PRINCETON SYSTEMSFEST PROCEEDINGS

CS-TR-224-89

February 1989

Princeton SystemsFest

Proceedings

February 24, 1989
Scanticon-Princeton

These are the proceedings of the Princeton University Department of Computer Science **SystemsFest**, a one day workshop on computer systems. Included are abstracts of the talks, as well as short writeups of the discussion following each talk. It is our goal that the workshop develops into an annual event during which researchers can present and discuss new ideas. Given the growing number of systems researchers within our department, and the many ways their work interweave, an event like **SystemsFest** should increase the communication and promote the collaboration among them.

There are a few things that we would like to see happen as a result of this workshop:

- Speakers leaving with a strengthened interest in their new ideas, through feedback during and after their talk as well as contacts with others who wish to see the work proceed.
- Students who do not yet have research topics finding ideas and advisors among the presentations.
- Everyone gaining a better understanding of the systems research being undertaken at Princeton, especially our guests from the Industrial Affiliate program.

A final, perhaps more important goal is that everyone leave with a feeling of a day well spent.

We wish to stress that this event is **not** a formal conference. The goal is to have open discussions about new ideas. If those in the audience have questions, suggestions, or concerns they should be voiced. Due to time constraints, question periods will have a limited time. However, if the moderator of a session suggest that the discussion be moved into another room, view this as encouragement and not admonishment.

These proceedings are provided so that attendees familiarize with the background of the talks *before* coming to the workshop, so that speakers will be able to spend most of their time outlining *new* ideas.

The organizing committee would like to thank all those involved in making **SystemsFest** a reality. First, to all the researchers who submitted ideas and to the ones who will be presenting them. Second, to all the faculty members that are supporting the event. Third, to Prof. Rafael Alonso, Prof. Hector Garcia-Molina and Prof. David Hanson for their advise and encouragement. Finally, to Prof. Robert Sedgewick, Mrs. Charlotte Ansted-Jameson and the secretarial staff, for their help with the administrative aspects of organizing an event of this type.

The SystemsFest Organizing Committee

Matt Blaze
Chris Clifton
Luis Cova
Carl Staelin

List of Attendees

From the Princeton University Department of Computer Science

Robert Abbott	Sarantos Kapidakis
Rafael Alonso	Makoto Kobayashi
Andrew Appel	Eleftherios Koutsofios
Daniel Barbara	Kriton Kyrimis
Brad Barber	Andrea LaPaugh
Youakim Bitar	Richard Lipton
Matt Blaze	Sally McKee
Matthias Blumrich	Jeff Naughton
Alvaro Campos	Karin Petersen
Mara Chibnik	Jim Plank
Chris Clifton	Christos Polyzois
Luis Cova	Norman Ramsey
Dimitris Doukas	Jonathan Sandberg
Hector Garcia-Molina	Dimitrios Serpanos
Michael Golan	Patricia Simpson
Mordecai Golin	Annemarie Spauster
Mark Greenstreet	Carl Staelin
Paul Haahr	Andrew Tolmach
David Hanson	Jenny Zhao

From Elsewhere at Princeton University

Mark Stewart	Applied Math

Industrial Affiliates

Anton Dahbura	Bell Labs
Stu Feldman	Bell Communications Research
Arding Hsu	Siemens Research
Christos Nikolau	IBM
Maylee Noah	IDA
Stephen North	Bell Labs
Andrew Reibman	Bell Labs
Norman D. Winarsky	SRI Sarnoff Laboratory

Princeton University Computer Science Department
SystemsFest
"Our Current Research"

Friday, February 24th, 1989
Scanticon-Princeton
FINAL PROGRAM

8:00-8:45 am	**Breakfast**
8:45-9:00 am	**Opening Remarks** by Chris Clifton
9:00-10:40 am	**Distributed and Operating Systems**
Moderator:	Dave Hanson
Luis Cova	Homogeneity in Very Large Distributed Systems
Annemarie Spauster	Reliably Delivering Ordered Multicasts
Carl Staelin	Dynamic Global File Allocation
Paul Haahr	New Thoughts on Assemblers and Linkers
10:40-11:00 am	**Morning Break**
11:00 am - 12:40 pm	**Software Tools**
Moderator:	Rafael Alonso
Sally McKee	Debugging on the Gnot
Dave Hanson	Tools for Literate Programming
Eleftherios Koutsofios	A Graphics Editor for Technical Pictures
Chris Clifton	SpeedBrowsing
12:40-2:00 pm	**Lunch**
2:00-3:40 pm	**Database Management Systems**
Moderator:	Luis Cova
Hector Garcia-Molina	A Probabilistic Relational Data Model
Daniel Barbara	Implementing a Knowledge Base System
Rob Abbott	Scheduling Real-Time Transactions with Disk Resident Data
Pat Simpson	Characterizing Database Structure
Rafael Alonso	Connecting Heterogeneous Databases
3:40-4:00 pm	**Afternoon Break**
4:00-5:40 pm	**Parallel Systems**
Moderator:	Matt Blaze
Dimitrios Serpanos	Design of a PRAM System
Jeff Naughton	Shared Single Level Store
Andrew Appel	Parallel Functional Languages
Richard Lipton	Programming Arrays of Identical Processors
6:00-8:00 pm	**Dinner**
8:00-8:15 pm	**Closing Remarks** by Carl Staelin
8:15-9:00 pm	**Evening Session**
Invited Speaker:	Stu Feldman, Bell Communication Research *"Real Life"*

All sessions will take place in *Conference Room K.* Room M4 will be available all day for spontaneous discussions. Breaks will be in the coffee break area between Rooms K and M4. All meals (including breakfast) will be in *The Copenhagen Restaurant.*

Distributed and Operating Systems Session

The first session started off with Luis Cova discussing protocol development in very large heterogeneous distributed systems. The point was made that in such systems, standards may not exist or be hard to enforce. He introduced ideas on developing protocols "on the fly", as opposed to using pre-existing standards. This led to a number of questions, in particular involving what standards must exist in order for this to work at all. Norman Ramsey asked what sort of experiments could be tried in the lab in order to test these ideas. Richard Lipton pointed out that in such an arrangement even alphabets might differ. Brad Barber suggested that as a bottom level, some sort of standard identification message might be needed. Mark Greenstreet suggested the problems involved with unknown machines -- should we trust strangers? Christos Polyzois commented that such a system could lead to Anarchy, to which Luis responded that what was being provided was Freedom.

Annemarie Spauster talked on the subject of *Ordered Multicast*. In her model, trees of *Multicast groups* are used to enforce ordering. As all messages to the group must go through the root, the root can impose an order on the multicasts. Richard Lipton asked how sparse are typical multicasts -- are they close to broadcast, or often a single message? Annemarie wasn't sure, but would like to hear from anyone who has applications which would use multicast. Rafael Alonso suggested that in networks such as Ethernet, everything is a broadcast anyway. Annemarie noted that much of the cost was involved in processing the message, and this could be cut by a multicast. Hector Garcia-Molina noted that much of the network traffic in an ethernet is between a workstation and file servers, for which multicast would be useful. Christos Nikolau suggested that installation of different versions of a long-running distributed application would be a good application for multicast. Another question was how much is multicast affected by topology. Annemarie's answer to this was "a lot". However, this seemed to be an advantage of multicast -- it could take advantage of diverse topologies. David Hanson asked about multiple recipients on a processor. This turns out to be a related, but independent problem. Brad Barber asked if unordered multicasts would be useful, and would the distribution graphs for ordering be of use. Mark Greenstreet suggested that the cost for ordered multicast really isn't that great anyway. Rafael Alonso suggested looking more closely at topology. Paul Haahr asked how the analysis had been done. Annemarie said that she had chosen small, random groups. Karin Peterson asked if this would unfairly load the nodes chosen as the *root* of the distribution graph. **Load shifting** trees would be a possible answer to this. Daniel Barbara asked if this work had application to existing distributed database protocols.

Carl Staelin's talk looked at shifting "hot" files on disk so as to spread the access among available devices. Richard Lipton asked if files were moved in whole or in part. Carl said that this was a decision that would have to be made, depending on the characteristics of the operating system in question. Another question was if replicating, rather than moving, files would be appropriate. Carl answered with an emphatic **yes**, noting that consistency would have to watched -- the replication may have to be ended at the first write, if the write frequency is high. Christos Nikolau asked about the frequency of measuring file use. Carl noted that studies have shown that caching (very frequent measuring of use, with no history) perform better than static file allocation (infrequent measuring of use), which suggest that frequent measurement is desirable. Rafael Alonso noted that at least in Unix, files are typically read in full, and static allocation is a well-

studied problem. He also pointed out that many writes are to temporary files. Kriton Kyrimis also asked "how does it play on Unix". Paul Haahr asked how this applied to disk striping. Carl said that he was looking into it. Norman Ramsey asked how this differed from caching. Carl pointed out that this was sophisticated caching. Andrew Tolmach pointed out that this related to the question of caching entire files versus blocks. Brad Barber pointed out that this approach would help with writes as well as reads, which caching wouldn't.

Paul Haahr talked about redefining the tasks of assemblers and linkers, particularly now that few people write assembly code, and what is really needed is a compiler back-end. In particular, much of the functionality that is now assigned to assemblers could be better done by linkers. Rafael Alonso asked how this would affect inter-language linking. Paul suggested that for dynamic languages it may be too difficult a problem but for static languages this could be interesting work. Andrew Tolmach noted that incremental linking is another area that could use work. Richard Lipton asked if assembler source needed to be human readable, and why in fact do assemblers need to work from source files, preventing "on the fly" assembly of program-generated code. Paul suggested that this is due to the two-pass nature of current assemblers. However, files may still be necessary due to the fact that assemblers are typically multi-pass. Stu Feldman noted that linkers are also multipass. Paul suggested that some of the passes are due to overlays and other techniques which are no longer necessary with large virtual memories. Mark Greenstreet asked how much time is spent "asciising" the assembly source. Stu suggested that little time (10% of total assembly) is spent "parsing" the assembly source, but gains could be made by storing assembly source in a more machine-friendly form.

Homogeneity in Very Large Distributed Systems

Luis Cova

Very Large Distributed Systems (VLDS) are distributed systems consisting of tens of thousands of computers or even more. The computers may be part of smaller distributed systems and belong to different administration domains. One important issue to consider in this type of system is the inherent **heterogeneity** in hardware, software, and users.

It has long been recognized that **network transparency** is the fundamental concept of any type of distributed system. While it is easily achieved for networks under a single administrative control, through the enforcement of standards and homogeneous components (e.g., Cambridge Distributed System [Needham1982], V-kernel [Cheriton1988], LOCUS [Popek1981]) the concept is less feasible for systems with multiple administrations, each controlling a subset of the computing sites. For the latter type of environments, network transparency has been achieved at the transportation and session levels by protocol standardization (e.g., TCP/IP), and usually, some degree of transparency has been achieved at the presentation level with the use of remote sessions (e.g. telnet), remote executions (e.g. rsh), and remote procedure calls (RPC).

Since in a VLDS there are potentially tens of thousands of machines, it is difficult to have complete homogeneity across all the components, neither it is desire in many cases. What it is require is to establish the role of homogeneity in this type of environment, i.e., set the **lowest common denominator** that allows the components to interact among themselves. There are others researchers also looking for this common denominator.

Several Researchers have proposed sets of protocols to be used by different vendors and implementors to develop their distributed applications. Therefore allowing interoperability across machines [Zimmermann1980], [SUN1988], . Others researchers have proposed homogeneous kernels to support distributed operations [Cheriton1988], [Turnbull1987], [Schmidtke1982]. A third camp represented by the HCS project [Notkin1988] works on a basic set of network services (mail, filing, printing, naming, authentication and remote computation) that adapt to the demands of a heterogeneous environment, mainly through dynamic binding.

My idea is to generalize the above approaches by using a procedure that allows entities of a VLDS to negotiate the type of interaction they will hold (e.g., decide the transmission protocol to be used.) I call this procedure a **bootstrap** negotiation process. This process must be based on a simple language to express basic operations and ways to iterate to higher levels of interactions. The common language should be simple, which means a restrictive type of interaction. This common language should be use to agree on some higher level of cooperation among entities. Implementation of this language for new members would not take much effort. Even more, there could be more than one common language (more that one standard) and nodes that know several standards could help other nodes to "learn" new common languages.

An example where the bootstrap negotiation process will solve many questions is in the discussion between state-less and state-full interactions. This is another aspect of an interaction that should be negotiated. State-less interactions are easier to implement (no crash recovery mechanism necessary,) but cost more since each message has to be self-contained. Often using state-full protocols will reduce the costs of communication. Again the nodes can agree on what style of interaction they are going to use and what type of state they are going to save. The protocol should also allow each node to check with the

other nodes to make sure that they are all in the same state.

Another aspect of VLDS is that it is difficult for an entity of the system to know what happen when something goes wrong in another entity, e.g., maybe a new operating system was brought up in another machine or a new RPC protocol is being used. By way of the bootstrap negotiation process, the interaction between these entities could again be resumed from the lowest common level, iterating to the appropriate level of cooperation. Therefore allowing new configurations to take place.

References

Needham1982.

Needham, R. M. and Herbert, A. J., *The Cambridge Distributed Computing System,* Addison Wesley (1982).

Cheriton1988.

Cheriton, David R., "The V Distributed System," *Communications of the ACM* **31**(3) pp. 314-333 Association for Computing Machinery, (March 1988).

Popek1981.

Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings Eight ACM Symposium on Operating System Principles,* pp. 169-177 (December 1981).

Zimmermann1980.

Zimmermann, H., "OSI Reference Model: The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications* **COM-28**(4) pp. 425-432 (April 1980).

SUN1988.

, "ONC/NFS Protocol Specifications and Service Manual," Part No. 800-3084-10, SUN Microsystems, Inc. (Revision A, of 26 August 1988).

Turnbull1987.

Turnbull, Martin, "Support for Heterogeneity in the Global Distributed Operating System," *Operating System Review* **21**(2) pp. 11-21 SIGOPS, (April 1987).

Schmidtke1982.

Schmidtke, F. E., "A Communication Oriented Operating System Kernel for a Fully Distributed Architecture," *Pathways to the Information Society. Proceedings of the 6th International Conference on Computer Communication,* pp. 757 - 762 North-Holland, (1982).

Notkin1988.

Notkin, David, Blank, Andrew P., Lazowska, Edward D., Levy, Henry M., Sanislo, Jan, and Zahorjan, John, "Interconnecting Heterogeneous Computer Systems," *Communications of the ACM* **31**(3) pp. 258-273 Association for Computing Machinery, (March 1988).

Reliably Delivering Ordered Multicasts

Annemarie Spauster

This is joint work with Hector Garcia-Molina.

The Problem

In distributed systems a *multicast group* is a collection of processes that are the destinations of the same sequence of messages. These messages may originate at one or more source sites and the destination processes may run on one or more sites, not necessarily distinct. Each source message is addressed to the multicast group (as opposed to individual sites or processes). The multicast protocol ensures that the messages are delivered to the appropriate processes.

For some applications, the multicast protocol must provide guarantees regarding the order in which messages are delivered to the destination processes. The strongest one that we have considered is the *multiple group ordering* property.

> *Multiple group ordering.* If messages m_1 and m_2 are delivered to two processes, they are delivered in the same relative order, even if they come from different sources and are addressed to different but overlapping multicast groups.

Aside from ordering messages consistently, it is often essential that the multicast protocol exhibit some *reliability* properties. Various degrees of reliability are possible. For example, if a site fails, it may be important that the other sites in its multicast groups still get the group messages according to the required ordering property. It may or may not be necessary to guarantee that the failed site get missed messages upon recovery. Just as important as providing reliability is providing it *efficiently*. Especially during failure-free operation (by far the most likely situation), it is essential that the overhead of the reliability mechanism be minimal.

The Propagation Graph Algorithm

The solution we propose to the multiple group ordering problem attempts to strike a compromise between previous solutions that are either fully distributed [BJ87] (and require many messages and long delay) or centralized [CM84] (and suffer from bottlenecks). The propagation algorithm orders messages using a collection of nodes structured into a *message propagation graph* (in particular, a forest). Each node in the graph represents a computer site. The graph indicates the paths messages should follow to get to all intended destinations. Instead of sending the messages to the destinations and then ordering them, the messages get propagated via a series of sites that order them along the way by merging messages destined for different groups. Eventually, all messages end up at their destinations, already ordered. The key idea is to use sites that are in the intersections of multicast groups as the intermediary nodes.

In Figure 1 we show a propagation graph for the following set of multicast groups. There are nine sites: a, b, c, d, e, f, g, h and j and eight destination groups:

$$\alpha_1 = \{c,d\},\ \alpha_2 = \{a,b,c\},\ \alpha_3 = \{b,c,d,e\},\ \alpha_4 = \{d,e,f\},\ \alpha_5 = \{e,f\},$$
$$\alpha_6 = \{b,g\},\ \alpha_7 = \{c,h\} \text{ and } \alpha_8 = \{d,j\}.$$

The originator of a message for a multicast group (the source) sends it to the member of the group that is of least depth in the tree (the primary destination). A site that receives a message propagates it down any subtree that contains members of the message's destination group. In Figure 1, if a source wants to send a message to group α_3, it sends the message to α_3's primary destination, d, d sends it to c and e, and c sends it to b. Along its journey, the α_3 message is merged with messages for the other groups on its path by their

primary destinations. Note that messages do not necessarily flow down to the bottom of the tree. For instance, g only receives α_6 messages. More details on the propagation graph algorithm can be found in [GS88].

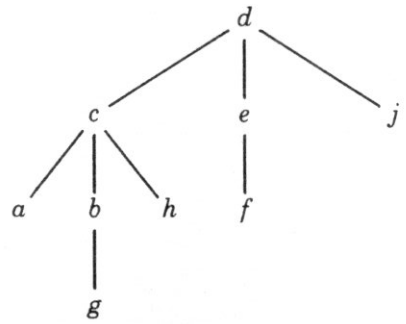

Figure 1

The Talk

With the solution to the ordering problem in hand, we will address the question of reliability.

References

[BJ87] K.P. Birman, T.A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February 1987, pp. 47-76.

[CM84] J. Chang, N.F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 251-273.

[GS88] H. Garcia-Molina, A. Spauster, "Message Ordering in a Multicast Environment," Technical Report CS-TR-161-88, Princeton University, June 1988. To appear in *Proceedings of the IEEE Ninth International Conference on Distributed Computing Systems*, June 1989.

Dynamic Global File Allocation

Carl Staelin

The problem is to automatically lay out files within a disk farm so that the overall performance of the system is globally optimal. The environment is a complicated one, with a variety of hooks available to the system to determine optimal policies. In addition, the process will be dynamic, in order to adapt to changing requirements.

Essentially, the system consists of a set of devices, with possibly varying operating characteristics, on which it can distribute data from a set of files. The probabilities of accessing any given file are strongly skewed, so that some files are dramatically hotter than other files.

The problem consists of three major sub-problems, some of which have been solved by other researchers. The first problem is to describe the I/O system so that the program is able to minimize contention, and to maximize performance. The second problem is to detect which files are heavily accessed, and to detect patterns of use for frequently accessed files. The third problem is to dynamically manage the storage space so that performance is maximized.

The first problem has been largely solved by work at IBM, and this area is interesting merely because of the wide variety of new devices which have become available recently, such as the jukebox and the electronic disk. With the advent of both massive online storage similar to the automated tape systems, massive storage systems with reasonable performance become feasible. However, the truly interesting development comes from the electronic disk, which may be used to provide fast reliable updates.

The second problem has not really been addressed, and it implies that some data collecting capability is built into the file system. The problem is to determine which data is most helpful for predicting future file use. Some work has been done in this area, but these results are preliminary, and more detailed data collection should be done.

The last problem is the truly interesting problem - how to dynamically manage data in a complex environment. Given the data collected in problem two, how can the system globally layout files to maximize performance (and reliability?). First of all, there has been substantial work at IBM on solving the static version of this problem, in a greatly restricted environment which precludes interesting solutions such as striping and which primarily focuses on load balancing. Some initial strategies are described below, and their use depends on how the data is accessed, and on the size of the file.

For (large) files which are always accessed in a linear fashion, bit striping the data so that each disk has one bit of each word is an attractive solution. First of all, the throughput is multiplied by the number of bits in the word (the extent of the striping). Secondly, reliable operation in the face of catastrophic (independent) disk failure can be accomplished simply by adding an extra disk which stores some checksum information.

For small files, the best solution might simply to hope that the truly hot files will stay in a cache most of the time, and then to allocate the files near the center of each disk in order to minimize seek times. In addition, the system should probably consider these files as being relatively mobile, since they are easily moved and since the hot files will have a dramatic effect on load balancing. In other words, the system might initially focus on balancing the load simply by moving the small hot files around.

For large files which are sometimes accessed randomly and sometimes scanned in a linear fashion, block striping might be an attractive solution. In this case one disk might have blocks 0, 8, 16, ... and another disk might have blocks 1, 9, 17, ... for a given file. For the random search the delay is similar to the delay for a normal file, while for the sequential scan the throughput is similar to the bit striped case.

References

1. Carl Staelin, *File Access Patterns*, CS-TR-179-88, Department of Computer Science, Princeton University, Princeton, N.J. 08540.
2. Joel Wolf, *The Placement Optimization Program: A Practical Solution to the DASD File Assignment Problem*, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, N.Y. 10598.

New Thoughts on Assemblers and Linkers

Paul Haahr

Assemblers and linkers are among the oldest and most frequently used of system programs. They are also among the most forgotten. While compiler technology has progressed significantly in the past decade, linkers and assemblers have changed little if at all. The few changes which are occurring in the design of these programs are generally being made to support dynamic linking and shared libraries, rather than altering the traditional breakdown of tasks. I am suggesting changes to the design of programming systems which would shift some tasks from compilers and assemblers to linkers, because it is more natural to handle those tasks late in the process of program creation.

My comments are meant to be taken in the context of the UNIX† operating system on large (at least 32-bit) linear address space machines, though many apply to a wider set of programming environments. Other environments, however, place different requirements on linkers. For example, on machines without virtual memory or not enough physical memory (i.e., PCs running MS-DOS) linkers normally handle overlays. Segmented memory architectures also change the function of linkers and assemblers.

The changes in design I am suggesting involve moving some tasks from compilers and assemblers to linkers. In doing so, the form of assembler output (and linker input) would change from relocatable object modules to little more than tokenized versions of assembler input. Also, it may be useful to add an optimization flag to linker invocation similar to the -O option commonly used to request compiler optimization.

Doing Compiler Post-Processing at Link Time

Address displacements are usually calculated by an assembler. Typically, an assembler must compute at least the size of both operand and branch displacements, and, quite commonly, must calculate the numerical value of these displacements, at least for branches within one function. However, for inter-module references, it is impossible to guess correctly the size of displacements, leading to assemblers that make the most pessimistic assumptions—i.e., all procedure calls use 32-bit jumps, all operands are specified as 32-bit addresses. In many cases, these assumptions introduce poor code.

Assemblers may synthesize complex instructions from simpler ones, but often a better job could be done by a loader. For example, on the MIPS R2000 architecture[8] loads from 32-bit addresses or with 32-bit offsets are done in several machine instructions, but the assembler provides one load instruction that, depending on context, will generate the appropriate machine instructions. To optimize the case of fetching from global and static variables (the most common case of 32-bit references in load and store instructions), the MIPS compiler set provides for a global data area which contains "small" static objects, so that they can be referenced in one instruction using a 16-bit offset. However, the compiler must know whether to place objects in this region, and it does so based on a user supplied hint—place objects of n or fewer bytes in the small data region—and changing the size requires recompiling all modules in a program. The compiler will tell the user if too many objects (more than 64K bytes worth) have been allocated to the small data region, or give a good hint size to maximally use the region[11]. If the linker constructed addressing modes and allocated this data region itself, this awkward communication between linker and compiler could be avoided. A similar technique to what I am proposing is used by Ken Thompson in the 2c compiler for the Motorola 68020, and by David Wall in the Mahler compiler series for the Titan[13, 12].

In the current generation of processor architectures, code has often has to be reorganized after code generation, to improve pipeline performance; for example, instructions may have to be moved

† UNIX is a registered trademark of AT&T.

into branch or load delay slots[10, 7]. Conventionally, reorganization is done by the compiler or the assembler. However, linkers may have more information useful in code reorganization, i.e. target addresses or their contents.

Moving Work from Compilers to Linkers

Procedure call can be an expensive operation. One factor which influenced the design of current RISC processors is the time-consuming call instructions on the VAX architecture[4, 9]. Much of the cost of procedure call is saving registers. Often, register windows or an on-chip top of stack cache[5] are used to make calling inexpensive. On machines with conventional simple register organizations, interprocedural optimization can make up for the lack of such hardware support, and sometimes perform better[14]. Some approaches to interprocedural optimization have been done in a separate post-compiler, pre-linker stage[3] while others have shifted more work to the linker[12]. Problems with making such optimizations outside the linker arise because the post-processor may not have information for all modules. Moreover, this extra stage will normally compile exactly the same information as a linker would, but do so independently of the linker, hence the same job is being done twice. (This result is not a necessary consequence of separate organization, though it is likely in practice.)

Procedure inlining can also be done inside linkers. In this way, a compiler would not have to pay attention to inlining, other than to possibly flag certain procedures as appropriate to inline.

Bringing these ideas to their logical conclusion leads to the idea of doing code generation at link time based on intermediary code generated by the compiler. Some work in this area has already been done[13, 1]. More investigation is needed in this idea, especially as to whether it introduces too much complexity in the linker or makes link-time a bottleneck in program development.

Shared Libraries and Dynamic Linking

Shared libraries and dynamic linking are often discussed in one breath. However, they are best thought of as two concepts that have a synergistic relationship: the most useful application of dynamic linking may be shared libraries, and shared libraries may be best used when they are dynamically linked. However, the combination of dynamic linking and shared libraries can also create performance problems in a demand paging environment[6].

Dynamic linking simply requires that a program have some access to its own symbol table—either on disk or in RAM—as well as that of the code to be linked in. Support code to do this in C on UNIX requires only about 200 lines of C code that invokes the linker as a separate process. Doing it efficiently, so that one function at a time may be linked in, rather than dynamically loading an entire library at once, may require writing a linker subset as a library that can be loaded with normal applications. In principle, the idea is straightforward.

Shared libraries can be easy or hard to implement depending on how dynamic they are. A library that is always loaded at the same location and makes no references to variables or procedures local to the program loading it can be used with no dynamic linking. Libraries that are mapped at arbitrary addresses and make use of symbols defined in a non-dynamic module require extensive dynamic linking, which may touch many or all pages in an address space and lead to thrashing.

Support for New Tasks

Many programming languages allow a programmer to specify initialization code for particular modules or data structures; for example, module bodies in Modula or Oberon, and constructors in C++. Implementing this with a conventional linker requires special post-linker processing, or invoking a tool which then invokes a linker, in addition to startup code in the language's run time system. If a linker provides a primitive that creates a list collected from various object modules, initialization can be done easily with little or no post-processing and trivial run time library

support.

Linkers transmit symbol table and type information from compilers to debuggers. In a conventional linker, this support is grafted on, and requires contortions from both compilers and debuggers[2]. With better linker and assembler support, compiler information and data structures could be stored directly in an image.

Compatibility

If the changes suggested here are implemented, many existing object files would be made obsolete, even if target image formats do not change. In order to maintain compatibility, wrapping programs which would take old object files and translate them to new linker (or assembler) input would be possible. Depending on which of the above ideas were implemented, there would be more or less work in the wrapper program.

References

1. Manuel E. Benitez and Jack W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 329-338 (June 22-24, 1988).

2. Tom Cargill, "Pi: A Case Study in Object-Oriented Programming," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 350-360 (August 1986).

3. Fred C. Chow, "Minimizing Register Usage Penalty at Procedure Calls," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 85-94 (June 22-24, 1988).

4. Digital Equipment Corporation, *Vax Architecture Handbook*. 1981.

5. David R. Ditzel, Hubert R. McLellan, and Alan D. Berenbaum, "Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor," *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems*, (October 5-8, 1987).

6. Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks, "Shared Libraries in SunOS," *Proceedings of the Summer USENIX Conference*, pp. 131-146 (June 1987).

7. John L. Hennessy and Thomas R. Gross, "Code Generation and Reorganization in the Presence of Pipeline Constraints," *Proceeding of the Ninth Conference on Principles of Programming Languages*, pp. 120-127 (January 1982).

8. Gerry Kane, *MIPS R2000 RISC Processor Architecture*, Prentice-Hall (1987).

9. David A. Patterson and David R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* **8**(6) pp. 25-33 (October 1980).

10. David A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM* **28**(1) pp. 8-21 (January 1985).

11. MIPS Computer Systems, *MIPS Language Programmer's Guide*. 1986.

12. David W. Wall, "Global Register Allocation at Link Time," *Proceedings of the SIGPLAN '88 Conference on Compiler Construction*, pp. 264-275 (July 1986).

13. David W. Wall and Michael L. Powell, "The Mahler Experience: Using an Intermediate Language as the Machine Description," *Proceedings of the Second Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 100-104 (October 5-8, 1987).

14. David W. Wall, "Register Windows vs. Register Allocation," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 67-78 (June 22-24, 1988).

Software Tools Session

Sally McKee started off this session with a discussion of a debugger for Plan 9 which she has built; in particular stressing some of the special features of Plan 9 and how they can be exploited in designing a debugger. Christos Nikolau suggested maintaining computational history for debugging. Rafael Alonso suggested that this system is natural for distributed debugging

Dave Hanson discussed tools for combining documentation and programs ("literate" programming.) The comment was made that such systems are good for large production systems. Hector Garcia-Molina asked what web does, and this was explained. Dick Lipton said that after the fact annotation is harder - documentation systems should make it easy to use them from the start. Paul Haahr asked why these systems only work with monolithic programs. Mark Greenstreet asked why programs need to be read in a fixed order. It was pointed out that coding is only a small part of software life cycle. Norman noted that Hanson's system seems to work on 25k line programs.

Eleftherios Koutsofios discussed a tool for building structured pictures (such as trees) based on a combination of a markup language to specify constraints, and a WYSIWYG display of the result. Carl Staelin suggested having WYSIWYG tools to modify constraints Mark Greenstreet said that similar problems occur in VLSI CAD systems

Chris Clifton introduced ideas for combining Hypertext-style browsing with database-style queries. Luis Cova suggested that people like two-dimensional spaces and that the user should be able to specify how to navigate. Karin Petersen suggested introducing a "meta node" for navigational information. Dick Lipton identified two modes for lookup; either you know what you want and look in index (easy), or you browse through lots of text quickly (harder). It was suggested that books are much better for this latter type of browsing than text stored on computers. Mordecai Golin suggested keeping track of previous indexes. Stu Feldman suggested looking at work done by Kim Fairchild at MCC. Rafael Alonso said that he thinks people can handle three-dimensional views of the database with practice. Pat Simpson suggested looking at spacial data management systems for information on three-dimensional data browsing. Stu said that keyword searches are more successful at locating data than spacial metaphors.

Debugging on the Gnot

Sally McKee

Plan 9 is a new computing environment under development at Bell Labs.[1] The environment is expected to accommodate small or large numbers of users, and the goal of the project is to provide a system to encompass all of AT&T's research and development. Rather than trying to build a big system by patching together a lot of small UNIX's, they are building one big UNIX from a lot of smaller systems: clusters of file and execute servers are connected by high speed networks, with lower speed distribution networks connecting the user interfaces to the servers.

The present user interface is a Motorola 68020-based, grey-scale bitmap terminal with mouse and network connections (most people would call it a diskless workstation) called a Gnot.[2] One of the Gnot's interesting features is a hardware instruction counter (it actually counts instruction pre-fetches) intended for research into hardware assists for program debugging.

As in 9th Edition Unix, Plan 9 supports a file system, /proc, that allows one to obtain information about and interact with living processes.[3] /proc contains a directory for each active pid; so, for example, information about the process with pid 12345 is found in directory /proc/12345. Each of these directories contains 5 files:

ctl	a write-only file for control messages (signals don't exist)
tac	a file corresponding to the hardware instruction counter
mem	the process's virtual address space (some parts of which are read-only), along with appropriate parts of kernel memory
text	a read-only link to the executable file that created the process
proc	a read-only copy of the kernel's proc structure for this process.

The /proc interface makes it easy for one program to access the address space of another; this is especially important in interactive debugging, where the debugger and the object being debugged are separate processes. This talk discusses the implementation of the /proc file system on the Gnot and how a debugger might make use of these files.

References

1. David L. Presotto, "Plan 9 from Bell Labs -- The Network," *EUUG conference proceedings*, (April 1988).
2. Bart N. Locanthi, "This is Gnot Hardware," Technical Memorandum #11276-870629-04TM, AT&T Bell Laboratories (October 1988).
3. T. J. Killian, "Processes as Files," *USENIX conference proceedings*, (summer 1984).

Tools for Literate Programming

David Hanson

Until recently, programs were not "read". That is, programs were prepared with the tacit assumption that their primary "reader" was the computer. Recently, several researchers have been looking at programs as a form of literature that is intended for human consumption first and computer consumption second. Not surprisingly, current programming languages and programming environments support this new view poorly.

I am currently investigating tools and techniques for integrating program fragments and explanatory text into a new form of "program". For the present, this work is using current high-level languages, such as C, and document formatting systems, such as **TeX** and **troff**. In addition, I am investigating tools, such as editors, that facilitate this integration. The goal of this work is to drastically reduce the effort required to write and document programs while at the same time greatly increasing their literate value. In the long term, a more interesting research direction---and a more important one---is to investigate the effects of the new literate view of programs on programming languages and environments. Such investigations may lead to the design and implementation of languages and environments that alter dramatically the way software is produced.

A Graphics Editor for Technical Pictures

Eleftherios Koutsofios

I am currently working on an editor for technical pictures.

Technical pictures (for example the picture of a graph, a tree, or a network layout), tend to be both very precise and very complex. Drawing a 100 node tree is impossible to do in a WYSIWYG system, both because drawing by mouse isn't very precise and because drawing and connecting 100 nodes is a lot of work. On the other hand, describing such a tree using a program is very simple. After all, in many cases the tree would represent some state of an algorithm so it should be trivial to change the program that implements the algorithm to also generate the tree as output. Once the user has generated a draft version of a picture using a program, it is generally desirable to use a WYSIWYG editor to make small changes to it.

The editor I'm working on will present two views of a picture to the user: the program view and the WYSIWYG view. It will allow the user to make changes to either view and will take care of keeping the two views up to date.

The language of the editor will be object oriented. Each object will specify constraints between it and other objects. Constraints are very appropriate for describing such pictures, since technical pictures have structure and symmetry that can be easily expressed by constraints.

The design of this system presents several interesting problems. From the user interface point of view, the main problems are how to use the mouse, and how to make it clear to the user which part of the program corresponds to which part of the picture.

References

1. Brian W. Kernighan, *PIC - A Graphics Language for Typesetting, Revised User Manual.*
2. Greg Nelson, "Juno, a constraint-based graphics system," *SIGGRAPH*, pp. 235-243 (1985).
3. Ivan Sutherland, "Sketchpad, A Man-Machine Graphical Communication System," PhD thesis, MIT (Jan 1963).
4. Christopher J. Van Wyk, "A high-level language for specifying pictures," *Transactions on Graphics* **1**(2)(April 1982).

SpeedBrowsing

Chris Clifton

Hypermedia systems[1, 2, 4-7] are growing in popularity. Typically access to data in such systems is through a *browsing* interface, in which users follow pointers between documents at their leisure. In large document bases this leads to the problem of being "lost in hyperspace", in which the user has no idea of where to start to find the desired information.

We are working on a document database which encourages non-navigational queries while still allowing the pointer-chasing approach of browsing[3]. However, we have done little on a user interface to such a system. In this talk I will present some ideas for a window and menu based approach to querying a hypermedia database that allows the user to specify properties of the desired documents and quickly find them, skipping the intermediate browsing steps.

First thoughts suggest that there are two separate parts to each query; the *properties* of the desired documents, and the *scope* of the query. Document properties can be specified using a "Query By Example"[8] style of interface. In particular, a menu-driven interface can be built which relies on the database catalog/schema to provide "hints" to the user in constructing queries. Scope can be handled in many ways, either through a visual approach based on a displayed "database graph" showing the structure of the data, or through some type of written specification of the type of links to follow.

Applications we have envisioned include on-line libraries, multiple authoring systems, and technical manual browsers. In the talk I will present ideas for this interface, and look for feedback on how *you* would use such a system.

References

1. Robert M. Akscyn, Donald L. McCracken, and Elise A. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Communications* **31**(7)ACM, (July 1988).

2. S. Christodoulakis, M. Theodoridou, F. Ho, M. Papa, and A. Pathria, "Multimedia Document Presentation, Information Extraction, and Document Formation in MINOS: A Model and System," *Transactions on Office Information Systems* **4**(4) pp. 345-383 ACM, (October 1986).

3. Chris Clifton, Hector Garcia-Molina, and Robert Hagmann, "The Design of a Document Database," pp. 125-134 in *Proceedings of the Conference on Document Processing Systems*, ACM, Santa Fe, New Mexico (December 5-9, 1988).

4. L. N. Garrett, K. Smith, and N. Meyrowitz, "Intermedia: Issues, Strategies, and Tactics in the Design of a Hypermedia Document System," pp. 163-174 in *Computer-Supported Cooperative Work Conference Proceedings*, , Austin, TX (December 1986).

5. Danny Goodman, "The Two Faces of Hypercard," *MacWorld*, pp. 123-129 (October 1987).

6. Frank G. Halasz, Thomas P. Moran, and Randall H. Trigg, "NoteCards in a Nutshell," in *Proceedings of the CHI+GI '87 Conference*, ACM, Toronto, Canada (April 5-9, 1987).

7. Randall H. Trigg and Mark Weiser, "TEXTNET: A Network-Based Approach to Text Handling," *Transactions on Office Information Systems* **4**(1) pp. 1-23 ACM, (January

1986).

8. Moshé M. Zloof, "Query by Example," in *Proceedings of the 1975 National Computer Conference*, AFIPS Press, Anaheim, CA (May 19-22, 1975).

Database Management Systems Session

A Probabilistic Relational Data Model

Hector Garcia-Molina

It is often desirable to represent in a database entities whose properties cannot be deterministically classified. We develop a new data model that includes probabilities associated with the values of the attributes. The notion of missing probabilities is introduced for partially specified probability distributions. This new model offers a richer descriptive language allowing the database to more accurately reflect the uncertain real world. The basic relational-like operators for the model are defined and their correctness is studied.

IMPLEMENTING A KNOWLEGE BASE SYSTEM

Daniel Barbará

Department of Computer Science
Princeton University
Princeton, N.J. 08544

Abstract

In the past years, a lot of attention has been devoted to the use of techniques from artificial intelligence to construct programs with a limited capacity of reasoning in a particular realm of knowledge. Such expert systems require the treatment of information as knowledge and use the AI techiques for reasoning, problem solving and question-answering. When these systems are intended to be used in practical situations, the realization is made that a lot of progress is needed in the use of implementation techiques commonly adressed in Database technology. From these realization, researchers have become interested in the integration of the two technologies, giving way to the concept of Knowledge Base Management Systems (see [BM85].) Here the database consists of production or inference rules, and it must be queried for the next rule to apply.

Many systems must deal with estimates of uncertainty of knowledge provided perhaps by the expert itself. For instance, rules can be of the form: "If 'a' then 'b'," , possibly with a numerical or linguistic qualifier that indicates the belief of the expert in the rule. The system must be also able to arrive to conclusions and to justify them even when the evidence is in conflict. New instances of data might be conflicting and increase the degree of uncertanty. This is unavoidable, since the domain of interest is likely to evolve with time. (New roads are built, new facts are discovered about the patient symptoms, troops are moved, communication links are repaired.) Systems that do not manage uncertainty have a serious problem dealing with conflicting data. Even when the conflict is detected (which is in itself a costly operation), it is difficult to decide where the error resides.

Several techniques have been used to deal with uncertainty in expert systems, ranging from non-numerical ones to the use of probability theory. (See [S86] for a survey.) This work plans to investigate the use of the probabilistic relational data model (PDM) described in [BGP89], as a tool to build knowledge bases that deal with uncertainty.

The first step in this direction will be the implementation of a database management system based on PDM and residing on main memory on the GigaSun. The reason for these last requirement has been adressed repeatedly [BM85]: there are thus far no succesful means of taking advantage of large databases for knowledge purposes. The problem with storing such knowledge bases in disks is that access to the objects is essentially random. Thus, massive main memory is likely to improve the acess time and the practical usage of these systems.

Another topic we would like to address in this work is the cooperation among a collection of knowledge based systems which exchange information. Here again the PDM can serve as the framework to integrate estimates about knowledge coming from different sources in a network of knowledge systems.

References

[BGP89] D. Barbará, H. Garcia-Molina, and D. Porter, "A Probabilistic Relational Data Model," submitted to the 15th VLDB.

[BM85] M. Brodie, and J. Mylopoulos, Editors, "On Knowledge Base Management Systems," Springer-Verlag 1985.

[S86] D.J. Spiegelhalter, "A Statistical View of Uncertainty in Expert Systems," in "Artificial Intelligence & Statistics" W. Gale, editor, Addison-Wesley 1986.

Scheduling Real-Time Transactions with Disk Resident Data

Robert Abbott
Hector Garcia-Molina

A *real-time database system* (RTDBS) processes transactions with timing constraints such as deadlines. The system guarantees serializable executions while at the same time minimizing the number of transactions that miss their deadlines. Conventional database systems differ from RTDB ones in that the former do not take into account individual transaction timing constraints in making scheduling decisions. Conventional real-time systems, on the other hand, differ from RTDB systems in that they assume advance knowledge of the data requirements of programs and their goal is to guarantee *no* missed deadlines[2]. However, they do not guarantee data consistency. Such systems are called hard real-time. RTDB systems can be useful in many applications, one of which is air traffic control. The system that directly controls aircraft (e.g., to avoid collisions) is a hard real-time system. However, there are a lot of additional data that must be handled by a RTDB, including weather reports, flight schedules, traffic patterns, and so on. Transactions on this data have deadlines. For example, a pilot may want to compute fuel requirements taking into account current winds and flight routes. The deadline of such a transaction would be the scheduled flight departure time. If on route the plane approaches a storm, the pilot may request a revised flight plan, taking into account current weather, remaining fuel, and the status of the destination airport. The deadline would reflect how close the plane is to the storm. Missing these deadlines in very undesirable but not an immediate disaster (the plane can leave late or can circle in the air waiting for information). More important than missing a few deadlines is guaranteeing that the database is consistent. For instance, getting the wrong flight plan is worse than getting the right one a bit late. Other applications for RTDB include threat analysis in military systems and program trading in financial systems.

There are many new and challenging problems in designing a RTDB. Two of these problems were studied in [1]: transaction scheduling and concurrency control. In particular, that paper presented several algorithms for resolving lock conflicts and for determining in what order to execute available transactions. The algorithms were studied via detailed simulations. Two major assumptions were made in that work: (a) the database was memory resident, and (b) only exclusive locks were available.

In this paper we continue our investigations of real-time scheduling and concurrency control. Assumptions (a) and (b) have been dropped, a new set of algorithms has been developed, and some additional issues and measures have been considered. The new results, we believe, provide substantial additional insights into the operation of RTDB systems.

Allowing the database to reside on disk, with a portion residing in a main memory buffer pool, introduces more interesting questions that one might initially imagine. For instance, the disk is now a resource that transactions must compete for. How are the disk requests to be scheduled? Do the same real-time priorities that worked for CPU scheduling work for disk scheduling? Some disk controllers do scheduling on their own (trying to minimize head movement). Does this interfere with the real-time scheduling? Since transactions now are suspended more frequently (lock waits and IO waits), there are more opportunities for CPU scheduling. How do the CPU scheduling algorithms respond? Finally, transaction commit must be considered. That is, transactions must flush their dirty pages to disk and write log records. What priorities should these operations receive?

Should the log be placed on a separate disk?

Shared locks also introduce a new set of challenging questions. With exclusive locks only, conflicts always involve a pair of transactions, the holder and the requester. The conflict can be resolved by comparing the priorities (e.g., earliest deadline) of each. With shared locks, the holder can actually be a set of concurrently reading transactions, each with different deadlines. If the requester needs an exclusive lock, what is to be done? What priority does the group have? If the requester needs a shared lock, it could be granted immediately, but there may be other transactions already waiting for exclusive locks. How are the priorities of the waiting transactions compared against that of the new requester? Should the new requester be granted the shared lock or not?

We have extended the algorithms of [1] to cope with disk data and shared locks. In addition, we have studied concurrency control algorithms not considered initially, including one that *promotes* transactions that are blocking higher priority transactions. Finally, we have also considered two supplementary measures (in addition to mean number of missed deadlines). One is the mean tardiness of transactions, i.e., average time by which transactions miss their deadlines. The second is the response of the system to a batch of transactions that arrive at once. Such an "input step function" emulates a severe overload situation. Such overloads may not be frequent, but having algorithms that can cope with them gracefully is important.

References

1. Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," *Proceedings of the Conference on Very Large Database Systems*, pp. 1-12 VLDB, (August 1988).
2. Zhao, W., K. Ramamritham, and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *Transactions on Computers* **C-36** pp. 949-960 IEEE, (August 1987).

Characterizing Database Structure

Pat Simpson

It goes without saying that the structure, or schema, of a database can be expressed without regard to the actual data contained in it at any particular time. Given a schema, many instantiations are possible. Likewise, given a data model many schemas are possible, all of which can be expressed in the common data definition language of that model. Let us take this idea one step further. Suppose we have many independent databases whose schemas instantiate different data *models*. Is it possible to express (in a data definition *definition* language, perhaps) all of the data models in existence? Or, alternately, is it possible to express *any* schema, regardless of model, in some common and universally understandable form? If so, such descriptions could constitute a basis for interoperability among heterogeneous database systems.

We will make the distinction between interoperability of separate DBMSs and the construction of a heterogeneous DBMS. The latter entails the construction of a global schema which in turn requires resolution of conflicts in both schemas and data. We are concerned here with characterizing databases, not necessarily for the purpose of integrating them, but simply for the purpose of representing them to outsiders. As a concrete example of an application requiring general database characterization, consider the following implementation of remote browsing:

Let us assume that there exists a universally accessible communications network (such as the Integrated Services Digital Network being constructed over the next several years) and a large number (thousands or millions) of independent computer systems, many of which have information to share. In previous work[1] we've discussed mechanisms for direct retrieval of remotely held information via explicit queries, based only on a knowledge of the topics or subject areas to which the host's information is relevant. This kind of retrieval can be done in a black-box fashion – the querier need not establish an interactive connection with the host, and no knowledge of database structure is needed. On the other hand *browsing* activity, while not requiring *a priori* knowledge of subject area, is closely involved with the *structure* of the data being browsed – the links, attributes, and other relationships connecting elementary data items.

A few points should be made with regard to remote browsing. First, a global schema among such a large number of databases is almost certainly not constructible. (The unconvinced are referred to Kent[2] for exhaustive evidence against the existence of a single consistent world view.) Second, if each host were to be browsed individually, browsers would potentially need to learn a different browsing interface for each host. In a large network this is not practical. Since the browsed data must be transmitted anyway before it can be viewed by the browser, it may be desirable to implement the actual browsing function locally (i.e. on the browser's system) *after* the data has been transmitted rather than at the host site.

We propose to implement remote browsing locally, in the following manner. Each would-be browser defines, on his/her own system, a single self-customized browsing interface. (This may be the same interface as is used to view local data). A host transmits, not only the data to be viewed, but also structural information about that data (its schema) in a universally understood format. A remote browsing application running locally accepts both schema and data as input and permits the browser to view the data via the interface to which s/he is accustomed.

The diversity of data models to be represented may be extreme, including (for example) the standard database models – relational, hierarchical, network, and entity-relationship – in their various incarnations; logic databases, consisting of facts and rules; hypertext systems with arbitrarily named links; indexed full text retrieval systems; historical (time- or version-based) databases; unstructured sets of files with simple search and retrieval mechanisms like "grep" and "awk"; and specialized data structures such as the patricia trees used to store the Oxford English Dictionary. Is there a single "universal data model" that is sufficiently general to incorporate many existing models? Or can we identify a few well-characterized schema types which cover most of the possible schemas? In this discussion we'll take a long hard look at the basic elements and characteristics of data and data relationships in an effort to extract some unifying concepts.

References

1. Patricia Simpson and Rafael Alonso, "Querying a Network of Autonomous Databases," Technical Report CS-TR-202-89, Princeton University (January 1989).
2. William Kent, *Data and Reality: Basic Assumptions in Data Processing Reconsidered,* North-Holland Publishing Company, New York (1978).

Connecting Heterogeneous Databases

Rafael Alonso

When cooperating organizations decide to share the information in their databases, it is not uncommon for them to find that the data stored in their respective computer systems are kept in incompatible database management systems. To allow users to query the combined database in a straightforward manner, a mechanism is needed that will mask the details of each particular systems and present a homogeneous interface to the collection of database systems. A *heterogeneous database system* enables users to transparently access the data contained in a multiplicity of differing databases. During my talk I will discuss some of the issues involved in the design and implementation of a heterogeneous DBMS. I will also describe briefly the previous solutions proposed for this problem, most of which are algorithmic in nature. Finally, I will sketch a possible new approach to this problem, which involves the use of an expert system to encode schema information in order to perform query translation.

Parallel Systems Session

This session opened with Jeff Naughton discussing work he is doing with Kai Li on *Shared Single Level Store*, which combines virtual memory and persistent storage ideas on a multiprocessor system. Rafael Alonso asked if it was appropriate to make everything transactional and persistent. Jeff suggested that certain portions of the memory could be so designated. Rafael also asked how this differed from the approach in *Linda*. Jeff suggested that the concept was similar, but the programming paradigm encouraged by the system is different. Mark Greenstreet suggested that this system would ease context switches. Chris Clifton asked how this would ease the writing of parallel applications. Jeff noted that lightweight processes (threads) are a natural programming paradigm for this system, and that I/O is also eased. Richard Lipton noted that this was a nice model, but if it was significantly slower than approaching the machine directly it would probably gain little acceptance. Given the problems highly parallel machines are used for, performance is more of an issue than ease of use. Andrew Appel suggested that this model would make it easy to move processes which are communicating closer together, thus possibly gaining performance over a static process allocation. Stu Feldman noted that this was a tempting paradigm for a wide class of problems. Mark Greenstreet suggested that it would be useful for simulating cellular automata.

Dimitrios Serpanos followed this talk with a hardware talk: an implementation of a parallel architecture using *Pipelined RAM*. This system scales well to both a large number of processors and a large distance. Rafael Alonso asked what applications would be appropriate for this system. Dimitrios noted that this really provided shared memory, and any shared memory application would run on PRAM. Rafael also asked if this would be best for systems which only synchronized occasionally. Richard Lipton responded yes. Paul Haahr asked what the eventual scale of such a system would be. Dimitrios said that there was no physical limit.

Andrew Appel talked about ideas in parallelizing functional languages. He pointed to a number of problems and solutions in this area. It was generally concluded that we are much farther along in automatic parallelization of functional languages than in imperative languages. Brad Barber asked if user specification of side effects (or the lack thereof) was necessary, or if this could be inferred from the type system. Andrew responded that this was an extremely difficult, and occasionally impossible, problem. Norman Ramsey suggested that side effects are rare, and could possibly be treated as special cases. Andrew noted that this might be reasonable, depending on the cost of handling the side effects. Chris Clifton expressed concern that these systems often required a small amount of user input in order to function properly, and that there was room for the user to "shoot themself in the foot". Andrew said that in some of the systems, the compiler could "check" the users assertions. Mark Greenstreet asked if it might be desirable to allow the user to specify where parallelism is appropriate.

The final talk in the session was Richard Lipton on programming arrays of processors. The talk was based on the **Xilinx** chip, which is composed of many simple processors which can be individually customized and connected. This gives many of the advantages of custom chips at a much lower cost. He suggested that a spreadsheet-style program could be used to program this chip, with each processor representing a location in the spreadsheet, and the *formula* for that cell the program for the processor. He ended the day with a demo of such a spreadsheet, and showed how a simple counter could be quickly and easily developed.

The Design of a Distributed Shared Memory

R.J. Lipton and D.N. Serpanos *
Computer Science Department
Princeton University

1 Introduction

PRAM [1] is a new shared memory model which offers high performance and scalability. The main novelty of *PRAM* is that it does not enforce coherence at the hardware level. This allows *PRAM* systems to overcome the main limitations of conventional shared memory systems: low scalability both in distances between processors and in numbers of processors. Following a RISC-like philosophy [Hen81], the burden of synchronization and mutual exclusion has been moved to software, while the hardware is tuned for fast data transfers. A detailed description of *PRAM* is in [LS88].

The current *PRAM* system connects two IBM PC/AT's through *PRAM* boards installed on their busses. The two boards implement pipelined communication over fiber links (hence the name *PRAM*) achieving data transfer rates up to 2.5 MBytes/sec. The system is fault tolerant since error detection/correction mechanisms are used at both the hardware and the software level.

In order to demonstrate the ability of *PRAM* systems to share memory among many processors we are building a switch that allows more than two processors to be connected simultaneously. The specifications and the design of such a switch is presented in this paper. The switch does not place any restrictions on the types of the connected processors, however the messages sent through the switches must conform to our given format. No delay bounds are assumed.

*This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

[1] Pipelined RAM

2 Specification of the Switch

Before we can specify the design of our switch we need to review how *PRAM* operates. Suppose that two processors P_1 and P_2 use *PRAM* to share memory. Then each keeps a local copy of the shared memory. In order to read, each processor just reads its own local copy. In order to write, each processor just writes its own local copy and at the same time writes over the link to the other local memory. It does not wait nor does it synchronize in any way its actions with the other processor as other conventional shared memory systems do [AB84] [Kat85]. The advantage of this approach is high performance: the current prototypes can transfer 2.5 MBytes/sec. Note, this does not depend on large block sizes: this performance is achieved even for single writes of one word. The disadvantage is that the shared memory can become incoherent. The key point is that [LS88] shows that compiler technology can be used to shield this incoherence from the programmer. Performance is not affected by incoherence, since in most common parallel programs write contention is a small fraction of the total memory requests [EK88].

Clearly, there is nothing fundamental about two processors, i.e. the *PRAM* method can be used to share memory among more than two processors. However, in order to do this without modifying the current *PRAM* implementation we require a special switch box. Such a box is connected to n processors each with a *PRAM* memory card. When a processor writes to the shared memory its card will send the switch a message. The switch will then pass this message on to all the other processors. Thus, rather than sending out $n-1$ messages each card only sends out one message: the switch box automatically broadcasts the messages to the other boxes.

There are four practical issues that make this box interesting: First, it must be fault-tolerant. Thus, the switch must be able to handle transmission link errors and other kinds of errors. Second, the switch must be able to handle "flow-control". Any implementation of the box will contain a variety of internal queues. Since in a real implementation these queues are finite, the box must be able to start and stop links without overflowing the internal queues. Note, both these issues arise in the current two processor implementation but are greatly simplified since only two processors are involved.

Third, the box must be programmable. Consider the case of four processors: suppose that P_1 and P_2 wish to share memory and P_3 and P_4 wish to share memory. Then each time P_1, for example, writes to its shared memory not only will P_2 get a message but P_3 and P_4 will also. This is clearly a potential waste of the link bandwidth. Thus, it is critical that the switch allow the programmer to selectively decide which memory is shared and by whom. We consider this kind of memory mapping critical.

Finally, the box must be able to be used not only with processors but also with other boxes. Clearly, the key advantage of this ability is that we can build very large shared memory systems if boxes can be used in this recursive manner.

It turns out to be possible to achieve this provided one is very careful in exactly how the switches are implemented. The principal problem is one of potential deadlock. Without careful design it is possible for sets of boxes to signal each other in such a way that all stop forever.

3 Design of the Switch

The switch, currently under implementation, has 8 inputs and 8 outputs and operates in 2 modes:

1. **Propagation Mode:** where messages are being selectively broadcasted to the rest of the processors and

2. **Programming Mode:** where messages are used to program the switch.

Since all incoming messages have the same format: 16 address bits and 16 data bits (see fig.2), the *Programming Mode* is selected by the use of a specific address. So the *Programming Mode* is mapped into shared memory.

The data and control flow path of the switch is shown in fig.1. The path is replicated for the 8 input/ output pairs, except the bus and the Priority_Resolution Module which are unique in the switch.

Messages come into the receivers (RCVR[I]) asynchronously at a maximum rate R and they are forwarded to the input queues (IN_Queue[I]). Then they get in the IN modules which in turn assert their request (REQ[I]) lines. The Priority_Resolution module selects one of the requests and by sending a GRANT signal it instructs the selected module to put its message on the bus. All the OUT modules latch the bus signals simultaneously and depending on the mode of operation they either read their memory to find whether they are going to send the message to their output queue (Propagate Mode) or they check whether they have to update their memory by writting into it (Programming Mode). Note that only one memory is updated with one *Programming Message* and the rest of the modules disregard the message. If the mode is *Propagate* and the memory lookup shows that the message has to be transmitted, they forward the message to their output queue (OUT_Queue[I]) and from there to the transmitter (XMTR[I]).

Except for the above messages, which are generated by the processors connected to the switch (or to a network of switches), there are other messages too, which are generated by the switch boxes themselves. These are the **Exception** messages. They are of 3 types: **Error**, **STOP** and **START**. The *Error* message is generated whenever a transmission link failure occurs. The receiver which is connected to the link "senses" the failure and generates a message with an address identifying the link. The message is immediately transmitted to the system (switch or processor) which transmits over that link and it also pushes the message to its input queue to be broadcasted to the rest of the connected

systems. In the meantime special circuitry turns the receiver off for as long as the link is failed, so that no disturbed data are received.

The *STOP* message is generated by an input module whenever its associated input queue gets Half-Full, because there is potential danger of overflow (Data Overrun). The *STOP* message is immediately sent to the system that is connected to the specific input module (the sender), so that it can temporarily suspend its transmission. When the input queue goes Under-Half-Full again, the input module sends a *START* message to the sender in the same way as before, so that it can resume transmission.

The output queues are in danger of overflowing too. This problem is solved by having the Priority_Resolution module stop granting input requests, whenever a number of output queues get Half-Full. The *Priority_Resolution Algorithm* implemented is fair and efficient as it keeps servicing incoming requests at the highest possible rate. It also plays an important role in the avoidance of a deadlock when 2 switches communicate through a 2-way connection (fig.3).

As far as performance is concerned, the cycle of the switch box is less than $1/R$, which guarantees that at least one request of each of the 8 inputs is serviced.

4 Conclusions

The design for the presented switch box supports memory sharing among heterogeneous processors which can be geographically separated, since it does not enforce (or assume) any delay bounds. The most important characteristic is *scalability* as it supports connections to other switch boxes.

When designing an interconnection of switches, one should be very careful about the topology of the resulting network. We can easily prove that cycles in the network can result in unrecoverable deadlocks. One can easily build recursive structures that do not contain cycles and operate correctly, as for example the tree structure in fig.4.

References

[AB84] J. Archibald and J.L. Baer. An Economical Solution to the Cache Coherence Problem. In *11th Annual International Symposium on Computer Architecture Conference Proceedings*, 1984.

[EK88] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluations. In *15th Annual International Symposium on Computer Architecture Conference Proceedings*, 1988.

[Hen81] Hennessy J.L. et al. MIPS: A VLSI Processor Architecture. Technical Report TR-223, Stanford University, November 1981.

[Kat85] Katz R.H. et al. Implementing a Cache Consistency Protocol. In *12th Annual International Symposium on Computer Architecture Conference Proceedings*, 1985.

[LS88] R.J. Lipton and J.S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

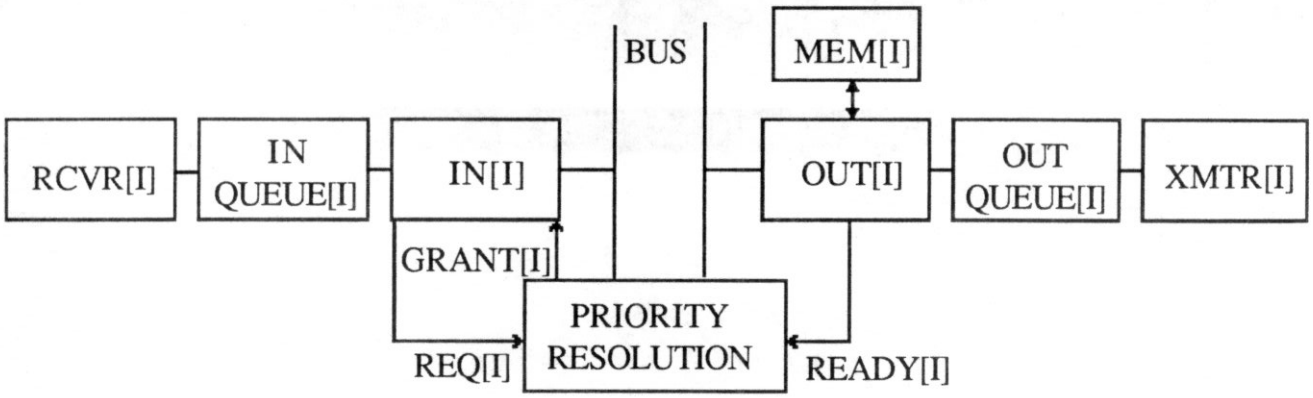

Fig.1: Block Diagram of the Switch

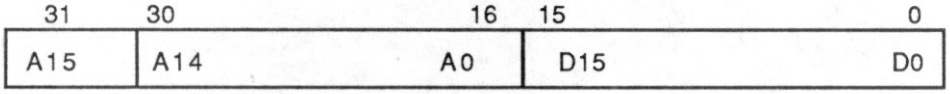

Fig.2: Message Format

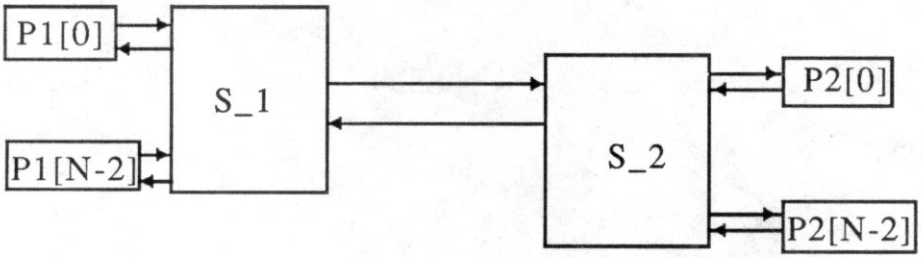

Fig.3: 2-Switch Connection

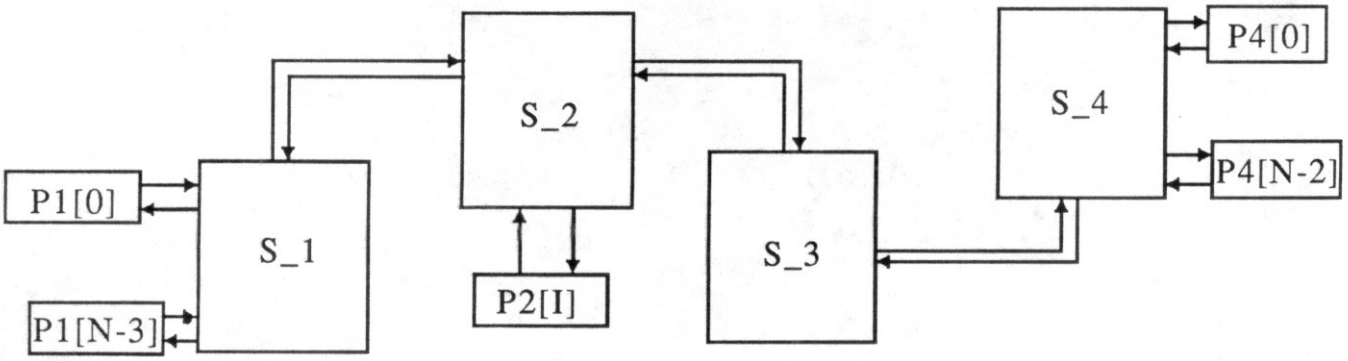

Fig.4: Tree Structured Network

SSLS: Shared Single Level Store

Kai Li and Jeff Naughton

A Shared Single Level Store (SSLS) provides message-passing multiprocessors with a global, persistent memory space. The address space is *shared* in that a reference to a given memory address means the same thing regardless of which processor in the multiprocessor makes the reference, and it is *single level* in that there is no notion of secondary or disk storage.

The SSLS simplifies the programming of multiprocessors in two main ways:

- The programmer need not program explicit data allocation and transfer among processors, and
- The programmer need not program explicit I/O.

With a properly implemented SSLS, to the programmer the multiprocessor appears to have a large, global memory that persists between program invocations and even machine crashes.

We currently in the process of the design and implementation of a SSLS for two target multiprocessors: a network of DEC Firefly multiprocessrs, and an Intel iPSC-2 hypercube.

Parallel Functional Languages

Andrew Appel

Because side-effects are rare or absent in functional programming languages, order of evaluation of program fragments is much freer. For this reason, it is natural to use functional languages as a basis for parallel programming.

Functional programming languages come in two styles: "pure" functional languages, in which no side effects (assignments, etc.) are permitted, and "impure" functional languages, in which side effects are permitted but rarely used. SASL and the new language Haskell are examples of pure languages, and Scheme, ML, and FX are examples of impure languages.

Both kinds of languages can be parallelized, but in much different ways. The pure languages pose no restrictions on order of evaluation, so that nonstandard orders or concurrent execution is possible without direction from the programmer. Thus, it may be possible to parallelize programs completely automatically. The impure languages may require explicit programmer specification, e.g. using the "futures" concept of MultiLisp, the "effects" specifications of FX-87, or the CCS-like structures of PFL.

The Standard ML of New Jersey compiler, developed at Princeton and Bell Labs, will be an ideal base for research into both kinds of parallelism. It is efficient, robust, and has a clean compiler and run-time system with few obstacles that might stand in the way of shared-memory parallelism[1-7].

References

1. Andrew W. Appel and Trevor Jim, "Continuation-passing, Closure-passing Style," in *Proc. 16th Symp. on Principles of Prog. Languages*, (Jan. 1989). (also Princeton CS-TR-183-88).

2. Andrew W. Appel, "Allocation without Locking," *Software -- Practice & Experience*, (to appear). (also Princeton CS-TR-182-88).

3. David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon, "FX-87 Reference Manual," MIT/LCS/TR-407, MIT (Sept. 1987).

4. Robert Harper, David MacQueen, and Robin Milner, "Standard ML," ECS-LFCS-82-2, Univ. of Edinburgh (1986).

5. Sören Holmström, "PFL -- A functional language for parallel programming, and its implementation," Report #7, Univ. of Goteborg (Sweden) (Sept. 1983).

6. Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall (1987).

7. David A. Kranz, Robert H. Halstead, and Eric Mohr, "Mul-T: A High-Performance Parallel Lisp," in *SIGPLAN '89 Conf. on Prog. Lang. Design & Implementation*, (to appear).

Programming Arrays of Identical Processors

Richard J. Lipton

It is now possible to build large arrays of simple boolean processors. We will soon have such a machine at Princeton with over 10,000 such processors. Its potential performance is in excess of 40,000 MIPS. The central problem is: how can we program such an array? This talk will focus a number of open problems concerning the programming of such arrays. These include practical issues as well as open theoretical questions.