transmission delays. Furthermore, we must limit the skews among the transmission of the different media.

PROCESS SCHEDULING IN LOOSELY-COUPLED COMPUTER NETWORKS

Rafael Alonso
Luis Cova
Kriton Kyrimis

Department of Computer Science
Princeton University
Princeton, N.J. 08544
(609) 452-3869

ABSTRACT

A computational environment in widespread use is that of a loosely-coupled local area network (typically an Ethernet) of high performance workstations (such as SUN† workstations). It has been observed that such networks have the potential for becoming inexpensive parallel engines, especially for users whose applications show a coarse parallelism (i.e., large grained parallelism). Furthermore, it seems that such systems are usually underutilized, i.e., many of the machines on the network are not in use at any one time. Our current research aims at helping users with applications displaying large grained parallelism to schedule their tasks and make efficient use of these idle processors.

Our work has proceeded along a number of lines. The first involves the exploration of load sharing policies. As a user starts up several parallel tasks, it is desirable for those jobs to be scheduled automatically, and in such a manner that each of them can obtain as many processing cycles as possible. A load sharing mechanism can ensure that idle workstations across the network can be used by a parallel application in a user-transparent manner. We have built such a mechanism [ALON86] and have used it to experiment with a variety of load balancing strategies. This work has concerned itself with load balancing (i.e., making sure that the available work is evenly spread throughout the network). This may not be appropriate for an environment where users own their individual machines; in that situation some users might be willing to share cycles, but not at the expense of slowing down their private computations. We are now studying techniques for scheduling in such networks [ALON87a].

We have recently started on a related topic, that of the placement of parallel tasks in networks of multiprocessing workstations (i.e., workstations such as the DEC Firefly or the Xerox Parc Dragon). In such environments, the scheduling decision is a two-level one, especially if there are different costs to communicate on the same machine than across the network. For some applications that require a large amount of inter-task communication it might be best to cluster all the computational threads on the same machine, even if excess processing cycles are available elsewhere, while in other instances the

† SUN is a trade mark of SUN Microsystems, INC.

computational component is the main processing bottleneck.

Our work in this area consists of a joint project with researchers at Bell Communications Research. For this project, DUNE [PUCC1987], a multiple processor system, is being used. Dune supports transparent process migration, both within a multiprocessor and across the network. We are currently exploring a variety of scheduling algorithms that take advantage of the process migration capability of the system to allocate several parallel threads automatically on behalf of a user.

Lastly, we have also studied the issues involved in process migration. For many applications, it will be true that, during some phases of the computation, there will be a large number of parallel tasks, which will then dwindle in number to very few. In this situation, it is desirable to spread initially all the tasks across the available machines and, when there are only a few left, migrate those tasks away from each other (if they happen to be on the same processor) or towards the more powerful machines. We have designed and implemented a process migration mechanism for a network of SUN workstations [ALON87b]. We are presently building tools that utilize the process migration functionality of our system. For example, we are building a mechanism that will periodically scan the machines on the network and ensure that processes that have used many CPU cycles in a short time do not run in the same processor if at all possible.

References

[ALON86]
Rafael Alonso, Phillip Goldman, and Peter Potrebic, "A Load Balancing Implementation for a Local Area Network of Workstations," *Proceedings of the IEEE Workstation Technology and Systems Conference*, 1986.

[ALON87a]
Rafael Alonso and Luis Cova, "Sharing Jobs Among Independently Owned Processors," Technical Report, Department of Computer Science, Princeton University, 1987.

[ALON87b]
Rafael Alonso and Kriton Kyrimis, "A Process Migration Implementation for a UNIX† System," Technical Report CS-TR-092-87, Department of Computer Science, Princeton University, 1987.

[PUCC87]
Marc Pucci and James Alberi, "The Architecture of the DUNE Multiple Processor System: An experiment in Generalized Interprocessor Communication," Technical Report, Bell Communication Research, 1987.

† UNIX is a trademark of Bell Laboratories.

Durra: Language Support for Large-Grained Parallelism

Mario R. Barbacci,
Charles B. Weinstock, and
Jeannette M. Wing

Software Engineering Institute and
Department of Computer Science
Carnegie Mellon University,
Pittsburgh, PA 15213

We are interested in a class of real-time, embedded applications in which a number of concurrent, large-grained tasks cooperate to process data obtained from physical sensors, to make decisions based on these data, and to send commands to control motors and other physical devices. Since the speed of, and the resources required by each task may vary, these applications can best exploit a computing environment consisting of multiple special- and general-purpose, loosely connected processors. We call this environment a *heterogeneous machine.*

During execution time, *processes*, which are instances of tasks, run on possibly separate processors and communicate with each other by sending messages. Since the patterns of communication can vary over time, and, since the speed of the individual processors can vary over a wide range, additional hardware resources in the form of switching networks and data buffers are also required in the heterogeneous machine. The application developer is responsible for prescribing a way to manage all of these resources. We call this prescription a *task-level application description.* It describes the tasks to be executed and the intermediate queues required to store the data as it moves from producer to consumer processes. A *task-level description language* is a notation for writing these application descriptions.

To support this large-grained parallelism, we have designed and implementated Durra [1], a task-level description language. We are using the term "description language" rather than "programming language" to emphasize that a task-level application description is not translated into object code in some kind of executable "machine language" but rather into commands for a run-time scheduler. We assume therefore that each of the processors in a heterogeneous machine has languages, compilers, libraries of (reusable) programs, and other software development tools that cater to the

---

Arpanet addresses: barbacci@sei.cmu.edu, weinstock@sei.cmu.edu, wing@k.cs.cmu.edu

special properties of a processor's architecture. Durra's support environment is responsible for coordinating the use and interaction of the separate software environments of the individual processors.

There are three distinct phases in the software development process for a heterogeneous machine: (1) the creation of a library of tasks, (2) the creation of an application description, and finally (3) the execution of the application. During the first phase, the developer breaks the application into specific tasks (e.g., sensor processing, feature recognition, map database management, and route planning) and writes code implementing the tasks. For each implementation of a task, the developer writes a Durra *task description* and enters it into the *library*. Developing programs for some of the more exotic processors involves selecting algorithms appropriate to a processor's architecture, and then painstakingly testing and tuning the code to take advantage of any special features of the processor. For example, an application might use a matrix multiplication task written in assembly for a systolic array processor while simultaneously accessing a database of three-dimensional images maintained by a program written in C running on a workstation. Developing these programs is a slow and difficult process and Durra facilitates their reuse in multiple applications.

During the second phase, the user writes a Durra *application description*. Syntactically, an application description is identical to a compound or structured task description and can be stored in the library and used later as a component task in a larger application description. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands. During the last phase, the *scheduler* executes a set of commands which are produced by the compiler. These commands instruct the scheduler to download the task implementations, (i.e., code corresponding to the component tasks) to the processors and issue the appropriate commands to execute the code.

In our presentation, we will illustrate the main features of Durra through examples, the existing implementation of tool support for Durra, followed by preliminary conclusions and directions for future work. Further details on the language can be found in the Durra reference manual [1] and an overview paper [2].

---

[1] M.R. Barbacci and J.M. Wing: "Durra: A Task-level Description Language", Technical Report CMU/SEI-86-TR-3, Software Engineering Institute, and Technical Report CMU-CS-86-176, Department of Computer Science, Carnegie Mellon University, December 1986.

[2] M.R. Barbacci and J.M. Wing: "Durra: A Task-level Description Language", in Proceedings of the 16th International Conference on Parallel Processing, Pheasant Run Resort, St. Charles, Illinois, August 1987.

Agora: Heterogeneous and Multilanguage Parallel Programming

Roberto Bisiani and Alessandro Forin

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Extended Abstract

The solution of many real-life problems encountered in science and industry requires the integration of parallel programs written in different languages and running on heterogeneous machines. We call the development of such systems *heterogeneous parallel programming*. For example, sensor data acquisition and signal processing might have to be integrated with planning, or electrical circuit simulation might have to be integrated with expert system technology. The goal of the **Agora** project is to facilitate heterogeneous parallel programming. Agora's support is both in terms of *operating system level* mechanisms that can be used to implement heterogeneous parallelism and in terms of *programming environment* functionalities that facilitate the management of parallel programs. This paper describes the former, see [3] for a description of the latter.

We call the operating system level mechanisms **Agora Shared Memory**, since they are based on a shared memory model of parallelism. In order to simplify the explanation of the **Agora** Shared Memory we will use an example abstracted from a speech recognition system that has been successfully programmed in **Agora** [1].

The structure of the fragment of speech recognition system that is used as example is sketched in Figure 1. This subsystem receives phonetic hypotheses and generates sentence hypotheses. Two components, Word Matcher and Sentence Parser, are best implemented in C and the other two in Lisp. The aggregate computation power required by the four components to achieve real time execution is about $2 * 10^8$ instructions for each second of speech [2], with half of the computing power being used in the Word Matcher. Each of the components can be decomposed into parallel computations in many different ways and both large and small granularity decompositions are necessary.

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5167, and monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-0163. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

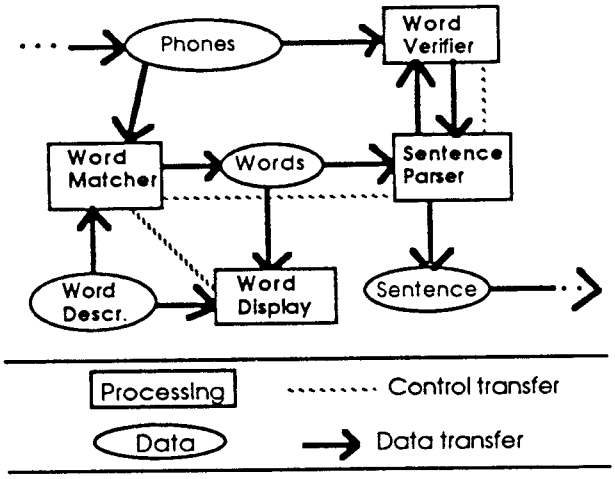


Figure 1: Example of a Parallel, Heterogeneous System: Speech Recognition.

A satisfactory implementation requires a multiprocessor that can execute programs with both C and Lisp components. The Word Matcher requires a tightly coupled architecture while the Word Display can be run on a single processor that is loosely coupled with the rest of the system. The Word Matcher communicates with the other components using a data-flow style of communication; the Sentence Parser and Word Verifier communicate as server and client.

There are a number of tools that could provide support for the implementation of the example, but none of them has all the necessary characteristics. The tools used by the AI community (possibly with the exception of ABE [7]) are centered on a single computational model

(e.g. production system languages), are based on a single language (e.g. Loops [5]), or have no support for parallel processing (e.g. SRL [8]).

One common way to tackle multilanguage applications with these tools is to implement a Lisp module that calls all the modules that are programmed in different languages. This solution has a number of drawbacks that make it unsuitable to our purposes:

- the structure of each module depends heavily on the other modules, e.g. the sentence parser would have to be explicitly programmed to activate the word display;
- the access of complex data structures from different languages must be handled by the user code.
- there is no easy way to parallelize the system to increase performance.

The tools used by the operating system community to link heterogeneous parallel programs (e.g. Matchmaker [9], Sun RPC [11]) have a different shortcoming: some of them support multilanguage parallel processing on heterogeneous architectures (e.g. Mach/Matchmaker), but they are geared only towards applications that can be efficiently cast into a client-server relationship between modules.

As in the sequential solution, the structure of each module depends heavily on the others since each module must be programmed to be able to explicitly deal with the requests of the other modules. Debugging is difficult since there are no tools to conveniently examine the data flowing between modules or to deal with more than one process at a time. Moreover, in current implementations on general purpose systems, communication is rather expensive since there is a message passing overhead even on shared memory architectures (currently about 2ms for a general purpose 1 MIPS machine).

Agora's Approach

Agora takes a different approach: first, concurrent modules share data structures independently of the computer architecture they are executed on and of the language they use; second, concurrent modules exchange control information by using a pattern-directed technique. Our hypothesis is that these two characteristics facilitate heterogeneous parallel programming. The only way to verify it is by implementing real systems and evaluating the effort required and the quality of the result.

Figure 2 shows how the example can be implemented with Agora.

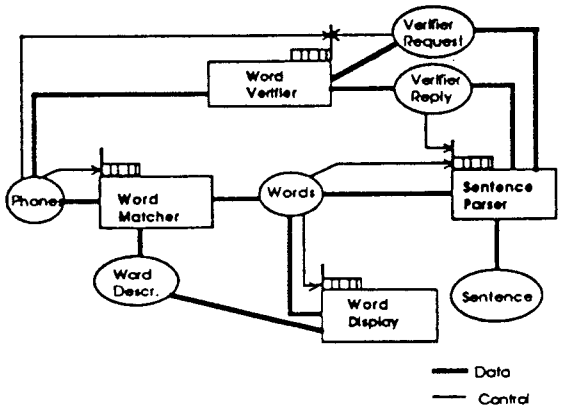


Figure 2: Agora's Implementation of the Example.

Shared Data

The ovals indicate shared data structures. These structures are allocated in Agora's shared memory and their structure is known by Agora's system code. Agora provides *standard functions* to create, destroy, read and write data structures as procedural extensions of each of the supported languages. Depending on the language, these functions can be more or less merged with the language syntax and semantics. For example, object-oriented languages like C++ [10] and Portable Common Loops [6] give the opportunity of blending Agora's functions more than their non-object-oriented counterparts (see [4] for an example). Users can also define *custom access functions* that are translated by Agora into each language and are available to all the modules that need them.

The description of data structures and access functions is processed by Agora and stored in its database where it is visible by all the tools in the environment, e.g. a debugger can interpret it to access data in the same way a user module does. Agora also generates a description of the data structures and a translation of the access functions for each language. The programming environment automatically includes the translated descriptions and access functions at compilation and link time.

Control

The boxes in Figure 2 represent concurrent computations. Each computation (agent) has a queue where Agora stores requests for activation of the agent. The agent is free to dequeue an activation whenever it wants and branch to different parts of its code depending on the kind of activation dequeued. In the example of Figure 2, the arrival of a new element in the *Words* data structure generates an activation for both the Word Display and the Sentence Parser agents. Activation patterns can be set by agents at any time, or by the user via the user interface. In the latter case, none of the agents involved need to be aware of it. This is a major feature

of Agora's handling of control information, since it maintains as much independence as possible between the modules of a system.

Multiple styles of computation, including control-driven, can be programmed using the basic Agora mechanism. For example, a context can be used to pass parameters back and forth between agents. In the example of Figure 1, the Sentence Parser uses this *remote procedure call* mechanism to communicate with the Word Verifier.

Current Status

The **Agora Shared Memory** has been operational since September 1986 and is used daily in the development of a large speech recognition system (about 100,000 lines of code and developed by 15 researchers). Agora currently runs on DEC Vax, IBM RT PC, Sun, Encore Multimax and all possible combinations of these machines. The languages currently supported are C, C++ and CommonLisp.

Conclusions

Here are some of the hypotheses that we are exploring in Agora:

- *the same model* can be used for both small and large grained parallelism;
- *shared memory is a viable communication abstraction* even between modules implemented in different languages;
- *a structured shared memory can be implemented with reasonable efficiency* on non-shared memory architectures and across heterogeneous machines;
- *pattern directed invocation is a convenient control mechanism* for a shared memory model.
- *multiple styles of computation,* including control-driven, can be programmed using the basic **Agora** mechanisms.

References

1. D. Adams and R.Bisiani. "The Carnegie-Mellon University Distributed Speech Recognition System". *Speech Technology* 3, 2 (April 1986).

2. Ananthamaran, T. and Bisiani,R. Hardware Accelerators for Speech Recognition Algorithms. Proceeedings of the 13th International Symposium on Computer Architecture, IEEE, June, 1986.

3. Bisiani,R., Alleva, F., Correrini, F., Forin, A., Lecouat, F., Lerner, R. Heterogeneous Parallel Processing, The Agora Programming Environment. Tech. Report CMU-CS-87-113, Carnegie-Mellon University, Comp. Science Dept., March, 1987.

4. Bisiani,R., et.al. Heterogeneous Parallel Processing, The Agora Shared Memory. Tech. Report CMU-CS-87-112, Carnegie-Mellon University, Comp. Science Dept., March, 1987.

5. Bobrow,D.G. and Stefik,M.J. A Virtual Machine for Experiments in Knowledge Representation. Xerox Palo Alto Research Center, April, 1982.

6. Bobrow D.G., et al. CommonLoops: Merging Lisp and Object-Oriented Programming. Proceedings of OOPSLA'86, Sigplan Notices Vol.21 Nov 86, Portland, Oregon, September, 1986, pp. 17-30.

7. Erman, L. et.al. ABE, Architectural Overview. In *Distributed Artificial Intelligence, Research Notes in Artificial Intelligence*, Pitman Publishing Ltd., 1987.

8. Fox, M.S., McDermott, J. The Role of Databases in Knowledge-Based Systems. Robotics Institute, Carnegie-Mellon University, 1986.

9. Rashid, R.F. An Interprocess Communication Facility for Unix. Report , Carnegie-Mellon University, Comp. Science Dept., June, 1980.

10. Stroustrup, B.. *The C++ Programming Language.* Addison-Wesley Publishing Co., 1986.

11. Sun Microsystems. Sun Remote Procedure Call Specification. Tech. Rept., Sun Microsystems Inc., 1984.

Simulation and performance evaluation of heterogeneous parallel robotic systems

Francois Bitz and Jon A. Webb

16 September 1987

Robotic systems are growing increasingly complex, in response to a desire for

- Increasing computer power.
- Increased flexibility of human interaction.
- Increased variety of sensors and motor control devices.

In response to this, the designer of such a system has had to construct heterogeneous networks of computers, which may incorporate simple real-time processors for motor and sensor control, powerful computers for image and signal processing, and general-purpose workstations for user interaction. The machines may be connected by a variety of communications media, including dedicated buses for closely coupled computers, and local area networks for computers that are less tightly coupled. Not only can the performance of each node vary, but also such important features as their operating systems, i/o throughtput and interfacing can be very diverse.

Achieving good real-time performance in such a system is difficult. The complexity of the system and the desire to make it useful for research makes it difficult to impose hard real-time constraints on the performance of individual modules, in order to apply traditional real-time systems methods to optimizing performance.

Instead, the designer of such a system may first construct it, then try to determine the constraints on performance. In doing so, he immediately discovers that:

- Bottlenecks in system performance, such as I/O bottlenecks, may not be discovered until the system is actually constructed. Moreover, these systems represent some of the most complex and critical applications of computers.
- Discovering the source of bottlenecks is difficult, since the interaction of different modules within the system cannot be observed without changing performance. Non invasive tracing techniques are usually not possible to implement.
- Answering questions such as the effects of improved hardware or different placement of modules on parallel computers is difficult, since the interaction of different modules can lead to significant second order effects in system performance.

It is therefore essential to use appropriate tools as early as possible in the design phase of such a system. Such tools should allow the designer to evaluate performance as well as give him the flexibility of changing the placement and characteristics of each component. For example a task might be able to run on any of the nodes by itself, but where it is eventually placed will affect the performance of the overall system.

The most appropriate tool is a simulator that can address these design issues. The object of performance evaluation and simulation is to determine the parameters which maximize the effectiveness of the system resources through improved throughput, resources utilization and response time.

We have implemented such a simulator in a high level language, namely C++, an extension of C with concurrent task facilities. The simulator allows multiple machines to be simulated concurrently. Each machine can run multiple tasks concurrently as well have its own operating system and scheduling scheme, such as FIFO, prioritized, or round-robin. The simulator can simulate such complex real time constructs as interrupts, semaphores and rendezvous. Tasks and machine communications can be implemented through queues (a basic object in C++) which simulate the communication media of the real system (e.g ethernet, mailboxes, or shared memory). Efforts are also under way to facilitate the user interface to the simulator through the use of a code generator. This becomes more crucial as the number of nodes increases since generating code is a very repetitive and error prone operation.

The simulator has been used to simulate the real-time control system of the Martin Marietta Autonomous Land Vehicle system in a component that used a Sun 3/160, the Carnegie Mellon Warp machine, and three standalone MC68020 processors to detect obstacles in laser scanner data. Simulation results suggested performance improvements by moving modules from the Sun to the standalone processors, therefore achieving greater parallelism.

We intend to model a demonstration of CMU's Autonomous Land Vehicle (Navlab) from which we have been able to gather real measurements (including task times, i/o throughput, and communication traffic). The simulator will be run in order to compare how well the model corresponds to the real system. The simulator will then be used to predict the performance of a Navlab demonstration which includes a sophisticated road following algorithm and obstacle avoidance. This demonstration will use some of the modules of the first demonstration with major hardware and software upgrades. The simulator will also predict how such a vision system will perform on other computer architectures. In this first version of the simulator module placement will be first done by the user. However one of the goals of the simulator and performance evaluator is to maximize performance given a set of constraints such as number of machines and communication medium. Therefore it is desirable to describe the different modules in a higher level language. We intend to benefit from some of the work done in the Software Engineering Institute's Durra project in the way tasks and modules are described. Another potential utilization of our simulator can be found in Carnegie Mellon's HET project in which a large number of heterogenous machines are connected together through fiber optic links and 16 by 16 optical crossbars.

We intend to use the simulator to address questions of

- Module placement, where modules can be placed on different computer nodes. Of great importance are the effects of translating a routine running in a general purpose computer to a specialized machine such as the Warp array.
- Communications network changes, especially including performance improvements resulting from the use of a reliable, dedicated real-time network in place of the unreliable Ethernet.
- Computer changes, especially including the division of parallel computers into multiple

parallel machines of smaller size. Preliminary results indicate that such a bifurcation could lead to an improvement in performance of up to two.

- Assessment of how much prior information is needed about each of the real components of the system in order to get reasonable good match between simulated and real performance. In general it is possible to reduce the complexity of routines so that it is not necessary to write the routines as they would appear in the real code. Sometimes it is even acceptable to reduce a single routine to a delay() or run() statement which will give acceptable estimation of the performance of the overall system.
- Simulation speed and computer requirements for simulation of a large number of machines. Of particular interest is the possibility to distribute the simulation over and array of processors (distibuted simulation).

Run-Time Monitoring of Tasking Behavior Using a Specification Language

Douglas L. Bryan
Computer Systems Laboratory
Stanford University

1 The Specification Language

TSL (Task Sequencing Language) is a language for specifying sequences of tasking events occurring in the execution of distributed Ada[1] programs. Such specifications are intended primarily for testing and debugging of Ada tasking programs, although they can also be applied in designing programs. TSL specifications are included in an Ada program as formal comments. They express constraints to be satisfied by the sequences of actual tasking events.

The general form of a specification is as follows:

when *activator-compound-event*
then *body-compound-event*
before *terminator-compound-event* ;

Informally, a specification has the meaning:

> Whenever the events specified by *activator-compound-event* occur, then the events specified by *body-compound-event* must occur before the events specified by *terminator-compound-event*.

A compound event is an expression constructed from *basic* events. A basic event can be thought of as an atomic action performed by a task. For example, "A calls B at E" is a basic event. The operations available for forming compound events include sequencing, conjunction, disjunction, and iteration.

The following is an example specification taken from a simulation of an automated gas station:

```
<< Pump_Protocol >>
when ?P accepts Operator at Activate
   then ?P accepts ?C at Start_Pumping =>
        ?P accepts ?C at Finish_Pumping =>
        ?P calls Operator at Change
before Operator calls ?P at Activate;
```

This specification places constraints on the actions of pump tasks.

[1] Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

2 Implementation Issues

The following are the goals of the TSL run-time monitor implementation:

1. automatically monitor for common kinds of problems such as deadness errors
2. allow the observation of events at a programming or specification language level, rather than at an architecture level
3. detect and report the violation of specifications
4. report problems as soon as possible after their actual occurrence
5. provide useful diagnostic information
6. minimize the effects on the underlying computation being observed

3 An Implementation

There are two major tools which comprise the Stanford prototype implementation of the TSL: the compiler and the run-time monitor. The compiler transforms TSL source code into Ada code which constructs data structures and interfaces the underlying computation to the TSL run-time monitor. (See figure 1.) During execution, the monitor is called. Using these methods, the TSL system is portable and can be used in conjunction with other Ada tools.

A token graph representation of TSL specifications is computed during compilation, and constructed during execution. These token graphs form the internal representation of the constraints placed on the computation. One token graph is built for each compound event. The graphs include a labeled arc for each basic event. At run-time, the monitor *matches* the observed behavior of the distributed system against these graphs, and determines when specifications are violated. Matching is performed by moving tokens from node to node, across arcs. Whenever a token reaches the finish node of the *body-compound-event* or the *terminator-compound-event*, the monitor determines if the specification has been violated.

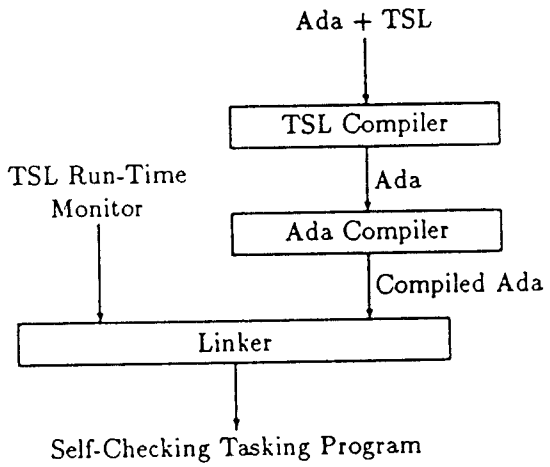


Figure 1: TSL Front-End.

The monitor also includes a user interface which allows one to interactively query the state of a run-time data base and the token graphs themselves. At any time during the computation, the user may examine the state of the graphs. The tokens on the graphs provide the user with a complete history of the computation, as it relates to the specifications. Similarly, when a specification has been violated, the state of the graphs provide the user with the chronology of events causing the violation. In this way, the TSL run-time monitor provides the capabilities of both a monitor and an interactive, specification level debugger.

4 Current Status

The prototype implementation of the TSL compiler and run-time monitor has been completed. This implementation has shown the feasibility and utility of specification level debugging of multi-tasking programs.

The interactive user interface of the monitor preserves the name space of the underlying computation. Events are reported, and the user requests information, using the names given in the Ada and TSL program. The violation of TSL specifications are reported during the execution of the final event causing the violation. That is, violations are reported as soon as they occur. At that time the user can interact with the monitor to determine the complete sequence of events leading up to the violation.

In the current implementation of the monitor, the specification checking code forms a critical region which is executed by the tasks of the underlying computation; the implementation relies on the fact that events are reported in a synchronous manner. During any user interaction, the tasks of the underlying computation are suspended. Thus, the monitor forms a bottle-neck, often causing tasks in the underlying computation to block.

5 The Event Reporting Problem

The fourth and sixth goals above are the main factors used to determine the architecture of the run-time monitor. It is desirable to report specification violations when they occur, and preserve program state while the user determines the cause of the violation. The simplest way to preserve state is to suspend the underlying computation. However, any such suspension has a drastic effect with respect to the minimal interference goal.

The problem is, are events to be reported to the monitor in an synchronous or an asynchronous manner? If asynchronous communication is selected, how does this effect the correctness of the specification checking code?

In a distributed system, certain events will always happen in a predetermined order. For example, some task must call another task before the second task

can accept the call. We refer to these event pairs as *connected events.* (Most events in a distributed computation are not connected. For example, if two tasks each call a third task, the order in which the calls occur is usually insignificant.) The means in which events are reported to the specification checking code must preserve the *connectedness* of events. The current implementation preserves connectedness simply by blocking tasks while an event is being processed.

6 Solutions Under Development

A number of monitor implementations are currently being studied or developed which provided alternative solutions to the report/interference trade-off. One alternative is to dedicate one or more processors to the monitor and make event reporting asynchronous. By doing so, we can reduce the processing overhead associated with the processors executing the underlying program as well as minimizing the blocking of tasks when events are reported.

Another alternative involves the distribution of the monitor itself. By executing the monitor on each available processor, the monitor on a given processor need only be concerned with a subset of tasks comprising the total computation. This approach reduces the processing requirements of a given execution of the monitor.

A new approach to monitoring TSL specifications is also being studied. In this approach, each specification is transformed into an Ada task. Each task would then be concerned with the monitoring of a single specification. In this way, the run-time monitor itself can be reduced to a common user interface called by these tasks. This approach relies on the Ada run-time system to perform load balancing and scheduling of monitor tasks.

Preserving connectedness is also being studies at the language, rather then implementation, level. It may be desirable to extend TSL and allow the user to specify connected event pairs. Then, under asynchronous event reporting, the monitor can shuffle the event stream to preserve connectedness. Such an approach would both extend the capabilities of the language and minimize the assumptions made by the monitor implementation.

References

Helmbold, D.P. and Luckham, D.C.
Debugging Ada Tasking Programs.
IEEE Software 2(2):47-57, March, 1985. In Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments, pp.96-110. IEEE, St. Paul, Minnesota, October 15-18, 1984. Also published as Stanford University CSL TR 84-263.

Ledoux, C., and Parker, D.S.
Saving Traces for Ada Debugging.
In Proceedings of the Ada International Conference '85. Cambridge University Press, 1985.

Luckham, D.C., Helmbold, D.P., Meldal, S., Bryan, D.L., and Haberler, M.A.,
Task Sequencing Language for Specifying Distributed Ada Systems, TSL-1.
Proceedings of the ESPRIT Conference "Parallel Architectures and Languages Europe", Eindhoven, The Netherlands, June 1987, Springer Verlag Lecture Notes in Computer Science, Vol. 259. An unabridged version of this paper published as Stanford University CSL TR 87-334.

Peterson, J.L.
Petri Nets.
Computing Surveys, 9(3), September, 1977.

Eager Evaluation in a Program Development Environment

Rick Bubenik
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas

We define eager evaluation as the execution of computations prior to the time they are required, with their results being stored in a temporary location. When at some later point those computations become necessary, we check if the eagerly computed results are still valid and if so, return them immediately without additional computation. Eager evaluation has the potential of providing very fast response time at relatively low cost in an environment where:

1. There are frequently idle computational resources so that speculative computations can be carried out without interference with other tasks.
2. There is a high likelihood of being able to predict the computations that will be necessary.

We believe these requirements are often met in a workstation environment where the program development procedures are described by some declarative description such as a *makefile*. Typically, in a workstation environment, most of the time the majority of machines are relatively idle. Consider what happens when a user is modifying several program files that compose some application. Typically, the user will edit the files, save the new versions, then rebuild the executable by issuing the *make* command. The rebuilding process usually involves generating object modules from each of the program source files, then linking these into a final executable file. When eager evaluation is applied to this environment, the evaluator anticipates the need to recompile each of the source files as new versions are saved and also anticipates the need to regenerate the final executable from the new objects. Then, when the user types make, the results can be returned as soon as possible.

More generally, we assume that the overall computation consists of a number of subcomputations whose relative order of execution is specified by an *execution* dependency graph. The individual subcomputations are carried out by one or more processes with no shared memory between subcomputations. These processes can perform arbitrary side effects by sending messages to server processes. The order in which side effects occur determines a *side effect* dependency graph. There is no communication between subcomputations other than indirectly, through side effects.

In order for the eager evaluation to be correct, we require that

1. No side effects become visible before the computation is *mandated* (requested by the user).

This research was supported in part by the National Science Foundation under grant DCR-8511436 and by an IBM Faculty Development Award.

2. After the computation is mandated, the side effects become visible as if the computation was executed normally at the time that the computation was mandated. In other words, the side effects should become visible in a serial order that is consistent with the side effect dependency graph, with the input set of the computation as of the time the computation was mandated.

We propose *encapsulations* as a mechanism for supporting eager evaluation. All processes carrying out part of a particular eager evaluation belong to the same encapsulation. Side effects remain invisible until the encapsulation is mandated. *Subencapsulations* can be used for grouping related activities. For example, the command(s) used to bring each *target* in a *makefile* up to date can be placed in a separate subencapsulation. When a subencapsulation is mandated, the effects concealed within it become visible to the external world. This facility is useful when only a portion of the eager computation is requested by the user.

An *encapsulation coordinator* monitors whether the various *make* dependencies remain satisfied, starts computations in encapsulated processes when dependencies are no longer satisfied, and logs the server-encapsulated process interactions in the order they occur.[1] If during eager evaluation the coordinator notices that one of its computations was performed in error, it undoes the corresponding log records, and (potentially) restarts the computation.

When an eager computation is mandated, the coordinator executes in two phases: a *consistency check* phase and a *writing* phase. In the consistency check phase, all read interactions are checked to determine whether the information on which eager evaluation was based is still valid. If a check fails (because the item read has since been modified), some parts of the computation need to be redone. If all checks succeed, the write phase begins. During the write phase, the side effects are made visible in the order in which they were logged. Since these effects were logged in the order they occurred, and since incorrect computations have been undone, the order in which side effects appear is correct in the sense we described above.

Unlike client processes, for which encapsulations are totally transparent and require no modifications, server processes have to be modified to participate in encapsulations. Essentially, they must log relevant interactions with the coordinator, and record output in a temporary location. As an example, consider how the file server can be modified. The file server handles encapsulations by checking all incoming requests. A request from a nonencapsulated process is handled normally, requiring no additional overhead. Requests from encapsulated processes are either handled normally or forwarded to an associated *encapsulation manager*, depending upon the nature of the request. The encapsulation manager then takes the responsibility of concealing side effects. When new files and directories are created, the desired name is mapped into a temporary name. All subsequent accesses to these files are redirected to the temporary versions. When a file is opened, the encapsulation manager sends a request to the file server to open the appropriate version. The file server returns a `fileid`, which the encapsulation manager then passes back to the requesting computation. All future read and write requests specifying this `fileid` do not have to be forwarded to the encapsulation manager, but rather can be handled directly by the file server. Consequently, encapsulations do not impose (significant) overhead on what we conjecture to be the vast majority of file server operations—reads and writes. Other operations requiring special attention include deletions, renames, and certain query operations.

We believe that encapsulations are a more appropriate abstraction to support eager evaluation than *atomic transactions*. Although atomic transactions provide another mechanism for hiding side effects and ordering them appropriately, we believe that if an atomic transaction were used

[1] In fact, it only needs to record a limited subset of the interactions.

to encapsulate an eager evaluation, with the transaction committing (and hence making its side effects visible) when the computation is mandated, several problems would ensue:

1. If only a portion of a large eager computation is requested by the user, it would be impossible to commit only a subset of a transaction in order to return just the requested results. Alternatively, if separate transactions are used for each portion, results computed in one portion would not be accessible in another (i.e. an output file, such as an object file, would not be accessible as an input file to some later stage of the computation).
2. If some of the subcomputations require terminal input, subsequent to terminal output, it would be necessary to make some of the side effects visible before commit time, in contrast with the requirement that side effects be made visible atomically. We anticipate that the eager evaluator will block the computation in the case of terminal input (until mandate time), then make all previously computed side effects visible and continue executing normally.
3. More generally, there seems to be a fundamental contradiction, between the atomic commit of transactions, and our desire to make side effects visible in an order that is consistent with the side effect dependency graph. In particular, we feel that it should be possible for the user to abort the computation after observing some partial output. This would not be possible if the transaction had committed by virtue of the computation being mandated.
4. We believe that the cost of atomic commitment, especially in the case of a distributed two-phase commit, far exceeds what is needed for encapsulations. Much of the savings comes from reduced I/O and protocol overhead since individual side effects can be made visible in isolation.

None of this precludes taking advantage of transactions to support non-idempotent operations or to improve reliability.

Previous work on eager evaluation has largely concentrated on applicative environments. Our work is different in that we explicitly deal with side effects, and in that the grain of computation considered for eager evaluation is much larger. We believe that with a large grain of computation, the potential for eager evaluation increases significantly, since the overhead involved in dealing with the evaluations and masking side effects becomes relatively less important. Eager evaluation has also been incorporated in some other programming environments. However, the type of environment considered has typically been of the *tightly coupled* variety, where the environment has tight controls over the commands executed and the files accessed. These environments appear to have an easier job supporting eager evaluation due to the tighter controls. However, they do not appear to generalize easily to support eager evaluation of arbitrary computations.

In summary, we have described our concept of eager evaluation and its application in a programming environment. We have proposed encapsulations as a mechanism for supporting eager evaluation and outlined why we believe it would be superior to atomic transactions for this purpose. We are currently implementing eager evaluation for *make* running under the V-System to get some experimental evidence about the cost and the potential of eager evaluation in this environment.

An Object-Oriented Approach to Remote Procedure Call Stub Generation for Heterogeneous Environments

Eric C. Cooper
Michael B. Jones

Computer Science Department
Carnegie Mellon University

Extended Abstract

Construction of stub generators is currently a time-consuming, error-prone task: the state of the art is analogous to that of compiler construction before the advent of parser generators and retargetable code generators. We are engaged in research to advance the technology of stub generation, by approaching the problem with two new ideas. Although both have been explored in other areas of computer science and software engineering, we believe their application to the design and construction of stub generators is novel.

The first concept is parameterization. A particular stub generator can be classified according to various attributes, including

- the data definition language (DDL) it accepts,
- the external representation it uses,
- the internal form it uses,
- the target language it produces, and
- the marshaling conventions it expects.

We believe that these attributes should be viewed as parameters to the stub generation process, just as BNF grammars, intermediate languages, and machine descriptions have come to be viewed as parameters to the compilation process. Following the analogy further, we are led to the idea of a stub generator generator, like a compiler compiler: a higher-order tool that one uses to produce stub generators with particular choices for the above parameters.

The second concept is object-oriented design. The parameters we propose are complex structures; it is not immediately clear how to represent them. Table-driven schemes have been used in previous work for some of these parameters, but the approach appears limited and difficult to extend to the other attributes.

We plan to use the ideas of type inheritance and polymorphism present in object-oriented languages such as Simula, Smalltalk, and C++. The inheritance structures that can be expressed in object-oriented languages provide another way of representing the knowledge needed by a program, in addition to conventional modularization techniques such as abstract data types, modules, or packages. We propose to use type inheritance in stub generator

construction to encode choices of DDL, external representation, target language, internal form, and marshaling conventions.

The advantage of this approach is that it allows the design decisions for a particular parameter choice to be implemented at a high level in the type inheritance hierarchy, while factoring out the details implied by the choices of the other parameters. For example, one can implement code that defines some aspect of the marshaling conventions (the argument passing scheme, say) in terms of generic target language operations. The particular marshaling routines can then be generated by inheriting both these marshaling conventions and a particular choice of target language. If a different target language is mixed in, the code for the marshaling conventions need not change, and vice versa. This separation appears difficult to achieve in stub generators programmed in conventional languages. We believe it will yield an order of magnitude simplification in the construction of stub generators for heterogeneous environments, in which multiple DDLs, target languages, target machines, and external representations are the norm.

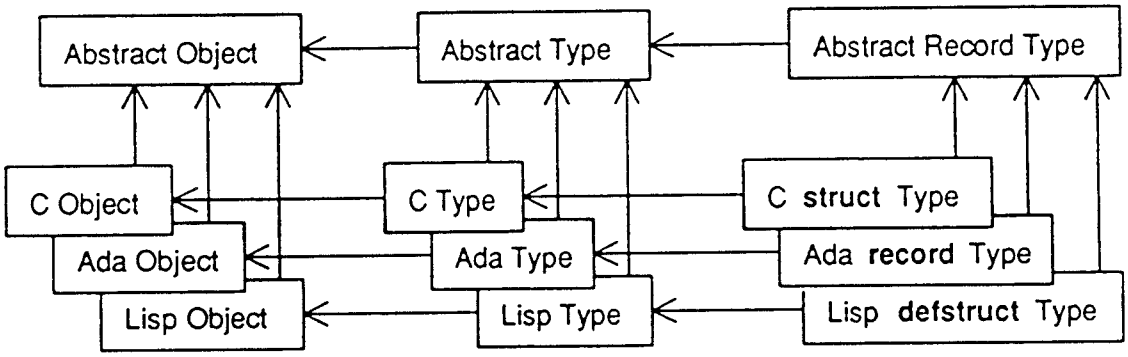


Figure 1: Class refinements for data type representation

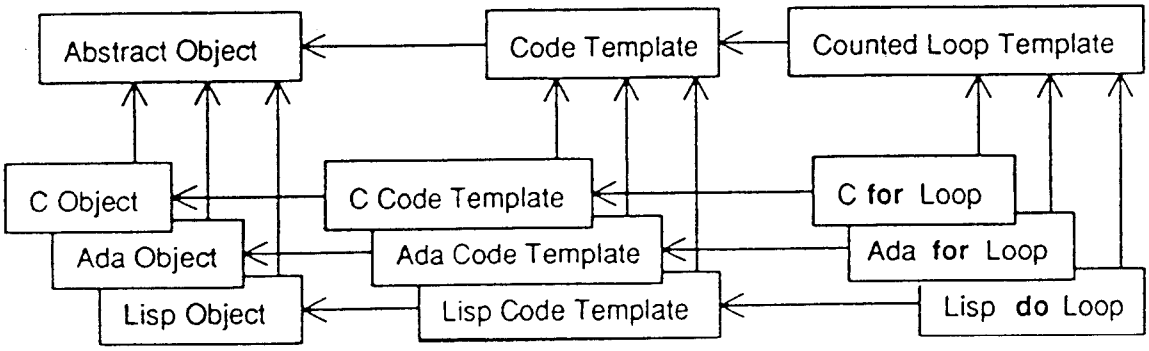


Figure 2: Class refinements for code representation

Research Plan

The first phase of our research project is to construct a prototype of a type transformation system, using an object-oriented approach. This will be a general-purpose tool for transforming typed data from one representation into another, with applications to remote procedure call marshaling and foreign function call interfaces. Type representations will be described in a language-independent fashion; code for type transformations will then be generated using language-specific code generators.

The second phase is to build a prototype multi-language code representation and generation system for use with the above type transformation system. This will provide a method of representing code templates in a language-independent fashion. Constructs such as assignments, type transformations, blocks, loops, conditionals, and procedure calls will be representable. Code generation will again be done by language-specific code generators. This will initially be used to represent and generate code for RPC interfaces of various kinds.

Our intent is to use an object-oriented approach for building both prototypes. Refinements of the class hierarchy will be used to represent refinements of specifications. For instance, a language-specific representation for a data type is a refinement of the language-independent declaration for that type; an Ada *for* loop is a refinement of an abstract counted loop. Figures 1 and 2 illustrate possible class refinements for type and code representation. We intend to implement the prototypes in C++, for several reasons: C++ is portable, commonly available, and produces efficient code. More importantly, it allows a fine grain of control over the declared objects and operators, including overloading of built-in operators.

A number of open problems must be solved during the course of this research in order to build reusable stub generators. We must find a way to describe type representations and remote procedure call formats independently of specific DDLs such as Matchmaker, Sun RPC, or Courier. We must also investigate how to specify type transformations in a way that is flexible enough for an environment of heterogeneous application programs, programming languages, and machine types.

Background

The authors have designed and implemented a number of stub generators and remote procedure call systems, including Courier, Matchmaker, and Flume (the DEC SRC stub generator). In the area of programming language design and implementation, we have worked on parallelism (C threads, Ada tasks) and exception handling in C++.

SAND87-2325A

Coupling a Network Computing Resource to a VLSI Placement Problem

Carl Diegert

Sandia National Laboratories, Albuquerque NM 87185

We describe a problem and its successful assault by a single user exploiting many computers, focusing on the strengths and weaknesses of (to borrow Apollo's term) the network computing environment in which we worked. Our solution was the best entered in a recent IEEE/ACM place-off competition, beating contestants using timeshared computers (many users sharing single computer) and contestants using workstations (computer allocated to single user). The aggregate compute power of the network allowed us more experimentation and search that workstation contestants. Mainframe contestants, however, had more compute power available to them than the compute power we applied from our network. The strength of our network computing is identified as its convenience in carrying out our ideas, experiments, and analyses. Efficiency in coupling the network compute cycles to the problem is ranked as relatively unimportant.

The competition problem was to give physical locations on a two-dimensional integrated circuit chip for about 3000 predesigned pieces (standard cells) of a given microprocessor design. The contest administrator then (ran the computer code that) interconnected these pieces, completing the physical design of the microprocessor chip. Our winning placement solution produced a microprocessor design with both the smallest chip area and the least amount of interconnect wire.

With a bit more abstraction than we actually used, the problem is to search through 3000! ways to assign the predesigned cells to grid locations on the chip, looking for an assignment (*numbering*) that will produce a small chip. This enormous discrete optimization problem is nasty in that attempts at greedy search quickly get stuck in local minima. The problem is challenging because the real objective of chip size is far too difficult to compute frequently during the search: statistical abstractions must be used for guidance. Stochastic search techniques addressed both the nasty and the challenging aspects of the problem.

The power of the network computing environment was in its convenience in setting up, executing, and analyzing experiments over variations in search technique, objective

function, etc. Designed experiments were necessary because the search techniques were stochastic: a desirable change in a parameter, say, is not apparent from comparing a single new run with an old run. Instead, trying a new idea comprised a sequence:

- design an experiment;
- set up a computer run for each experimental sample point;
- execute the independent runs, usually in parallel;
- analyze results.

The Apollo network single-level store and the network's remote process facilities were adequate for our pioneering effort. Madhat, the code that searched for placement solutions, includes an flexible input parser. Madhat can digest problem setup commands that other tools generate from a sample space of parameters. Execution of the parallel runs was tedious and wasteful, but workable. With each Madhat run leaving results in the same subdirectory, we quickly and easily wrote codes for analysis of each experiment.

We accepted great inefficiency in coupling network computer cycles to our problems, as the network resource was entirely justified by the capabilities and productivity it offered its daytime interactive users. We didn't have, and didn't stop to develop a clever *location broker.* Instead, we resorted to using only network nodes that could complete a run overnight (or, on Fridays, over a weekend), and scheduled only a single run for a particular node on a particular evening (or weekend). The quantity and mix of nodes to be available on a given evening did influence the design of the experiment conducted that evening. We did this mostly by *asking around*, and with face-to-face negotiations and verbal agreements with other (human) users of the network resource. Adaptation of the allocation occurred by our collective human experience, and at most resulted in changes from one evening to the next.

Synchronization of parallel Madhat runs occurred at most a few times each day. Synchronization in this broad sense ranged from

a. judge which of several runs stopped with a numbering likely to lead to a good chip, and use this numbering as the starting numbering for more parallel runs,

to

b. note a high-level problem in subsequent completion of physical designs from a batch of solutions (a problem with feed-through cell management), introduce a

new nonlinear term in Madhat's objective function, set up an experiment with variations on the new term, and execute the experimental runs.

Synchronizing once a day was effective, but more frequent synchronization might produce a better solution, or might get to an equally good solution quicker. If the computing network offered better support, we might try a *type a* synchronization a couple of times an hour. The human thought needed at most synchronization points would still be limited by the single user's ability to understand what the computers had done, to develop a new idea, and to express execution of a new experiment to the computers. For the most part, these human interactions set the pace of our progress. We would have welcomed an intelligent *location broker*, and other network computing niceties, but we doubt that they would have gotten us to our solution much sooner, or that they would have gotten us to a better solution.

To couple yet more power to the same problem perhaps we need to move to a fourth environment, an environment with one problem, many computers, and many users. The computing network would still allocate its computing resource to execute experiments, now given by its several users. The network would now facilitate the users building on each other's methods and results. Steps forward, then, accidental or clever, might be more frequent. Borrowing from a Minsky title, this new height in integration could be called societal computing.

(Abstract unclassified, presentation unclassified.)

Transaction Processing in Heterogeneous Distributed Databases.

Ahmed K. Elmagarmid
Computer Engineering Program
121 Electrical Engineering East Building
The Pennsylvania State University
University Park, PA 16802
ahmed@psuecl.bitnet
(814) 863-1047

A heterogeneous distributed database is a system of interconnected DBMSs that use different strategies for data and transaction management. Though issues such as universal query languages, and global view and schema integration have been investigated, transaction management issues introduced by the integration of separate database management systems into one global database have not been widely addressed in the literature.

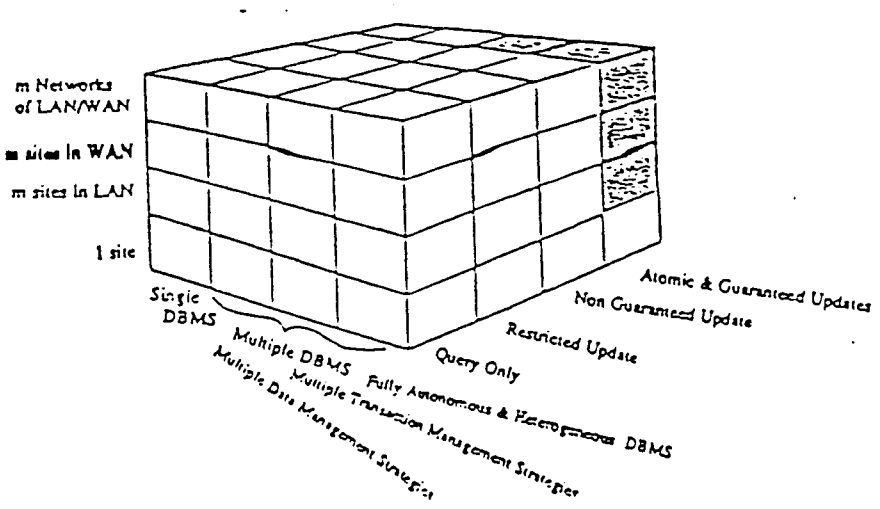
Probably most disturbing to me is the general misunderstanding in the database community as to what a heterogeneous database really is. Many people seem to refer to distributed databases as heterogeneous databases.

Of interest to me are questions relating to transaction support in the heterogeneous database environment. Two basic approaches are possible in order to integrate differing DBMS's. The first approach integrates transaction management policies into one global transaction manager that handles subtransactions accessing the heterogeneous database. The second is based on hierarchical composition of transaction management policies. In the latter approach, software is added on top of existing systems whenever needed (Gligor and Luckenbaugh, Interconnecting Heterogeneous Database Management Systems, IEEE Computer January 1984).

The heterogeneous database research group at Penn State consists of three students along with myself. We are looking at concurrency control (Y. Leu), recovery (D. Mannai), and deadlock issues (I. Mahgoub). In this workshop I would like to discuss the following set of related questions:

- Which of the problems are due to DBMS integration and not due to heterogeneity?
- How is the consistency of heterogeneous databases defined?
- How does serializability apply to heterogeneous databases?
- How strict must the definition of database autonomy be? How does it complicate transaction management issues?
- How important are atomic updates in this environment? How often are they expected?
- Are we likely to have generalized solutions to the problem of concurrency control and recovery?
- Specific algorithms we have developed for concurrency control and recovery in HDDBs.

In the discussion we would like to consider all possible systems depicted in the figure below. Especially in the shaded areas.


m Networks of LAN/WAN
m sites in WAN
m sites in LAN
1 site
Single DBMS
Multiple DBMS
Multiple Data Management Strategies
Multiple Transaction Management Strategies
Fully Autonomous & Heterogeneous DBMS
Query Only
Restricted Update
Non Guaranteed Update
Atomic & Guaranteed Updates

Hierarchical Process Composition

Stuart A. Friedberg
Computer Science Department
University of Rochester
stuart@cs.rochester.edu

1. The HPC System

The primary goal of the Hierarchical Process Composition (HPC) project is to provide tools for building, monitoring and maintaining long-lived, complex, distributed applications [LeF85a, LeF85b]. HPC is an experimental system for structuring applications, rather than a manager of applications or an application itself. It has roughly the same relationship to its clients and host operating systems that the X window system has to its. Where X provides an abstraction of nested windows, HPC provides an abstraction of nested processes.

The ultimate target environment is long-haul distributed systems: systems with substantial, variable communication delay and connectivity, and with independent site failures of significant frequency. This environment is further characterized by physical and administrative autonomy, and hetereogeneous hardware and software. The Xerox or DARPA internets and mobile packet radio networks are good examples.

HPC uses only two basic host facilities: conventional (heavyweight) processes and network interprocess communication. Large grain, loosely coupled processes are natural building blocks for distributed computations. The interactions between processes are subject to the same restrictions as interactions between hosts. They run autonomously, concurrently and asynchronously, communicate only through explicit shared interfaces, and know only their own state.

HPC builds applications from these large-grain processes and network protocols. First, cooperating processes are joined by creating communication channels between them. This composition leads to something like a dataflow graph. Each process can have several distinct ports, each presenting different functions or network interfaces. Second, groups of related processes are encapsulated as abstract HPC "objects" and treated exactly like single processes. This leads to a hierarchy or tree of active entities, where the leaves are real processes and the internal nodes represent larger and more complex activities with some degree of internal parallelism.

Our use of abstraction and composition is not novel. There are many design and analysis tools which describe a system at several levels of abstraction, where a black box at one level is a group of components with specific interrelationships at the next lower level (SARA, for one). This basic structuring also appears in several proposals for programming languages (DPL-82, PRONET), and in at least one implemented distributed operating system (CONIC). What is new or interesting about HPC?

- HPC process structure (the abstraction hierarchy and composition graphs) is completely dynamic. The HPC system is not a language for describing static structures, but a set of tools for building, maintaining, modifying, and tearing apart applications during execution. [Fri86]
- Everything in HPC is designed for an asynchronous, failure-prone environment. Rather than attempting to provide transparent synchronization and reliability at the HPC level,

explicit reports of failures and other unexpected events are provided to applications. Each application's manager can decide on the appropriate recovery or control policy and use HPC primitives to implement it.

- HPC runs on heterogeneous host systems. Each host's resources may be different, and no host is obligated to provide any specific network protocol or executable process image. Type information is used to prevent improper combination of host resources.
- Access control is based on application structure. The same hierarchy which defines abstraction is used to define protection domains. The agents and contents of a domain are immediately obvious, unlike access control list or capability list systems, and positive control of every domain is intrinsic to the HPC system.

2. Observations and Problem Areas

We are satisfied the HPC design meets its major goals. However, in the course of implementing and experimenting with the system, we found some problems related to HPC implementation on the one hand and application management and programming on the other.

To implement dynamic process structure while preserving abstraction, the HPC server needs to set up and tear down network connections without the cooperation of the processes being connected. We call this general capability *third-party connect.* Emulation of third-party connect for network protocol suites that do not support it is expensive, yet it is critical to reconfigurable, modular software. Designers of future protocols must separate the session and transport layers more carefully. [Fri87]

Since large-grain processes are loosely coupled, they should not have to synchronize often. However, they must resynchronize occasionally to apply end-to-end control, (re)authentication, flush transactions, indicate urgent data, and so forth. Dedicating communication channels to infrequent synchronization is wasteful, but the alternative is synchronizing out-of-band in the channels used for data, and many networking protocols do not support OOB communication. I would like to hear approaches for dealing with the general problem of OOB or synchronization marks.

Argus, Eden, and others, started with the viewpoint that interactions between distributed entities should be synchronous. We began with, and still hold, the opposite view: distributed interactions are intrinsically asynchronous. However, writing a program with multiple asynchronous interations is notoriously difficult, and we now provide a lightweight task library to support the illusion of synchronous interactions. As a result, *many* processes using HPC are structurally similar to Argus guardians, although the HPC system knows nothing of this internal structure, and atomic transactions are not provided (or desired). Our conclusion is that the grain of parallelism appropriate for programming (given existing methods and paradigms) is smaller than the grain appropriate for efficient use of distributed resources.

Even using lightweight tasks to simplify the programming, writing robust managers for survivable applications remains extremely difficult. The problem is coping with arbitrary asynchronous events (like process failure) when the primitive actions available in response are themselves asynchronous. At the moment, there is too little experience with actual managers to consider special languages or tools. Exploration of sample applications and their run-time management may be the most important use of the prototype HPC implementation.

3. Status

The HPC project began in mid-1984 as a "1 and a fraction" person project. Much of the last three years has been taken up with design issues, especially the interactions between distribution and control, to ensure a small set of features would support a wide variety of application management policies.

Currently we have a "wizard mode" prototype implementation running on Sun and Vax Unix hosts. All communication between parts of the system uses standard IP protocols, and applications can actually be spread across the DARPA internet, but the HPC server itself is not distributed. Over half the code, and by far the least attractive part, is dedicated to networking support and the client and host interfaces. (It has been a matter of discussion whether building on top of Unix or on top of bare hardware would have been more productive.)

There are several directions HPC-based research could take. Having this toolkit begs the question of how it can best be used, and experimention with various control policies for distributed applications is the most interesting research program. Second, it was always our intention to distribute the HPC service itself, but time and effort prohibited a full development of the distributed protocols required. This remains a challenging area, but one we don't feel obligated to tackle in the near future. Third, there were a number of design issues which we solved expediently but not properly. At some point a redesign that satisfies both our current frustrations and coming experiences with client control policies would be appropriate. And finally, there is always the desire to do "the last 10 percent" and distribute a high quality system for others to use.

[Fri86] S. A. Friedberg, "Control of Dynamic Process Structure - Policies and Mechanism", HPC Project Report 6, University of Rochester, October 1986.

[Fri87] S. A. Friedberg, "IPC for Modular Software Requires a Third Party Connect", Tech. Rep. 220, University of Rochester, June 1987.

[LeF85a] T. J. LeBlanc and S. A. Friedberg, "HPC: A Model of Structure and Change in Distributed Systems", *IEEE Transactions on Computers C-34*, 12 (December 1985), 1114-1129.

[LeF85b] T. J. LeBlanc and S. A. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems", *Proceedings 5th International Conference on Distributed Computing Systems*, Denver, Colorado, 13-17 May 1985, 26-34.

LARGE GRAINED DATA-DRIVEN APPROACH TO MULTIPROCESSOR PROGRAMMING†

Jean-Luc Gaudiot and Liang-Teh Lee

Computer Research Institute
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-0781

(213) 743-0249

Extended Abstract

The purpose of our research efforts as described in this paper is to investigate software methodologies for multiprocessor systems programming by using a data-driven approach to solve the problem of runtime scheduling. Indeed, the data-flow model of computation offers the potential for virtually unlimited parallelism detection at little or no programmer's expense. It has been applied to a distributed architecture based on a commercially available microprocessor (the Inmos Transputer). Some initial performance results of our system have been described in [Gaudiot *et al* 86] and [Gaudiot and Lee 87]. These results will be used for a comparison of the communication

† This research was supported in part by the U.S. Department of Energy under grant DE-FG03-87ER25043. The views presented herein are solely the author's and are not necessarily endorsed by the U.S. Department of Energy.

cost, degree of parallelism, and execution time of a matrix multiplication example, with and without loop unrolling among the different stages of partitioning.

A complete programming environment which translates a complex data-flow program graph into occam as well as a set of instructions for our simulator has been developed. A graph generator creates a program structure graph (PSG) and a data-flow graph (DFG). In accordance with the PSG and DFG, the code generator generates both the *occam* program and a set of simulation instructions. We will describe in detail the mapping from the SISAL (Streams and Iteration in a Single Assignment Language) high-level constructs into the low-level mechanisms of the Transputer. Synchronization between different processes, array handling problems, relay processes and some important program structures, such as vector operations, WHILE REPEAT / REPEAT UNTIL loops, and SELECT operations will all be discussed.

In order to increase the utilization of the Processing Elements in the system, maximize the parallelism and minimize the communication costs, several optimization techniques will be considered. The partitioning issues (granularity of the graph) will be presented and several solutions based upon both data-flow analysis (communication costs) and program syntax (program structure) are proposed and have been implemented in our programming environment. Based on the program structure and on heuristics, a high level partitioning algorithm which lumps together several actors to form the macro-actor and generates a partitioned data-flow graph can be implemented. The partitioning algorithm proceeds recursively: it traverses the PSG until the tree is exhausted. A large grained parallelism is obtained by the execution of all macro-actors concurrently upon data-flow principles of execution.

To achieve better performance, the following approaches have been studied in our research:

- *Communication cost thresholding:* lumping together of those partitions between which communication costs greater than a specified value to reduce the number of partitions and the total communication cost of the system.

- *Unrolling of loops*: for array operations, unrolling the loop body to

obtain a corresponding speedup.

- *Static and dynamic allocation:* making further partitioning and considering the type of interconnection networks, such as mesh and Hypercube connections, to achieve an efficient task assignment at compile time and runtime respectively.

For testing and analyzing of our graph allocation and optimization schemes, a set of benchmark programs, matrix operations, Livermore Loops, etc., have also been performed on a deterministic simulator to evaluate the performance of the translator on our proposed architecture (TX16).

References

[1] J. L. Gaudiot, M. Dubois, L. T. Lee, and N. Tohme, The TX16 : A Highly programmable multi-microprocessor architecture, *IEEE Micro*, October 1986.

[2] J. L. Gaudiot and L. T. Lee, Multiprocessor systems programming in a high-level data-flow language, *Proceedings of the Conference on Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, June 1987.

[3] J. L. Gaudiot and L. T. Lee, A methodology for multiprocessor systems programming, *submitted to the Journal of Parallel and Distributed Computing*, 1st revision.

SCHEDULING IN DISTRIBUTED SYSTEMS

Thomas B. Gendreau
Department of Computer Science
Vanderbilt University
Box 1679, Station B
Nashville, TN 37235

A central problem in distributed systems is the scheduling of processes onto processors. This problem is motivated by issues such as load balancing, parallel algorithm requirements, algorithm-architecture matching, and utilization of resources. Without a satisfactory solution to the distributed scheduling problem, the creation of efficient large grained parallel algorithms will not be possible.

Most work in distributed scheduling (in Local Area Network (LAN) environments) treats processes in the system as if they were independent entities. In many systems this is a reasonable assumption. However, if a system (either LAN based or message-passing multiprocessor based) wants to provide an environment for the development of parallel algorithms, then this is not a reasonable assumption. In a parallel application, processes will have certain relationships with other processes in the algorithm. These relationships can be described in terms of concurrency relations and communication relations. The concurrency relation indicates how much of the processes' work can be done concurrently. For processes that are not directly related it may be possible that all their processing can be done concurrently. For processes that share information the frequency of communication will be an important feature in determining the amount of work that can be done concurrently. For example, we could have two processes in the algorithm that do not communicate with each other and whose only purpose is to compute some result and send it to a third process. In this case the work of both processes can be done concurrently. At the other extreme we could have two processes that work in lock step with process A computing a result and sending it to process B and waiting for process B to do its work and returning a result back process A. In this case there is no concurrency between the processes. The communication relationship indicates the amount of information that is exchanged between processes.

In scheduling the processes of a parallel algorithm, the system will be able to make more intelligent decisions if it has information about the concurrency and communication relations and other features (e.g., expected lifetime or process creation patterns) of the processes in the algorithm. If the number of processes and the relationships between the

processes of the algorithm are known before execution, then this information can be provided to the system before the application runs. On the other hand, the number of processes and their relationships may be data dependent and thus not known until run time. In this case the system needs ways to gather information about the behavior of the program. This may be done by having the application program communicate with the operating system about important changes in behavior or by having the operating system learn about the behavior of the program.

When an application process creates a new process at run-time it may be able to inform the operating system about certain characteristics of the new process. These characteristics could include information about the relationship between the new process and existing processes and information about the new process' potential to create other processes. In order to do this appropriate protocols for the operating system/application processes communication will have to be developed. Some primary issues in this area include the discovering of what information is very helpful to the operating system and creation of algorithms that can use this information with a tolerable amount of overhead.

Having the system learn about the behavior of parallel programs is attractive because the programs are rarely developed to be run only once. Given this fact we can consider the possibility that the system can gather information from previous runs of a program. This information can be used in the management of future runs of the program. It has been recognized that some parallel programs go through a certain number of phases. These various phases may be characterized by different patterns of process creation and destruction and by changes in the concurrency and communication relationships. Ideally, the management programs would be able to identify that a program was about to exhibit a certain type of behavior and take actions (e.g., migration of processes) that would allow the program to run more efficiently. This is obviously an ambitious goal and a number of questions must be answered. First, what information should be kept from the previous runs? Second, how can the information be analyzed? Third, is the cost of collecting information during the running of the application and the cost of running the management programs justified by the increased performance of the applications?

We are currently investigating the appropriate operating system/application program interface and the problem of having the system learn about program behavior in LAN and message-based multiprocessor environments. We also examining ways in which *bidding* [Farb73, StSi84], *drafting* [NiXG85], or the *gradient model* [LiKe87] algorithms can be modified to make use of greater information about program structure and process relationships.

An additional issue to consider is a justification for dynamic creation of processes by the parallel algorithm. It has been an implicit assumption of this work that the flexibility provided by data dependent run-time process creation is useful. For example, a process that finds it has a large amount of computation to perform may want to create another process to do part of the work on another processor in parallel. Ideally this is an attractive concept. An important open question is at what frequency can dynamic process creation be handled efficiently. We hope that our research in the above problems will give us some insight into this question.

REFERENCES

[Farb73] Farber, D. J., et. al., "The Distributed Computing System," *Proc. Compcon Spring 73*, pp. 31-34, 1973.

[LiKe87] Lin, F.C.H. and Keller, R.M., "The Gradient Model Load Balancing Method," *IEEE Trans. on Software Eng.*, pp. 32-38, Jan. 1987.

[NiXG85] Ni, L. M., Xu, C. and Gendreau, T. B., "A Distributed Drafting Algorithm for Load Balancing," *IEEE Trans. on Software Eng.*, pp. 1153-1161, Oct. 1985.

[StSi84] Stankovic, J. A. and Sidhu, I. S., "An Adaptive Bidding Algorithm for Processes, Clusters, and Distributed Groups," *Proc. of the 4th Int'l Conference on Distributed Computing Systems*, pp. 49-59, May 1984.

Expressing Large Grained Parallelism Using Operator Nets

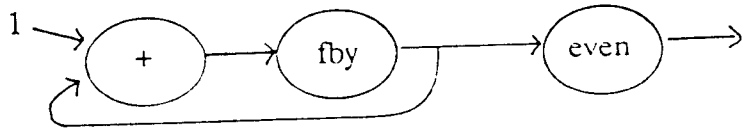
J.I. Glasgow, G.H. MacEwen and D.B. Skillicorn
Queen's University, Kingston

Introduction

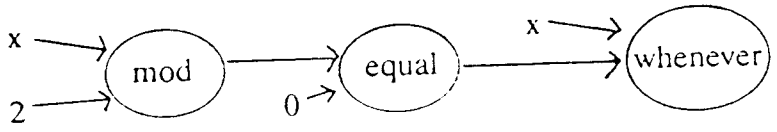
The graphical language, *operator nets* [Ashcroft85], provides a method for describing interprocess communication and parallelism in a distributed computing environment. An operator net consists of a set of nodes and a set of directed arcs corresponding to infinite sequences of data values from some underlying algebra. A program in the language consists of a set of equations that relate the output arc of a node to a function applied to the input arcs of the node. These equations can themselves be considered a language: the functional language Lucid [Wadge85].

A behavioral semantics for operator nets has been defined [Glasgow 1987a] in which properties of a distributed system are expressed in the operator net model in terms of the *histories* of an operator net and *events* that occur in such a net.

Operator nets can be used to express either fine or large grained parallelism. In the behavioral model for operator nets, a node and its associated equations are considered a process that consumes input sequences and produces an output sequence. These process nodes can either correspond to operators (fine grained) or to Lucid functions of any complexity (large grained). Each arc of a net is interpreted as a communication channel that carries messages from one process to another. For example, consider the Lucid function that inputs a sequence and returns the even values in the sequence: *even(x) = x whenever (x mod 2) equal 0;* This function can be represented as a single process and used to calculate all of the positive even integers as illustrated in the following operator net.



In this operator net we have three processes which could potentially be computing in parallel. For more fine grained parallelism we refine the function node into a subnet containing only operators, i.e.



so the resulting net would have five rather than three processes.

Current research in using operator nets to specify parallelism in distributed systems is centered around three projects: 1) Specifying and verifying security properties of computer systems; 2) Specifying real-time systems using Lucid and operator nets; and 3) Developing a formal theory of operator nets for reasoning about distributed systems. In the remainder of this abstract, we summarize each of these projects.

SNet Multilevel Secure System

SNet is a multilevel secure system being designed as part of a project investigating methods for specifying and verifying security properties of computer systems [Glasgow 1985,1987b, MacEwen 1987]. In particular, we are interested in methods that allow a natural decomposition of a security model into component models and then into functional components that can be verified and implemented independently from other models and components. Security properties of SNet have been specified and verified using operator nets. This approach has been particularly successful since it has allowed us to specify abstract constraints, using a behavioral semantics for operator nets, and concrete executable constraints using a Lucid specification.

The SNet design comprises host machines, secure terminal servers, and secure downgraders connected via an untrusted network. The current prototype contains three hosts, one downgrader, and one terminal server based on NS32000 processors connected via an Ethernet. The Lucid specification contains approximately fifty nodes of varying functionality. The implementation is a network of Concurrent Euclid processes that mirrors the structure of the operator net specifications.

Real Time Specification Using Operator Nets

This project involves the development of a methodology for specifying real time systems using Lucid and operator nets [Skillicorn 1986]. Given any Lucid functional specification of a system, the approach constructs two operator nets that describe the early and late time constraints of the system. These operator nets are sets of equations that capture all of the real time properties of the system and can be solved for any of the variables, given values for the others. For example, it is possible to answer questions of the form: what execution speed is needed to achieve a given set of input and output timings? Because the real time specification is written in Lucid, all of the formal techniques we have developed can be applied to the real time part, as well as the functional part. Thus it is possible to prove properties of the real time specification. Because the specification is executable, it is relatively easy to locate performance bottlenecks and places where the real time constraints are missed. We are working towards using our formal theory to allow statements about architectural constraints to be made and results concerning the relationship of architecture and performance to be proven.

Formal Theory of Operator Nets

One of the major problems with formal verification is that the languages used to reason about programs differ greatly from those in which systems are built. The underlying foundation of Lucid as a programming language was to provide a programming and proof technique that shared a single coherent structure. This was accomplished by defining the semantics of Lucid completely denotationally with mathematical properties such as referential transparency. Unfortunately, the program transformation rules provided by Lucid are sufficient for only a very limited kind of formal reasoning. We are currently developing a a proof system based on a behavioral semantics for operator nets. This theory will allow us to formally verify that Lucid specifications correspond to abstract specifications written in a logic language for operator nets.

The formal theory for operator nets is based on a behavioral semantics that intuitively models computations in a distributed system. This model has been extended to also allow for reasoning about knowledge, where knowledge is defined as a function of a process's initial knowledge, input history and reasoning capability.

References

[Ashcroft 1985]
E.A. Ashcroft and R. Jagannathan, "Operator Nets", in *Proceedings of IFIP TC-10 Working Conference on Fifth-Generation Computer Architectures,* North Holland, 1985.

[Glasgow 1985]
J.I. Glasgow, G.H. MacEwen, "A Two-level Security Model for a Secure Network", *Proceedings of the Eighth National Computer Security Conference,* Gaithersburg, Sept 1985.

[Glasgow 1987a]
J.I. Glasgow, G.H. MacEwen, "A Computational Model for Distributed Systems Using Operator Nets", *Proceedings of Parallel Architectures and Languages Europe* (PARLE) Conference," Eindhoven, pp. 243-260, June 1987.

[Glasgow 1987b]
J.I. Glasgow, G.H. MacEwen, "The Development and Proof of a Formal Specification for a Multi-level Secure System," *ACM Trans. on Computer Systems,* Vol. 5, No. 2, May 1987.

[MacEwen 1987]
G.H. MacEwen, V. Poon, J.I. Glasgow, "A Model for Multilevel Security Based on Operator Nets", *Proceedings of IEEE Symposium on Security and Privacy,* Oakland, April, 1987.

[Skillicorn 1986]
D. Skillicorn, J.I. Glasgow, "Real-Time Specification Using Lucid", Department of Computing and Information Science, Queen's University, Technical Report, 1986.

[Wadge 1985]
W.W. Wadge, E.A. Ashcroft, "Lucid the Dataflow Language", Academic Press, 1985.

Mentat: A Prototype Macro Data Flow System †

Andrew S. Grimshaw, Jane W. S. Liu, and Mark D. Thomas
Dept. of Computer Science, Univ. of Illinois, Urbana, Illinois 61801

Mentat [1] is an object-oriented macro data flow system designed to facilitate parallelism in distributed systems. The macro data flow model of computation [2,3] is similar to the traditional, large grain data flow model [4–7] with two differences: 1) some macro actors are persistent and maintain their internal state between firings, and 2) program graphs are dynamic. Mentat objects implement macro actors. Each object implements an actor for each member function of the object class. Mentat program graphs are constructed at run time. Graph nodes are actors, each of which may be elaborated into an arbitrary subgraph at run time by the node. Graph structure information is carried with the tokens. Thus, control of graph execution is completely decentralized. Parallelism is gained when different portions of the graph execute on different processors.

The Mentat programming language is an extended C++ [8]. C++ was chosen for several reasons. It is simple, efficient, object-oriented, and has no parallel constructs already built in. The objective of the extensions is to facilitate the writing of data driven objects and the construction of program graphs. These extensions are implemented by a preprocessor that translates the extended language programs into C++ programs augmented with calls to Mentat library routines. The library routines interface with the Mentat virtual macro data flow machine. Once compiled, the programs can then be executed on the virtual machine. The preprocessor provides for the definition of actors and independent objects, the automatic detection of macro data flow, the generation of code to construct program graphs, and optional programmer control over scheduling decisions.

The Mentat programming language consists of the following four principle extensions:

(1) the keywords *Mentat*, *persistent* and *regular* in class specifiers
(2) *select/accept* statements
(3) the predefined member function *main()* in class definitions
(4) implicit generation of subgraphs

These extensions make the power of data driven computation easily accessible to programmers.

The keyword *Mentat* in the class specifier indicates that the class is to be a Mentat class. Furthermore, the Mentat class may be declared either *persistent* or *regular*. The syntax for Mentat class definition is

[persistent|regular][Mentat] class-specifier

Instances of persistent classes maintain state between firings, whereas instances of regular classes do not. Each instance of a Mentat object has a separate thread of control. The member functions of the Mentat classes implement actors. Mentat objects are similar to monitors [9]; no two actors for a particular instance of a Mentat object may execute simultaneously.

In standard C++, there is only one thread of control. As a consequence member functions will always be executed when called. Mentat objects must be able to specify which operations are candidates for firing. *Select/accept* statements are added for this purpose. Guards can depend on the contents of local variables and the arguments of the member functions. By including the

† This work was partially supported by NASA Contract NAG-1-613.

arguments of the member functions in guards we provide the programmer with additional scheduling flexibility.

The programmer may also define a *main()* procedure for persistent Mentat classes, e.g., account::main() {...};. The main() procedure is started by the underlying machine once the object has been instantiated. The main() procedure is the active portion of the persistent object. It represents the thread of control in the object, and when it terminates the object is destroyed. Under most circumstances the main() procedure will be used as an outer control loop determining which operations to accept. If no main() procedure is present the preprocessor will generate one with an accept statement for each member function.

One of the purposes of the preprocessor is to detect data flow between actors and to generate code to construct data flow subgraphs. Data flows between two actors when either the result of one actor is used as an input to another, or when one actor invokes another actor. The preprocessor detects data flow by imposing an implicit single assignment rule on all variables used on the right hand side of expressions involving Mentat objects. We call these variables result variables. We call the Mentat operation that produces the result the source operation. Each time a result variable is used on the right hand side a new instance of the variable is created. Then, when the result variable is used on the left hand side of an expression an arc is created between the source operation and the expression on the right hand side. Graph construction proceeds at run time until either a result variable is *forced* or a *return_to_future* is encountered. A result variable is forced when it is used on the right hand side of a strict function. A return_to_future indicates that the invoked operation is complete.

We have implemented a prototype virtual macro data flow machine to execute macro data flow programs on a ten processor Encore Multimax. We plan to use this prototype to evaluate the functionality and performance of the macro data flow model, and as a test bed for the preprocessor. The preprocessor is currently in the design stage. A prototype version of the preprocessor will be complete by December, 1987. The feedback obtained from implementing the prototype version and from the evaluation will be used to reshape and refine our design for another version of the prototype Mentat system (preprocessor and virtual machine) to be implemented on a network of workstations, and on a multiprocessor system.

References

[1] Grimshaw, A. S., and J. W. S. Liu, "Mentat: An Object-Oriented Macro Data Flow System," To appear in *Proceedings of the 2nd Conference on OBJECT ORIENTED PROGRAMMING Systems, Languages, and Applications*, ACM, 1987.

[2] Liu, J. W. S., and A. S. Grimshaw, "A Distributed System Architecture Based on Macro Data Flow Model," *Proceedings Workshop on Future Directions in Architecture and Software*, May 7-9, 1986.

[3] Liu, J. W. S., and A. S. Grimshaw, "An object-oriented macro data flow architecture," *Proceedings of the 1986 National Communications Forum*, September, 1986.

[4] Dennis, J., "First Version of a Data Flow Procedure Language," MIT TR-673, May, 1975.

[5] Gaudiot, J. L., and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proceedings of the 1984 IEEE Conference on Distributed Systems*, 1984.

[6] Brock, J. D., Omondi, A. R., and D. A. Plaisted, "A Multiprocessor Architecture for Medium-Grain Parallelism," *Proceedings of the 6th International Conference on Distributed Systems*, pp.167-174, May, 1986.

[7] Babb, R. F., "Parallel Processing with Large-Grain Data Flow Techniques," *IEEE Computer*, pp. 55-61, July, 1984.

[8] Stroustrup, B., *The C++ Programming Language*, Addison Wesley, Reading, MA, 1986.

[9] Hansen, P. B., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, pp. 199-207, vol. SE-1, no. 2, June, 1975.

Avalon: Language Support for Reliable Distributed Systems [1]

Maurice P. Herlihy and Jeannette M. Wing
[1]Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable and recoverable. *Serializability* means that transactions appear to execute in a serial order, and *recoverability* means that a transaction either succeeds completely or has no effect. A transaction that completes all its changes successfully *commits*; otherwise it *aborts*, and any changes it has made are undone.

Avalon is a set of linguistic constructs designed to give programmers explicit control over transaction-based processing of atomic objects for fault-tolerant applications. These constructs are being implemented as as extensions to C++ [5]. The constructs include new encapsulation and abstraction mechanisms, as well as support for concurrency and recovery. The decision to extend an existing language rather than to invent a new language was based on pragmatic considerations. We felt we could focus more effectively on the new and interesting issues of reliability and concurrency if we did not have to redesign or reimplement basic language features, and we felt that building on top of a widely-used and widely-available language would facilitate the use of Avalon outside our own research group.

A program in Avalon consists of a set of **servers**, which resemble Argus *guardians* [4]. A server encapsulates a set of objects and exports a set of **operations** and a set of **constructors**. A server resides at a single physical node, but each node may be home to multiple servers. An application program explicitly creates a server at a specified node by calling one of its constructors. Rather than sharing data directly, servers communicate by calling one another's operations. An operation call is a remote procedure call with call-by-value transmission of arguments and results. When a server receives an operation call, it creates a short-lived "light-weight" process to execute the operation. A server can also provide a special **background** operation called by the system after it is created.

The objects managed by a server may be stable or volatile. *Stable* objects survive crashes, while *volatile* objects do not. Internally, the storage managed by an Avalon server is organized as a three-level hierarchy consisting of volatile, non-volatile, and stable storage. Objects are updated in fast *volatile* storage, which does not survive crashes. Slower *non-volatile* storage, such as a disk, is used as a backing store for pages in volatile memory. Non-volatile memory survives *soft crashes*, but not *hard crashes*. Finally, *stable* storage, such as replicated disks [3], is used to keep a log of updates to stable objects. Stable storage survives all expected crashes.

---

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

Syntactically, a server resembles a C++ class definition, where the objects correspond to class members, the operations correspond to member operations, and the constructors correspond to constructors. At the statement level, Avalon provides primitives to begin and end transactions, either in sequence or in parallel. Each transaction is identified with a process.

Avalon also supports nested transactions. A transaction commits only if all its children commit or abort; a transaction that aborts aborts all its children, even those that have committed. A transaction's effects become permanent only when it commits at the top level. Thus, a subtransaction's effects need not be written to stable storage until its top-level transaction commits. Nested transactions can be used to make applications more robust. For example, if a subtransaction aborts, the parent transaction need not abort, but can execute an alternative subtransaction. Nested transactions also increase the level of concurrency within a single transaction since subtransactions may execute concurrently.

In Avalon programs, each data object performs its own synchronization and recovery. A transaction is guaranteed to be atomic if all the objects it manipulates are *atomic objects*. Avalon provides a set of built-in atomic data types that resemble typical built-in types (e.g., arrays and records), but these data types guarantee atomicity as well. Avalon also provides primitives to assist programmers in implementing their own atomic types. Serializability and recoverability are implemented for the built-in atomic types by Camelot facilities such as locking protocols, new value/old value logging, and commitment protocols.

A novel aspect of Avalon is that concurrency control is governed by a property called *hybrid atomicity*. Informally, hybrid atomicity requires that transactions be serializable in the order they commit. Hybrid atomicity is a *local* property; if each individual atomic object is hybrid atomic, then the system as a whole will be atomic. Hybrid atomicity encompasses a variety of concurrency control protocols. For example, hybrid atomicity is automatically ensured by two-phase locking protocols [1], but programmers can achieve higher levels of concurrency and availability by taking the transaction ordering explicitly into account [2]. To assist programmers in implementing their own hybrid atomic data types, Avalon provides a built-in transaction identifier type *tid*. The tid type provides a restricted set of operations that facilitates run-time testing of serialization orders and the state of transaction commitment. A second novel aspect of Avalon is that programmers may define type-specific *commit* and *abort* operations for user-defined atomic data types. The system automatically applies commit or abort when transactions terminate. User-defined commit and abort operations are particularly useful for defining application-dependent synchronization and recovery protocols that enhance concurrency and efficiency by exploiting specialized properties of the data type.

References

[1] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger.
The Notion of Consistency and Predicate Locks in a Database System.
Communications ACM 19(11):624-633, November, 1976.

[2] M.P. Herlihy.
Comparing How Atomicity Mechanisms Support Replication.
In *Proceedings of the 4th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. April, 1985.

[3] B. Lampson.
Atomic transactions.
Lecture Notes in Computer Science 105. Distributed Systems: Architecture and Implementation.
Springer-Verlag, Berlin, 1981, pages 246-265.

[4] B.H. Liskov, and R.W. Scheifler.
Guardians and actions: linguistic support for robust, distributed programs.
Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[5] B. Stroustrup.
The C++ Programming Language.
Addison Wesley, 1986.

Emerald: A Language to Support Distributed Programming

Norman C. Hutchinson
Department of Computer Science
University of Arizona

Emerald is an object-based language and system designed for the construction of distributed applications. The principle feature of Emerald is a uniform object model appropriate for programming both private local objects and shared remote objects. Emerald objects are fully mobile, and can move from node to node within the network, even during an invocation. Despite this highly mobile nature of objects, invocation of an operation on an object is location independent: the programmer need not know the location of an object when invoking it. Emerald also supports an *abstract* type system that concentrates on the specification, not the implementation of objects.

Emerald's goal is to simplify distributed programming through language support, while also providing acceptable performance and flexibility, both locally and in the distributed environment. Like Eden [3], Emerald's model of computation is the object. Objects are an excellent way to structure a distributed system because they encapsulate the concepts of process, procedure, data, and location. In contrast to a several existing distributed programming languages and systems that support separate computational models for local and distributed entities, Emerald supports a single object model. Emerald objects include private entities such as integers and Booleans, as well as shared, distributed entities such as compilers, directories, and entire file systems. All objects are programmed using the same model, and have identical invocation semantics.

While we believe that programmers deserve the consistency of semantics offered by a single object model, we do not accept the common criticism of object-based systems: namely, that they are too slow. To a limited extent, the Emerald compiler is capable of analyzing the needs of each object and generating an appropriate implementation. For example, an array object whose use is entirely local to another object may be implemented using shared memory and direct pointers, while another array that is shared globally requires a more general (and expensive) implementation that allows remote access. These multiple implementations are generated by the compiler from the same source code depending on the needs of a particular object. This approach simplifies the programmers task since he sees a uniform model, while providing an implementation whose cost is appropriate for the functionality required of each object.

One novel aspect of Emerald's uniform object model is its support for *fine-grained* mobility. Mobility in the Emerald system differs from existing process migration schemes in two important respects. First, Emerald is object-based and the unit of distribution and mobility is the object. While some Emerald objects contain processes, others contain only data: arrays, records, and single integers are all objects. Thus, the unit of mobility can be much smaller than in process migration systems. Object mobility in Emerald therefore subsumes both process migration and data transfer. Second, Emerald has language support for mobility. Not only does the Emerald language explicitly

recognize the notions of location and mobility, but the design of conventional parts of the language (e.g., parameter passing) is affected by mobility.

In traditional process migration systems, process are normally migrated by other entities (e.g., load managers) without their knowledge. In fact, it is often a major design goal of such systems to make it impossible for a process to notice that it has been migrated. In contrast, location is an attribute of each Emerald object and language primitives exist to move objects to new locations and determine the current location of any object. Making location part of the language semantics allows Emerald to be used for constructing applications such as load balancers and replicated servers that wish to manipulate location to increase their performance or fault tolerance.

We have recently instrumented the Emerald mail system to investigate the benefit of light-weight mobility for a particular application. The results of this are reported in [6]. Briefly, we compared the performance of moving mail messages addressed to users on other machines to that of accessing them remotely. For a typical (synthetic) workload, mobility allows the number of remote invocations to be cut in half, and the total number of network packets sent to be reduced by 33%.

The Emerald language supports the concept of *abstract type*. The abstract type of an object defines its interface: the number of operations that it exports, their names, and the number and abstract types of the parameters to each operation. For example, the abstract type *Directory* specifies that directories implement the operations Add, Lookup, and Delete. Further, Add requires a string and an object (of arbitrary type), Lookup takes a string and returns an object (again of arbitrary type), and delete requires just a string. We say that an object *conforms* to an abstract type if it implements at least the operations of the abstract type, and if the abstract types of the parameters conform in the proper way.

Since abstract types capture only the specifications of objects (and not their implementations), they permit new implementations of an object to be added to an executing system. This is important for long-lived distributed applications such as mail systems, file systems, and window systems since it allows new kinds of objects to be fitted dynamically into a system without bringing the system down and restarting it. To use a new object in place of another, the abstract type of the new object must conform to the required abstract type. Note that each object can implement many different abstract types, and an abstract type can be implemented by many different objects.

Emerald has been implemented under 4.2BSD Unix on Vax and Sun computers, and is currently running on small networks at the University of Arizona, the University of Copenhagen, Denmark, and the University of Washington. A small number of applications have been implemented: a mail system, a shared calendar system, a file system, and a replicated name server. In addition, several load-sharing style applications have been implemented to experiment with light-weight mobility.

We are continuing work with Emerald along two major fronts. The first concerns replication. Emerald performs automatic replication of *immutable* objects (those that may not change their state over time). We have more recently been working on extensions to Emerald to support replicated *mutable* data. We wish to take advantage of the semantics of operations (in particular commutativity) to reduce the communication required to keep multiple replicas synchronized when they are updated. We are particularly interested in finding a clean language framework for dealing with replicated objects.

Secondly, we are interested in a stand-alone implementation of Emerald. Our current implementation on top of Unix does not allow us to evaluate the intrinsic costs of particular language features because of the large overhead associated with sending a network message under Unix. The

x-kernel being developed by Larry Peterson and myself at Arizona will provide a framework for constructing kernels that have specialized communication requirements. Customizing an Emerald kernel using the x-kernel as a base will allow us to better understand the fundamental costs of the abstractions that Emerald provides.

An overview of the Emerald language is given in [1]. The rationale for the design, and a description of the compiler algorithms used to deduce appropriate implementations are in [4]. The type system is described in [2]. An overview of the object migration facility is in [6], and the details of the implementation of the run-time system including garbage collection are in [5].

References

[1] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, ACM, October 1986. Published in SIGPLAN Notices, vol. 21, no. 11, November 1986.

[2] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987. Also Technical Report 86-02-04, Department of Computer Science, University of Washington.

[3] Andrew P. Black. Supporting distributed applications: experience with eden. *Proceedings of the Tenth ACM Symposium on Operating System Principles*, December 1985.

[4] Norman C. Hutchinson. *Emerald, An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, WA 98195, January 1987. Also, Technical Report 87-01-01, Department of Computer Science, University of Washington.

[5] Eric Jul. *Object Mobility in Emerald.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1988. In preparation.

[6] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 1987. to appear, Also University of Washington, Computer Science Technical Report 87-02-03.

Modelling Time Dependent Behavior in Parallel Software Systems

Debra S. Lane

University of Calfornia at Irvine
Department of Information and Computer Science
Irvine, CA 92717

A great difficulty in building distributed systems lies in being able to predict what the system behavior will be. A distributed or communicating system is defined here to be one in which the hardware consists of a set of processors each with their own memory, connected by some communication medium (there is no shared memory), and the software is assumed to be of the CSP (Hoare's Communicating Sequential Processes) type. The problem is that while it is easy to understand how each process behaves in and of itself, it is nearly impossible to predict all the ways in which the processes will interact and influence each other's execution. It is necessary to understand their interaction in order to determine how the system behaves (so that one might convince oneself or others that the system performs as intended).

In the past few years some theories have been proposed to model features of communicating systems. Milner's Calculus of Communicating Systems (CCS), Winskel's Synchronization Trees (ST), Hennessy's Acceptance Trees (AT), and Hoare and Brooke's theory of communicating processes are examples of formal models of such systems. All of these models concentrate on modelling observable properties of a system.

This paper presents a new representation of communicating systems called Event Dependency Trees (EDT) that models the time dependent nature of such systems. None of the representations mentioned above explicitly represent time but time is precisely the factor that introduces so much variability and complexity into such software and systems. Many models in computer science assume that events occur instantaneously, but here it is assumed that every event occurs with a certain time delay represented explicitly by an event name and a variable for the time delay. Communication events are important because that is how processes interact. Events preceding the communication events, even if they are only executions of sequential pieces of code, are also very important, however, because they determine the exact manner in which the communication events will occur.

Besides modelling time explicitly, EDT differs from CCS, ST, and AT in its representation of system behavior. Both CCS and ST represent system behavior

as interleavings of events. The combine tree operation in those models produces the set of interleavings. AT represents the system as a state–transition graph. The tree combine operation in AT takes two state–transition graphs and produces a larger one. In EDT, the system behavior is represented as a partial ordering of events. The combine tree operation in EDT produces the partial ordering of events in a way that indicates how particular sets of events contend with each other to produce the various execution paths.

EDT show the right amount of information about system behavior, not too much as in an interleaving representation, and not too little as in a state–transition model. It is possible to identify each execution path by its *unique* event ordering. In interleaving many event orderings produce the same execution path because many times it is irrelevant that some event occurred before or after another since they don't influence each other's execution. EDT shows exactly those events that influence each other's execution and also those that are not related.

EDT also provides answers to the questions "Why is one execution path chosen over another?" or "How is a particular execution path chosen?" The answer is that some set of events occurs before a different, contending set of events. CCS, ST, and AT all show the possible execution paths but indicate only that they arise because of nondeterminism. What is the source of such nondeterminism? There are two ways in which nondeterminism arises in such systems: (1) through the use of guarded commands, and (2) through the use of the communication constructs. EDT models the nondeterminism that arises through the use of communication constructs in CSP–type languages.

In EDT processes are represented as trees where the nodes of a tree represent system states and the arcs represent the execution of system events. An event is one of three types: (1) execution: represents the execution of a sequential piece of code (with no communication constructs), (2) communication: represents the execution of a message passing construct, or (3) the null event. Communication events are further subdivided into send, receive, and synchronized communication events. In addition, each event has an associated time delay, represented by some variable such as t.

The following notation is used:

1) $\overrightarrow{e[t]}$ denotes a sending communication event that takes time t.

2) $\overleftarrow{e[t]}$ denotes a receiving communication event that takes time t.

3) $\overleftrightarrow{e[t]}$ denotes a synchronized communication event that takes time t.

4) $e[t]$ denotes an execution event that takes time t.

5) τ_0 denotes the null tree, which is also the null event.

These are the only events that can occur in EDTs. Using this model, all portions of the computation that take time are accounted for.

Labelling trees is subject to some restrictions, which are not described here. However, note that each event has a name e, a time t, and a type that is in the set $\{exec, send, recv, sync, null\}$. The name of the null event, which is also the null tree, is ϵ or the empty string, and the time of the null tree is 0. The functions *name*, *type*, and *time* when applied to an event, return the respective information about that event.

Three operations are defined on trees: a prefix operation that allows a tree to be prefixed by an event producing a new tree (prefixing an event to the null tree results in a tree with a single arc labelled by the new event); a combine operation that takes two trees and produces a new tree; and a remove operation that takes two trees and removes one tree from the other. The combine operation is a very important one in that it preserves the relevant information that indicates how execution paths arise as a function of event orderings. A set of event dependency trees along with the combine operation is shown to form an algebraic group. In this model, two trees are defined to be equivalent if they are isomorphic to each other.

Once the formal model is defined the question becomes how does one use it. EDT is useful for performing some types of analysis. It is assumed that a programmer codes a piece of software. The software is then transformed into an EDT representation. At this point other algorithms are invoked to analyze the "software" for various properties or information. One type of information, which the model was designed explictly to produce, is the set of execution paths, identified by their unique event ordering. Once one has this information it becomes possible to ask questions of the form, "Will this execution path ever occur?", or in other words "Does event x always occur before event y, and if so what in the system causes it?"

Another type of analysis familiar to all is the detection of deadlock or proving the absence of deadlock. The algorithm detects two types of deadlock, deadlock due to wrongful use of the synchronization primitives, and deadlock due to timing aspects of the system.

EDT is a formal model of distributed or communicating systems that predicts how CSP–type processes will interact. Although it appears that EDT is a model of software, assumptions about how the system impacts the execution of the software is a crucial aspect of the model, the primary assumption being that events take time that could differ from execution to execution. From an EDT model of software one can identify each execution path by its unique event ordering. This provides some insight as to how one might reason about whether certain events and ultimately execution paths can occur. The model supplies potentially important information for the design and construction of distributed, parallel software systems.

LGP2 Position Paper

Paul J. Leach
apollo computer inc.
330 Billerica Road
Chelmsford MA 01824

Introduction

At Apollo, we have experience in three areas that are relevant to the creation of systems support for large grained parallel computations on a network of workstations and servers. First, we have collected from users a set of parameters that they consider useful in the selection of machines to be used to perform parallel computations. Second, we have identified some policy axes that mechanisms for machine selection need to support. Putting these two together, we created an architecture that allows the coexistence of many different policies, and for the user extension of the set of selection criteria. Finally, we have implemented several parallel applications that make use of load balancing.

Policies and Parameters

By polling potential users of large grain parallelism, we discovered that they had applications that would want to discriminate between potentially useful nodes on the following bases. CPU speed and load were the primary criteria mentioned, but the ability to distinguish between "foreground" and "background" load were also desired. Disk speed and load were secondary criteria: in fact, available disk space seemed to be more important; this may be because disk performance is not as variable in our environment as CPU performance. Main memory size was a criterion, but actual memory load measures were not requested at all, perhaps because of the prevalence of virtual memory in our environment, or perhaps because of the unavailability of good measures. Finally, the software configuration on a node would sometimes be a factor.

The users of nodes, in addition to wanting to limit remote process creation on their nodes to times of low load, however defined by the interaction of some of the above measures, also wanted to be able to force remotely created processes into the background, and to take interactive use of the node into account.

Policy axes: policies variations that the mechanisms need to support: individual autonomy; group ownership of nodes and the desire to limit their use to the owning group. Inter-user protection (or the lack thereof) would place constraints on simultaneous use of a node by different principals. Even if no-one were locally using a node, protection of files from remote processes would be an issue, especially on some sinlge user operating systems. Another axis is individual responsiveness versus group throughput.

Architecture

The basic mechanism that supports large grain parallel computations is just the ability to create processes on other nodes in the network. Policy is enforced by having each node retain the power to determine who and under what conditions remote processes may be created on that node. At the next layer up, a registry of nodes and the selection criteria they meet is kept as a "hint" mechanism for quickly locating suitable nodes.

Control over remote process creation

On each node, a control file is present in a known place, which contains the parameters defining allowable usage of the node. Terminology:

OWNER: the node owner; a person who can change the access permissions/availablity criteria for the node. Represented by write rights to the control file.

USER: a person who is allowed to use the node, assuming the rest of the criteria are met. Represented by having read rights to the control file.

KBUSER: the person at the keyboard (unless the "server" option is used).

HOLDER: the first person to start an unsafe program is the holder. Usually, it is the KBUSER, if there is one.

FRIEND: a person who is trusted by the holder of the node to run unsafe programs while the holder of the node is using it. The current implementation defines a friend to be a person in same project as the holder, or same person as holder, but with a different project or organization ID.

INVOKER: a program that invokes other programs (i.e., a DSEE builder).

SAFE: trusted; a program you are willing to let others run while you are using the node is said to be safe.

SAFE DIR: a directory holding programs that are assumed to be safe unless explicitly identified to the contrary.

FORCE: the owner (or holder) of a node can ignore load considerations by using FORCE to create a process onto a node. This is primarily for debugging purposes.

SLOT: a division of computational power of a node into units; at most one remote process can be created for each available slot on node.

Normally, you can create a process on a node if:

1) you have permission.
2) you are compatible with the other users of the node.
3) the node has a low enough load.
4) the keyboard user has not reserved the node for him/her self or friends.

Permission: You have to be the owner or a user.

Compatibility: You have to be the holder or a friend of the holder, or be running a safe program. If "only_safe_friends" is set, then even friends need to be running safe programs (see below).

Load:

a. There must be a slot available; the number of remotely created processes must not exceed "max_slots"
b. The CPU use must be less than "cpu_max" (less than "cpu_max_kb" if there is a KBUSER).
c. The keyboard user must not have typed anything for more than "min_idle" minutes.

Reservation:

a. If the KBUSER has "reserved" the node, then no-one can create a process, regardless of how low the load is.
b. If the KBUSER has "reserved_friends" then only he/she and his/her friends can create processes, regardless of how low the load is.

Load Balancing

Each node "volunteers" its computational power when its local policy module decides that it would accept the creation of some remote processes. A node volunteers is by registering its availability with what we call the compute slot allocator (CSA). When a node volunteers, it also informs the CSA of selection criteria that might be of interest to potential users of the node. The CSA maintains an attribute/value database for these criteria, and allows potential users to query the database as part of the selection process. Current load is one of the pre-defined attributes. However, nodes and users can create new attributes at will, allowing new selection criteria at any time. The CSA's database is regarded only as a source of hints about the state of nodes.

Implementation

We have implemented a process creation server, called the server process manager (SPM), that has essentially the policy manager described above, and does simple CPU load calculations. The Apollo software engineering system (DSEE) can use this facility to do parallel makes. The CSA is currently not implemented, so each DSEE user provides a list of candidate nodes, each of which DSEE polls to determine load; the least loaded are selected. Many of the policy ideas came from users who disliked having DSEE use their node for makes while they were trying to get work done.

A Programming System for Heterogeneous Distributed Environment

Insup Lee
General Robotics and Active Sensory Processing Lab
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

A programming system (DPS) designed to facilitate the implementation and execution of distributed programs is being developed. The purposes of DPS are to facilitate the development of large distributed programs consisting of programs written in different languages (currently C, LISP and Prolog) and to allow the programmer to exploit large-grain parallelism by distributing programs to different processors[2]. The system hides heterogeneity in the underlying programming languages, architectures and operating systems from the programmer. The underlying distributed system consists of a loosely coupled heterogeneous mix of computers including VAX 11/785, MicroVAX II's, Symbolics and HP 3000 connected by an Ethernet.

The salient features of DPS are that processes in a distributed program may be written in the appropriate language for the task, and that the configuration of these processes into a distributed program is separated from programming of individual processes. A DSP program consists of a set of communicating processes written in C, LISP, or Prolog. The configuration of the program is specified using a distributed configuration language[3]. The configuration language provides a simple and efficient way to synthesize a set of component programs to form a distributed program; that is, it supports programming-in-the-large for distributed programs. A configuration written in this language identifies component sequential programs and specifies process interconnection. To simplify the loading and execution of a distributed program, it also identifies resources needed for execution and specifies process assignment constraints. The compiler uses this information in determining process allocation, freeing the programmer from details about the underlying system. The run-time support of the programming system ensures that processes acquire resources before start executing and handles distributed termination.

The system currently supports message based communication between programs written in C, LISP, and Prolog on Ultrix[1], proprams written in LISP on Symbolics,

[1]Ultrix is a trademark of Digital Equipment Corporation.

and programs written in C on HP 3000[1]. To support typed messages, we have implemented a typed data communication package, which is a set of functions that provide the ability to transfer complex data structures between processes, with type and structure retained, even between dissimilar systems. For Prolog programs, we have also implemented remote predicate invocation [4]. We plan to implement remote procedure call for programs written in C and LISP.

References

[1] I. Lee, N. Batchelder, J. Chang, K. Mok, S. Ngai, J. Rozario, Q. Tan and K. Tran, *DPS User's Manual*, GRASP Lab Internal Memo, Dept. of Computer and Information Science, University of Pennsylvania, 1986.

[2] I. Lee and S.M. Goldwasser, "A Distributed Testbed for Active Sensory Processing", Proc. 1985 Int. Conf. on Robotics and Automation, St. Louis, Mo. March 1985.

[3] I. Lee, N. Prywes, and B. Szymanski, "Partitioning of Massive/Real-Time Programs for Parallel Processing", in Advances in Computers, Edited by M.C. Yovits, Prentice-Hall, 1986.

[4] I. Lee, O.M. Yasuda, and G. Hager, "Remote Predicate Invocation for Distributed Prolog Programs", TR-87-76, Dept. of Computer and Information Science, University of Pennsylvania, 1987.

The synchronous language SIGNAL.

P. Le Guernic, A. Benveniste

IRISA/INRIA Campus de Beaulieu
35042 RENNES Cedex
FRANCE

SIGNAL is a Real-Time Programming Language designed at IRISA to describe and implement algorithms onto multiprocessors systems. Firstly defined to realize Real Time Signal Processing applications, SIGNAL adresses a larger field of needs for programming tools and especially in the areas where automata are used.
Based upon formal properties, SIGNAL is used as the major support for correctness verification, sequential simulation and repartition of algorithms.To implement an application, the following method is applied:

1/ specification of the algorithm in the synchronous language SIGNAL; at this stage, we have a specification where the synchronisations and the parallelism capabilities have been analysed.
2/ this first step provides as a byproduct a FORTRAN program, which can be used for standard simulation purposes.
3/ tools are available to help the programmer in designing a multiprocessor implementation, while controlling the required modifications of the original program.

The language.

SIGNAL is a data-flow like declarative language. It is defined upon a small set of primitive operators acting on two kind of expressions:

1. **Signal expressions** (ie expressions with dated sequences of values as operands) define primitives cyclic processes (named **Generators**) in a definitionnal equation style; generators produce output Signal from input ones in a synchronous composition (ie calculus are assumed to have a zero-duration). For this purpose, the programmer is provided with two classes of operators:
   i) Natural extensions of standard functions (+, x, ...) to sequences of values for which signals are constrained to be synchronous;
   ii) A small and complete set of temporal instructions to generate the control part (synchronisation and logic) of the program:
   . *delay* operators, acting as fifo-registers;
   .*when* operators, to delete data according to the value of a boolean control signal;
   .*default* operators, to merge two signal with an implicit priority, (specified to avoid non functionnal behaviour).

2. **Processes expressions** define new processes from smaller ones in a **block-diagram building** style (à la Milner); two processes communicate by zero duration exchange of values. They are defined by using the following operators:
   -*Renaming* of signals (input- and output-) which give new external names to named signals;
   -*Connection* of signals which define an input signal as being the output signal identically named in the process; connections allow broadcasting of values; each input has no more than one output connected.
   -*Compostion* of processes putting together two sets of signal definition equations; input signals with the same name in the operands are stated to be identical; operands may not have two output signals with the same name.

This set of instructions provides suitable mechanisms for event based under- and oversampling of signals.

The group works in cooperation with the project **Signal Processing Architectures** (Michel Sorine, leader) at INRIA-Rocquencourt. This work is supported by CNET (French National Agency for Telecommunications).

These expressions may be structured by the means of **Process Declarations** compound of:

-an **Interface** which gives its name and describe the set of its external signals;
-a **Body** which contains a processes expression
-a set of **Local declarations** of signal and subprocesses.

The compiler.

In addition to standard verifications and calculus (types, context,...), compiling SIGNAL programs involve a static calculus of logical time properties and the production of the timed precedence graph.

1 Logical time properties.

Signals handling in a SIGNAL program state logical time constraints defining a system of algebraic equations over the finite field Z/3Z, its **clock calculus**:

i)to each signal S is associated a variable which value denotes at every moment of a virtual clock:
- -the absence of a value in S when 0,
- -the TRUE value for boolean S when 1,
- -the FALSE value for boolean S when -1,
- -the presence of a value in non-boolean S when 1;

ii)to each of the generators, a model of equation is associated;
iii)rules of equations composition are simply deduced from the semantic of processes expressions.

The correctness of the program in regard to parallel computation (starvation, nondeterminism) are studied on its associated equations sytem by using effective algorithms relevant to computational algebraic geometry.

2 Precedence Graph.

A SIGNAL program define a set of output signals from input ones using data-flow variables. The precedence graphe, associated with a program, is defined by a one-to-one function from the set of calculus to a set of nodes; an edge exists between two nodes N1 and N2 if and only if the result of the calculus associated to N1 is used in the calculus associated to N2; moreover, this edge is labelled with the clock denoting the moments when the dependance is effective. The absence of circular definition of signals is verified using the graph, before a sequential FORTRAN Program is generated for simulation.

The Multiprocessor implementation.

The pair {clock calculus, conditional dependence graph} is the convenient level of compilation for studying processor allocation.

At first, we define the notion of a functionnal subgraph as being a subgraph in which each input node (node preceded by an outer one) precedes each output node (node preceding an outer one); such a functionnal subgraph may be translated in any sequence of its nodes according to a greater order than its reflexive closure; every local optimisation depending upon the structure of the processor may be used. Functionnal subgraphs may be calculated by local algorithms.

We intend to define a set of tools to help the programmer in implementing signal programs on a multiprocessor by using hierarchical organisation of the graph. Functions are the atoms of allocation; the set of the atoms is partitionned in synchronous subset ; the set of these subsets is recursively partitionned with recpect to the inclusion of clocks. The first level of the hierarchy represents the architecture. This work is in progress at this time, whith two target architectures: the first is based upon Transputers and Signal Processors, the second is an IPSC.

The group works in cooperation with the project **Signal Processing Architectures** (Michel Sorine, leader) at INRIA-Rocquencourt. This work is supported by CNET (French National Agency for Telecommunications).

Optimistic Algorithms for Replicated Data Management

Darrell D. E. Long

Computer Systems Research Group
Department of Computer Science and Engineering
University of California, San Diego

Extended Abstract

1 Introduction

In a distributed system, data are often replicated for protection against site failures and network partitions. Through the use of replication, increased availability of data and reliability of access can be obtained. When data are replicated at several sites an access policy must be chosen to insure a consistent view of the data so that it appears as though there were only a single replica of the data. The view presented to the user must remain consistent even in the presence of site failures and network partitions.

The simplest consensus algorithm is *static majority consensus voting* [2]. Static majority consensus voting provides consistency control and mutual exclusion, but does not provide the highest possible availability of data since it requires that a majority of the sites to be reachable for an access request to be granted.

An attempt to remedy the short-comings of static majority consensus voting, known as *dynamic voting*, was introduced by Davčev and Burkhard [1]. Their algorithm improved the performance by allowing quorums to be adjusted automatically during system operation. The method that we propose, called *Optimistic Dynamic Voting*, operates on possibly out-of-date information, hoping for the best. It can be shown that the scheme provides mutual exclusion and that data consistency is preserved. There are many benefits to our scheme, including efficiency and ease of implementation.

2 Optimistic Consensus Algorithms

The family of algorithms that are known collectively as *dynamic voting* [1,3,4] represent an ideal by which we can measure more realistic consistency control algorithms. The dynamic voting schemes previously described rely on instantaneous information about the state of the system. Such information is unachievable even is the best of circumstances, and our experiments have shown that attempting to approximate the connection vector lead to unacceptable loads being imposed on the sites.

Our analyses indicate that maintaining state information at each access produces availability of data comparable to dynamic voting with a connection vector. Using information that is out-of-date does not affect the consistency of the data, but does sacrifice some availability of data. Since the method that we propose propagates connectivity information when an access is successfully made, the amount of availability of data that is lost is related to the rate at which the data is accessed.

The basis of our scheme is the algorithm for detecting whether the access request is originating within the majority partition. Since there is at most one majority partition, mutual exclusion is guaranteed and consistency is preserved. There are three sets of information that must be maintained: the partition sets, P_i, which represent the set of sites which participated in the last successful transaction, a transaction number, t_i and a version number, v_i, attached to each site.

Algorithm 2.1. *Algorithm for deciding whether the current partition is the majority partition.*

1. Find the set of communicating sites, call it R.
2. Request from each site $i \in R$ its partition set P_i, transaction number t_i and version number v_i.
3. Let $Q \subseteq R$ be the set of all sites with version numbers that match that of the site with the highest transaction number.
4. Let P_m be the partition set of any site in Q.
5. If the cardinality of Q is greater than one half the cardinality of P_m, or is exactly one half and contains the maximum element of P_m then the current partition is the majority partition.

The advantage of the algorithms that we propose is that they are nearly as efficient as static majority consensus in terms of the number of messages sent, and that their implementation is simple. There are no

assumptions made about the state of the network other that which can be found by examining the partition sets and version numbers. We have an advantage over the scheme proposed by Jajodia [3,4] in that we can, by simply changing step five of the above algorithm, incorporate lexicographical ordering or topological information into the decision process. Our early analyses indicate that topological sensitivity can greatly improve the performance of Optimistic Dynamic Voting.

3 Stochastic Analysis

In this section we present an analysis of the availability of data provided by our scheme. The previous work on estimating the availability of replicated data managed by dynamic voting schemes had assumed idealized consistency control algorithms that possessed instantaneous information about the system state.

The availability of data provided by optimistic dynamic voting is related to the availability of data provided by lexicographic dynamic voting by the rate at which access requests occur. As the access rate increases, the information available to our scheme regarding the system state becomes closer to the true state of the system and the availability of data increases. So long as the access rate is greater than the failure rate the performance of our scheme is very good; regardless of the access rate it is always superior to static majority consensus voting.

Theorem 3.1. *The availability of data afforded by Optimistic Dynamic Voting, $\mathcal{A}_O(n)$, approaches the availability of data afforded by Lexicographic Dynamic Voting, $\mathcal{A}_L(n)$, as the access rate approaches infinity.*

Our algorithm performs asymptotically as well as the original lexicographic algorithm. This can be shown by direct manipulation for small numbers of sites, as it is below for three replicas. Here ρ represents the ratio of the failure rate to the recovery rate, and ϕ is the ratio of the access rate to the recovery rate.

$$\begin{aligned}\lim_{\phi\to\infty} \mathcal{A}_O(3) &= \lim_{\phi\to\infty} \frac{2\rho^4 + \phi\rho^3 + 6\rho^3 + 3\phi\rho^2 + 11\rho^2 + 4\phi\rho + 6\rho + \phi + 1}{(\rho+1)^4(2\rho+\phi+1)} \\ &= \frac{\rho^3 + 3\rho^2 + 4\rho + 1}{(\rho+1)^4} \\ &= \mathcal{A}_L(3)\end{aligned}$$

And it can be shown for any number of replicas based on a general form of the state diagram.

Our method is simple and efficient. It provides consistency control, and more generally, mutual exclusion. The availability of data and the reliability of access afforded by our method is superior to static majority consensus voting for only a small increase in network traffic. We feel that because of this, and because of the simplicity of the implementation, that our policy will replace static majority consensus voting as the method of choice for replicated data consistency and mutual exclusion.

References

[1] D. Davčev and W.A. Burkhard "Consistency and Recovery Control for Replicated Files," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, (1985), 87–96.

[2] D.K. Gifford, "Weighted Voting for Replicated Data," *Proceedings of the Seventh ACM Symposium on Operating System Principles*, (December 1979), 150–161.

[3] S. Jajodia, "Managing Replicated Files in Partitioned Distributed Database Systems," *Proceedings of the Third International Conference on Data Engineering*, (February 1987), 412–418.

[4] S. Jajodia and D. Mutchler, "Dynamic Voting," *ACM SIGMOD International Conference on Data Management*, (May 1987), 227–238.

The FileNet System

Martin S. McKendry
FileNet Corporation
Costa Mesa, California

The FileNet® system is a tightly-coupled distributed processing system used for managing document images. The system exploits the very high storage capacities of optical disks to store images: a single 12" disk can hold 2.6 gigabytes of data, or 40,000 compressed images, each occupying 70 kilobytes. Optical disks are managed in an OSAR® (Optical Storage and Retrieval Unit, aka 'jukebox') library which can hold up to 200 disks. The maximum total capacity of an OSAR is thus 500 gigabytes, or 8 million images.

Each OSAR is controlled by an OSAR server. The server is a 68000-series processor, with memory, magnetic disk, and local area network. Servers also manage databases that map user-specified indexing data to image locations on optical disk. Access to the system is through workstations with bit-mapped displays. Workstations are usually diskless. All servers and workstations run a variant of Unix[1], although because the system is closed, this is not visible to users. The system's distribution is also invisible to users. To a user, the entire system appears to function as a single unit. The WorkFlo® system can be used to program the flow of documents between users of the system.

The FileNet product has been shipping since 1985. There are now approximately 100 systems installed worldwide. Major applications include mortgage processing, credit card operations, customer support, and management of technical drawings.

The first FileNet system was designed for small numbers of users, with few systems expected to support over 32 workstations. This rapidly proved insufficient, and much larger systems are now becoming common. It is clear that very large systems are desirable to many customers. Thus, the next challenge is expanding the system to support several hundred users. Supporting these users while maintaining the unity of the system is difficult. We expect to combine mechanisms for highly efficient caching, dynamic load distribution, and fault tolerance. Where necessary, we will exploit application semantics to control the costs of these mechanisms.

System Structure

The system manages distribution and associated parallelism in two distinct sets of mechanisms. A distributed file system supports most operations typically associated with file systems. Alongside the file system, an RPC-based

---

0. FileNet, OSAR, and WorkFlo are registered trademarks of FileNet Corporation. Specifications subject to change without notice.

1. Unix is a trademark of AT&T

mechanism provides applications with a means for managing distribution.

The file system is similar to Locus [Locus 81], except that at present it does not support replication. It implements location transparency, and presents users with a single view of the file system (i.e., the mount structure is global). Servers are the critical resource in the system, because much work is logically centralized at servers. Thus, the file system attempts to move all processing possible to workstations. In this spirit, directory traversals are performed at workstations. A sophisticated caching mechanism for file control blocks (inodes) and file contents permits most file management to be performed without contacting the server storing a file.

The FileNet application makes demands of the file system that differ substantially from those of a typical program development environment. We have exploited these requirements already, and intend to do more so in the future. For example, we intend to provide a form of replication that avoids the overhead of generalized schemes. In this approach, we will support replicated read-only files (all code files), and replication of storage services for temporary files. Because most permanent data is stored in data bases that bypass the file system, supporting only these limited subsets of the general replication problem will address most needs. We will still have to support mutable replicated data for access paths, configuration structures, etc., but we will not incur the overhead of generalized approaches. In the cases we must support, we are able to exploit application semantics to reduce costs. Usually, reconfiguration to accomodate failures can proceed concurrently with application processing.

The second major portion of the FileNet system that supports distribution is the RPC-based mechanism presented to applications. This mechanism is used to access most application services. It uses a global name server to bind applications to their services. An open protocol (currently XNS) is then used to communicate with services.

Databases are accessed through this mechanism. At present, no distributed databases are supported, and there are no distributed consistency mechanisms (other than those supporting the filesystem). This has been acceptable under current loads and processing requirements.

Application requirements are truly distributed. In typical FileNet systems, individual machines support bit-mapped user interfaces, storage of files on magnetic disks, storage of images on optical disks, control of special devices such as scanners and printers, and databases mapping indexing data to optical disk storage locations. Thus, a typical user query involves several machines: the user's (diskless) workstation may have to contact its operating system server for paged-out data, an index server to process a document query, and an optical storage server to retrieve a document image. Depending on the structure of the file system and the devices required to process the query, additional machines may be involved. Because most users access the same databases and optical disks, server loading is critical to the system's performance.

Summary

The FileNet system is an example of a distributed system that operates as a single unit. In many cases, accepted industry practices are inadequate to supporting the system's application demands. In other cases, a lack of generality in application semantics can be exploited in ways not possible by designers of more open systems. In future, we expect to continue this approach,

using industry standard techniques where possible, and developing our own techniques as dictated by the requirements of the system's applications.

Reference

[Locus 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., LOCUS: A Network Transparent, High Reliability Distributed System, *Proceedings of the Eighth Symposium on Operating System Principles*, published as SIGOPS Operating System Review, Vol 15, No. 5, October 1983.

Author's Address:

Martin S. McKendry
FileNet Corporation
3530 Hyland Avenue
Costa Mesa, California 92626

Usenet: ...ihnp4!trwrb!felix!martin
...hplabs!felix!martin

Requirements for the Performance Evaluation of Parallel Systems

Michael K. Molloy
CMU

Determining or predicting the performance of distributed and parallel systems has been difficult in the past. This will not change in the near future unless several prerequisites are met. Often, the designers of distributed and parallel systems are too busy solving problems in a design to worry about the performance of the final system. In the cases where performance has been addressed, prediction has been impossible because of the lack of knowledge about the actual use of the systems. (i.e. Will there actually be 'hotspots' in parallel systems built around multi-stage switching networks?) In order to properly address questions about the performance of a current or future system, a clear understanding of the workloads and their patterns is needed.

Before any realistic performance prediction can be accomplished, a more extensive base of experimental knowledge must be established. However, before an experimental base can be established, new measurement techniques and appropriate metrics must be defined. Many different systems have been measured by researchers, but the measurements tend to be self-serving and incomparable with each other. Guidance on what to measure, how to measure it, and how to report it is clearly necessary. As an example, consider the hardware monitors in the RP3 project at IBM. The hardware monitor is built into the system from the initial design, an admirable trait. However, the monitor simply keeps a histogram (separate counts) of the control lines internal to the architecture. This allows the analyst to find out how often certain actions occur in the system, but nothing is known about the sequence of actions (most importantly the sequence leading up to a critical event). A circular buffer holding the last few control patterns would have gone a long way to extending the monitor's usefulness.

Unfortunately, the problem is not as simple as adding some features to existing systems. It is a chicken and egg problem. How can we specify what should be measured and how, if we don't know what is going to happen? On the other hand, how can we find out what is going to happen, if we don't measure anything? The answer is a two phased approach. First, a methodology to measure a large universe of information in a condensed form is developed. Second, more specific probes, both software and hardware, are designed to zero in on possible problems or unusual phenomena uncovered (necessarily incompletely) in the first phase. The study of advanced systems at HP, SUN microsystems, and IBM has started with the acquiring of massive traces using large (640MB) highspeed (200MBps) memory arrays to meet the storage requirements.

As an example of the infancy of the measurement methodology, consider the recent problems with the ARPANET. After changing the ARPANET naming schemes to include domains and nameservers, the ARPANET quickly became overloaded. It is still overloaded and will cause problems for some time. No measurements were made for several months. No determination of the exact cause of the load increase

was made for several more months. The corrective action necessary has still not been determined. Yet the ARPANET was been an established system for many years and has a dedicated network management facility. The problem is found in the fact that the management facility had tools to locate normal communication problems and test IMP processors, but no mechanism to study the ARPANET as a distributed processing environment.

There is some hope. An example of an improved monitor design is the HP4972 LAN analyzer for ethernets. By limiting the scope of the environment, the design for the 4972 resulted in a flexible and powerful monitor. The design begins with the input filter concept for restricting sampling. It expands this with the concept of the circular buffer and storing triggers. It is therefore possible to sample some subset of the packets, buffer and store the two packets before some trigger (like an error, particular address, or collision). Such monitoring sessions are set up using a high level programming language for the acquisition and generation of data. This makes the monitor flexible enough to be used in evolving environments.

Proving Real-Time Communicating Sequential Processes Correct

K.T.Narayana
Department of Computer Science
Whitmore Laboratory
The Pennsylvania State University
University Park, Pa16802, USA
(814)863-0147 narayana@psuvax1.uucp

The seminal paper by Hoare[8] on a notation for Communicating Sequential Processes (CSP) introduced input and output commands as fundamental language concepts. Since then, programming based on message passing has been extensively studied[1, 6, 10, 18]. The regime has established itself as *distributed programming* and has been distinguished from concurrent programming in that each process does not share variables with others and cooperation is achieved using message passing. There have been significant advances made towards a formal theory for understanding the design, construction and verification of distributed programs [2-5, 7, 12, 15, 20].

When the basic notation of CSP is augmented with the *wait* commands, it offers capabilities for programming real-time distributed applications. There has been a proliferation of languages (both concurrent and distributed) which seek to facilitate the programming of real-time distributed systems. In spite of the availability of high-level languages and the programmed real-time applications, real-time programming continues to suffer from the absence of an adequate mathematically founded methodology for specification, design, construction, and verification. Recently, attempts have been made to alleviate this problem[9, 11, 17] in a denotational context by providing real-time models.

The first significant methodological advice for the construction of real-time systems comes from Wirth. In his paper on real-time programming[19], Wirth offers the following advice- "In order to keep real-time programs intellectually manageable, we recommend that they first be designed as time-independent multiprograms and that only after analytic validation they be modified in isolated places, where the reliance on execution time constraints are simple to comprehend and document." The remark though made in the context of real-time concurrent programming seems extremely relevant even for real-time distributed programming.

Our concern in respect of formal correctness of a real-time program, in the light of the above, would be to address the separation of concerns *as far as possible* and to coalesce the reasoning to a unified whole when it becomes imperative to do so. Here again the notion of what constitutes a specification of real-time program seems to be important. The simplest way of looking at a specification of a real-time distributed program is to regard that each individual process establishes a given timing behaviour and the distributed program establishes a functional behaviour. For a certain class of real-time programs, the

functional behaviour of the program in the real-time model tends to be the same as that in the interleaving model. Thus, while timing behaviour can be stated consistent with the real-time model, it is enough to state the functional behaviour with respect to the interleaving model for this class of programs. When we extend the class of programs to those for which the functional behaviour in the real-time model is different from that in the interleaving model, then we need a stronger specification for the functional behaviour. Further, the specification of the timing behaviour should be given consistent with the functional behaviour in the real-time model. We can address a third category of specification in which we can speak about the collective timing and functional behaviour of every process in each computation of the program. This form of specification is the strongest of all. Thus, the specification regimes reflect the grades of difficulty in proving the correctness of the real-time program.

By their very nature, real-time models tend to be complex. A proof regime offered only in the context of real-time models makes the task of proving real-time distributed programs daunting. On the other hand, interleaving models have the advantage that they are weak. Further, correctness theory for distributed programs based on interleaving models is well understood. Thus, a proof methodology which seeks to draw upon a proof of the functional behaviour of the program in the interleaving model shall have definite advantages in easing the task of proving real-time distributed programs.

Thus, we approach the problem of the design of a proof system for real-time distributed programs with the following steps.

a) Firstly, we develop a proof system $\mathfrak{C}$ for reasoning about the time behaviour of individual processes. In the proof outlines of the time behaviour of processes, we make assumptions about the state at various points, and further we make assumptions about the waiting behaviour of i/o commands. Assertions in the proof system $\mathfrak{C}$ are structured more on the lines of the *time predicates* of Nielson[14]. A meta-variable t_i identified with process P_i captures the notion of the advancing time of process P_i.

b) Secondly, we prove the logical correctness of the program using the *Cooperation Test* based proof systems of Apt, Francez and De Roever[2] and the total correctness proof system of Apt[3]. We may as well have chosen any other proof system, for example of Levin and Gries[12]. The central ideas remain the same, but their articulation may be different.

c) Finally, we couple the two proof systems together with capabilities

i) for validating assumptions about the state in the proof outlines of processes established using $\mathfrak{C}$,

ii) for obtaining the exact waiting behaviour of each of the processes at their i/o commands, and

iii) for restricting the states of the program to those obtainable in the real-time models only.

Step (c) ensures consistency with respect to real-time models. Essentially, step (c) consists of formulating a system of equations (linear) which involve existential elements quantifying the waiting behaviours of processes at each of its i/o commands. We call these equations *Characteristic Equations* of the system. Then we define *simultaneous solutions* to these characteristic equations which are *acceptable*. The *acceptability* criterion assures consistency with respect to the *Maximum Parallelism* model[16] of communication and prohibition of unnecessary waiting of processes.

In summary, we develop a proof system for real-time CSP[13]. We adhere to the central elements of AFR proof system[2,3] by requiring that assertions do not share variables. Further, assertions in the proof system consist of two parts; one treating timing aspects and the other dealing with the functional behaviour. As in AFR system, we make assumptions about the timing behaviour of i/o commands. We make provision for performing waiting analysis of the real-time program as part of the proof process. The waiting analysis provides for the determination of the waiting behaviours of processes at the i/o commands. This particular aspect eases the programmer from obtaining an assertional structure for the exact waiting behaviour of i/o commands by a priori analysis. Further, the waiting analysis part of our proof system could be automated. A strong invariant I_R introduced into the proof system serves, more or less, the same purpose in the real-time model as the global invariant I does in the AFR system under the interleaving model. We show by examples the use of the proof system for several classes of real-time programs. The proof system we develop is a total correctness proof system.

References

1. Andrews,G.R, Synchronizing Resources, *Trans. Prog. Lang and Systems 3*, (Oct,1981), pp. 405-430.
2. Apt,K.R, Francez,N and De Roever,W.P, A Proof System for Communicating Sequential Processes, *Trans. Prog. Lang and Systems* 2, (1980), pp. 359-385.
3. Apt,K.R, Proving Correctness of CSP Programs- A Tutorial, *International Summer School on Control Flow and Data Flow Concepts of Distributed Programming*, Munich, July, 1984.
4. Barringer,H, Kuiper,R and Pnueli,A, Now You may Compose Temporal Logic Specifications, *ACM-Annual ACM Symp. on Theory of Computing*, , 1984, pp. 51-63.
5. Brookes,S.D, Hoare,C.A.R and Roscoe,A.W, A Theory of Communiating Sequential Processes, *J. ACM 31*, 3 (July 1984), pp. 560-599.
6. Feldman,J.A, High Level Programming for Distributed Computing, *Comm. of the ACM 22*, (Jun,1979), pp. 353-368.
7. Frances,N, Lehman,D and Pnueli,A, A Linear History Semantics for Languages for Distributed Programming, *TCS 32*, (1984), pp. 25-46.
8. Hoare,C.A.R, Communicating Sequential Processes, *Comm. of the ACM 20*, 8 (1978), pp. 666-676.
9. Huizing,C, Gerth,R and De Roever,W.P, Full Abtraction of a Real-Time Denotational Semantics for an OCCAM- like LANGUAGE, *POPLS*, , 1987.
10. Inmos Ltd, *The OCCAM Language Reference Manual*, Prentice Hall International, 1984.

11. Koymans,R, Shyamasundar,R.K, De Roever,W.P, Gerth,R and Arun Kumar,S, Compositional Denotational Semantics for Real-Time Distributed Computing, *Information and Control*, . To appear. Also in 1985 Conference on Logics of Programs, LNCS 193.

12. Levin,G and Gries,D, A Proof Technique for Communicating Sequential Processes, *Acta Informatica 15*, 3 (1981), pp. 281-302.

13. Narayana,K.T, Towards Proving Real-Time Communicating Sequential Processes Correct, Research Report, Department of Computer Science, Pennsylvania State University, Aug 1987.

14. Nielson,H, Proof Systems for Computation Time, *FST&TCS-3*, , Dec,1983, pp. 258-273.

15. Pnueli,A, Application of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, *LNCS 224*, (1986), pp. 510-584, Springer-Verlag.

16. Salwicki,A and Muldner,T, On the Algorithmic Properties of Concurrent Programs, *LNCS 125*, (1981), , Springer-Verlag.

17. Shyamasundar,R.K, Narayana,K.T and Pitassi,T, Semantics for Nondeterministic Broadcast Networks, ICALP, July, 1987.

18. *Reference Manual for the Ada Programming Language*, United States Department of Defence, Washington, 1983.

19. Wirth,N, Toward A Discipline of Real-Time Programming, *Comm. of the ACM 20*, 8 (1977), pp. 577-583.

20. Zwiers,J, De Roever,W.P and Boas Peter Van Emde, Compositionality and Concurrent Networks Soundness and Completeness of a Proof System, *ICALP 1985 LNCS 194*, (1985), pp. 509-519, Springer Verlag.

Research in Parallelism at The University of Washington

David Notkin
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

September 14, 1987

The Department of Computer Science at the University of Washington has become increasingly involved in parallel computing research. Through a new NSF Coordinated Experimental Research Program grant, *Effective Use of Parallel Computing*, we have just acquired a Sequent Symmetry 81 multiprocessor, and we expect 10 DEC SRC Firefly multiprocessor workstations to arrive imminently. In the following sections we briefly describe two current related research efforts, concerning (1) the integration of heterogeneous computer systems and (2) systems that support various styles of distributed and parallel programming.

Heterogeneous Remote Procedure Call The Heterogeneous Computer Systems Project [6] is exploring ways to reduce the costs of integrating diverse system types into a computing environment. The general approach has been to build both high-level and low-level services that are flexible enough to accommodate multiple existing standards or models of computation. We have made progress in several areas including a remote procedure call facility [3], a naming facility [15], a remote computation service [5], a mail service [17], and a distributed file service.

The HCS RPC (HRPC) facility supports the emulation of existing RPC systems by allowing different transport protocols, control protocols, data representations, and binding protocols to be mixed and matched. The stubs of HRPC clients and servers are written in terms of abstractions of these protocols and representations; the abstractions are bound to a specific set of choices (for instance, TCP/IP transport and Xerox Courier data representation) when clients are bound to servers. This allows a single HRPC client to communicate with multiple servers written in various existing RPC systems, and vice versa.

One problem with RPC systems is the synchronous nature of the RPC paradigm, which has the potential to serialize processing in servers. To permit more parallelism, we have developed an abstract interface to light-weight process packages that allows us to employ various existing implementations in a manner similar to that by which we emulate different RPC systems.

The basic HRPC facility has been running for over a year. The HRPC system itself runs on UNIX systems, including VAXes, SUNs, and the Tektronix 4404 family. Run-time support includes SUN RPC (with both TCP/IP and UDP) on VAXes and SUNs and Courier RPC on Xerox Dandelions. We plan to accommodate the Fireflys and their RPC system when they arrive.

Our ongoing projects include: developing HRPC support for both Lisp and Smalltalk-80; exploring the construction of HRPC intermediaries called bridge servers that perform the necessary protocol and data translations to allow existing clients and servers to speak indirectly when they cannot speak directly; and investigating the utility of HRPC's support for heterogeneous data representations in a local context, supporting calls between languages that use different data representations [14].

Distributed and Parallel Programming Systems For a decade our department has been actively engaged in research in object-oriented systems and languages. The design and implementation of the Eden system, which was the first distributed object-based system [1] and provided location

transparency and object mobility, led to the Emerald system [8,9], which provided a new programming language specifically designed for distributed programming. The novel features of Emerald include (1) its use of a single object model for programming both small, passive, local objects (such as arrays) and large, active, distributed objects (such as mail systems), and (2) its support for fully mobile objects [12]. Emerald is highly efficient and Emerald invocations execute in approximately procedure call time on a MicroVAX. More recently, a Distributed Smalltalk system has been prototyped to examine issues in extending the Smalltalk environment to multiple networked machines [2].

Our experience with object-oriented systems has led to an examination of the use of object-oriented languages for parallel programming. We have recently designed and prototyped an environment called *Presto* [4] that currently runs on our 10-processor Sequent. Presto extends the C++ language to run in a multiprocessor environment. In Presto, objects encapsulate the notion of abstract data types, i.e., protected data that is operated on only by a set of procedures in the object. To promote parallelism, Presto adds the notion of the thread object, which is the the fundamental unit of execution. A Presto object can create multiple threads either to execute within itself or to invoke other objects in parallel with its execution. Presto also provides synchronization objects so that simultaneously executing threads can coordinate their activities. Threads are implemented in a highly-efficient way that permits their use in medium- to fine-grained computations. For example, early measurements show that applications that use hundreds of threads can perform competitively on our 10-processor machine.

Conclusion and Acknowledgments This abstract describes the work of many faculty and students in the department. Our particular expertise, in the context of the overall UW effort, is the Heterogeneous Remote Procedure Call system, programming systems for distributed and parallel environments, and programming environments for parallel computers.

This work is supported in part by the National Science Foundation under Grants DCR-8352098, DCR-8420945, and CCR-8611390, by an IBM Faculty Development Award, by the Xerox Corporation University Grants Program, and by the Digital Equipment Corporation External Research Program.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Trans. on Softw. Eng.* (Jan. 1985).

[2] J. K. Bennett. The Design and Implementation of Distributed Smalltalk. To appear, *Proc. of the Second Symp. on Object-Oriented Programming Systems, Languages, and Applications.*

[3] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. To appear, *IEEE Trans. on Softw. Eng.*

[4] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. Tech. Rep. 87-09-01, Dept. of Comp. Sci., Univ. of Washington (Sept. 1987).

[5] B. N. Bershad and H. M. Levy. Remote Computation in a Heterogeneous Environment. Tech. Rep. 87-06-04, Dept. of Comp. Sci., Univ. of Washington (Jun. 1987).

[6] A. P. Black, E. D. Lazowska, H. M. Levy, D. Notkin, J. Sanislo, and J. Zahorjan. Interconnecting Heterogeneous Computer Systems. Tech. Rep. 87-01-02, Dept. of Comp. Sci., Univ. of Washington (Jan. 1987).

[7] A. P. Black. Supporting Distributed Applications: Experience with Eden. *Proc. of the 10th Symp. on Operating Sys. Princ.* (Dec. 1985).

[8] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. Proc. of the Symp. on Object-Oriented Programming Systems, Languages, and Applications (Oct. 1986).

[9] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Trans. on Softw. Eng.* (Jan. 1987).

[10] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup Versus Efficiency in Parallel Systems. Tech. Rep. 86-08-01, Dept. of Comp. Sci., Univ. of Washington (Aug. 1986). To appear, *IEEE Trans. on Computers.*

[11] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. on Softw. Eng.* (May 1986).

[12] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. To appear *Proc. 11th ACM Symp. on Operating Sys. Princ.* (Nov. 1987).

[13] A. N. Habermann and D. Notkin. Gandalf Software Development Environments. *IEEE Trans. on Softw. Eng.* (Dec. 1986).

[14] D. Notkin and W. G. Griswold. Enhancement through Extension: The Extension Interpreter. *Proc. of the ACM SIGPLAN '87 Symp. on Interpreters and Interpretive Techniques*, pp. 45-55 (Jun. 1987).

[15] M. Schwartz, J. Zahorjan, and D. Notkin. A Name Service for Evolving Heterogeneous Systems. To appear *Proc. of the 11th Symp. on Operating Sys. Princ.* (Nov. 1987).

[16] L. Snyder. Parallel Programming and the Poker Programming Environment. *Computer* (Jul. 1984).

[17] M. Squillante and D. Notkin. A Mail System for Local, Heterogeneous Environments. Tech. Rep. 87-07-04, Dept. of Comp. Sci., Univ. of Washington (July 1987).

Supertransactions

Calton Pu
Department of Computer Science
Columbia University
New York, NY 10027

Increasing interconnection of computer systems produces heterogeneous distributed systems. To cope with heterogeneity in hardware, we port the same software (e.g. the Unix operating system) to different machines. However, integrating similar but different software packages presents another challenge. In this abstract, we propose the supertransaction approach to accommodate heterogeneity in distributed transaction processing systems. For brevity, we use the term *database* in the broad sense, to denote general transaction processing systems.

We define *supertransactions* as atomic transactions spanning more than one database. A supertransaction is atomic in the same sense of normal transactions; concurrent access should be serialized, and database consistency recovered from crashes. We call the components of the supertransaction component transactions, which run on element databases. In contrast, a nested transaction has subtransactions running in the same database.

If all the element databases are implemented the same way, supertransactions are the same as known as distributed transactions, for example, R* and TABS. A more interesting possibility is a supertransaction running on element databases of different origins. In other words, a supertransaction should support atomic updates across heterogeneous databases.

We introduce the design of *Superdatabase*, a heterogeneous database system to update different element databases consistently [1]. We assume that each element database provides local transaction processing, including crash recovery and concurrency control. Our approach is based on hierarchical composition (Figure 1). The element databases are the leaves, while the superdatabases are the internal nodes, extending crash recovery and concurrency control to integrate different elements.

Each element database must satisfy two composibility conditions. The first is on crash recovery: the element database must understand some kind of agreement protocol, for example,

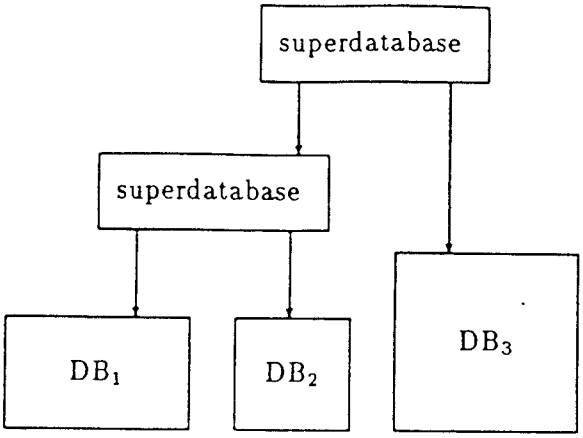


Figure 1: The Structure of Superdatabases

two-phase commit. The second condition is on concurrency control: the element database must present an explicit serial ordering of its transactions to the superdatabase. Fortunately, explicit serial ordering is easy to obtain from all major concurrency control methods (two-phase locking, timestamps, and optimistic concurrency control). For example, timestamps represent an explict serial ordering. A timestamp at the beginning of the shrink phase in two-phase locking also captures an explicit serial ordering.

Given element databases satisfying the above two conditions, the superdatabase that composes element databases can carry out two-phase commit (or any other agreement protocol understood by the elements) for crash recovery. To compose concurrency control, the superdatabase checks the explicit serial ordering of transactions from all elements, making sure they are serialized in the same order for all supertransactions.

This brief summary of superdatabase architecture only outlines a simple implementation of supertransactions. Detailed algorithms and refinements are described in another paper [1].

References

[1] C. Pu.

Superdatabases for Composition of Heterogeneous Databases.

Technical Report CUCS-243-86, Department of Computer Science, Columbia University. June 1987.

parmake and dp:
Experience with a distributed, parallel implementation of make

Eric S. Roberts and John R. Ellis
Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

Large software systems are typically developed as a set of smaller modules that are easier to manage individually. In order to provide automatic support for building a complete system and for keeping track of the dependencies between modules, facilities like the **make** program developed by S.I. Feldman at Bell Laboratories prove extremely useful. In an environment that permits concurrency (either through the use of multiprocessors or by using several machines on a distributed network), modular decomposition also provides a considerable opportunity for speedup, since the compilations of independent units can usually be performed concurrently.

The hardware base for this work is the Firefly—a shared-memory multiprocessor workstation developed at DEC's Systems Research Center to serve as our principal computing resource. Typically, each Firefly workstation contains five MicroVAX-II processors. There are currently 90 Fireflies on the local Ethernet, including two Firefly file servers. The Firefly operating system includes a remote file mechanism that provides transparent access to any file on any machine. Each Firefly has a local disk containing a partial Unix directory tree. User directories and portions of the public readonly directories (**/bin**, **/lib**, etc.) are stored on the local disk. The rest of the public directories are accessed transparently through remote symbolic links to the file servers. The most frequently accessed public files and programs are copied locally on each Firefly, kept up to date by nightly daemons. This arrangement approximates a cache-based distributed file system (which we are building).

To take better advantage of the available processing power, we have implemented **parmake**—an extension of the traditional **make** facility from Unix that provides for concurrent execution of those operations which have no mutual dependencies. Moreover, **parmake** can also take advantage of the facilities provided by our distant process facility **dp** to export some of that processing to idle Fireflies in the local area network.

The feasible orderings of the independent tasks are determined by topologically sorting the dependency graph provided by the **Makefile**. For the most part, the **Makefile** is the same as that used for the traditional **make** utility and requires no changes. In our early experience with **parmake**, however, we discovered that the local **Makefile** discipline often relies on the implicit left-to-right ordering, and we have added a backward-compatible syntax to allow programmers to make such dependencies explicit.

Within the set of feasible orderings, **parmake** uses a set of heuristics to balance the load on the local processors, while **dp** controls the scheduling of remote tasks based on machine-loading statistics. The heuristics are controlled by several parameters that reflect the relative cost of the independent operations. For example, the initial cost of invoking the **dp** mechanism (6 seconds) is large in comparison to the incremental cost of starting a new distant process (1 second) once the **dp** mechanism is initialized. To account for this, **parmake** does not invoke **dp** until the number of pending tasks reaches a relatively large threshold; once started, however, this threshold is reduced substantially to provide better load balancing.

The combination of **parmake** and **dp** provides capabilities similar to those of several other projects, including Locus at UCLA, Apollo's DSEE, the V system at Stanford, and Andy Tanenbaum's distributed make at CWI/Amsterdam. Our system is unique in two respects. First, it is compatible with the standard version of **make** and does not need to analyze the actual operation steps to provide speedup. Second, it is designed for use in a distributed network of multiprocessors and must therefore consider the proper balance between local and distributed concurrency.

Initial timings show that **parmake** reduces considerably the time required to recompile large systems. The following table demonstrates the speedup for a large-scale recompilation consisting of 238 Modula-2+ files drawn from various library packages at SRC. These files contain approximately 65,000 source lines, independent of imports.

processes	local	distant
1	1.00	0.95
2	1.68	
3	2.20	
4	1.90	
5	1.05	4.7
10		8.7
15		12.1
20		13.5
25		12.6
30		12.9

Table 1
Speedup for Modula-2+ Compilation

The "local" column shows the speedup using 1 to 5 concurrent local processes relative to the single process case. Even though the Firefly has 5 processors, the maximum speedup was 2.2. This is due to the large memory demands of our Modula-2+ compiler, which typically uses 5 megabytes or more of virtual memory to compile a file. Fireflies currently have 16 megabytes (of which several megabytes are required for the operating system), so running more than three simultaneous compilations results in thrashing.

The "distant" column shows the speedups as more concurrent distant processes are used, with each distant process on a separate idle machine. The processes read the source files from the single controlling machine and write the objects back to the local disk on the controlling machine. As the table demonstrates, maximum speedup occurs with approximately 20 distant processes, which provides about 65% utilization. When more processors are used, the processors on the controlling machine and the network bandwidth become limiting factors and no further improvement is seen.

The speedup is, however, strongly dependent on the specific nature of the computation being executed. The table below presents similar timing information for the recompilation of the X11 library, which consists of 194 C files. The actual source files contain only 8,400 lines, but the included files raise the total line count after preprocessing to 167,000:

processes	local	distant
1	1.00	0.82
2	1.88	
3	2.49	
4	2.70	
5	2.86	4.1
10		5.6
15		5.9
20		5.8
25		5.8
30		5.8

Table 2
Speedup for X11 Library (C-based)

Since the C compiler is much smaller than the Modula-2+ compiler, virtual memory is no longer a bottleneck, and the "local" column continues to show improvement throughout the 1 to 5 range. The C compiler also performs much more I/O relative to the amount of computation. The result of this is that the local compilation quickly becomes limited by the disk speed. Distributing the five process case to five machines results in a significant performance advantage, since each machine has an independent local copy of the libraries on **/usr/include**, and there is similarly no contention for **/tmp**, since this is also local to each machine. Even with this distribution, however, we do not see speedups above 5.9, since the time required to read the source files from the controlling machine limits the available parallelism.

Our experiments have demonstrated that it is possible to achieve considerable improvement in performance by adding local and distributed parallelism to the standard tools used to control recompilation. Moreover, the performance advantage increases along with the ratio of computation to I/O, as it does, for example, in optimizing compilers. We also expect that this performance will improve when we complete our current work on cache-based distributed file systems. A research report with more details on our experiments is forthcoming and will be available from SRC.

Extended Abstract

Coda: A Resilient Distributed File System

M. Satyanarayanan
James J. Kistler
Ellen H. Siegel

Department of Computer Science
Carnegie Mellon University

Distributed file systems have grown in importance in recent years. As our reliance on such systems increases, the problem of availability becomes more acute. Today, a single server crash or network partition can seriously inconvenience many users. *Coda* is a distributed file system that addresses this problem in its full generality. It is designed to operate in an environment such as the Andrew system at CMU [3, 5, 2], where many hundreds or thousands of workstations span a complex local area network. Coda aspires to provide the highest degree of availability possible in such an environment. An important goal is to provide this functionality without significant loss of performance.

Like Andrew, Coda distinguishes between clients from servers and uses caching of entire files as its remote access mechanism. In addition to improving scalability, whole-file transfer simplifies the handling of failures since a file can never be internally inconsistent. Coda masks server failures and network partitions to the fullest extent possible. Failures during a file operation are totally transparent at the user level unless the operation requires data that is neither cached locally nor present at any accessible server.

Aggregates of files, called *Volumes* [6], are replicated at multiple server sites. When a file is fetched, the actual data is transferred from only one server. However, the other available servers are queried to verify that the copy of the file being fetched is indeed the most recent. After modification, the file is stored at all the server replication sites that are currently accessible. To achieve good performance, Coda exploits parallelism in network protocols. We have an implementation of a parallel RPC mechanism that is capable of using multicast, if available. This mechanism can transmit files in parallel to multiple sites.

Consistency, availability and performance tend to be mutually contradictory goals in a distributed system. Coda will provide the highest availability at the best performance. A close examination of the way files are shared in an actual file system indicates that an optimistic policy regarding consistency is likely to be successful. Two principles guide the design of consistency mechanisms in Coda. First, the most recently updated copy of a file that is physically accessible must always be used. Second, although inconsistency is tolerable, it must be rare and always detected by the system. We may experiment with heuristics based on file access patterns to resolve simple cases of inconsistency. As in Locus [4], inconsistency is detected by the use of version vectors. However, Coda uses atomic transactions at servers to ensure that the version vector and data of a file are mutually consistent at all times.

At the present time Coda is in the detailed design phase. The implementation of the parallel RPC mechanism has been completed, but the bulk of the design and implementation work remains to be done. This includes areas such as recovery from failures, detection and resolution of inconsistency, file transfer protocols, and support for partitioned operation. The evaluation of Coda along the dimensions of performance and resiliency will also require considerable effort. Although much work remains, we expect that our use of the Andrew file system as a base, Camelot [7] for transaction support, and Mach [1] for operating system support will simplify implementation.

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 *Proceedings of the Summer Usenix Conference*. July, 1986.

[2] Howard, J.H., Kazar, M.J., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N. and West, M.J.
Scale and Performance in a Distributed File System.
In *Proceedings of the 11th ACM Symposium on Operating System Principles*. November, 1987.

[3] Morris, J. H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S. and Smith, F.D.
Andrew: A Distributed Personal Computing Environment.
Communications of the ACM 29(3), March, 1986.

[4] Popek, G.J. and Walker, B.J.
The LOCUS Distributed System Architecture.
The MIT Press, 1985.

[5] Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z. and West, M.J.
The ITC Distributed File System: Principles and Design.
In *Proceedings of the 10th ACM Symposium on Operating System Principles*. December, 1985.

[6] Sidebotham, R.N.
Volumes: The Andrew File System Data Structuring Primitive.
In *European Unix User Group Conference Proceedings*. August, 1986.
Also available as Technical Report CMU-ITC-053, Information Technology Center, Carnegie Mellon University.

[7] Spector, A.S., Thompson, D., Pausch, R.F., Eppinger, J.L., Duchamp, D., Draves, R., Daniels, D.S. and Bloch, J.J.
Camelot: A Distributed Transaction Facility for Mach and the Internet.
Technical Report CMU-CS-87-129, Department of Computer Science, Carnegie Mellon University, June, 1987.

ABSTRACT

Liuba Shrira
MIT Laboratory for Computer Science
Cambridge, MA. 02139
July 27, 1987

Dr. Liuba Shrira has received her M.Sc. and Ph.D. from Computer Science Dept. Technion Haifa, Israel. Her M.Sc. thesis was one of the first distributed implementations of CSP. Her Ph.D. thesis was on methodological construction of distributed and reliable algorithms. Since 1986, Dr. Shrira has been a postdoctoral research fellow at the Laboratory of Computer Science, MIT with Programming Methodology and Theory of Distributed Computing research groups. Dr. Shrira is one of the participants in the Mercury heterogenious distributed systems project at LCS. The main interests of Dr. Shrira are in the methodological design and analysis of distributed systems.

Recent Work

Within the Mercury project, Dr. Shrira has worked on the communication mecanism of the system. The Mercury heterogenious system aims at a general class of applications written in a wide variety of languages. The approach is to connect programs in a flexible and efficient way by a new communication mechanism called *stream*. This new mechanism combines the advantages of remote procedure calls and message passing. Remote procedure calls have come to be the preffred method of communication in a distributed system because programs that use procedure are easier to understand and reason about than those that explicitely send and receive messages. However, remote calls require the caller to wait for a reply before continuing, and therefore can lead to lower performance than explicit message exchange.

Streams allow a sender to make a sequence of calls to a receiver, without waiting for the reply to the previous call before making the next. The stream guarantees that the calls will be delivered to the receiver in the order they were made and that the replies from the receiver will be delivered to the sender in call order. Provided that the receiver executes the calls so that they appear to occur in call order, the effect of making a sequence of calls in the same as if the sender waited for the reply to each call before making the next.

However, new linguistic mechanisms are needed to use streams. For example, suppose

```
a := p(x)
b := q(y)
```

are two calls on the same stream, and what is wanted is to begin the call of q immediately after the call of p has been made. How can this be indicated? How can the results of the two calls be picked up without

error or confusion? What happens if one of the calls signals an exception? Finally, suppose a communication problem makes it impossible to complete one of the calls; how is this indicated?

A new kind of data type called a *promise* was invented to integrate streams into programming languages. Promises support an efficient asyncronous remote procedure call for use by components of a distributed program. They are also useful as a general way of allowing a caller to run in parallel with a call and to pick up the results of the call, including any exceptions it raises, in a convenient manner. Thus, promises preserve the merits of organizing programs using procedures and procedure calls without sacrificing the performance benefits of streams [LS].

Independent of the Mercury project, Dr. Shrira has worked on a new efficient fault taulerant data replication schema. The schema improves availability of the system by exploiting the semantic knowledge of the application to relax the up to date consistency constraint. An interesting class of applications was identified and the schema was given a rigorous specification and correctness proof. [LLS].

Dr. Shrira also worked on modular specifications of network protocols [FLS]. The work analyzed a network synchronyzining algorithm by B. Awerbuch designed to be used as subcomponent in derivation of other protocols. Modular specification and correctnes proof were given to the algorithm which enable them to be reused in specifications and proofs of the derived protocols .

References

[LS] Liskov, B., Shrira, L., "Promises: Linguistic Support for Efficient Assynchronous Procedure Calls in Distributed Systems", submitted to POPL 87.

[LLS] Ladin, R., Liskov., B., Shrira, L., "A Technique for Constructing Highly-Available Services ", accepted to journal of Algorithmica.

[FLS] Fekete, A., Lynch, N., Shrira, L., "A Modular Proof of Correctness for a Network Synchronizer ", In proc. Second International Workshop on Distributed Algorithms, Amsterdam, May 87.

Scheduling Parallel Programs On A Distributed System

John A. Stankovic
Don Towsley
Gary Rommel

Dept. of Computer and Information Science
University of Massachusetts

August 1987

Abstract

In the near future it will be common to see local area networks with uni-processors and multi-processors. There is also a growing trend to program applications by decomposing them into multiple parallel tasks of large granularity. If these multiple tasks are assigned to different processors, then it becomes a distributed program. However, in spite of the potential parallelism, distributing a program can easily result in a decrease in performance. This decrease is due to such factors as extra delay in communication between the various parallel tasks, operating system overheads such as context switches, and delays imposed on the parallel tasks by the scheduling algorithm. It is obvious that the total response time for a parallel application is only as good as its slowest component. All these issues complicate the effective use of local area networks for large grain parallelism.

In our project we have been studying the scheduling of large granularity parallel programs on distributed systems where some of the nodes are multi-processors. We have approached this complicated problem in three related ways: one analytical, one based on implementation, and one based on simulation.

The analytical work considers a job to be composed of multiple, parallel tasks generated by a fork-join construct. The parallel tasks do not communicate with

each other, except at the last phase of execution (the join). The analytical results derive closed form solutions for response time of the fork-join job. These results show that for uni-processors, scheduling fork-join jobs under processor sharing should be done at the job level and not at the task level. We also derive analytical solutions that show that the opposite is true for multiprocessors, i.e., scheduling fork-join jobs under processor sharing should be done at the task level and not at the job level. One implication of these results is that if a job with multiple tasks is moved from a multi-processor to a uni-processor, then the job should no longer be treated as a collection of parallel tasks. We were also able to derive analytical results for fork-join jobs on a multiprocessor which compare processor sharing with first come first serve (FCFS) scheduling. We find that FCFS exhibits better performance than processor sharing over a wide range of systems. We also studied the situation where there are two classes of jobs and where a specific number of processors is statically assigned to each of these classes. The results demonstrate that in a multi-processor a static assignment of processors by classes must be avoided. This latter result gives rise to the next aspect of out project.

Current multiprocessing scheduling algorithms are quite limited, and usually treat all tasks as independent. This could be a mistake in many circumstances. In the implementation part of our work, we have developed a dynamic, multi-class, multi-processor scheduling algorithm which we intend to implement on our SEQUENT machine under MACH. The implementation has been delayed until we obtain a version of MACH for the SEQUENT. The algorithm supports the simultaneous execution of short jobs, long jobs, jobs with many parallel and communicating tasks, and those jobs which require a dedicated set of processors. The algorithm separates policy from mechanism and is highly parameterized for ease of tuning in different environments. It does require lightweight processes. In addition, the algorithm makes use of the insight gained from the analytical models. This algorithm does not consider scheduling across the network. It is necessary to integrate such a local, multi-processing, scheduling algorithm into a distributed setting. Special problems arise when attempting to integrate local multiprocessing scheduling with distributed scheduling, especially when jobs are composed of parallel and communicating tasks. This gives rise to the simulation phase of our study.

The simulation study removes the restriction found in the analytical models that parallel tasks don't communicate with each other. In the simulation study we investigate various types of communicating parallel programs with both synchronous and asynchronous IPC. We have developed focused addressing and bidding algorithms that specifically address some of the major issues of such programs. A major characteristic of this algorithm is that the scheduling modules at each site *negotiate* either

to cluster highly communicating tasks, and/or to distribute tasks across the network when we predict that those tasks would benefit from executing on separate processors. Again, insights provided by the analytical results are used in formulating some of the scheduling policies of this algorithm. To date, in this part of the work, we have only considered communicating parallel tasks on a local area network of uniprocessors. Future work will attempt to integrate this scheduling algorithm with local multi-processing scheduling algorithms. The simulation program is implemented.

Marionette: Support for Highly Parallel Distributed Programs in Unix

Mark Sullivan
University of California, Berkeley,
Berkeley, CA 94720.

Extended Abstract

A parallel algorithm can be implemented as a set of processes executing concurrently on many loosely-coupled processors. *Marionette* is a facility that simplifies the construction of such parallel, distributed programs. It includes a library providing a high-level interface to the Unix facilities for remote process creation, interprocess communication, and asynchronous I/O. This interface resembles Sun Remote Procedure Call [1] both in syntax and in its use of the XDR protocol [2] for machine independent data representation, but is oriented more toward multiprocess parallel programs than client/server interactions.

Marionette supports a master/slave model of distributed computation. It requires a distributed program to be divided conceptually into a *foreground,* containing the program's main thread of control, and a *background,* in which functions may be executed concurrently with the main program and with each other. The foreground and background are essentially duplicate copies of the program's address space except that global variables in the background are read only. The main thread invokes functions in the background using a non-blocking library call. When one of these functions completes, the main thread accepts the result parameters into the foreground with a second library primitive. It may either poll or block until background function results become available. Using a configuration file, the library determines at run time the number of processors available to the program. Attempts to invoke more functions in the background than there are processors available return an error code.

The program may declare any global data structure to be *shared* between the foreground and background. Like all global data, shared data structures may only be modified by the foreground thread. If the foreground thread then notifies Marionette of the modifications with another library call, subsequent background invocations will operate on the updated version of the data structure.

In sum, the Marionette library provides:

- transparent initialization of remote processes.
- a means for the programmer to request that certain of his functions be executed in parallel.
- flow control in the event that the program requests more parallel operations than it has machines available to execute them.
- a high degree of fault-tolerance. If some processors fail or become over-loaded, performance degrades, but correct execution continues without user intervention. Similarly, if additional processors become available, the program may take advantage of them.
- a mechanism for maintaining replicated data structures at all sites executing the program.

In addition to the library, Marionette includes two utility programs to smooth over some of the mechanical concerns of distributed programming. A parallel compilation utility ensures that consistent versions of the program binary files are available to the processors that will execute the program. This utility must copy source code to file systems accessible to each processor and compile these sources in instruction set of each processor. A second utility helps make debugging less difficult by simulating execution of the multiprocess program in a single Unix process. This process can then be monitored with the standard Unix debugging tools.

The master/slave semantics enforced by Marionette limit communication between the parallel components of the program to data passed into and out of the background by the main thread.

This organization simplifies the programmer's synchronization task, though the foreground thread might become a performance bottleneck in communication-bound programs. The library primitives are flexible enough to allow a programmer to implement a parallel algorithm without knowing the number, type, and relative speeds of the processors that will eventually execute the program. Processor heterogeneity is handled by XDR and the parallel compilation utility. Through the shared variable mechanism, a program can cache large data structures at remote processors. The library assumes responsibility for keeping the data structures up to date with regard to the functions scheduled at the processor.

Marionette provides the most performance benefits to programs that can be decomposed into many small, independent operations. When the number of operations is much larger than the number of processors, faster or more lightly loaded processors will become available for scheduling more frequently, hence take on a larger proportion of the work. Real applications that may be structured in this manner include "ray-trace" rendering in graphics [5] and Monte Carlo simulation techniques used, for example, in Chemical Physics [4].

Currently, a prototype library, a parallel compilation utility, and some debugging tools run on a network of Vaxes and Sun workstations. Work on a distributed implementation of the UgRay ray-tracing renderer [3] using Marionette is nearly complete. Future efforts will explore the limits to parallelism imposed by our decision to synchronize communication through the foreground thread.

1. B. Lyon. "Sun Remote Procedure Call Specification", Technical Report, Sun Microsystems, Inc., 1984.
2. B. Lyon, "Sun External Data Representation Specification", Technical Report, Sun Microsystems, Inc., 1984.
3. D. Marsh, "UgRay: An Efficient Ray-Tracing Renderer for UniGrafix", Technical Report UCB/Computer Science Dpt 87/360, University of California, Berkeley, May 1987.
4. A. Wallqvist, B. Berne and C. Pangali, "Exploiting Physical Parallelism Using Supercomputers: Two Examples from Chemical Physics", *Computer 20, 5* (May 1987).
5. T. Whitted, "An Improved Illumination Model for Shaded Display", *Communications of the ACM 23, 6* (June 1980), 343-349.

The PHARROS Project

by

John Van Zandt
RCA Advanced Technology Laboratories

The PHARROS Project (Parallel Heterogeneous Architecture for Reliable Realtime Operating Systems) is currently underway at RCA's Advanced Technology Laboratories. The goals of the project are to develop an operating system and associated distributed architecture to support applications which are distributed across next-generation networks of heterogeneous parallel processors. The project is also concerned with methodology and tools to assist the applications development within the context of this system. The focus of this project is on applications which straddle the boundary between signal and data processing.

To this end, we have constructed a demonstration system consisting of a Connection Machine, a BBN Butterfly, a VAX cluster, and a WARP, along with a set of workstations, all networked together using an Ethernet. Next year we will be replacing the Ethernet network with direct connections between the processors and the Butterfly using multiple VME buses, modeling a tightly-coupled network as will be seen in next generation distributed systems with the Butterfly switch and shared memory as the interconnection system. The Butterfly processors will be used as processing resources for both the distributed operating system and for the application.

This year, a large signal processing and tracking application is being implemented on top of this system. The application is being decomposed into many interdependent tasks which will take advantage of the heterogeneous parallel processors in the network. The modeling of the performance and estimations of the communication requirements along with other measures will guide the granularity to be supported by the architecture and operating system. As part of this task we are developing a set of tools to assist the programmer in distributing the application. Also, performance monitoring tools will visually guide the programmer to better understand the complex interactions of the application as it executes in the parallel environment.

Research in Distributed Systems

William E. Weihl

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
(617) 253-6030
weihl@xx.lcs.mit.edu (Arpanet)

September 8, 1987

My recent research in distributed systems has been focused in two main areas: distributed transaction management, and heterogeneous distributed systems. These two areas are discussed in more detail below

1 Distributed Transaction Management

In earlier work, I developed an approach to the design of loosely coupled transaction systems that supports the modular design of highly concurrent applications. The approach, which builds on earlier work on data abstraction, involves organizing programs around *atomic data types*. The design decisions involved in designing a system can be divided into *global* and *local* decisions. Global decisions constrain the entire system, while local decisions affect individual types. A global decision that must be made involves the choice of a *local atomicity property*, which characterizes the behavior required of the different atomic types in a system to ensure that they cooperate to ensure global atomicity. Given this choice, new types and transactions can be added to the system without modifying existing types or transactions, and atomicity is still guaranteed. In other words, systems are extensible.

Extensibility is an important attribute of a system. Performance, however, is equally important. One of the potential problems with transaction systems is that the level of concurrency can be relatively low; in some applications this can be a serious problem. Atomic types can be used to alleviate this problem by using the semantics of a type in designing the concurrency control and recovery algorithms for the type. The specification of a type can be analyzed to determine the concurrency permitted for transactions using objects of the type; this analysis can be used as feedback during the design process to modify the specification of a type if the permitted level of concurrency is not adequate to meet the performance demands of the application. Furthermore, the implementation of a type can be modified safely to permit any level of concurrency up to the limits imposed by the type's specifications; thus, a type can be implemented initially in a simple way that permits relatively little concurrency, and then re-implemented to permit more concurrency if it turns out to be a concurrency bottleneck.

My current work has several goals:

- To design powerful, efficient, and easy to use mechanisms for implementing atomic data types.
- To develop more general concurrency control and recovery algorithms.
- To understand the interactions between, e.g., concurrency control and recovery.

These goals are mutually supportive; for example, the attempt to design new mechanisms and algorithms creates a need for a deeper understanding of the algorithms and their interactions. Some of the algorithms I have developed illustrate interesting interactions between concurrency control and recovery; I would like to understand these interactions better, with the ultimate goal of generalizing the algorithms and developing better mechanisms for implementing them.

I have also been involved in a major effort (in part with Nancy Lynch and Michael Merritt) to develop formal models for describing and analyzing distributed transaction systems. We have already described and analyzed a variety of concurrency control, replication, and orphan elimination algorithms. While the model we have used to date allows us to analyze algorithms that cope with aborts of transactions, it does not include a notion of a site crashing. We are currently working on modelling crashes and analyzing algorithms that cope with crashes. In this work, as in the work described above, I am particularly interested in modularity issues: what is the approriate decomposition of the system into pieces, and what are reasonable correctness criteria for each of the pieces?

2 Heterogeneous Distributed Systems

A number of us at LCS have also been working on a project (formerly called the LCS Common System, now called Mercury) aimed at solving some of the problems of heterogeneous distributed systems. We have been particularly interested in heterogeneity at the level of the programming languages. Our work to date has focused on two issues: the semantics of data types, and the communication model.

Data types present an obvious problem in a heterogeneous system: different languages have different notions of data types, with different underlying representations, yet some method must still be found for them to communicate. A basic premise of our approach at this point is that communication interfaces between heterogeneous components must be described in language-independent terms. We have designed a language-independent type system that is expressive and that permits a flexible connection with each individual language. Earlier work typically placed serious restrictions on the set of types and the use of type constructors, and provided relatively inflexible translations between local types and the types used for communication. We are currently working on extending this work to permit, for example, polymorphic interfaces.

In trying to develop a semantic model for the data types used in communication, we came to the conclusion that these types are fundamentally different from the types used for local computation. Types used for local computation are frequently viewed as consisting of a set of values and a set of operations. (In a language like Ada, a module might define several types and some operations together, so the operations might not be associated with a single type.) Types used for communication, however, are best viewed simply as sets of values. Defining the semantics of communication types by associating operations with them can lead to serious problems as systems evolve. This has implications for single-language systems such as Argus (which currently does not distinguish between types used for communication and types used for local computation), since the issue of evolution arises regardless of the number of languages involved. We are currently redesigning the data communication mechanism in Argus to provide better support for evolution by making a clear distinction between the two kinds of types.

Our initial discussions about communication models led to the conclusion that existing high-level models, such as remote procedure call (RPC), are not adequate for a wide enough range of applications (for example, driving a remote display, or transferring large amounts of data). As a result, we have designed a new communication model that integrates RPC and byte-stream protocols into a single semantic framework. The model allows a client to decide whether a call should be performed immediately, in which case the system attempts to minimize the delay for the call, or whether it should be *streamed*, in which case the system is free to buffer the call in an attempt to maximize throughput.

The semantics of the communication mechanism guarantee that calls sent on the same stream appear to be executed in the order in which they are sent. Thus, a client can stream one call and then stream additional calls without waiting for the results of the first call, but still be sure that the calls appear to execute in the order in which they were made. Of course, this makes sense only if the arguments of the later calls do not depend on the results of the first call.

The choice to stream a call is made entirely by a client: servers can be written more or less as they

would be in the absence of streaming. In addition, a server needs to provide only a single interface, rather than one interface for clients who want to use RPC and another for clients who want to use byte-stream protocols.

Our mechanism permits clients to *pipeline* remote calls, taking advantage of the concurrency between the sender and receiver of a message, and of the buffering capabilities of the network and the communication protocols. For some applications, pipelining can result in dramatic improvements in performance. An interesting open question, however, involves the applicability of pipelining: for what kinds of service interfaces can streaming be used profitably? For example, if typical uses of a service require a client to receive the results of one call in order to compute the arguments for the next call, pipelining could not be used to advantage. In the few cases we have examined, we have been able to modify the service interfaces so that clients can pipeline calls. My hope is to develop a small set of general transformations of this sort, with the result that pipelining can be used for a wide range of applications.

Submitted to the Second Workshop on Large Grained Parallelism,
Hidden Valley, PA, October 11-14, 1987.

Programming Language Features for Resilience and Availability

C. Thomas Wilkes and Richard J. LeBlanc

Distributed Systems Group, School of Information and Computer Science
Georgia Institute of Technology, Atlanta GA 30332-0280
Internet address: {wilkes, rich} @ stratus.gatech.edu Voice: (404) 894-6170

Extended Abstract

Since late 1981, the Clouds project at Georgia Tech [Allc83, Dasg87] has been occupied with the design and construction of a reliable *multicomputer*, that is, a unified environment over loosely-coupled distributed resources in which reliable applications may be constructed. The research goals of this project include decentralized cooperative control, location independence for data as well as processing, and *failure tolerance* of computations. Failure tolerance implies the *resilience* of data despite node crashes, the *availability* of resources despite partial failures of the system, as well as continued *forward progress* of jobs in the system. The Clouds architecture offers several features in support of these goals, including support for passive objects, capability-based object access, location-transparent object invocation, nested and toplevel actions (transactions), and customizable as well as automatic synchronization and recovery mechanisms.

In support of programming the levels of the Clouds system above the kernel level, we have designed and implemented a systems programming language called *Aeolus* [LeBl85, Wilk85, Wilk86]. The purposes of the Aeolus language include: providing abstractions of the Clouds features of objects, actions, and processes; providing access to the recoverability and synchronization features of Clouds; and serving as a testbed for the study of programming methodologies in action/object systems. The combination of Aeolus and the Clouds kernel provides support for resilient objects.

Aeolus support for objects includes a hierarchy of *object classifications* which share a common implementation and invocation syntax. The support in Aeolus for elements of this hierarchy ranges from completely automatic synchronization and recovery (the paradigm presented by most other systems offering support for resilience), through programmable synchronization and recovery based on object semantics, to "lightweight" objects—living in the address space of their creators—in which recovery support has been "optimized out."

A similar hierarchy of support for actions and action/object interactions is included in Aeolus. The constructs for programmer specification of resilience properties support the separation in Clouds of *failure atomicity*—the "all-or-nothing" behavior of atomic actions—and *view atomicity*, in which actions are prevented from observing the uncommitted results of other actions. Failure and view atomicity together form the traditional notion of *serializability*; we believe their separation in Clouds provides a powerful means of increasing the efficiency of actions as a reliability technique, especially in development of resilient structures for use in operating systems [McKe85]. This characteristic is exploited in the linguistic features of Aeolus.

Recently, we have been using Aeolus to examine availability issues in Clouds [Wilk87]. We have developed a scheme for deriving replicated objects from single-site specifications which we call *Distributed Locking*. This scheme addresses the issues of control of concurrency and state consistency among the replicas in a system in which objects may have arbitrary structure; in Clouds, objects may be *logically nested* in an arbitrary manner, in the sense that an object may hold capabilities to other objects. Distributed Locking consists of a *methodology* for deriving a replicated implementation from the single-site version, as well as a *mechanism* to support this methodology. In accord with the Clouds philosophy in other areas, it does not assume any particular *policy* for replication control (*e.g.*, quorum consensus).

The methodology of Distributed Locking consists of two steps: the programmer writes a single-site implementation of an object with appropriate Aeolus/Clouds lock mode compatibility specifications for synchronization; then, an *availability specification* (**availspec**) is provided separately for the object, which supplies information about the object's replication properties. (The **availspec** is described in more detail below.)

The mechanism provided by Distributed Locking also consists of two parts:

1. when an action obtains a lock on an object, the system also obtains locks on some subset of its replicas, according to a user-specified policy;
2. when an action commits, the object state is copied to the subset of replicas locked in step (1), according to another user-specified policy.

The policies for locking and state copying among replicas used in the DL mechanism may be specified by the programmer in an **availspec** as handlers for the *lock* and *copy* events, respectively. These may consist of one of several default policies (*e.g.*, the quorum consensus or available copies algorithms), or the programmer may specify custom handlers using the same system-supplied primitives which we have developed for programming the default handlers. When a quorum consensus-style algorithm is used for a lock event, the programmer may also specify the relative availabilities of the modes of each lock type declared by the object.

Other Clouds researchers have been concerned recently with the issue of forward progress in Clouds. A scheme called *Parallel Execution Threads* (PET) has been developed which essentially provides replication of actions as well as objects [Aham87, Aham87a]. PET may be regarded as a generalization of the so-called "hot spares" scheme. Our current research includes specifying how PET may be controlled by the Clouds system; this functionality is to be embedded in a subsystem which we call the *Fault Tolerant Job Scheduler*.

REFERENCES

[Aham87] Ahamad, M., P. Dasgupta, R. J. LeBlanc, and C. T. Wilkes. "Fault-Tolerant Computing in Object Based Distributed Operating Systems." *PROCEEDINGS OF THE SIXTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS* (IEEE Computer Society), Williamsburg, VA (March 1987): 115-125.

[Aham87a] Ahamad, M., and P. Dasgupta. "Parallel Execution Threads: An Approach to Fault-Tolerant Actions." TECHNICAL REPORT GIT-ICS-87/16, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, March 1987.

[Allc83] Allchin, J. E. "An Architecture for Reliable Decentralized Systems." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983. (Also released as technical report GIT-ICS-83/23.)

[Dasg87] Dasgupta, P., R. LeBlanc, and W. Appelbe. "The Clouds Distributed Operating System: Functional Description, Implementation Details, and Related Work." TECHNICAL REPORT GIT-ICS-87-28, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, July 1987.

[LeBl85] LeBlanc, R. J., and C. T. Wilkes. "Systems Programming with Objects and Actions." *PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS*, Denver (July 1985). (Also released, in expanded form, as technical report GIT-ICS-85/03.)

[McKe85] McKendry, M. S. "Ordering Actions for Visibility." *TRANSACTIONS ON*

SOFTWARE ENGINEERING (IEEE) 11, no. 6 (June 1985). (Also released as technical report GIT-ICS-84/05.)

[Wilk85] Wilkes, C. T. "Preliminary Aeolus Reference Manual." TECHNICAL REPORT GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986.)

[Wilk86] Wilkes, C. T., and R. J. LeBlanc. "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System." *PROCEEDINGS OF THE 1986 INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES* (IEEE Computer Society), Miami, FL (October 1986): 107-122. (Also available as Technical Report GIT-ICS-86/12.)

[Wilk87] Wilkes, C. T. "Programming Methodologies for Resilience and Availability." PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987. (In progress.)

A COARSE-GRAINED DISTRIBUTED MULTIPROCESSING SYSTEM

Joan M. Wrabetz
SRI International

Problem Statement

Users of distributed systems are developing an increasing number of applications that are computationally intensive, but they are unable to obtain reasonable performance for those applications on distributed systems. One way to improve performance is to increase the amount of parallelism used in the system. However, the existing programming languages and operating systems do not provide adequate support for the development and execution of applications that require shared use of the available processing resources. Current models of interaction in distributed systems are not appropriate to parallel computations, and must be replaced with models which provide high-speed communication primitives and support for parallel operations.

At the same time, special-purpose stand-alone processing hardware is being incorporated into heterogeneous networks of multiprocessors, workstations, and parallel-processors. The developers in these environments do not have the tools or the underlying operating systems to support decomposition of computationally intensive applications into tasks that can functionally use the best capabilities of the heterogeneous processing elements. In these distributed network environments, programming languages and methodologies are required that allow developers to design applications that make the best use of the processing capabilities, both in terms of available parallelism and processing hardware capabilities.

Objective

At SRI, we are developing a model for computing on these distributed networks of heterogeneous and autonomous processing elements. Our model is designed to address two goals. The first is to provide a programming environment and methodology that facilitates the development of parallel code to execute computationally intensive applications. This involves developing a model for parallel computation that is appropriate for the underlying distributed architecture.

The second goal is to provide an underlying execution system that interprets the computing model and applies it efficiently to the distributed architecture. Current distributed systems support interprocess communication but not efficient distributed computation. While these systems may provide the capability of distributed or parallel computation, no support is provided for automatic allocation and execution of processes. The execution system must make distributed computation both feasible and efficient by providing these capabilities. One should expect the performance of a distributed parallel architecture to achieve multiprocessor performance.

Approach

In order to provide a programming environment that does not require users to program in entirely new ways, yet allows the user to take advantage of parallelism, we have combined visual programming methods with the writing of sequential code to produce coarse-grained tasks. Because communications and task management overhead can be high on distributed system, it is not efficient to handle fine-grained tasks and thus coarse-grained parallelism provides a reasonable tradeoff between achievable parallelism and task execution and communication overhead.

In order to support the efficient execution of coarse-grained parallel tasks on heterogeneous and autonomous processors, our computing model addresses both the method of execution of tasks and the method of allocation of tasks. The method of execution of tasks on the graph can take one of two forms. Execution can be data-driven, where tasks are executed on a first-come-first-served basis as input data becomes available. Alternatively, execution can be demand-driven, where tasks are only executed if their output data is required for subsequent tasks. The choice of execution mechanism will affect the level of fault-tolerance, the capability for real-time execution, and the role of external input and output handling. Further, the relative execution costs incurred by each of these execution mechanisms within the runtime system of a distributed system of processors must be addressed.

The allocation of tasks to processors in the system is dynamic in order to efficiently utilize autonomous and heterogeneous processing capability. Incorporating status collection to provide the information necessary for load balancing in allocation and subsequent execution of tasks is a requirement for distributed system architectures. While the effectiveness of load-balancing may be significantly reduced by the communications costs incurred for status propagation, the effective capacity on each host may differ greatly because nodes are autonomous and may be executing separate local functions. The load at each host must therefore be considered in task allocation. Further, because processing resources are heterogeneous, the absolute processing capability at each site must also be taken into account.

The runtime system for graph execution must provide the capability to support 1) information exchange between tasks on arbitrary processors, 2) task activation as determined by the computing model, and 3) task allocation and execution onto arbitrary processors. Our initial focus is to develop a runtime system that uses the applicative properties of the graph of an application to dynamically allocate and execute that graph on a distributed network of autonomous nodes. Applicative execution of the tasks on the graph will allow the runtime system to efficiently utilize the available processing resources. Execution proceeds by traversing the graph with tasks being activated and synchronized by input data. Communication between tasks is limited to exchange of input and output data, and can be easily accommodated by message passing. This method of communication is deemed more natural for distributed systems and their underlying interprocess communications primitives than other approaches commonly used in shared-memory multiprocessors. Further, elimination of data access by reference removes the need for shared address spaces between remote processes.

By virtue of our design, the runtime system also takes advantage of the freedom from side-effects possible with applicative programming, and the locality of data implicit in the graph in dynamically allocating and executing tasks. In a system with moderate to high communication costs, exploitation of these properties is imperative. Finally, the design of the execution system includes the capability for status collection to support task allocation, and for voluntary allocation of processing capability by autonomous nodes. We intend to take advantage of existing work for both of these functions. [1][2]

---

[1] Wrabetz, J., Schreier, L., and Davis, M., "An Overview of an Experimental Distributed System", submitted for publication to IEEE Transactions on Software Engineering, 1987.

[2] Ezzat, A., Agrawal, R., "Making Oneself Known in a Distributed World", Proceedings of the 1985 International Conference on Parallel Processing, Aug., 1985, pgs. 139-142

tation can do without dynamic tests except in situations where they are explicitly required.

Polymorphism allows the essentials of algorithms to be formulated independently of a particular application. The so obtained algorithms are re-usable to handle objects of different properties without loss of the benefits of static typing.

Cardelli [85] and Barendregt [86] both present languages based on the state of the art of static typing and polymorphism. Though such languages do with comparatively few and simple concepts, they can be shown to cover the data abstractions of Ada as well as the classes of object oriented languages [Cardelli 85]. We can show that with minor additions such languages may also allow the description of distributed systems and communication in a structured way, with the security of static typing.

The role of language in systems development and verification

In the evolution of programming and specification languages we can distinguish three stages with respect to language support in systems development. Languages in the first stage give no support at all. Languages in the second stage—which are accompanied by powerful run-time systems—allow or enforce automatic inclusion of "run time checks", synchronization, garbage collection, stack administration, etc. to help to detect design errors or avoid error-prone programming. Languages in the third stage, however, *syntactically enforce consistency* to make run time checks unnecessary. One way to achieve this is static typing. The necessary syntactic restrictions and the enforced redundancy will be taken as a benefit rather than an obstacle if the language not merely rules out inconsistent systems but also supports the development of consistent ones or allows to write re-usable algorithms. Polymorphism here is a valuable tool.

With respect to *algorithmic correctness*, we are used to *third stage* languages. If contemporary languages support *timing* at all, the corresponding constructs are hardly beyond second stage (cf. Wirth [1977]). Parnas observed in [1985] that the support of reliability has not yet developed beyond the first stage.

Our approach

Static typing and polymorphism as described by [Cardelli 85] or [Barendregt 86] are based on a mathematical theory of types general enough to cover not only data types (i. e. sets of values), but also timing and reliability attributes.

A first generalization step lead to our present language [Wupper 87]. It allows us to formally state timing requirements and take them into the development process to arrive at systems guaranteeing to fulfil these requirements without dynamic time checks or synchronization. The algorithms so derived are polymorphic and can be reused in contexts with different functionality and different, but structurally similar, timing requirements.

References

Barendregt 86
H. P. Barendregt, M. van Leeuwen:
Functional Programming and the Language TALE
in: Current Trends in Concurrency (J. W. de Bakker et al., eds.), LNCS 224, Berlin 1985

Birolini 85
A. Birolini: *Qualität und Zuverlässigkeit technischer Systeme*, Berlin New York Tokyo 1985

Cardelli 85
L. Cardelli, P. Wegner: *On Understanding Types, Data Abstraction, and Ploymorphism*
ACM Comp. Surv. 17 (1985), Nr. 4, pp. 471-522

Koymans 83
R. Koymans, J. Vytopil, W. P. de Roever: *Real-Time Programming and Asynchronous Message Passing*, ACM, 2nd Symp. Princ. Distr. Comp., Montreal, Aug. 1983

Leveson 86:
Nancy G. Leveson: *Software safety: Why, What, and How*, Comp. Surv., 18, No. 2, June 1986, pp.125-163

Parnas 85
D. C. Parnas: *Software Aspects of Strategic Defense Systems*, C.ACM 28 (1985), No. 12

Wirth 77
N. Wirth: *Toward a Discipline of Real-Time Programming*, C.ACM 20 (1977), No. 8

Wupper 87
H. Wupper, J. Vytopil: *An Approach towards Formal Treatment of Reliability in Systems Specifications*, Fachtagung "Requirements Engineering", Gesellschaft f. Inf.., St. Augustin, 20.-22. 5. 1987; KUN, Informatica, Report no. 93

Static Typing of Temporal and Reliability Attributes in Distributed Systems

Hanno Wupper, Jan Vytopil
Catholic University Nijmegen[1]

Distributed computer systems which control physical processes must be prompt (in "real time") and reliable. For their methodical development one needs a specification language that supports treatment of formal promptness (timing) and reliability requirements and that can (i) serve as a basis for development and verification, (ii) provide a formalism for theoretical investigation of properties of distributed reliable real-time systems, and (iii) allow the exchange of re-usable algorithms. — A first prototype of such a language shall be presented. It is based on a generalization of the concepts of static data-typing and polymorphism. It allows to associate, in systems specifications, attributes describing temporal properties with the components and sub-components of the systems to be specified in a similar way as data types are associated with expressions and sub-expressions in strongly typed languages. Such a specification will be syntactically correct only if it is consistent with respect to temporal properties and guarantees that the specified system will fulfil the stated requirements.

This prototype of a specification language can be used for the development and verification of such distributed real-time systems that have to react at fixed moments or within a fixed period. It has mainly to be developed to show that a consistent language based on the principles of static timing and reliability typing can indeed be defined. Future versions will also contain constructs for systems with variable temporal behaviour and will, moreover, treat reliability attributes besides the temporal ones.

Clarification of terms

Timing requirements for real-time systems must not only include qualitative statements about the necessary temporal order of activities, but moreover quantitative statements with respect to physical time (e.g. the duration of an activity in seconds) [Koymans 83].

Reliability in computer science often is defined as the probability that a certain component functions correctly over a certain period of time. This definition gives rise to four questions: (1) Does it cover 'Reliability' in the sense of natural language (i. e. is it a sufficient basis to allow to decide whether we can "rely" on a system)? (2) If we accept the definition: How do we obtain the reliabilities of the building blocks of systems (in other words: what means "to function")? (3) How does a system's reliability depend on its structure and the reliabilities of its components. (4) How can we formulate reliability requirements for the overall system?

Question (1) is extra-mathematical and shall not be addressed. In any case this probabilistic approach is widely accepted to approximate Reliability closely enough to justify further research. For hardware components, engineering disciplines have contributed a lot to (2) [Birolini 85]. (3) is purely intra-mathematical and has been studied well: If components of known reliabilities and known average repair times are assembled in a given way, the overall reliability can be computed by statistical means. This has lead to approved methods to include redundancy in systems in order to increase reliability. The requirement that redundant components be really independent is usually not checked formally, however. Analysis of accidents often reveals that their cause was not an unforeseen failure of a basic component but an illegal interference between components assumed to be independent [Leveson 86]. Though reliability is a probability, (4) cannot simply be dealt with by stating one number for a whole system. It is more realistic—and common practice—to separately require reliabilities differing in order of magnitude for different sub-functions of a complex system. Reliability requirements are in itself something complex, closely linked to the system structure. A language that allows to formally establish that link, is still missing, however.

Static typing associates an attribute ("type") with certain or all sub-expressions of a text; conventionally this is a "data type" saying something about the set of values the expression may possibly assume. If such a text has been proved to be syntactically correct, this ensures that during execution all function applications will be well-defined and that the corresponding implemen-

[1]Informatica V, Toernooiveld
6525 ED Nijmegen, The Netherlands