SUBMICRON SYSTEMS ARCHITECTURE
SEMIANNUAL TECHNICAL REPORT

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

5235:TR:86

Computer Science Department
California Institute of Technology

December 1986

SUBMICRON SYSTEMS ARCHITECTURE

Semiannual Technical Report

Department of Computer Science
California Institute of Technology

5235:TR:86

5 December 1986

Reporting Period: 16 March 1986 to 15 November 1986

Principal Investigator: Charles L Seitz

Faculty Investigators: James T Kajiya
Alain J Martin
Robert J McEliece
Martin Rem
Charles L Seitz

Sponsored by the
Defense Advanced Research Projects Agency
ARPA Order Number 3771

Monitored by the
Office of Naval Research
Contract Number N00014-79-0597

SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This document is a summary of the research activities and results for the eight month period, 16 March 1986 to 15 November 1986, under the Defense Advanced Research Project Agency (DARPA) Submicron Systems Architecture Project. Technical reports covering parts of the project in detail are listed following these summaries, and can be ordered from the Caltech Computer Science Library.

1.2 Objectives

The central theme of this research is the architecture and design of VLSI systems appropriate to a microcircuit technology scaled to submicron feature sizes. Our work is focused on VLSI architecture experiments that involve the design, construction, programming, and use of experimental message-passing concurrent computers, and includes related efforts in concurrent computation and VLSI design.

Additional background information can be found in previous semiannual technical reports [5052:TR:82], [5078:TR:83], [5103:TR:83], [5122:TR:84], [5160:TR:84], [5178:TR:85], [5202:TR:85], [5220:TR:86].

1.3 Highlights

Some highlights of the previous 8 months are:

- All sections of the Mosaic C designed and laid out (sections 2.2.3 and 4.1).
- Cantor programming language operational and in use (sections 2.2.2 and 3.1).
- Joint projects initiated with two companies to build prototype second generation "cubes" (section 2.3).
- New results in compiling multiprocess programs to self-timed VLSI designs (section 4.2).

2. Architecture Experiments

2.1 Cosmic Cube Project

W C Athas, Michael Lichter, Wen-King Su, Chuck Seitz

The Cosmic Cubes and Intel iPSC continue to run very reliably, with researchers at Caltech and at other DARPA sites using them for application programming projects. Usage has been moderately heavy.

Our own most intensive use has been for event-driven simulations of designs for the second generation message-passing multicomputers (see sections 2.3 and 3.4).

A system supporting Cantor, a fine grain concurrent object-oriented programming language, is now available and in regular use on the Cosmic Cubes and iPSC (see section 3.1).

2.1.1 Hardware and System Software Status

Neither the 64-node nor 8-node Cosmic Cubes has exhibited a hard failure in this 8-month period. The cubes have now logged 2.3 million node-hours with only 3 hard failures. The calculated MTBF of the nodes of 100,000 hours reported before these machines were constructed was conservative at above the 99% confidence level. A node MTBF in excess of 500,000 hours is probable, and can be stated now at a 70% confidence level. The systems will have to operate for another 2 years with a similar or smaller failure rate for us to be able to state a 500,000 hour MTBF with 90% confidence.

A 500,000 hour MTBF for hard failures in a node corresponds to an MTBF for a 64-node system of only approximately a year, which is hardly a problem in a system in which the mean time to repair is a few minutes. The reason for our interest in the *post facto* reliability experience with the Cosmic Cubes is to accumulate and share some experience that may be applicable to our own and other researchers's experimental prototypes. The three hard failures observed have been in chips, two in 64K dRAMs and one in a 74C-series line driver. There have been no connector failures. Systems with comparably complex nodes in current technology, and at a 1024-node scale, would exhibit an MTBF of about three weeks, which we regard as acceptable for a system of that scale and performance.

There have been no significant changes made in the cosmic cube system software in this period, except for the Intel iPSC compatibility package discussed below. There have been a large number of network problems reported by users outside of Caltech, but very few cube problems.

2.1.2 Intel Cube

Our Intel iPSC I d7 (128 nodes) is proving to be a valuable resource for the project. This machine was contributed to the Submicron Systems Architecture

Project as a part of the license agreement between the Caltech and Intel, and are accessible via the ARPAnet to other DARPA researchers who may wish to experiment with them. To request an account, please contact chuck@vlsi.caltech.edu.

Intel has gone through several revisions of their hardware and software since we recieved our iPSC last year, with improvements being made at each step in reliability, speed, and usability.

Since this last summer we have been running as a user option a beta release of the R3.0 node kernel, and are now running the production release. This new kernel is written in C, and is accordingly much easier for us to experiment with than was the R2.0 kernel. It is also much faster. The largest component of the message-passing latency on the iPSC cubes is startup time in the node operating system, and it has been significantly reduced in R3.0. The iPSC nodes are for integer computations about 4 times faster than the cosmic cube nodes. The iPSC message latency for short messages is now about 1ms, about half that of the cosmic cubes, and asymtotically one fourth as large for long messages. The cosmic cube was faster than the iPSC with R2.0 for short messages, but the relationship between communications and computing performance in these two systems is now similar.

Computing problems run on the iPSC show performance up to about 100 times faster than a VAX11/750 when good load balance and message locality are achieved in the concurrent formulation.

The cosmic environment has been generalized to support space-sharing on the iPSC and compatibility with the cosmic cube message functions. Except for programs that use the Cosmic Cube's more elaborate process spawning mechanisms, the same programs can be compiled to run on either the Cosmic Cubes or iPSC. These new system features are described in the July 1986 edition of the "C Programmer's Guide to the Cosmic Cube" [5203:TR:85].

Wen-king Su is now writing a new version of the cosmic environment package based on the new "reactive" message primitives discussed in section 2.3. Michael Lichter will be doing the same with the cosmic kernel and/or the iPSC kernel.

2.1.3 Applications

Many new applications have been written to run on our cubes. The Cantor system runs on both types of cubes, and quite a few of the newest applications have been written in Cantor (see section 3.1).

We continue to make the cubes available to selected guest researchers for problems that appear to be particularly interesting or difficult.

Researchers in the Aeronautics department at Caltech, under the direction of Professors Brad Sturtevant and Tony Leonard, continue to report very good results on some difficult fluid mechanics computations. One of their most successful applications employs a Monte Carlo method for computing local interactions for

molecular dynamics, rarified gas (such as jet plumes), and granular flow, with a non-uniform grid to balance the load. They have also gotten very good results in experiments with the same lattice gas automaton approach to fluid mechanics computations that has been developed for and applied to the Connection Machine.

Researchers at the University of Washington, under the direction of Professor Larry Snyder, have ported the Poker programming system to the cubes.

Researchers at USC/ISI, under the direction of Dr David Mizell, have been regular users of the cubes. Chuck Seitz, Bill Athas, and Wen-King Su gave a one-day cube programming class at ISI on 10 July 1986.

2.2 Mosaic Project

Bill Athas, Charles Flaig, Fritz Nordby, Steve Rabin, Steve Roskowski, Don Speck, Wen-King Su, Chuck Seitz

A large fraction of the project's resources and effort have been devoted to the Mosaic project over the past eight months, and a great deal has been accomplished.

2.2.1 Working Toward a 16K-node Mosaic C System

Mosaic is a message-passing MIMD multicomputer similar to the Cosmic Cubes, but with the stipulation that the node be integrated onto a single chip. We regard this node size as the most interesting design point for this architecture from the viewpoint of exploiting VLSI technology. The stipulation of single-chip nodes limits the storage for each node so that relatively fine grain concurrent programming techniques must be used.

We are working toward building a 16K-node Mosaic system using nodes fabricated in 1.2μm CMOS technology, with a near-term milestone of a 1K-node system with nodes fabricated in 2μm CMOS.

The 16K-node system will be built as a 3D routing mesh ($32 \times 32 \times 16$). Each node will include 16K bytes of storage, a 16-bit instruction processor, and channels that perform wormhole routing on the mesh. The mesh topology allows the nodes and their communication to operate in a locally bounded skew clocking scheme. Our circuit simulations show that we can achieve a clock rate of 40MHz for this design in 1.2μm CMOS, which translates in the microcycle simulations to an instruction rate of about 12 MIPS. The channels send 2 bits per clock period, corresponding to a channel rate of 10 MB/s. In aggregate, the 16K-node Mosaic will have 256MB of storage, a peak performance of 200,000 MIPS (6,000 Mflops in floating point subroutines), and a bilateral bisection communication bandwidth of 5,120 MB/s.

The 1K-node system is both a near-term milestone and a programming development system. It will be organized as a 32×32 2D routing mesh. We will include as much storage per node as the 2μm fabrication technology and the quality of our

RAM designs allows, most likely about 6KB per node. Clock rate based on circuit simulations will be 20MHz.

We have calculated an MTBF for the 16K-node system of about 400 hours (three weeks). Although this MTBF is actually quite good for a system with such phenomenal performance, it is a source of some concern in that we estimate that the repair time for this 3D assembly will be an hour or so. It will be possible to use the working nodes of the system in spite of faulty nodes under the same "space-sharing" mode of operation that we have developed for the Cosmic Cubes.

The calculated MTBF of the 1K-node 2D system, essentially the same circuit and packaging technology, but with fewer package pins per chip, is nearly 10,000 hours. It is accordingy unlikely that we will observe enough failures in its operation to give us a better estimate for the 16K-node system. However, if we observe more than 2-3 failures in its first year of operation, we will know that we have a more serious problem in the 16K-node system than we have calculated.

2.2.2 Mosaic Programming Systems

The Cantor Model

The concurrent programming model for the Mosaic is a fine grain object-oriented model. Since the development of the Cantor programming system, based on this model, we are confident not only that we can program relatively fine grain systems such as the Mosaic without extraordinary efforts, but that these fine grain systems are efficient and surprisingly general. See section 3.1 for a discussion of Cantor.

The Cantor execution model fits message-passing ensembles very well, and has also dictated many features of the Mosaic C node architecture. Although Cantor is an "Actor" language, we do not always use Actor terminology to describe this model. Cantor execution deals with objects and messages. A Cantor object is normally at rest, and executes its program code, called its definition, in reaction to receiving messages. The object takes a finite time to complete a series of actions that may include sending messages, spawning new objects, and executing code with possible side-effects that change the object's persistent variables.

The message-driven or reactive character of object scheduling allows a very streamlined run-time system that must be resident in each node. This system manages incoming and outgoing message queues, and runs objects according to the contents of the incoming message queue. This system can be implemented in about 1000 bytes of Mosaic code. In order to be able to perform this message handling efficiently, the Mosaic C processor is a "two sequence" machine with two program counters and two overlapping register banks, so that it can process interrupts from the message system with zero context switching overhead.

Although the run-time system is assisted by the compiler in determining the placement of new objects, object placement is assumed to be little better than

random. The performance of the message network with non-localized message traffic is accordingly very important, and is accomplished by the "routing automata" channels (see sections 4.1 and 4.2).

Typical Cantor programs that have been run on the Cosmic Cubes and iPSC have many thousands of objects. The typical size of an object is much less than 100 bytes in the object table to represent the persistent variables, and several hundred bytes of code for each object definition. Thus the storage size even of the nodes of the 1K-node Mosaic C is adequate to support Cantor programs. Messages are typically short, 20 bytes or less, so the message system is optimized for short messages.

Mosaic C Simulators

The Mosaic detail simulation environment, meant principally for system development, supports the simulation of Mosaic ensembles with arbitrary connectivity, and with elements that are loaded with variable microcode and bootstrap programs. There is a simulated host interface to the ensemble, and provisions for loading data into or unloading data from the ensemble. Debug commands allow one to monitor communications and to examine the state of each computer in the ensemble. The simulation is, to the best of our knowledge, absolutely accurate on a clock cycle by clock cycle basis, in order to be able to replicate precisely any behavior of a Mosaic ensemble.

The cost of this precision is performance. The current implementation of the Mosaic simulator executes roughly 50 simulated Mosaic cycles per VAX11/780 cpu-second. At this rate the Mosaic A v1.2 self-test and bootstrap program, which moves a march pattern through memory, requires roughly 6 VAX11/780 cpu-minutes multiplied by the number of machines in the ensemble being simulated. Fortunately, for self-test and bootstrap programs we are interested in ensembles of one element.

We have been working on providing a concurrent simulation environment for Mosaic ensembles, intended for application programming development, and able to run either on the cosmic cubes or Intel iPSC computers. Our goal, now that the Mosaic processors are so well characterized, is to perform macroinstruction rather than microinstruction level simulation, and be able to simulate full size (*e.g.*, 1024 node) ensembles on the 128-node iPSC at no worse than 1,000 times slower than a hardware ensemble.

This new Mosaic simulator has been written in C. It currently models the Mosaic B processor instruction set and Mosaic C communications network. The node architecture being simulated has the following attributes:

- Mosaic B instruction set (without channel addressing modes).
- Special memory locations control node initialization, interrupt control, and I/O, as in the Mosaic C. Channel I/O is done entirely by DMA operations.

The communications network simulated is the Mosaic C wormhole routing mesh.

The simulator is written to run on concurrent hardware. Each node simulates a submesh of Mosaics. The same distributed event-driven simulation algorithm reported in section 3.4 is used to schedule the node simulations in order to avoid simulating Mosaics that are in a busy-wait state. The simulator operates in several modes. At one extreme, ***realistic*** mode models communication hardware faithfully, including blocking in the network, whereas in ***high-performance*** mode message order is preserved but message delays are arbitrary.

The simulator is not yet availible for general use.

2.2.3 Mosaic C Chip Design

The continued poor yield experience with the *n*MOS Mosaic A and the rapid progress in the design and superior performance of the CMOS Mosaic C have persuaded us to drop the Mosaic A design in favor of the Mosaic C.

Some additional details of the Mosaic C chip design are reported in section 4.1. This section is only a summary.

The target technology for the Mosaic C is MOSIS SCMOS with $0.6\mu m \leq \lambda \leq 1.5\mu m$. Target maximum chip size is $36mm^2$, or $100M\lambda^2$ with $\lambda = 0.6\mu m$, and $16M\lambda^2$ with $\lambda = 1.5\mu m$. Speed, storage size, and top-level floorplan will necessarily vary with feature size.

The Mosaic C is composed of 3 main parts: RAM & ROM, channels, and processor.

- *RAM & ROM* – Of the preliminary designs investigated by layouts of critical parts and circuit simulations, 2T and 4T fast dynamic RAMs emerged as the most promising. The 2T RAM cell, essentially two 1T cells written to complementary values and sensed differentially, can be implemented in about $400\lambda^2$ per bit including decode and sense circuits, and with only one contact per bit in the RAM cell. As a backup, since the sense circuitry introduces some risk of having to go through several fabrication iterations on the 2T design, we first laid out a safer 4T RAM that requires about $600\lambda^2$ per bit. The ROM is straightforward and requires less than $100\lambda^2$ per bit.

- *Channels* – The channels section of the Mosaic C designed in the spring 1986 was abandoned because of its size and performance relative to a new organization based on "routing automata" (see section 4.3). Routing automata channels were laid out over the summer 1986. A unidirectional channel requires 4 wires, two data, one control escape, and one in the reverse direction for flow control. A 2D mesh requires two unidirectional channels for each bidirectional channel, and channels in each of 4 directions, for a total of 32 wires. Except for power, clock, reset, and LED drivers, all of the I/O from a Mosaic is through these channels. The channels convey 2 bits on each clock period. Layout size is about $3M\lambda^2$.

- *Processor* – The Mosaic C processor is a 16-bit machine similar to the Mosaic B processor in layout and microcode style, except that the channels are implemented separately. Also, the Mosaic C processor is a "two-sequence" machine: It has two program counters and two overlapping sets of registers to be able to switch instantly between task and channel processing contexts. In addition to the two program counters, the processor has 5 other addressing registers that share the same incrementer: refresh counter, channel receive pointer and limit register, channel send pointer and limit register. Also like the Mosaic A and B, the Mosaic C is controlled with a single large PLA, which in the current design is 96 implicants. Layout size is about $5M\lambda^2$.

The processor, channels, ROM, and peripheral circuits on the Mosaic C consume about $10M\lambda^2$. The amount of space left for storage depends strongly on feature size under our assumption of not wanting to push the die size above about $36mm^2$. The area remaining for RAM is about $6M\lambda^2$ or 1.5K bytes at $3\mu m$ feature size with the 2T RAM, $26M\lambda^2$ or 6K bytes at $2\mu m$ feature size, and up to $90M\lambda^2$ or 24K bytes at $1.2\mu m$ feature size. The amount of 4T that will fit is about 2/3 of these amounts. The $3\mu m$ version is not interesting for system building, but only for chip design verification. The $2\mu m$ version even with 4K bytes of memory is more than adequate for system building and programming experiments.

2.3 Second Generation Cosmic Cubes*

Chuck Seitz, Alain Martin, Bill Athas, Bill Dally†, Charles Flaig, Steve Rabin, Craig Steele, Wen-King Su

Over the past three years we have developed a number of new ideas for second generation medium grain size message passing systems based on commercial processor chips and RAM, low-latency communication devices such as the Torus Routing Chip (TRC) [5208:TR:86], [5209:TR:86], and a reactive programming model [5196:TR:85], [5209:TR:86], [5232:TR:86]. Over the past eight months these ideas have come together into a coherent design, and in September 1986 we started simultaneous joint projects with two companies, Intel and Ametek, to build prototype second generation message-passing multicomputer systems. The companies are both prepared to provide all necessary parts, assembly, and logistical support for constructing the prototypes to our specifications and designs, and to work with us in developing the system software.

Intel and Ametek have non-exclusive licenses to patents on the Cosmic Cube architecture and message passing mechanisms, several later patents, and non-exclusive

* This segment of our research is sponsored jointly by DARPA and by grants from Intel Scientific Computers (Beaverton, Oregon) and Ametek Computer Research Division (Arcadia, California).

† At Caltech through June 1986; now Assistant Professor of EECS at MIT.

resale licenses for the Cosmic Cube system software, and ongoing work at Caltech. Both companies are manufacturers of first generation message-passing multicomputers based on the Cosmic Cube, and are licensed to produce the second generation machines commercially. Because of proprietary concerns, the schedules and certain details of the designs are not included in this report, but have been reported elsewhere to the DARPA management both by our project and by the companies.

We are already well underway with these projects. Machines of 64 nodes will be contributed to the project, and will be on the ARPAnet just as our Cosmic Cubes and iPSC are currently.

Since the two machines come from the same basic architectural and system software design, they will be sufficiently similar that it will be possible to compile the same programs to run on either machine. In fact, because of the formulation of the low level message send and receive primitives, the user can provide a library of message handling routines that allows these machines to execute programs written for *any* multicomputer systems.

The two machines differ in their processors, hosts, what accelerators will become available, details of their software implementations, programming language support, and application packages.

2.3.1 Hardware Characteristics

The general hardware characteristics of these machines is that the node processors are about 10 times faster than the first generation machines, but the message performance for non-localized message traffic is about 1000 times faster. The design is tuned to minimize latency for short messages. The message network uses a low dimension graph rather than the binary n-cube, and wormhole routing. The machines can be scaled over a wide range of N, but a 256-node system is the "centerline" design point.

The nodes are an unsymmetrical two-processor design with a general-purpose instruction processor in the 5-10 MIPS range for user processes, and a microcode-driven message interface processor. Except for the message interface, the characteristics of a node are expected to be similar to those of an advanced workstation of the same period. Scalar floating point performance is in the 1-2 Mflop range, and can be extended by vector accelerators to the 10-20 Mflop range. Minimum memory size per node will be 1 MB, but expandable to 16 MB or more. We expect that most commercial machines would have 4 MB per node. Thus the "centerline" 256-node system would have a peak performance of almost 2,000 MIPS, 400 scalar Mflops, and as much as 5 Gflops with vector accelerators and vectorization in the process code. Total storage size of the 256-node system with the typical 4 MB per node is 1 GB.

We have the encouragement of both companies to suggest standards that would not compromise either company's proprietary approaches, including:

- The format of the functions that invoke operating system services.
- The format of error reports and error recovery routines.
- Host functions for allocating machines and submachines in space sharing modes.
- "Open" hardware and software interfaces for accelerators and I/O.

The second generation machines will readily accommodate multiple hosts and disks.

Another of the new features we plan to include as an option in these machines is a distributed frame buffer. For example, a 16×16 node machine will be able to refresh a 1024×1024 pixel video image from 64×64 pixel frame buffers in each node. This distributed frame buffer can be used both to create some visibility into the activities of the whole machine – to "animate" algorithms in execution –, as well as to provide an extremely high performance graphics output from computations performed by the system.

2.3.2 Software

One of the aspects of the design that we can discuss fairly freely, as we hope that this user interface will become a standard for message-passing multicomputers, is the system software. The very small network component of the message latency makes it essential that we streamline the message handling in the nodes. The way in which this is accomplished is to use a "message-driven" or "reactive" node kernel.

All messages are sent and received from a heap maintained by the kernel, and accessed both by the main processor and the message interface processor. Messages are sent by allocating message space in the heap by:

```
p = xmalloc(len);
```

then sending the message built in that space by:

```
xsend(p,node,pid);
```

which also deallocates the space. That is, `xsend(p,...)` is like `free(p)` except that it also sends a message. Thus there is no need for the time-consuming "feedback," as for example via the message descriptor lock variable in the Cosmic Cube.

Messages are received by:

```
p = xrecv();
```

which is semantically like `malloc`, except that the length of the block allocated is determined by the length of the message received. Space thus allocated can later be freed after the message is used, either by the `xfree(p)` function, or by building another message in the space and freeing the space by the `xsend(p,...)` function.

Just as the `malloc` function in the Unix environment may return the NULL pointer if there is no space available, `xrecv` function will return the NULL pointer

if no message is present. However, xrecv returns the NULL pointer only if there are no messages in the node's entire receive queue. Calling xrecv when the next queued message is for another object (process) causes the kernel to save the state of this object and start running the other object. Thus the appearance of xrecv in the code marks "choice points" for switching execution to another object, and it is in this sense that the scheduling is "message-driven." It is still possible for an object to do work while waiting for a message, as in:

```
if (p = xrecv()) digest(p);
else do_other_work();
```

So long as an object (process) is making progress, one does not force a context switch. The blocking receive is expressed as a busy-wait loop:

```
while ( !(p = xrecv()) );
```

which has no inefficiency, since if the NULL pointer is returned the node does not have anything else to do anyway.

Although this set of low-level primitives encourages object-oriented programming styles, it is easy to express any other message primitives in terms of these. For example, we have written a library of functions that exactly duplicates the Cosmic Cube message primitives, which are much more complex in including a type mechanism and allowing a process to exercise discretion in the messages it receives.

The results of this formulation of the message primitives are a very fast and simple node kernel that is also compatible with all existing "cube" programs.

A substantial part of the software necessary for the second generation Cosmic Cubes has been derived from the Cosmic Cube software. A new version of the cosmic environment based on these new primitives has been written and will be used as the host runtime system. We are working on a portable kernel that will need only to be adapted for each model of the new hardware. We are also writing libraries of the host message and process spawning functions for each of the older types of cubes.

These reactive primitives, although illustrated here as called from C processes, are essentially identical to those used in Cantor's intermediate code. Cantor will be particularly efficiently supported on these second generation cubes.

One of the advantages of these projects that use commercial microprocessors in the nodes is the ability to exploit software tools – compilers, interpreters, debuggers – already developed for that particular microprocessor. Although these machines do not offer the ultimate in performance per cost, as a system such as Mosaic does, their large storage per node and extensive software tools result in the most highly evolved and advanced software development environments that will be available on message-passing concurrent computers. In supporting fine grain concurrency on

these machines, we exploit the leverage in the software development environments in a way that will also be applicable to fine grain machines.

Taken together, the computing and communication performance, scalability, open interfaces, I/O capability, new features, and system software of these second generation machines represent to us fulfilling an "IOU" – a working demonstration of the capabilities we have said would be possible to include in a well engineered message-passing concurrent computer.

2.3.3 Other Multicomputers

We have developed under DARPA sponsorship a base design and set of portable system software into which one can drop other processors, such as fast RISC processors, and hosts. Our licenses with Ametek and Intel are non-exclusive, and it is possible that in the future that we may form joint efforts with other companies to develop multicomputers based on other processors.

3. Concurrent Computation

3.1 Cantor

Bill Athas, Chuck Seitz

The programming notation Cantor represents the confluence of our ideas about programming idioms and compilation techniques for fine-grain message-passing concurrent computers. The results of this work have been twofold. The reactive model for object-oriented programming first presented in the XCPL report [5196:TR:85] and subsequently refined in the Cantor experiment, has been cast in the well-known programming language C to be used as the basis for the next generation of message driven kernels for the "Cube" machines. The Cosmic Kernel and its ilk are easily expressed in terms of the reactive or message-driven kernel.

Secondly, Cantor exists as a programming system comprised of a compiler, code generator, and interpreter for a variety of computers, both sequential and concurrent. Cantor is a lexically scoped, dynamically typed, object-oriented programming notation. The objects of Cantor share data and synchronize exclusively by message passing. The delivery of messages between objects and the assignment of objects to nodes is handled jointly by the Cantor compiler and the interpreter. The programmer does not need to know the topology or number of nodes in the computer on which a Cantor program is executed.

The Cantor compiler is a recursive-descent compiler written in C, and is portable to any machine that hosts a C compiler. The code generator is also written in C and is a table-driven program. The interpreters exist in a variety of forms. For concurrent computers, the interpreter is partitioned into two parts, the node part and the host part. Both parts run under the auspices of the cosmic environment. Each node of the Intel iPSC or the Cosmic Cube hosts a node interpreter as one process. The loading and initialization of the the node interpreters is handled by the host interpreter, which is also a process. Managing I/O between the cube and user is also handled by the host interpreter.

For sequential computers there is a version of the interpreter for debugging purposes. There is also an interpreter designed to be coupled to the statistical analysis tool S. This interpreter is used for profiling Cantor programs under ideal conditions, to measure quantities such as the amount of concurrency present, the amount of message traffic present, the average proximity between objects, etc. These quantities are measured as a function of discrete time quanta called sweeps. Although the ideal conditions are most likely not representative of actual conditions, if a program performs poorly under profiling, then almost assuredly, it will perform poorly on an actual concurrent machine. The profiler is also useful for evaluating different strategies for assigning objects to processors.

A variety of applications programs have been written in Cantor including: a Gaussian elimination program, an N-Queens program, two prime sieves, a circuit

simulator, a finite field package, the game of Life, optimal ordering for matrix multiplication (dynamic programming), and Floyd's all points shortest path computation.

Plans for Cantor include a native code generator for a variety of 32-bit commercial microprocessors as well as generating native code for Mosaic C. A flow analysis program for Cantor to perform future flow, as described in our previous semiannual technical report [5220:TR:86], is also in preparation. Currently the compiler is able to "assist" the interpreter in placing objects by supplying pertinent information about the behavior of new objects to the interpreter. This information includes the maximum number of new objects the new object could in turn potentially create, called the New Factor (NF) , and the number of messages the new object could potentially send, called the Send Factor (SF). Both the Send and New Factor are measured on a per single message received basis. Simple heuristics based upon SF and NF can appreciably increase the locality between communication objects with only negligible impact on the load balancing.

The next step is to have the compiler "direct" placement rather than merely assist. The problem of future flow is an instance of value or constant propagation used by optimizing compilers. If the types of values used in a program are constant, then the result is type inferencing. Type inferencing is useful in Cantor since values can be type checked once at compile time, thus removing unnecessary type checking at runtime. If the values used in a program are constant, then any expression consisting completely of constants can be evaluated at compile time and replaced by a constant.

Expressions in Cantor can evaluate to new objects, which are called "futures". Propagating reference values at compile time is future flow.

3.2 The Sync Model for Parallel Execution of Logic Programming

Pey-yun Peggy Li, Alain J. Martin

The Sync Model is a multiple solution data driven model which realizes AND-parallelism and OR-parallelism in a logic program assuming a message-passing multicomputer system. Our initial studies of this model are now complete and reported in "A Parallel Execution Model for Logic Programming" [5227:TR:86].

A paper about the Sync Model is included as an appendix to this report.

3.3 Concurrent Simulated Annealing

Craig S. Steele, Chuck Seitz

The process placement optimization system previously reported is being implemented in a concurrent version for the Intel iPSC. A preliminary sequential version of the event-driven programming environment is presently running. Current work

is extending the system to support concurrency and data distributed among several processing nodes.

A simulation of the concurrent optimization has demonstrated remarkable insensitivity to database inconsistency. The data upon which process relocation decisions are based are local to the node upon which the process resides. The local record of the positions of distant but communicating processes may become "stale" as those processes move. Concurrent optimization coupled with communications system latency induces some error in cost calculations. Another deleterious effect of concurrency is the possibility of conflicting simultaneous attempts to move a particular process to other nodes. For a given size task, both problems are accentuated by increasing the number of nodes concurrently attempting optimization. With increasing concurrency, some threshold will be reached where the errors arising from data staleness and the inefficiency due to aborted conflicting moves will preclude any optimization.

Both effects have been simulated sequentially by delaying the actual application of process relocation decisions for some time after the decision is made. By the time the action is taken, the environment may have changed significantly due to previously queued relocations. Typical problems do not show significant deterioration in convergence rate until the simulated concurrency exceeds one optimizing node per process being optimized. Since any real application must have at least one process per node to fully utilize the hardware, this result is very favorable to the development of a concurrent programming environment which dynamically optimizes its own placement at runtime.

3.4 Object-oriented event-driven simulation

Wen-King Su, Chuck Seitz

Since March 1986, a canonical model of event driven simulation has been formulated. The model includes a set of reactive processes that represent the elements in a simulation, and a set of messages that represent the influences exerted by the elements. Though it remains to be proven that all event driven simulators can be derived from this model, we have derived all the simulators that we have studied so far, including those based on the fundamental algorithm developed by Bryant, and independently by Chandy and Misra, as well as the "time warp" scheme developed by Jefferson.

The condition for ensuring progress of the canonical simulator has been proven. Though specialized simulators may have a more relaxed progress condition, satisfying the progress condition of the canonical simulator is sufficient to ensure the progress of the simulator.

The model has a intimate relationship with object-oriented reactive programming. It was this study and the study done by Bill Athas on the Cantor language

that prompted us to re-think our concurrent programming environment, and which resulted in the message primitives described in section 2.3.2. Amongst other advantages, this formulation of the primitives allows an indefinite lazy evaluation strategy that still assures progress, typically expressed as:

```
if (p = xrecv()) simulate_and_optionally_send_outputs(p);
else send_accumulated_outputs();
```

A simulation package originally written to evaluate the performance of alternative network topologies and routing schemes for the second generation cubes is developing into a more general package for a broad class of discrete event simulations.

3.5 Adaptive Routing in Fine Grain Multicomputers

John Y. Ngai, Chuck Seitz

Our research continues the investigation of system architectures to support general purpose computing in massively concurrent machines. As was pointed out in previous reports, for progressively finer grain message passing concurrent machines, the performance of the underlying communication network becomes increasingly significant in determining the overall performance of the machines. In the extreme, the sending and receiving of messages *is* the computation, while the results of the computation are the corresponding *side effects* of message passing.

Our current investigation focuses in experimenting with strategies that enhance and sustain the performance of communication networks in moderate to heavy message loading situations. The routing networks used in the Mosaic and second generation cubes have now reached a limit of being as fast as possible for a given bisection while using deterministic (oblivious) routing. Any improvements on these networks will require an *adaptive* utilization of available network bandwidth to prevent buildup of local congestion (hot spots). Message routes are no longer deterministic, but are continuously perturbed by local message loading. Messages will tend to follow their optimal routes to destinations in light traffic loading but adaptively detour to longer but less loaded routes as local hot spots spring up. This approach is in contrast to *randomized* approaches suggested in the literature, and which completely destroy any message locality present in the original process placements.

Current work involves :

- The derivation of an analytic framework for the understanding of these adaptive routing strategies.
- Investigation of several related variants such as *cut-through* and adaptive *fragmentation* schemes.

- Measuring performance statistics through extensive simulation.
- Derivation and simplification of automata for the hardware implementation of these schemes.

In the future, we are also interested in investigating adaptive routing strategies to support efficent message delivery in arbitrary networks. These schemes would then help our understanding in devising routing schemes to handle message forwarding in faulty networks.

4. VLSI Design

4.1 Mosaic Elements

Bill Athas, Charles Flaig, Fritz Nordby, Steve Rabin, Steve Roskowski, Don Speck, Wen-King Su, Chuck Seitz

The overall organization of the Mosaic C chip was described in section 2.2. Here we fill in some details on our final attempts with the Mosaic A chips, and the various sections of the Mosaic C. The parts of the Mosaic C are being switch simulated and fabricated separately for design verification before assembling them into a single chip.

4.1.1 Mosaic A

The yield and voltage sensitivity problems with the *n*MOS Mosaic A reported in our last report still remain. We've determined that the voltage sensitivity occurs even on chips without RAM, and is specific to the amplitude of one clock phase. That phase is used in few places outside of the RAM, none of which should be intolerant of large variations in clock voltage swings. With MOSIS phasing out *n*MOS, and with the Mosaic C design being far along and a better system chip, it has become a moot point.

4.1.2 Mosaic C RAM

The RAM represents about 70% of the area of the 2μm version of the Mosaic C, and about 85% of the area of the 1.2μm version. The RAM is also determines the speed of the chip. Thus we have made the most concerted effort to minimize size and delay in the RAM.

The basic strategy is to develop a 4T dRAM that is a low risk design with a relatively large area, and a 2T dRAM that is a higher risk design but a relatively small area.

Of the 3 dRAM designs attempted by the VLSI class last year, the 4T design was the easiest, and gave us some usable layout, primarily the data bus interface. Addition of addressing logic and the test interface completed the layout. Verification began with visual inspection, leading to correction of myriad electrical problems, such as lack of substrate contacts, and progressing to MOSSIM, with the 35,000-transistor size exposing weaknesses in our tool suite at every step. It is nearly ready for first fabrication. Because it was designed by committee, the area on this version is larger than we predicted, about $800\lambda^2$ per bit.

Meanwhile, attempts to locate the voltage-swing sensitivity in Mosaic A pointed out a capacitive coupling problem in 3T dRAM cells that was exacerbated in double-metal CMOS. Reducing the coupling in the CMOS design entailed a large area increase. This destroyed the appeal of 3T dRAM, so we've abandoned that design.

It had other disadvantages, such as tiny stored charges, large amounts of peripheral circuitry, and timing differences from the 4T and 2T designs.

This leaves the 2T design as our only alternative for denser storage, which is convenient because the 2T dRAM is nearly compatible with the 4T dRAM, the main difference being the need for sense amplifiers. The part that gives us the most trouble is generating a sense ramp that must maintain the right rate and shape over a reasonably wide range of supply voltage. Our current design for the sense ramp generators works over only a narrow range, so must be improved.

The 2T dRAM is a variation of the popular 1T dRAM that we do not think we could build reliably in a logic process. The 2T dRAM cell is just two 1T cells with a differential pair of storage capacitors per bit, eliminating the need for a dummy cell and providing a more reliable 0.2 volt difference between the bit lines. It also lends itself to folded bit lines, resulting in easier I/O and a convenient layout pitch. Thus while the 2T dRAM is not the ultimate in density, at $\approx 450\lambda^2$ per bit, it is much easier to make work in a logic process than the 1T dRAM.

4.1.3 Channels

After abandoning the router design for the Mosaic C which was generated by the VLSI design class, work was started on an improved version based on routing automata in which the header represents offsets in a prefix encoding that spans multiple flits. This approach lowers the latency in each node, and also the area of the routing circuits.

The new channels design is synchronous, with a minimum of 4 wires in each direction between nodes. Two wires are used for data, one for control, and one for an acknowledge signal. The control wire is used to specify the data lines as containing a digit, or encoded signals for a null flit, the tail of a message, or turn instructions. The number of data bits per flit, and the number of dimensions of the mesh, can easily be expanded.

Each dimension of the router is made up of three parallel data paths, one for data from the + direction, one for data from the - direction, and a cross-dimension path for data from the previous dimension. Each data path has a controller and feeds into a switch matrix that can switch data flow to any of the matching output data paths. If the first flit is a turn control flit, a request is generated to switch data flow to one of the +/- output paths or to the path leading to the next higher dimension. In the +/- data paths, if the first flit is a digit it is decremented and a request is generated to switch data flow to the opposite output path. Input data continues to flow through these connections until the arrival of a tail flit, at which time the connection is closed.

The subcells of a complete router have been laid out and individually tested. They have also been composed into a minimum width 1-dimensional router in preparation for a fabrication test run. This composed layout is currently being checked

for errors and will proceed to fabrication after a it passes simulation.

The channels will be connected to the rest of the Mosaic C via a pair of simple finite state machines that make use of a simple cycle-stealing DMA scheme to keep up with the high data transfer rate of the channels. The first of these FSMs accepts 16-bit words from memory, converts them into the flit format required by the channels, and feeds them into the cross-dimension routing path of the first dimension of the router. The second FSM accepts the flits which emerge from the last dimension of the router and converts them into 16-bit words for transfer to memory. These FSMs have been specified but not yet designed.

4.1.4 Microcode

The microcode for Mosaic C implements an instruction set closely resembling the Mosaic B instruction set, however, the Mosaic C microcode supports the two-sequence processor architecture. The Mosaic C microcode also permits memory-mapped "devices" to access the address generation logic section of the the Mosaic C datapath during microcycles where the microcontroller is unable to use the storage cycle. The memory-mapped devices include the refresh counter and the channels interface.

The data registers of Mosaic C are split between 8 data registers that are common to both the system and user context, and a set of 8 data registers private to each context. The address registers of Mosaic C, contained in the address generation logic section, includes a program counter for each context (PC0 and PC1), a refresh address register (RA), a send message pointer (SMP) and a receive message pointer (RMP), and a send message end (SME) and a receive message end (RME) . The register pairs, (SMP, SME) and (RMP, RME) are specific to the channels interface. These registers are accessed from the instruction set by using a new source and destination mode for the move instructions called CONTROL.

The register pairs (SMP, SME) and (RMP, RME) are loaded under program control but incremented and tested inside of the address generation logic section under channel interface control. When the microcode is unable to use a storage cycle, control of the address generation logic section and the address bus is released to one of the memory-mapped devices. The device, *eg,* the channels interface, can perform a memory read or write, increment a pointer, and compare the new pointer value to one of the message end registers. If the end of a message has been released, an interrupt is generated to place the instruction processor into system context.

This scheme solves the problem of arbitrating the use of the address bus and allows the refresh counter and channels interface to share the program counter incrementer.

An instruction-level simulator for the Mosaic C microcode and datapath has been written. This simulator will soon be used to program the message-driven kernel for Mosaic.

4.1.5 Datapath

Major pieces of the of the Mosaic C datapath were laid out as a VLSI design class project before June. Since then, the datapath has gone through several redesigns. The address generation part of the data channels has been moved into the datapath as a set of DMA control registers. Arbitration of memory access is simplified because the datapath is the only source of memory address. Bus arbitration has been moved into the microcode engine which will now handle requests from the channels, the processor, and the refresh counter.

To accomodate this change, the address section of the datapath has been expanded and generalized. It now includes two pairs of start/stop address registers for the channels, one for the output DMA, and one for the input DMA. A read request from the channel causes the microcode engine to interleave a memory cycle at a point in microcode execution when memory access is not required, then fetch a word from memory and stuff it into the channel, and update the associated address registers. A write request behaves similarly, but would move a word from channel to memory instead. Since the PC is not being used at the time DMA occurs, the incrementer for PC is used to increment the start address register. In order to improve the robustness of the first silicon, efforts are made to recycle similar pieces through out the datapath at a small expense of area.

All but a few "glue" pieces of this new datapath has been completed and assembled together, and have been shown to be correct with the help of MOSSIM.

4.2 Compiling Programs into Self-Timed Circuits

Steve Burns, Peter Hazewindus, Alain J. Martin

Our main task during the past half year has been to experiment with the method we have developed for compiling CSP-like programs into self-timed VLSI circuits. The collection of algorithms that we have compiled into self-timed circuits now covers the whole spectrum of applications that one could possibly envisage for the method. No fundamental flaw or limitation of the method has been uncovered by these experiments.

However, before addressing the issue of automatic compilation, *ie*, before embarking on writing a compiler, we still have to solve two main problems. The first and most difficult one is related to the introduction of so-called "state variables." In all but very simple programs, the variables provided by the original CSP program and by the handshaking expansions of communication actions are not sufficient to identify certain states of the system. In such cases, extra variables called state variables have to be introduced. So far, the technique we have been using for the introduction of state variables still relies on the insight of the designer and may require a certain amount of backtracking. We are now looking for more systematic

algorithms for the introduction of those variables. As the reader suspects, this problem is strongly related to the state assignment problem in the theory of sequential machines.

The second problem is related to the last phase of the compilation *ie,* the identification of the program with gates and the implementation of the gates in VLSI. So far the method works as follows: once the program has been compiled into a "production rule set," we try to identify sets – mostly pairs – of production rules with the sets of production rules that define our basic repertoire of gates. When the set of production rules that controle a certain variable cannot be directly identified with the rules of any gate, we have to further massage the set of rules to make it fit. In particular, this massaging may consist of decomposing the rules into more elementary ones, by introducing extra variables. This is a step that we want to eliminate in an automatic compilation because it involves backtracking. At the moment, we think that we can easily eliminate this step if we don't restrict ourselves to a fixed set of gates. After all, one of the main advantages of VLSI is that we can create new operators at no cost. We can therefore take any set of production rules that control a certain set of output variables and implement it directly rather than in terms of more elementary gates.

The above refinement will require that we also refine our assembly and layout programs. So far the chips that we have fabricated have been assembled and laid out using a "home-made" assembler, designed by students as a project in Chuck Seitz's VLSI class. This assembler uses a standard cell library of self-timed elements in SCMOS technology. We have also started experimenting with using the MOSIS 'fusion' service in cooperation with Ron Ayres at ISI. In both cases, the refinement mentioned above will require from the assembler that it is able to assemble generic gates, that we call 'generalized C-elements'.

Once these two problems have been solved, the construction of an automatic compiler can be confidently considered.

4.3 Routing Automata

Chuck Seitz, Alain J. Martin

The switching elements of the brain, neurons, have large numbers of input and output connections, but employ simple signaling. VLSI switching elements, by contrast, tend to be limited in "fan-in" and "fan-out", but can interpret complex signals. If one decides to build complex VLSI networks in a way that exploits their strong point, one is led to the idea of a set of switching elements that we refer to as "routing automata."

A typical routing automaton accepts serial streams with prefix headers of high information density, such as the packets in message-passing concurrent computers, and produces streams that can be interpreted in the same way. For the purposes of

describing the way in which the message handling and routing circuits for message-passing concurrent computers can be developed from routing automata, let us illustrate this organizational approach with the same 2-dimensional bidirectional mesh network that we use in the Mosaic. The mesh network is acyclic in each direction and dimension, so a simple dimension order routing function – first x then y – will suffice to provide deadlock-free routing without recourse to virtual channels. However, one can employ the same organizational approach for k-ary n-cube (torus) networks, in which there is a cycle in each dimension, by using two virtual channels per physical channel.

We can assume for the time being that the packet consists of a sequence of flow control units, called *flits*, in which one can specify Δx in the first flit, Δy in the second flit, followed by message flits, the last of which is marked at the *tail* flit. Because this network is bidirectional, the Δx and Δy specification are signed numbers.

Because the routing proceeds first in x and then in y, one can compose a 2D routing chip out of two cascaded 1D routing automata. The first 1D routing automaton accepts a packet from its node computer. If the Δx read in the first flit is non-zero, it decrements the value and sends it in the $+x$ or -x direction according to the sign. If the header value is 0, the Δx part of the header is stripped and the remainder of the message is passed along to the y routing automaton, which operates identically. A message entering the x routing automaton at, for example, the $+x$ input is processed similarly. If the Δx value in the first flit is 0, the Δx flit is stripped and the remainer of the message is passed to the y routing automaton. If the Δx value is non-zero, it is decremented as the message is sent out the $+x$ output.

This structure would be nothing more than a nice organization for making routing chips but for the observation that the 1D routing automaton can in turn be composed of more elementary routing automata. One can build the 1D routing automaton out of two types of elementary routing automata that we refer to as decision (D) and merge (M). The D elements are typically required in several forms. The D and M elements are duals that can be thought of as similar to railroad switches. The M element is 2-input, 1-output, and passes entire packets from its inputs to outputs on a first-come first-serve basis. The flits of a packet can be thought of like coupled train cars, and only after the tail flit (caboose) of a packet has cleared the M element can another full packet be admitted from an input. The D element is 1-input, 2-output, and the most essential type for building routing networks processes the first flit of packet header as described above, stripping the flit and sending the packet to the side if the first flit is 0, and decrementing the first flit and passing the packet straight if the first flit is non-zero. A D_s (sign decision) is another 1-input 2-output elementary routing automaton that simply sends its input message to one of its two outputs according to the sign of the relative address in the first flit, and is needed to make routers for bidirectional networks.

The implementation of the elementary routing automata can be either synchronous or self-timed. There appears to be no limit to the variety of routing structures – including structures with properties of fault-tolerance, freedom from deadlock, and ability to perform message broadcast – that can be implemented as systems composed of elementary routing automata, each of which has only a few inputs and outputs.

One of the interesting properties of these networks is that the complex signals *at every point* have an identical sequential interpretation.

In addition to variations in the internal structure of these routing systems, one can employ different conventions for the sequential interpretation of signals. For example, in multicomputer communication networks, one might use different formats for header information.

The form of header with Δx in the first flit and Δy in the second flit requires that the number of bits communicated in a single flit be adequate to represent the maximal address difference between nodes in each dimension. Suppose one had a relatively large network, such as a 64 by 64 mesh of 4096 nodes, in which the communication bandwidth requirements between nodes required only 2 data bits to be sent with each flit. However, to be able to address any node it would be necessary to employ a flit of $\log_2 64 = 6$ bits for Δx or Δy, plus a sign bit. However, one may extend the representation of Δx and Δy (and additional dimensions) over several flits with the following technique.

Consider the case of 2 data bits, D0 and D1. These two bits are able to represent a radix-4 integer. An additional signal is required to distinguish "punctuation" symbols in the message stream, including the tail. Thus a minimum of 3 signals is required, for which we shall use the following of many possible encodings:

C	D0	D1	meaning	
0	0	0	blank	(represented as ".")
0	0	1	plus	(represented as "+")
0	1	0	minus	(represented as "-")
0	1	1	unused	
1	0	0	data 0	
1	0	1	data 1	
1	1	0	data 2	
1	1	1	data 3	

In addition, it is necessary to have one flow control signal communicated in the reverse direction of the channel in order to force transmission of data to pause in the event that the head of the message has become blocked.

A message that is to be conveyed by +5 in the x dimension and -2 in the y dimension is then encoded with the following string of characters from this alphabet:

```
...+0123+2-11+...
```

The message should be visualized as traveling from left to right, with its header on the right. It is surrounded by blank characters, and could also have blank characters intermixed inside the message as may occur for timing reasons.

The initial "+" indicates that the message is to travel in the $+x$ direction. The following "11" is the radix-4 representation for 5, as the message is to traverse 5 channels in the $+x$ direction. The following "-" indicates that the message is to travel next in the -y direction. The following "2" is the distance in the -y direction, in radix-4 notation. The following "+" indicates the beginning of the data part of the message. The following "0..3" is radix-4 data corresponding to the 8-bit binary message 00011011. The following "+" indicates the tail of the message.

What is remarkable about this encoding is that by processing overlapping pairs of symbols from this alphabet, a routing automaton is both able to decrement the remaining distance and detect when the remaining distance has reached 0. For example, when this packet enters the x routing automaton, it is sent out the $+x$ channel, and has the following representation on successive channels:

```
...+0123+2-11+...   entering 1st router
...+0123+2-10...    leaving +x output of 1st router
...+0123+2-3...     leaving +x output of 2nd router
...+0123+2-2...     leaving +x output of 3rd router
...+0123+2-1...     leaving +x output of 4th router
...+0123+2-0...     leaving +x output of 5th router
...+0123+1...       leaving -y output of 6th router
...+0123+0...       leaving -y output of 7th router
...+0123...         entering node at 8th router
```

There are many variations on this basic scheme, one of which is used in the Mosaic C routing system.

We are at the stage in this work where we have interesting examples of routing systems constructed from routing automata, and a deep interest from our architecture experiments in developing more advanced routing systems. What is needed is a theory of and systematic methods for transforming routing specifications into routing structures. We believe that, in addition to their application to the communication networks of message-passing concurrent computers, this framework is suggestive of new approaches to the design of VLSI chips with multiple pipelines.

4.4 Fast Mesh Routing Chips

Charles Flaig, Chuck Seitz

Work has started on a next generation of self-timed routing chips to provide a message system for message-passing concurrent computers whose nodes are based

on commercially available processors and memory.

The internal structure will consist of many stages of ultra-fast self timed FIFO pipeline, interspersed with slightly slower switching and pipelined header processing circuits, which operate in the manner of the Mosaic C router (section 4.1.3) and routing automata (section 4.3). The largest delay will probably be encountered in synchronizing data at the outputs, driving the pads, and waiting for an acknowledge signal. The internal pipelining will help relieve congestion in the system without adding an overly large amount of latency. It is expected that the chip will have a flit rate at least twice that of the synchronous router in the Mosaic C (which uses the same clock as the rest of the chip) and will also have the advantage of not being sensitive to clock distribution.

The routing chip will be optimized for an 8-bit wide data path, which requires at the interface 3 more lines for control, request, and acknowledge, for a total of 11 wires going in each direction between nodes. The low cycle time and wide data path will require a very fast buffering system between the routing chip and the connected processor in order to avoid slowing down the system.

4.5 SCMOS Pad Designs

Fritz Nordby, Chuck Seitz

As chip geometry scales down, the requirements of pad structures change. In chips with small feature sizes, the sizes of the output drivers decrease until they are small compared to the input protection structures. Because of this, output pads, traditionally protected by the sheer size of their associated driver transistors, are found to be in as much danger from input transients as the circuits connected to input pads, and the amount of chip area devoted to input protection structures increases to the point of near-absurdity.

Investigations of protection structures to attack these problems of chip area and output protection have yielded a new strategy for input and output protection, and which promise to allow wider and more varied on-chip use of input signals. This new strategy is based on the idea that static protection cannot be fully realized on-chip: the magnitude of the problem is simply too large to allow it to be handled without devoting a large area to protection structures.

Thus, rather than use an input resistor to *dissipate* the energy of the transient, a pair of large clamping transistors is placed adjacent to the pad so that any input transient is clamped directly to the power rail and its energy is *kept out* of the chip. This clamping structure is used on input and output pads alike, thus providing more protection for the outputs than would be provided by their drivers alone without significantly increasing the size of the output pad or interfering with the action of the output drivers. Inputs are further protected by a resistor and a second, smaller, clamp structure. This allows the pair of input clamps to act as a current divider,

and greatly decreases the size of any transient; and this decrease in transient size allows inputs to be connected to diffusions internal to the chip, since the size of any transient that gets past the input structure can be kept below the 0.6 volts turn-on potential of the diffusion-substrate or diffusion-well diode.

These new pad structures are presently being layed out, and will soon be fabricated and tested.

4.6 Polygon Rendering Chip

Rajiv Gupta, Al Barr, Chuck Seitz

This chip is designed to render polygons at speeds of 8M polygons per second. The rendering process consists of z-depth buffering and Gouraud shading of polygons on a raster scan display. The future version of the design will also permit us to perform Bit-blt (creating copies of arbitrary portions of the screen with one command) operations.

The chips will have 18 pins with a die size of approximately $36mm^2$. The chip is organized with on-chip memory to reduce communication bottlenecks, and 24 processors per chip, each handling a 20 pixel sliver of the screen. As is apparent, a large number of such chips would be required to render large screens. Thus with minimal additional support circuitry we will be able to sub-pixel – the only way of mimicking anti-aliasing in z-depth buffer systems.

We decided to partition the processor into 6 parts to ease in testing. Of these, 4 have been fabricated in $3\mu m$ SCMOS and have passed tests at 20 Mhz. The others along with the pre-processor and the post-processor are in the process of fabrication and testing.

California Institute of Technology
Computer Science Department, 256-80
Pasadena CA 91125

Technical Reports
Autumn 1986
Available from the Computer Science Department Library
Prices include postage and help to defray our printing and mailing costs.

Publication Order Form

If you wish to order any of the reports listed, complete this form and return it with your check or international money order (in U.S. dollars) payable to CALTECH. Prepayment is required for all materials.

____5231:TR:86 $2.00 *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*
Dally, Wm. J. and Charles L. Seitz

____5230:TR:86 $24.00 *Monte Carlo Methods for 2-D Compaction*, PhD Thesis
Mosteller, R.C.

____5229:TR:86 $4.00 *anaLOG - A Functional Simulator for VLSI Neural Systems*, MS Thesis
Lazzaro, John

____5227:TR:86 $18.00 *Parallel Execution Model for Logic Programming*, PhD Thesis
Li, Pey-yun Peggy

____5221:TR:86 $3.00 *Sync Model: A Parallel Execution Method for Logic Programming*
Li, Pey-yun Peggy and Alain J. Martin

____5220:TR:86 $4.00 *Submicron Systems Architecture*
ARPA Semiannual Technical Report

____5215:TR:86 $2.00 *How to Get a Large Natural Language System into a Personal Computer*,
Thompson, Bozena H. and Frederick B. Thompson

____5214:TR:86 $2.00 *ASK is Transportable in Half a Dozen Ways*,
Thompson, Bozena H. and Frederick B. Thompson

____5212:TR:86 $2.00 *On Seitz' Arbiter*,
Martin, Alain J

____5211:TR:86 $3.00 *Self-timed FIFO: An exercise in Compiling Programs into VLSI Circuits*
Martin, Alain J

____5210:TR:86 $2.00 *Compiling Communicating Processes into Delay-Insensitive VLSI Circuits*,
Martin, Alain

____5209:TR:86 $11.00 *VLSI Architecture for Concurrent Data Structures*, PhD Thesis,
Dally, William J.

____5208:TR:86 $2.00 *The Torus Routing Chip*,
Dally, William and Charles L Seitz

____5207:TR:86 $2.00 *Complete and Infinite Traces: A Descriptive Model of Computing Agents*,
van Horn, Kevin

____5205:TR:85 $2.00 *Two Theorems on Time Bounded Kolmogrov-Chaitin Complexity*,
Schweizer, David and Yaser Abu-Mostafa

____5204:TR:85 $3.00 *An Inverse Limit Construction of a Domain of Infinite Lists*,
Choo, Young-Il

____5203:TR:85 $9.00 *C Programmer's Guide to the Cosmic Cube*,
Su, Wen-King, Reese Faucette and Chuck Seitz

____5202:TR:85 $15.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5200:TR:85 $18.00 *ANIMAC: A Multiprocessor Architecture for Real-Time Computer Animation*, PhD thesis
Whelan, Dan

____5198:TR:85 $8.00 *Neural Networks, Pattern Recognition and Fingerprint Hallucination*, PhD thesis
Mjolsness, Eric

____5197:TR:85 $7.00 *Sequential Threshold Circuits*, MS thesis
Platt, John

____5196:TR:85 $5.00 *ECL: An Experimental Concurrent Language*,
Athas, Bill

____5195:TR:85 $3.00 *New Generalization of Dekker's Algorithm for Mutual Exclusion*,
Martin, Alain J

____5194:TR:85 $5.00 *Sneptree - A Versatile Interconnection Network*,
Li, Pey-yun Peggy and Alain J Martin

____5193:TR:85 $2.00 *Delay-insensitive Fair Arbiter*
Martin, Alain J

____5190:TR:85 $3.00 *Concurrency Algebra and Petri Nets*,
Choo, Young-il

____5189:TR:85 $10.00 *Hierarchical Composition of VLSI Circuits*, PhD Thesis
Whitney, Telle

____5185:TR:85 $11.00 *Combining Computation with Geometry*, PhD Thesis
Lien, Sheue-Ling

____5184:TR:85 $7.00 *Placement of Communicating Processes on Multiprocessor Networks*, Ms Thesis
Steele, Craig

____5179:TR:85 $3.00 *Sampling Deformed, Intersecting Surfaces with Quadtrees*, MS Thesis,
Von Herzen, Brian P.

____5178:TR:85 $9.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5177:TR:85 $4.00 *Hot-Clock nMOS*, Proc. 1985 Chapel Hill Conference on VLSI, pp 1-17
Seitz, Charles, A H Frey, S Mattisson, S D Rabin, D A Speck, and J L A van de Snepscheut

____5174:TR:85 $7.00 *Balanced Cube: A Concurrent Data Structure*,
Dally, William J and Charles L Seitz

____5172:TR:85 $6.00 *Combined Logical and Functional Programming Language*,
Newton, Michael

____5168:TR:84 $4.00 *Object Oriented Architecture*,
Dally, Bill and Jim Kajiya

____5165:TR:84 $4.00 *Customizing One's Own Interface Using English as Primary Language*,
Thompson, B H and Frederick B Thompson

____5164:TR:84 $13.00 *ASK French - A French Natural Language Syntax*, MS Thesis
Sanouillet, Remy

____5160:TR:84 $7.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5158:TR:84 $6.00 *VLSI Architecture for Sound Synthesis*,
Wawrzynek, John and Carver Mead

____5157:TR:84 $15.00 *Bit-Serial Reed-Solomon Decoders in VLSI*, PhD Thesis
Whiting, Douglas

____5148:TR:84 $4.00 *Fair Mutual Exclusion with Unfair P and V Operations*,
Martin, Alain and Jerry Burch

____5147:TR:84 $4.00 *Networks of Machines for Distributed Recursive Computations*,
Martin, Alain and Jan van de Snepscheut

____5143:TR:84 $5.00 *General Interconnect Problem*, MS Thesis
Ngai, John

____5140:TR:84 $5.00 *Hierarchy of Graph Isomorphism Testing*, MS Thesis
Chen, Wen-Chi

____5139:TR:84 $4.00 *HEX: A Hierarchical Circuit Extractor*, MS Thesis
Oyang, Yen-Jen

____5137:TR:84 $7.00 *Dialogue Designing Dialogue System*, PhD Thesis
Ho, Tai-Ping

____5136:TR:84 $5.00 *Heterogeneous Data Base Access*, PhD Thesis
Papachristidis, Alex

____5135:TR:84 $7.00 *Toward Concurrent Arithmetic*, MS Thesis
Chiang, Chao-Lin

____5134:TR:84 $2.00 *Using Logic Programming for Compiling APL*, MS Thesis
Derby, Howard

____5133:TR:84 $13.00 *Hierarchical Timing Simulation Model for Digital Integrated Circuits and Systems*, PhD Thesis
Lin, Tzu-mu

____5132:TR:84 $10.00 *Switch Level Fault Simulation of MOS Digital Circuits*, MS Thesis
Schuster, Mike

____5130:TR:84 $3.00 *LOG The Chipmunk Logic Simulator User's Guide*,
Gillespie, Dave

____5129:TR:84 $5.00 *Design of the MOSAIC Processor*, MS Thesis
Lutz, Chris

____5128:TM:84 $3.00 *Linguistic Analysis of Natural Language Communication with Computers*,
Thompson, Bozena H

____5125:TR:84 $6.00 *Supermesh*, MS Thesis
Su, Wen-king

____5124:TR:84 $4.00 *Probe: An Addition to Communication Primitives*,
Martin, Alain

____5123:TR:84 $14.00 *Mossim Simulation Engine Architecture and Design*,
Dally, Bill

____5122:TR:84 $8.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5120:TM:84 $1.00 *Mathematical Approach to Modeling the Flow*,
Johnsson, Lennart and Danny Cohen

____5119:TM:84 $1.00 *Integrative Approach to Engineering Data and Automatic Project Coordination*,
Segal, Richard

____5118:TR:84 $2.00 *SMART User's Guide*,
Ngai, John

____5114:TM:84 $3.00 *ASK As Window to the World*,
Thompson, Bozena, and Fred Thompson

____5113:TR:84 $4.00 *WoLery*,
Mead, Carver A

____5112:TR:83 $22.00 *Parallel Machines for Computer Graphics*, PhD Thesis
Ulner, Michael

____5106:TM:83 $1.00 *Ray Tracing Parametric Patches*,
Kajiya, James T

____5105:TR:83 $2.00 *Memory Management in the Programming Language ICL*,
Wawrzynek, John

____5104:TR:83 $9.00 *Graph Model and the Embedding of MOS Circuits*, MS Thesis
Ng, Tak-Kwong

____5103:TR:83 $7.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5102:TR:83 $2.00 *Experiments with VLSI Ensemble Machines*,
Seitz, Charles L

____5101:TM:83 $1.00 *Concurrent Fault Simulation of MOS Digital Circuits*,
Bryant, Randal E

____5099:TM:83 $1.00 *VLSI and the Foundations of Computation*,
Mead, Carver

____5098:TM:83 $2.00 *New Techniques for Ray Tracing Procedurally Defined Objects*,
Kajiya, James T

____5097:TR:83 $4.00 *Design of a Self-timed Circuit for Distributed Mutual Exclusion*,
Martin, Alain J

____5094:TR:83 $2.00 *Stochastic Estimation of Channel Routing Track Demand*,
Ngai, John

____5093:TR:83 $1.00 *Design of the MOSAIC Element*,
Lutz, Chris, Steve Rabin, Chuck Seitz and Don Speck

____5092:TM:83 $2.00 *Residue Arithmetic and VLSI*,
Chiang, Chao-Lin and Lennart Johnsson

____5091:TR:83 $2.00 *Race Detection in MOS Circuits by Ternary Simulation*,
Bryant, Randal E

____5090:TR:83 $9.00 *Space-Time Algorithms: Semantics and Methodology*, PhD Thesis
Chen, Marina Chien-mei

____5089:TR:83 $10.00 *Signal Delay in General RC Networks with Application to Timing Simulation of Digital Integrated Circui*
Lin, Tzu-Mu and Carver A Mead

____5086:TR:83 $4.00 *VLSI Combinator Reduction Engine*, MS Thesis
Athas, William C Jr

____5084:TM:83 $3.00 *Tree Machine: An Evaluation of Strategies for Reducing Program Loading Time*,
Li, Pey-yun Peggy, and Lennart Johnsson

____5082:TR:83 $10.00 *Hardware Support for Advanced Data Management Systems*, PhD Thesis
Neches, Philip

____5081:TR:83 $4.00 *RTsim - A Register Transfer Simulator*, MS Thesis
Lam, Jimmy

____5080:TR:83 $4.00 *Distributed Mutual Exclusion on a Ring of Processes*,
Martin, Alain

____5079:TR:83 $2.00 *Highly Concurrent Algorithms for Solving Linear Systems of Equations*,
Johnsson, Lennart

____5078:TR:83 $5.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5075:TR:83 $2.00 *General Proof Rule for Procedures in Predicate Transformer Semantics*,
Martin, Alain

____5074:TR:83 $10.00 *Robust Sentence Analysis and Habitability*,
Trawick, David

____5073:TR:83 $12.00 *Automated Performance Optimization of Custom Integrated Circuits*, PhD Thesis
Trimberger, Steve

____5068:TM:83 $1.00 *Hierarchical Simulator Based on Formal Semantics*, Proc Third Caltech Conf on VLSI
Chen, Marina and Carver Mead

____5065:TR:82 $3.00 *Switch Level Model and Simulator for MOS Digital Systems*,
Bryant, Randal E

____5055:TR:82 $5.00 *FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems*, MS Thesis
Ng, Charles H

____5054:TM:82 $3.00 *Introducing ASK, A Simple Knowledgeable System*, Conf on App'l Natural Language Processing
Thompson, Bozena H and Frederick B Thompson

____5052:TR:82 $8.00 *Submicron Systems Architecture*,
ARPA Semiannual Technical Report

____5051:TM:82 $2.00 *Knowledgeable Contexts for User Interaction*, Proc Nat'l Computer Conference
Thompson, Bozena, Frederick B Thompson, and Tai-Ping Ho

____5047:TR:82 $3.00 *Torus: An Exercise in Constructing a Processing Surface*, Proc 2nd Caltech Conference on VLSI
Martin, Alain

____5046:TR:82 $3.00 *Axiomatic Definition of Synchronization Primitives*, Acta Informatica 16, pp 219-235 (1981)
Martin, Alain

____5045:TM:82 $3.00 *Distributed Implementation Method for Parallel Programming*, Proc Information Processing '80
Martin, Alain

____5044:TR:82 $10.00 *Hierarchical Nets: A Structured Petri Net Approach to Concurrency*,
Choo, Young-Il

____5038:TM:82 $4.00 *New Channel Routing Algorithm*,
Chan, Wan S

____5035:TR:82 $9.00 *Type Inference in a Declarationless, Object-Oriented Language*, MS Thesis
Holstege, Eric

____5034:TR:82 $12.00 *Hybrid Processing*, PhD Thesis
Carroll, Chris

____5033:TR:82 $4.00 *MOSSIM II: A Switch-Level Simulator for MOS LSI User's Manual*,
Schuster, Mike, Randal Bryant and Doug Whiting

____5029:TM:82 $4.00 *POOH User's Manual*,
Whitney, Telle

____5021:TR:82 $5.00 *Earl: An Integrated Circuit Design Language*, MS Thesis
Kingsley, Chris

____5018:TM:82 $2.00 *Filtering High Quality Text for Display on Raster Scan Devices*,
Kajiya, Jim and Mike Ullner

____5017:TM:82 $2.00 *Ray Tracing Parametric Patches*,
Kajiya, Jim

____5016:TR:82 $4.00 *Bristle Blocks - Scrutinized and Analyzed*,
McNair, Richard and Monroe Miller

____5015:TR:82 $15.00 *VLSI Computational Structures Applied to Fingerprint Image Analysis*,
Megdal, Barry

____5014:TR:82 $15.00 *Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture*, PhD Thesis
Lang, Charles R Jr

____5012:TM:82 $2.00 *Switch-Level Modeling of MOS Digital Circuits*,
Bryant, Randal

____5001:TR:82 $2.00 *Minimum Propagation Delays in VLSI* , IEEE J Solid State Circuits
Mead, Carver, and Martin Rem

____5000:TR:82 $6.00 *Self-Timed Chip Set for Multiprocessor Communication*, MS Thesis
Whiting, Douglas

____4777:TR:82 $7.00 *Techniques for Testing Integrated Circuits*, PhD Thesis
DeBenedictis, Erik P

____4724:TR:82 $2.00 *Concurrent, Asynchronous Garbage Collection Among Cooperating Processors*,
Lang, Charles R

____4716:TM:82 $4.00 *Rectangular Area Filling Display System Architecture*,
Whelan, Dan

____4684:TR:82 $3.00 *Characterization of Deadlock Free Resource Contentions*,
Chen, Marina, Martin Rem, and Ronald Graham

____4675:TR:81 $7.00 *Switching Dynamics*, MS Thesis
Lewis, Robert K

____4655:TR:81 $20.00 *Proc Second Caltech Conf on VLSI*,
Seitz, Charles, ed.

____4654:TR:81 $12.00 *Versatile Ethernet Interface*, MS Thesis
Whelan, Dan

____4653:TR:81 $10.00 *Toward A Theorem Proving Architecture*, MS Thesis
Lien, Sheue-Ling

____4618:TM:81 $5.00 *Tree Machine Operating System*,
Li, Peggy

____4600:TM:81 $3.00 *Notation for Designing Restoring Logic Circuitry*, Proc Second Caltech Conf on VLSI
Rem, Martin, and Carver Mead

____4530:TR:81 $20.00 *Silicon Compilation*, PhD Thesis
Johannsen, Dave

____4527:TR:81 $11.00 *Communicative Databases*, PhD Thesis
Yu, Kwang-I

____4521:TR:81 $8.00 *Lambda Logic*, MS Thesis
Rudin, Leonid

____4517:TR:81 $7.00 *Serial Log Machine*, MS Thesis
Li, Peggy

____4407:TM:82 $3.00 *Experimental Composition Tool*,
Mosteller, Richard C

____4332:TR:81 $3.00 *RLAP, Version 1.0, A Chip Assembly Tool*,
Mosteller, R

____4320:TR:81 $7.00 *Hierarchical Design Rule Checker*, MS Thesis
Whitney, Telle

____4317:TR:81 $10.00 *REST - A Leaf Cell Design System*, MS Thesis
Mosteller, Richard C

____4298:TR:81 $7.00 *From Geometry to Logic*, MS Thesis
Lin, Tzu-mu

____4204:TR:78 $8.00 *16-Bit LSI Digital Multiplier*, EE Thesis
Masumoto, R T

____4191:TR:81 $4.00 *Towards A Formal Treatment of VLSI Arrays*, Proc Second Caltech Conf on VLSI
Johnsson, Lennart S, Uri Weiser, D Cohen, and Alan L Davis

____4128:TM:81 $2.00 *Shifting to a Higher Gear in a Natural Language System*,
Thompson, Fred and B Thompson

____4116:TR:79 $25.00 *Toward A Mathematical Theory of Perception*, PhD Thesis
Kajiya, Jim

____3975:TM:80 $3.00 *Rapidly Extendable Natural Language*,
Thompson, B H and Fred B Thompson

____3762:TR:80 $8.00 *Software Design System*, PhD Thesis
Hess, Gideon

____3761:TR:80 $7.00 *Fault Tolerant Integrated Circuit Memory*, PhD Thesis
Barton, Tony

____3760:TR:80 $10.00 *Tree Machine: A Highly Concurrent Computing Environment*, PhD Thesis
Browning, Sally

____3759:TR:80 $10.00 *Homogeneous Machine*, PhD Thesis
Locanthi, Bart

____3710:TR:80 $10.00 *Understanding Hierarchical Design*, PhD Thesis
Rowson, James

____3364:TR:79 $8.00 *Stack Data Engine*,
Efland, G and R C Mosteller

____3340:TR:79 $26.00 *Proc. Caltech Conference on VLSI (1979)*,
Seitz, Charles, ed

____2276:TM:78 $12.00 *Language Processor and a Sample Language*,
Ayres, Ron

Please fill in your name, address and amount enclosed below:

name _____

Address _____

City _____ State _____ Zip _____ Country _____

Amount enclosed $ _____

_____ Please check here if you wish to be included on our mailing list

_____ Please check here for any change of address

Return this form to: Computer Science Library, 256-80, Caltech, Pasadena CA 91125

The Sync Model:

A Parallel Execution Method for Logic Programming

Pey-yun Peggy Li
and
Alain J. Martin
Computer Science
California Institute of Technology
Pasadena CA 91125

March 1986

5221:TR:86

Abstract

The Sync Model, a parallel execution method for logic programming, is proposed. The Sync Model is a multiple-solution data-driven model that realizes AND-parallelism and OR-parallelism in a logic program assuming a message-passing multiprocessor system. AND parallelism is implemented by constructing a dynamic data flow graph of the literals in the clause body with an ordering algorithm. OR parallelism is achieved by adding special Synchronization signals to the stream of partial solutions and synchronizing the multiple streams with a merge algorithm. The ordering algorithm and the merge algorithm are described. The merge algorithm is proved to be correct and therefore, the Sync Model is proved complete, i.e., the execution of a logic program under the Sync Model generates all the solutions.

The research described in this paper was sponsored by the Defense Advanced Research Projects Adency, ARPA Order No. 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

1 Introduction

One way to improve the efficiency in the execution of a logic program is to exploit the potential parallelism, namely AND parallelism and OR parallelism, inherent to the program. In this paper, a method – called the "Sync Model" – is proposed for the parallel execution of logic programs on a message-passing multiprocessor system. The method realized both AND-parallelism and OR-parallelism. OR parallelism – the parallel execution of all clauses that are unifiable with the goal – is easier to realize than AND parallelism because the executions of OR clauses are independent of each other. On a message-passing system, the synchronization of the multiple solutions generated by different processes is the major problem in the implementation of OR parallelism. AND parallelism—the parallel execution of AND literals in a clause body—may result in binding conflicts for a variable shared by several literals.

Constructing a data flow graph is the most common approach for AND parallelism. By allowing exactly one producer for each shared variable, binding conflicts can be eliminated. One problem in the data flow approach is that the data flow graph is changed dynamically according to the binding values transmitted within the graph. When a variable is bound to a partially instantiated term containing another variable, binding conflicts may occur. Therefore, the data flow graph needs to be modified to enforce the "one producer per variable" rule to the new variable. In most computation models for concurrent logic programming languages, the data flow graph of literals in a clause body is constructed by the programmer through variable annotations. Alternatively, the data flow graph can be constructed automatically by the system; either dynamically such as in Conery's AND/OR process model [2] or statically such as in Chang and DeGroot's static data dependency analysis [1,3]. In the Sync Model presented in this paper, the data flow graph is dynamically constructed after each unification and is modified by adding "dynamic links" when partially instantiated terms are detected in a binding by using a run-time type checking algorithm similar to [3]. The algorithm is more efficient than [2] and the graph constructed by the algorithm reveals more parallelism than [1]. Optional variable annotations from the programmer may help constructing the data flow graph.

To implement both AND parallelism and OR parallelism in one model is a difficult task. The synchronization of partial solution streams in AND processes has never been solved satisfactorily. Either AND parallelism is suppressed by connecting sibling AND processes into a linear chain [7, 9] or OR parallelism is reduced by using backtracking [2]. In the Sync Model, a synchronization mechanism is proposed to synchronize the multiple partial solutions so that all the solutions of a

given problem will be produced without explicit request. Therefore, the Sync Model is a multiple solution data driven model.

The language we choose for the Sync Model is an extended logic programming language, called CLP, with optional variable annotations and a commit operator. Variable annotations, the input annotation ("?") and the output annotation ("!"), are used in the clause body to specify the producer and the consumer of a shared variable. The commit operator "→" is used to serialize the executions of two parts of the clause body. CLP is not designed as a concurrent language. The variable annotations and the commit operator are used to achieve more efficient execution under the Sync Model, but they are not required and do not change the semantics of the language. Therefore, although the Sync Model is designed for CLP, any Horn-clause logic program can be executed under the Sync Model.

The target machine for the Sync Model is a message-passing multiprocessor system with the processors interconnected into an augmented binary tree, called the *Sneptree* [8,5]. Since the mapping of an unbounded binary tree onto the Sneptree is done automatically and the mapping of a complete binary tree onto the Sneptree is always optimal, the Sneptree is an ideal architecture for the Sync Model.

One of the major distinction between the Sync Model and the computation models for other concurrent logic programming languages, such as Concurrent Prolog [13], is that in our Model, a process is suspended when waiting for an input from an input channel, while in Concurrent Prolog, a process is suspended when it attempts to unify a read-only variable with a non-variable term. In our approach, all the input variables are bounded before the unification so that the unification rule is not changed. In Concurrent Prolog and other similar approaches [11,12], the unification rules are modified to handle variable annotations. As a consequence, the variable annotations may be propagated to other non-annotated variables and a read-only variable may get instantiated in a unification.

The rest of the paper is organized as follows: In the next section, the language and the Sync Model are described. We also address, and propose solutions to, the main problem of constructing *the* data flow graph, i.e., binding to a partially instantiated term causes the data flow graph to be changed, as well as the synchronization problem of multiple partial solutions in the data flow graph. In sections 4 and 5, the two main algorithms of the Sync Model, i.e., the ordering algorithm and the merge algorithm, are presented. We also prove the correctness and completeness of the

merge algorithm and the Sync Model.

2 The Language and the Sync Model

2.1 The Language

The language, called CLP, (which stands for Concurrent Logic Programming), is an extended logic programming language with variable annotations and guarded clauses.

A CLP program is a finite set of *guarded clauses* of the form

$$A :- G_1, G_2, \ldots, G_m \rightarrow B_1, B_2, \ldots, B_n.$$

where A is called the *head* of the clause, $(G_1, \ldots, G_m)$ the *guard* of the clause, and $(B_1, \ldots, B_n)$ the *body*.

The guard of a clause may be empty. When the guard is empty, the commit operator is neglected. When both the guard and the body are empty, the clause is called a *unit clause*. Both the guard and the body are a set of literals. The two sets are separated by a commit operator, "$\rightarrow$". Declaratively, the commit operator reads like a conjunction: A is true if $G_1, \ldots,$ and G_m, as well as $B_1, \ldots,$ and B_n are true. Operationally, the commit operator forces the sequential execution of the guard and the body: a goal $A1$ which is unifiable with A can be reduced to $B_1, \ldots,$ and B_n if and only if the guard literals $G_1 \ldots G_m$ are evaluated to true.

A variable can be either a simple variable, or an *output variable* annotated by a postfix operator "!", or an *input variable* annotated by a postfix operator "?". Variable annotations are not allowed in the clause head. This restriction prohibits annotated variables from appearing in the unification. Therefore, Robinson's unification algorithm can be used directly without any modification. A variable is "shared" when it appears in more than one literal in the body. For a shared variable in the body, at most one literal containing that variable is allowed to have it annotated as output. Such a literal is called the *producer* of that variable, and the literals that contain input variables are called the *consumers* of those variables. The guard may not have any shared variable with the clause head or the body after unification — a guard evaluates to true or false without generating any outputs. But share variables between guard literals are allowed. Such a syntactic restriction separates the guard and the body into two independent parts which simplifies the implementation of our Model. In each CLP program, there is a goal with the form "$:- G$".

Unlike other parallel logic programming languages, the extra language constructs in CLP, the variable annotations and the commit operator, do not affect the semantics of the language. They can be used by the programmer optionally to achieve more efficient execution under the Sync Model. In order to prevent the semantics from being changed by the commit operator, when the restriction on the variables of the guard is violated, the system simply ignores the commit operator and executes the guard and the body in parallel.

The execution of a logic program is to construct and search the AND/OR tree of this program. For a given goal and a program, there exists a unique AND/OR tree which represents the complete search space of the goal. The Sync Model constructs a tree of processes corresponding to the AND/OR tree of the program and search the tree in breadth-first manner.

2.2 The Sync Model

The computation model of CLP, called the Sync Model, is a *process model.* Two types of processes are created and terminated dynamically during the computation. An *AND process* corresponds to a goal, and an *OR process* corresponds to a clause that is used to reduce a specific goal. A tree of interleaved AND and OR processes, called the *process tree*, is constructed corresponding to the AND/OR tree of the program. The initial goal is assigned to an AND process, which becomes the root of the process tree. For each clause whose head is unifiable with the goal of an AND process, one OR process is spawned to carry out the unification and the reduction of this OR clause. After unification succeeds in an OR process, the reduction of the goal is carried out by spawning one AND process for each literal in the body and then reducing the goals in the AND processes concurrently. If the clause in an OR process has a nonempty guard, a set of AND processes is spawned for each goal in the guard first. When all the AND processes for the guard successfully terminate, the OR process can spawn processes for the goals in the body and proceed. When any of the guard literals fails, the OR process fails. Therefore, full OR Parallelism is implemented in this model in the way of parallel unification of all the unifiable clauses, parallel evaluation of all the guard literals and parallel execution of all the OR branches that succeed in unification.

A leaf node of the process tree is either an OR process which fails to unify, or an OR process corresponding to a unit clause, or an AND process corresponding to a built-in predicate. An OR process containing a unit clause returns the variable bindings to its father AND process and terminates if it succeeds in unification. An AND process corresponding to a built-in predicate evaluates the predicate directly and sends the variable bindings to proper destination processes.

A non-leaf AND process succeeds when at least one of its OR descendants succeeds. It receives the bindings of its output variables from the descendants and sends them out to its father and all the sibling consumer processes of its output variables. A non-leaf OR process succeeds when all its descendant AND processes successfully terminate. It merges the results received from its descendants and then sends them to its father.

AND parallelism is implemented by dynamically constructing the data flow graph of the literals in the clause body. To avoid binding conflict in the parallel execution of sibling AND processes with shared variables, only one AND process is allowed to be the producer of a shared variable. All the other AND processes that also contain that shared variable are considered the *consumers* of that variable. A consumer process will suspend its computation until the values of its input variables have been received from their producers. A data flow graph of all the literals in the clause body, (so-called AND literals), is constructed such that a node represents an AND literal and an edge is directed from the producer of a shared variable to a consumer of that variable. As we shall see, the ordering algorithm will guarantee that the data flow graph is acyclic so as to avoid deadlock. Communication channels are added into the process tree to model the edges of the data flow graph. With the communication channels between sibling AND processes, the process tree is no longer a tree. We prove later that our process tree generates the same results as the corresponding AND/OR tree.

The input and output annotations in CLP are added to the program optionally by the programmer to help construct the data flow graph so that more efficient computation can be achieved. Without explicit variable annotations, the "left to right" order of the AND literals is used for selecting the producer of a variable. The explicit variable annotation should fulfill the two restrictions on the data flow graph: one producer per variable and acyclicity of the data flow graph. These can easily be checked syntactically.

Parallel execution of different OR processes may produce multiple solutions for the output variables of their father AND process. Those multiple solutions will be transmitted along the communication channels. Hence, we need some mechanism to synchronize the multiple inputs of a given AND process originating from different sources. In our computation model, any process that generates or collects a solution transmits the solution without requiring a request. Hence, our model can be viewed as a ***multiple-solution data-driven model.*** With this synchronization mechanism, we are able to incorporate both AND parallelism and OR parallelism without any form of backtracking.

2.2.1 Synchronization of Multiple Inputs of a Process

Multiple solutions for a variable may be transmitted through the communication channels in the data flow graph. If one AND process, say p, consumes two inputs from two different sources, we shall merge the two input streams to form all the input combinations of process p. Usually, the input combination of process p is the Cartesian Product of the two input streams. There is one exception — when the two input streams originate in the same process, the input combination of p is a set of Cartesian Products over certain portions of the two input streams that derive from the same output of the common ancestor. In the sequel, we call a set of paths that have the same starting process and the same ending process a *multiple path* between these two processes. In Figure 1, there are two paths (a, b, d) and (a, c, d) between process a and process d. If process a binds (X,Y) to (x_1, y_1) and (x_2, y_2), process b binds T to t_1 and t_2 with input x_1, t_3 with input x_2, and process c binds S to s_1 and s_2 with input y_1, s_3 and s_4 with input y_2, then the input combination for process d should be $(t_1, s_1), (t_1, s_2), (t_2, s_1), (t_2, s_2), (t_3, s_3), (t_3, s_4)$ instead of the full Cartesian Product of the two input streams. Observe that the first four input pairs of process d are derived from the input (x_1, y_1) and the remaining two input pairs are derived from (x_2, y_2). Because the two inputs of process d originate in the same process a, we shall form the Cartesian Product over the portions of the input streams which are generated by the same output pair of process a, e.g., (t_1, t_2) and (s_1, s_2), or (t_3) and (s_3, s_4). In order to derive the correct input combination, we mark process a as a Sync generator and the outputs generated by process a are separated by a special Sync signal. The Sync signals are then propagated through processes b and c, and reach process d in both inputs. Finally process d detects the same Sync signals in both inputs and then forms the Cartesian Product over the input portions which are enclosed by the corresponding pair of Sync signals.

After the data flow graph has been constructed, we determine all the multiple paths in the graph and mark the starting nodes of those paths as Sync generators. Different Sync generators generate different Sync signals. A process that receives two or more inputs from different channels merges the input streams according to the Sync signals carried in each input stream. The Sync signals may be duplicated during the merge process when they are nested in other Syncs. In the above example, process a is a Sync generator, hence the output streams generated by process a should be $(S_a, x_1, S_a, x_2, \text{END})$ and $(S_a, y_1, S_a, y_2, \text{END})$ respectively, where S_a represents a Sync signal generated by process a and "END" represents a special signal indicating the end of the

stream. Likewise, the output streams of process b and process c should be $(S_a, t_1, t_2, S_a, t_3, \text{END})$ and $(S_a, s_1, s_2, S_a, s_3, s_4, \text{END})$ respectively. Therefore, the input combination of process d becomes $(S_a, (t_1, s_1), (t_1, s_2), (t_2, s_1), (t_2, s_2), S_a, (t_3, s_3), (t_3, s_4), \text{END})$. Once a Sync signal is generated, it is propagated to (may be duplicated in) the other sibling AND processes through the communication channels in the data flow graph. The Sync signals will be removed at the father OR process before the output streams are sent out to higher level AND processes. Therefore, the Sync signals are local to the OR process and its AND descendants.

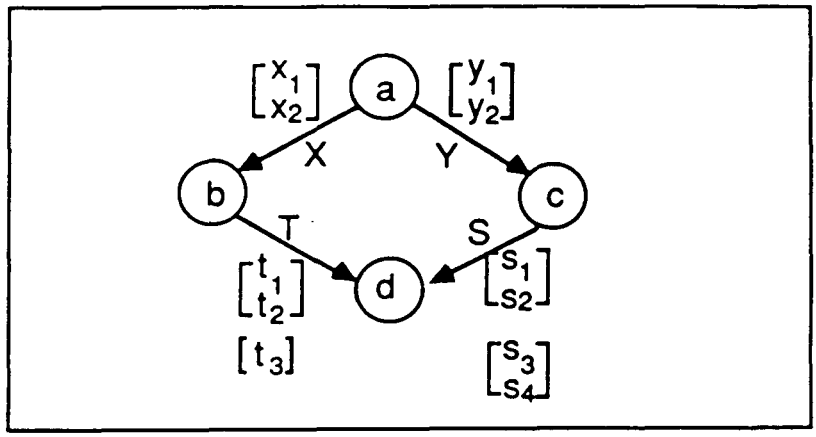

Figure 1. An Example with Multiple Path

2.2.2 Partially Instantiated Terms

When the producer of a variable binds the variable to a partially instantiated term, i.e., a term containing another variable, binding conflict may occur if that variable has more than one consumer. We solve this problem by adding so-called "dynamic links" into the graph to enforce the "one producer per variable" rule to the newly generated variable.

The data flow graph needs to be changed in two cases: (1) when a variable is bound to a partially instantiated term and this variable has more than one consumer, and (2) when two or more variables are bound to some terms containing the same variable. In both cases, one of the consumers of these variables is selected as the producer of the new variable and the dynamic links are directed from the new producer to all the rest of the consumers. The information about dynamic links is not provided during the construction of the data flow graph. Instead, such information is generated and sent to the selected producer of the new variable when an AND process binds some output variables to partially instantiated terms. A simple test on the binding values of all the output variables to test the above two cases is sufficient to determine whether dynamic links are

needed and how they are directed. Such a test is similar to DeGroot's type checking [3], except that we do the same check in every AND process without consulting the complex graph expression proposed by DeGroot.

The creation of dynamic links may introduce new Sync generators. The only process which may become a Sync generator is the producer of the new variable, which becomes a Sync generator when the node that binds a variable to the partially instantiated term is also a Sync generator. Those Sync Generators are identified and marked after the dynamic links are created. For more detail about dynamic links, see [6].

2.2.3 The AND process and the OR process

We briefly summarize the major tasks performed by an AND or an OR process. For full detail of the Sync Model, see [6].

AND process

- Call a merge algorithm to merge the input streams and bind the merged inputs to the input variables of the goal one at a time if the goal contains input variables.
- Perform type checking on the merged input and create dynamic links if necessary.
- Spawn OR processes and collect the results for each of the goals with bound input variables.
- Generate Sync signals to separate each of its outputs if it is a Sync generator.

OR process

- Unify the goal with a given clause.
- Return the bindings derived in the unification followed by an "END" to its father if the given clause is a unit clause.
- Construct the data flow graph of the guard literals and spawn AND processes for the guard if the given clause has a nonempty guard.
- Construct the data flow graph of the literals in the body and spawn AND processes for the body if all the AND processes for the guard return true or the given clause has an empty guard.
- Merge the partial solutions received from its descendants, remove the Sync signals and send the results to its father.

3 The Ordering Algorithm

By imposing that each shared variable has exactly one producer, we eliminate binding conflicts. To construct the data flow graph of AND literals, an Ordering Algorithm is applied in each OR process. The data flow graph is represented in two ways: by variable annotations in the literals and by a channel table containing the producer and consumer information of shared variables.

The Ordering Algorithm is performed in an OR process to construct the data flow graph of the AND literals after unification succeeds and the variables in the clause body are replaced by their binding values if they are instantiated after the unification. The Ordering Algorithm consists three major steps: (1) the construction of the data flow graph, (2) the refinement of the graph, and (3) the marking of the Sync generators. In the first step, variable annotations are used to determine the modes (input or output) of the uninstantiated variables in the AND literals. Initially, all the AND literals in the clause body are stored in an *Undecided Process List* (UPL). The algorithm determines the producer and the consumers of all the variables in the AND literals, adds annotations to all the variables, and then moves the literals to a *Fired Process List* (FPL). A Channel Table (CT) is also constructed to store the producer and consumers information of all the variables. Moreover, the literals are renumbered during this step so that their numerical order is consistent with their partial order in the data flow graph. In the second step, the data flow graph is further refined by creating "selective channels" and "True/False channels" for the literals that generate no output variables. As we shall see, this step is necessary to exploit the parallelism implied by the program so that a more efficient data flow graph can be constructed. In the third step, the algorithm searches for all the multiple paths in the data flow graph. If a multiple path is found, the algorithm marks the starting node of the multiple path as a SYNC generator. The complete algorithm will be elaborated in the remainder of this section.

Data Structure:

The following data structures are used in the algorithm:

- UPL – a list of AND literal and identifier pairs that are not fired yet†.
- FPL – a list of fired AND literals with all their variable arguments annotated. Each entry in the list contains an AND literal with annotated arguments, a Sync attribute, and a number attached to the literal to enforce a total order.

† "A literal is fired" means that a literal is moved from UPL to FPL and all its variable arguments are annotated.

- CT – a table of triples (Var,Producer,Consumers-list), to record the producer and consumers of a variable.
- S – a stack containing distinct nodes belonging to the paths starting from one specific node.

Besides, the AND literals are initially identified 1 to N from left to right in the clause body with the goal of the current OR process numbered 0.

Algorithm:

Step 0: Initialization

CT:= $\emptyset$; FPL:= $\emptyset$;

UPL:= list of all literals.

Step 1: Construction of the data flow graph:

In this step, the producer and the consumers of each shared variable are chosen and the variables in each literal are annotated.

A literal can be fired iff (1) all its input variables have a producer and the producers are already fired, and (2) the total number of output variables, input variables, and constant arguments of this literal is at least one. The first condition assures that a producer of a shared variable is always fired before the consumers of this variable. The second condition implies that the threshold [14] of each literal is one. If none of the unfired literals satisfies the above conditions, the leftmost unfired literal in the clause body is chosen to be fired next.

a. forall v_i: $v_i \in$ uninstantiated variables in the goal
   add $\langle v_i, [], [0] \rangle$ into CT;

b. forall l: $l \in$UPL
   do forall v_i: $v_i \in$ variable arguments in l
      do if $v_i \notin$ CT → if v_i is output annotated → add $\langle v_i, l, [] \rangle$ into CT
                             | otherwise → add $\langle v_i, [], [] \rangle$ into CT
                             fi
         | $v_i \in$ CT → if v_i is output annotated → CT.v_i.*producer* := l
                         | otherwise → skip
                         fi
         fi
      od
   od ;

```
c. index := 1;
   while UPL≠ ∅
   do    fired :=false;
         forall l: l ∈UPL
         do    forall v_i: v_i is unannotated ∧ CT.v_i.producer ≠[]
                     mark v_i as an input variable in UPL;
               b :=true;
               forall v_i: v_i is an input variable in l
               do    x :=CT.v_i.producer;
                     b := b ∧ (x ≠ [] ∧ x > N)
               od    {b = ∀v_i : v_i is an input variable in l: v_i has a producer and the producer is fired}
               if b∧(#constant arguments+#output variables+#input variables>0)→
                     {beginning of firing process}
                     newid := index + N;
                     forall v_i: v_i ∈ variableargumentsinl
                     do    if v_i is input → add newid into CT.v_i.consumer
                           |v_i is unannotated∨v_i is output → CT.v_i.producer := newid;
                                       mark v_i as an output variable in UPL
                           |otherwise → skip
                           fi;
                     od
                     UPL:=UPL−l;
                     FPL[index] := l;
                     index := index + 1;
                     fired :=true
                     {end of firing process}
               |otherwise→ skip
               fi
         od ;
d.       if ¬fired → l :=UPL[1];
                     do "firing process"
         |otherwise→skip
         fi
   od ;
e. forall v_i: v_i ∈ CT
   do    if CT.v_i.consumer = [] → CT:=CT−v_i
         |otherwise → skip
         fi
   od .
```

Step 2: Refinement of the Graph

If some literals have no output variables in the data flow graph constructed in step 1, extra links need to be added into the graph to make sure the true/false results of this kind of literals will be transmitted to the goal.

Let's assume p is such a literal and X is an input variable of p. In this step, we first attempt to add so-called *selective channels* from p to the rest of the consumer literals of X. These channels transmit only the values of X that make p true. Meanwhile, the links between the producer of X to the consumers of X except p are removed from the graph. If no selective channel is constructed for p, a *True/False channel* is added from p to the goal to transmit the results of p.

The insertion of selective channels should not cause cycles in the graph. To assure the acyclicity of the graph, we only add the selective channels such that the receiver of the channel is fired after all the antecedents of the sender. The antecedents of a literal are the producers of all the input variables of the literal.

forall l: $l \in$ FPL $\wedge$ l has no output variables
do $new := false$;
$prod := \emptyset$;
forall v_i: v_i is an input variable of l
add CT.v_i.*producer* into *prod*;
forall v_i: v_i is an input variable of l
do $c := CT.v_i.consumer$;
$cl := \{c_i | c_i \in c : (\forall p_j \in prod : c_i > p_j) \wedge c_i \neq l\}$;
if $l \in c \wedge cl \neq \emptyset \rightarrow$ add $\langle \overline{v_i}, l, cl \rangle$ into CT;
CT.v_i.*consumer* $:= c - cl$;
new :=true
|otherwise $\rightarrow$ skip
if
od ;
if $\neg new \rightarrow$ add $\langle t/f, l, [0] \rangle$ into CT
|otherwise $\rightarrow$ skip
fi
od .

Step 3: Marking of the Sync generators (Detection of the multiple paths):

A stack is built for each literal l in FPL that has more than one output channel. The descendants of l are pushed into the stack if they are not yet in the stack. This pushing process

continues until either all the descendants of l are in the stack or a descendant to be added to the stack is found to be already in the stack. In the second case, l is marked as a SYNC generator.

```
forall l : l ∈FPL ∧ #consumers(l) > 1
do    pt := 1; S := [l];
      while S[pt] ≠ ∅
      do    p := S[pt];
            forall v_i : v_i is a variable in p
            do    if CT.c_i.producer = p →
                        forall c_i : c_i ∈ CT.v_i.consumer
                        do if c_i ∉ S → push c_i into S
                           |c_i ∈ S → set Sync attribute of p to true in FPL; stop
                           fi
                        od
                  |otherwise → skip
                  fi;
            od
            pt := pt + 1
      od
od
```

The above algorithm always generates an acyclic data flow graph with one producer per shared variable. The ordering algorithm is correct for the following reasons:

1. The ordering algorithm selects exactly one producer for each variable.

2. The data flow graph generated in Step 1 is acyclic because a literal can be fired only when all the producers of its input variables have been fired (b is true in Step 1.c). Therefore, the producer of a given variable is always fired before all the consumers of that variable. The firing order of the literals implies their partial order in the data flow graph, thus, the graph has no cycles.

3. The refined data flow graph generated in Step 2 is acyclic because the redirected links do not create cycles in the refined graph. If a cycle were found in the refined graph, it would contain at least one redirected link, say (l_i, l_j). Let the cycle be $(l_i, l_j, \ldots, l_k, l_i)$, then l_k is the producer of one input variable of l_i and $l_j > l_k$ because a path exists from l_j to l_k. In Step 2, such a link (l_i, l_j) is never generated because l_j is excluded from cl. Therefore, the refined graph is also acyclic.

An Example

Figure 2 is a query:"Is there a student such that a professor teaches him two different *courses* in the same room?" for a data base of *S*tudents who take *C*ourses ($student(S,C)$), *P*rofessors who teach *C*ourses ($professor(P,C)$), and *C*ourses held on certain week*D*ays and *R*ooms ($course(C,D,R)$), [10]. To save space, the database of relations *student*, *course*, and *professor* are omitted here.

```
query(S,P):- student(S,C1),          (1)
             course(C1,D1,R),        (2)
             professor(P,C1),        (3)
             student(S,C2),          (4)
             C1≠C2,                  (5)
             course(C2,D2,R),        (6)
             professor(P,C2).        (7)
```

Figure 2. A Query for a database of students

To answer the query "$:- query(S,P)$.", we construct a process tree and map the initial goal to the root. In the OR process that is spawned by the root, we shall apply the ordering algorithm against the seven AND literals in Figure 2.

Since none of the variables are annotated in the definition of *query*, we select the producers of the shared variables by imposing the left-to-right order of the literals and as a consequence, the data flow graph constructed by Step 1 is shown in Figure 3.

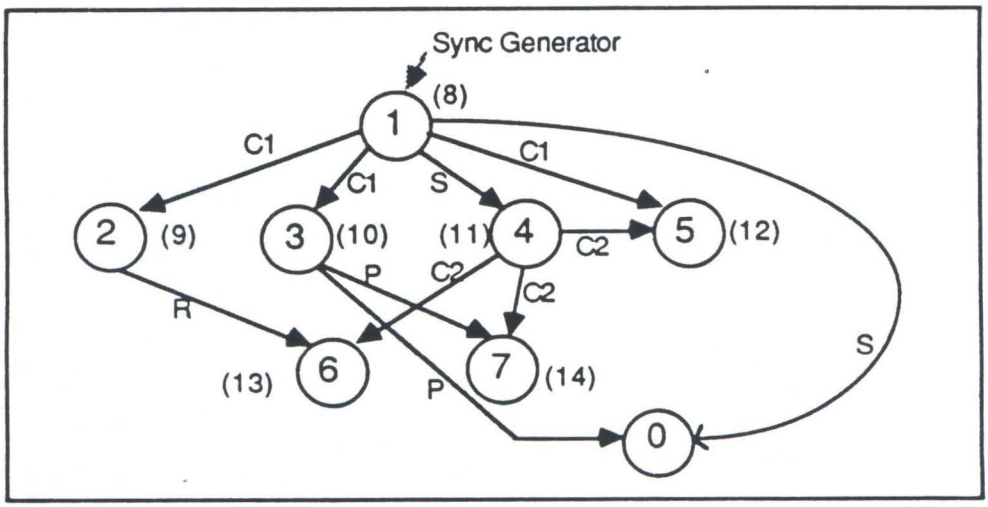

Figure 3. The Data flow graph of $query(S,P)$

In Step 1, the literals are renumbered so that their numerical order implies their partial ordering in the graph. The new identifiers of the literals are enclosed in the parentheses next to each node

in Figure 3. Notice that literals (5), (6), and (7) don't generate any outputs. After adding selective channels to these literals by Step 2, the refined data flow graph is shown in Figure 4. Comparing with the previous graph, we found that the variable $C2$ is transmitted sequentially from literal (4) to (5), (6) and (7) in the refined graph instead of transmitted in parallel in the original graph. At first glance, the refined graph seems to have less parallelism than the original one. In fact, the latter one is more efficient than the former one because literal (6) or (7) only receives the values of $C2$ such that literal (5) or (6) is proved true. Therefore, the values of $C2$ generated by (4) will first be filtered by (5), then sent to (6) and so forth. Unnecessary computations are avoided in (6) and (7) because invalid values of $C2$ won't be received by them. Also notice that no selective channels are constructed for $C1$ at literal (5) because the consumers of $C1$, (2) and (3), are both fired before the producer of $C2$, i.e., (4). To assure the acyclicity of the graph, the channels for $C1$ remain unchanged.

In Step 3, a stack is built up for literal (1). A multiple path is found when (5) is going to be pushed into the stack twice. Therefore, (1) is marked as a Sync generator. No more stack is needed because all the other literals have exactly one descendant each.

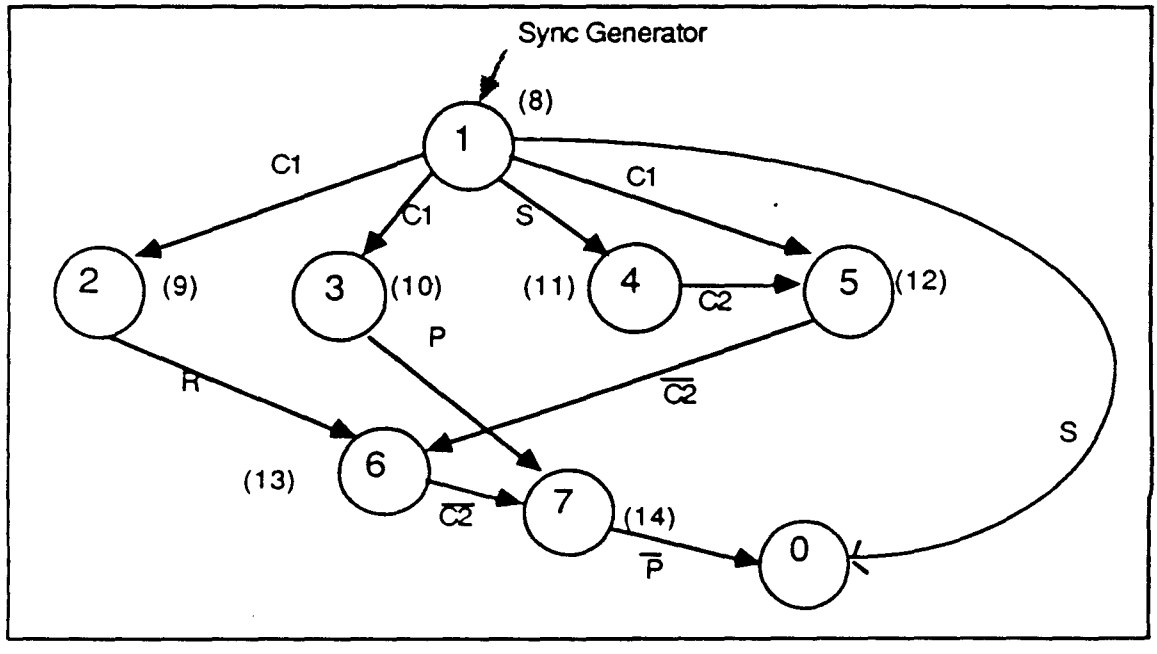

Figure 4. The Refined Data flow graph of $query(S, P)$

The average complexity of the ordering algorithm is $O(n \lg n)$ with n AND literals. In most cases, the AND literals in a clause body are almost-ordered, therefore, a linear complexity can be

achieved. For detail analysis of the complexity of the ordering algorithm, please see [6].

4 The Merge Algorithm

In the Sync Model, a process has to handle multiple input streams from different sources. For example, an OR process has to merge all the partial solutions from its AND descendants to form the solutions of this OR process, and an AND process needs to merge the input streams from other sibling AND processes to form input combinations to itself. It is particularly true for a nondeterministic program, in which multiple partial solutions may be generated, transmitted and validated by different processes. A merge algorithm that synchronizes the execution of all the cooperating processes is the crucial part of our Sync Model.

The merge algorithm in an AND process is basically the same as the one in an OR process. The only difference is that the input stream of the latter one may contain True/False values instead of variable bindings. In the following, the merge algorithm refers to the one in an AND process.

The merge algorithm operates only when there exist two or more input variables in a process. An input stream consists of SYNC signals, variable bindings, and an END signal at the end. A variable binding is a pair consisting of a variable name and its binding value. The SYNC signal carries the process identifier that identifies the generator of the Sync signal. SYNC signals are nested when the receiving node belongs to two or more different multiple paths. In essence, the merge algorithm forms a Cartesian Product over the input streams to form all the possible input combinations. When SYNC signals appear, the algorithm forms Cartesian Product over part of the input streams separated by pairs of identical SYNC signals. In other words, only the input elements in between the corresponding pair of SYNC signals can be combined and the input streams are thus synchronized by the SYNC signals.

In the rest of this section, the base-case algorithm (i.e., no input stream contains SYNC signals) is described in the next subsection. The Cartesian Product implemented as nested loops is inefficient because the process may keep waiting for the inputs from a slow channel. A more efficient algorithm is given in Figure 5. This algorithm reduces the waiting time by forming the Cartesian Product over the available portions of input streams while the rest of the inputs are not there yet. The general algorithm with input streams containing SYNC signals is presented in Section 4.2. Figure 6 is the general algorithm for two streams. The general algorithm is a recursive algorithm which recursively peels off SYNC signals in two streams and finally forms the Cartesian Product over the data inputs

enclosed by the innermost SYNC signal pair with the base-case algorithm. The algorithm for n streams can be derived by generalizing the two-stream algorithm. In the last section, a correctness proof for the n-input general merge algorithm is presented.

Throughout the algorithms, $buf[i,j]$ is used to represent the j-th input in the i-th input buffer, where $1 \leq i \leq n$ and n is the total number of input buffers. Each buffer is assumed to have enough capacity to store the whole input stream. The $index[i]$ points to the position which is currently being merged and $avail[i]$ points to the top of the available portion of buffer i. Procedure $put(entry)$ adds a new element *entry* into the output queue, where *entry* can be a SYNC signal, an array of n input bindings or an "END" signal.

4.1 Base-case Algorithm

Since the merge algorithm is operating concurrently with the receiving of inputs in each input buffer, the simple iterative loop implementation may be inefficient due to waiting for the inputs from a slow channel. A more efficient implementation is shown in Figure 5.

This algorithm forms the CP (abbreviation for Cartesian Product) over the available portions of the n input streams repeatedly. Whenever an input buffer receives new inputs, Procedure *cp* is called repeatedly to form the CP over the newly received inputs and the available portions of the other input buffers. Then $avail[j]$ is advanced to the location of the newest available input. The algorithm repeats the above operations for each input buffer until the new input in all the input buffers is "END".

```
{ Global Variables}
integer n;  { number of input buffers}
integer array index[1:n], avail[1:n];  { pointers}
input buffer buf[1:n,1:m];  { n input buffers with length m which are large enough to contain
                              the whole input streams}
buffer entry[1:n]; { a buffer to contain the next output}

{ Cartesian Product of the available portions of the n input buffers except the i-th buffer
  which is fixed to an element e}
procedure  cp(e,i);
begin
    entry[i]:=e;
    cp1(i,1)
end.

{ Cartesian Product over the available portions of buf[k] to buf[n] except buf[i]}
procedure  cp1(i,k);
begin
```

```
    [ k>n → put(entry)
    | k=i → cp1(i,k+1)
    | otherwise → l:=1;
                  *[ l≤avail[k] → entry[k]:=buf[k,l];
                                  cp1(i,k+1);
                                  l:=l+1
                   ]
    ]
end.

{ Main Program }
begin
    i:=1;
   *[ i≤n → index[i]:=1; avail[i]:=0; i:=i+1 ];
    i:=1;
   *[ ∃k: 1≤k≤n: buf[k,index[k]]≠'END' →
        *[ i≤n → *[ ¬empty(buf[i,index[i]])∧buf[i,index[i]]≠'END' →
                                                  cp(buf[i,index[i]],i);
                                                  index[i]:=index[i]+1
                  ];
                  avail[i]:=index[i]-1;
                  i:=i+1
         ]
     ]
end.
```

Figure 5. Base-case Algorithm

4.2 General Algorithm

If SYNC signals appear in at least one input stream, the general merge algorithm applies. We first present the general algorithm for two input streams and later show how to generalize the algorithm to n input streams.

In the ordering algorithm, the literals have been renumbered so that their numerical order is compatible with their partial order in the data flow graph. The linear ordering of the Sync signals in an input stream is always assured by the merge operation which performs an n-way merge on n input streams.

The general algorithm consists of two principal operations: merge on the same Sync signals and merge on different Sync signals. First, let two input streams contain the same Syncs, say S, and the two input streams are $A = (S, A_1, S, A_2, \ldots, S, A_n, \text{END})$ and $B = (S, B_1, S, B_2, \ldots, S, B_n, \text{END})$, then the merge result is a sequence of CP's over the corresponding portions of the two input sequences which are separated by a pair of consecutive S's, i.e.,

$$A \times B = (S, A_1 \times B_1, S, A_2 \times B_2, \ldots, S, A_n \times B_n, \text{END}) \qquad (1)$$

where A_j stands for a sequence of data inputs, so as B_j for $1 \leq j \leq n$, and $A_j \times B_j$ stands for the CP of A_j and B_j.

The second principal operation handles the merge of two sequences with different Syncs. Let two input streams be $A = (S1, A_1, S1, A_2, \ldots, S1, A_n, \text{END})$ and $B = (S2, B_1, S2, B_2, \ldots, S2, B_m, \text{END})$, and let S1<S2 so that S1 becomes the outer Sync in the merge result. The linear ordering of the Sync signals in a merged stream guarantees that the common Syncs appearing in two input streams are in the same order, therefore, the merge algorithm functions correctly. The merge result can be computed as follows:

$$\begin{aligned} A \times B &= (S1, A_1 \times B, S1, A_2 \times B, \ldots, S1, A_n \times B, \text{END}) \\ &= (S1, S2, A_1 \times B_1, S2, A_1 \times B_2, \ldots, S2, A_1 \times B_m, \\ &\quad S1, S2, A_2 \times B_1, S2, \ldots, A_2 \times B_m, \\ &\quad S1, S2, \ldots\ldots, S2, A_n \times B_m, \text{END}) \end{aligned} \tag{2}$$

The merge result is actually the CP of all the data inputs of the two streams when the two input streams contain different Syncs. In order to maintain the synchronization information, we first do the CP's over the whole input stream B and a portion of stream A, i.e. A_i for all i and separate the CP's by S1. In each $A_i \times B$, again we do a set of CP's of $A_i \times B_j$ for all j and separate them by S2. The CP "$A_i \times B_j$" contains no Sync signals, hence the base-case algorithm can be applied. In the result, the number of Sync signals S1 is preserved, i.e., n, and the number of Sync signal S2 is increased to $n \times m$ because S2 is nested inside S1.

The general algorithm for two input streams is recursively defined on the two principal operations. The Sync sequences of the input streams are linearly ordered, i.e., a Sync signal is larger than all the Syncs which are outer to it and smaller than all the Syncs inner to it. In each recursion, the outermost Sync signals of the two input streams are checked. If they are the same, the first principal operation is called. If they are different, the second principal operation is called. The merge algorithm is called recursively to compute each $A_i \times B_i$ in (1) or each $A_i \times B$ in (2). When the merge algorithm is called to merge two input streams without any Sync signals, the base-case algorithm is applied to get the CP. The merge result preserves the linear ordering of the Sync sequence. Figure 6 presents the major procedures of the merge algorithm: *merge*, and *scanto*. Procedure *merge* merges the input streams in $buf1$ and $buf2$, and puts the result in an output queue. Boolean function *sync* checks whether the given argument is a Sync signal or not. Procedure *merge* has a guarded command with four alternatives: (1) neither of the inputs contains

Sync signals: *cp* is called to derive the Cartesian Product, (2) either $buf1$ contains Sync signals and $buf2$ does not, or both inputs have Sync signals and the outermost Sync of $buf1$ is smaller than that of $buf2$: the second principal operation applies, (3) same condition as (2) with $buf1$ and $buf2$ switched: the second principal operation also applies with A and B switched, and (4) both inputs contain Sync signals and the outermost Syncs of the two inputs are the same: the first principal operation applies. Procedure *scanto* divides the input buffer into two parts by the first occurrence of *some specific* SYNC signal S. Procedure *cp* is the base-case merge algorithm which generates the CP of the data elements in two buffers.

```
procedure  merge(buf1,buf2)
begin
    [buf1=∅∨buf1="END"∨buf2=∅∨buf2="END" → skip
    |otherwise→ A:=buf1[1]; B:=buf2[1];
        [¬sync(A)∧¬sync(B) → cp(buf1,buf2)                          (1)
        |sync(A)∧(¬sync(B)∨(A<B)) → scanto(buf1,A,buf11,buf12);     (2)
                                    put(A);
                                    merge(buf11,buf2);
                                    merge(buf12,buf2)
        |sync(B)∧(¬sync(A)∨(B<A)) → scanto(buf2,B,buf21,buf22);     (3)
                                    put(B);
                                    merge(buf1,buf12);
                                    merge(buf1,buf22)
        |sync(A)∧sync(B)∧(A=B) → scanto(buf1,A,buf11,buf12)         (4)
                                  scanto(buf2,B,buf21,buf22);
                                  put(A);
                                  merge(buf11,buf12);
                                  merge(buf12,buf22)
        ]
    ]
end of procedure merge.

procedure  scanto(buf,S, buf1,buf2)
begin
     i:=2;
    *[ buf[i]≠S∧buf[i]≠"END" → buf1[i]:=buf[i]; i:=i+1];
     j:=1; N:=length(buf);
    *[ i≤N → buf2[j]:=buf[i]; i:=i+1; j:=j+1]
end of procedure scanto.
```

Figure 6. General Algorithm for two buffers

If there are more than two input buffers and some of them have one or more SYNC signals, the above algorithms can be generalized easily. With n input streams, in which each has an ordered Sync sequence, the merge algorithm applies recursively to remove the smallest Sync signal of the n outermost ones of the input streams one at a time. When the smallest Sync is common to several input streams, all those Syncs will be removed at once. When none of the input streams

contains Sync signals, the Cartesian Product over n input streams is performed. For instance, if $merge(buf1, buf2, \ldots, bufn)$ is called and a smallest Sync S is found in both $bufi$ and $bufj$, the following program is executed:

```
scanto(bufi,S,bufi1,bufi2);
scanto(bufj,S,bufj1,bufj2);
put(S);
merge(buf1,...,bufi1,...,bufj1,...,bufn);
merge(buf1,...,bufi2,...,bufj2,...,bufn);
```

The merge algorithm in an OR process merges the partial solutions received from its AND descendants to form all the legal solutions of this OR subtree. The partial solutions received from one AND descendant could be variable bindings or true/false values. The true/false values are used to select the merge result from other channels. If the value is true, the merge algorithm merges the partial solutions as usual. If the value is false, the merge algorithm skips the merge operation and returns false instead. In addition, the merge algorithm in an OR process eliminates all the Sync signals in the merge result so that the solution stream sent up to the father AND process contains no Sync signals.

4.3 Correctness Proof

In order to prove that the merge algorithm produces all the correct combinations of multiple inputs of a process, we shall define the syntactic structure of an "input stream" and give a formal treatment of how an AND process transforms one or more input streams into an output stream.

Definition: An *input stream* $\Sigma_R(D)$ can be defined recursively:

1. $\Sigma_\emptyset(D) \equiv D$

2. $\Sigma_{R\cup\{i\}}(D) \equiv \Sigma_R(\Sigma_{\{i\}}(D)) \equiv \Sigma_R((S_i, D_v)^{n_i}), \forall j \in R : i > j.$

where R is an ordered set of integers. Each element in R is a Sync that appears in the input stream. Let's call R the *Sync sequence* of this input stream. We slightly abuse notations and represent R by the array $R[i]$, s.t., $i < j \Rightarrow R[i] < R[j]$. Σ_R is an operator defined recursively over the input data, D, where D is the input stream with all the Syncs removed. Applying $\Sigma_{\{i\}}$ over D is to divide D into n_i groups and separate each group by a Sync S_i. Each group of input data, D_v, is called a *data segment*, which is uniquely identified by a vector, v. In (2), v is a vector of length $(r+1)$ where $|R| = r$ and $v[r+1] = k$ for $1 \leq k \leq n_i$. Therefore, D_v represents a data segment that is

produced by the k-th output of the Sync generator S_i. Besides, $(S_i, D_v)^{n_i}$ is a regular expression denoting the concatenation of the string (S_i, D_v) n_i times. Notice that the data segment D_v is changed every time the syntactic structure of the input stream is transformed. The above notation is used to represent the syntactic structure of an input stream. How D_v is changed by different transformations of the input stream will be explained later.

There are two ways of changing the structure of an input stream in our model. First, if an AND process is a Sync generator, the structure of the output stream is derived by concatenating an extra Sync signal to the Sync sequence of the input stream. Second, if an AND process has several inputs, say n, the structure of the merge output can be derived by an n-way merge of the n Sync sequences. Figure 7 shows the two possible transformations of an AND process given one input and one output. In Figure 7.a, the structures of the input and the output streams are the same because the AND process is not a Sync generator. In Figure 7.b, the AND process is a Sync generator which generates Sync S_i and the output stream has the structure $\Sigma_{R\cup\{i\}}$. Because of the total ordering of the Sync generators, i is guaranteed to be larger than any element in R. Figure 8 shows the input-output transformation of the merge algorithm, given n input streams. The Sync sequence of the output is derived by n-way merge of the n input Sync sequences. An AND process with n inputs and one output can be represented by one merge operation (Figure 8) followed by one of the two AND operations (Figure 7) depending on whether the AND process is a Sync generator or not.

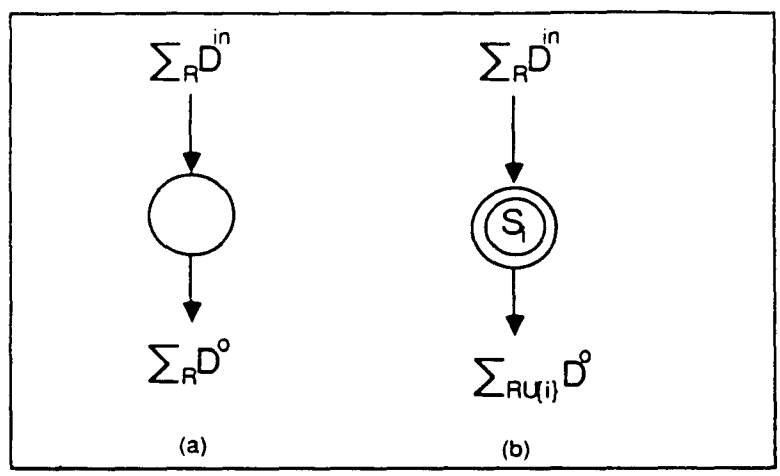

Figure 7. The transformation of an AND process with single input

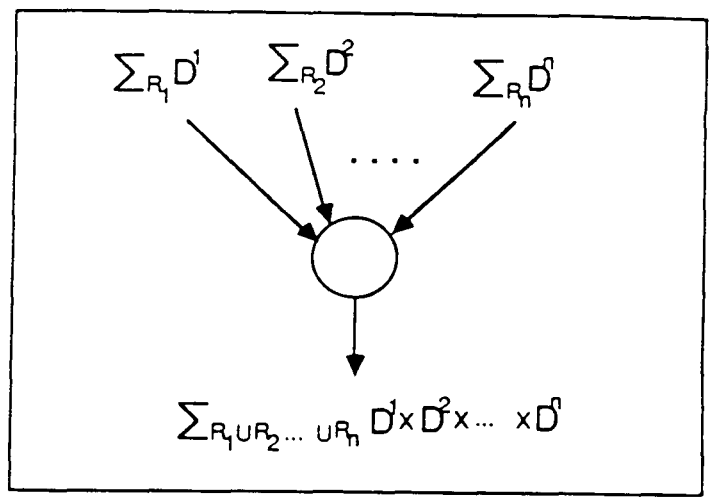

Figure 8. The transformation of the merge algorithm with two inputs

The data segments in the input stream are changed differently in the two transformations described above. Since we are only interested in the merge result, we'll only consider the second case, i.e., the transformation due to the merging of two input streams.

Definition: An *ordered union* operator "$\sqcup$" is defined as $R = R_1 \sqcup R_2$, where R, R_1 and R_2 are ordered sets (i.e., the elements in the set are sorted in ascending order) and $R = R_1 \cup R_2$. In other words, it is equivalent to a two-way merge.

Definition: An *ordered join* operator "$\underline{\sqcup}$" is defined as $v_R = v_{R_1} \underline{\sqcup} v_{R_2}$, where $R = R_1 \sqcup R_2$, v_R, v_{R_1} and v_{R_2} are vectors with length $|R|$, $|R_1|$ and $|R_2|$ respectively. v_R is the result of joining v_{R_1} and v_{R_2} on the common elements of R_1 and R_2. More precisely, $v_R = v_{R_1} \underline{\sqcup} v_{R_2}$ iff

1. $\forall i, j : R_1[i] = R_2[j] \Rightarrow v_{R_1}[i] = v_{R_2}[j]$ and
2. $v_R[i] = \begin{cases} v_{R_1}[j], & \text{if } R[i] = R_1[j]; \\ v_{R_2}[k], & \text{if } R[i] = R_2[k]. \end{cases}$

Theorem 1. *Given two input streams $\Sigma_{R_A}(D^a)$ and $\Sigma_{R_B}(D^b)$, the result generated by the merge algorithm is $\Sigma_{R_C} D^c$, where $R_C = R_A \sqcup R_B$. Moreover, D^c is defined as the Cartesian Product of D^a and D^b such that*

$$D^c_{v_c} = D^a_{v_a} \times D^b_{v_b} \qquad \text{with} \quad v_c = v_a \underline{\sqcup} v_b \tag{3}$$

Proof: Let the length of R_A and R_B be t_a and t_b respectively. This theorem can be proved by induction on the ordered pair (t_a, t_b), where $(t_a, t_b) < (t'_a, t'_b)$ iff $t_a < t'_a$, or $t_a = t'_a$ and $t_b < t'_b$.

It is easy to derive the proof from the program in Figure 6. The complete proof is given in [6]. ∎

From Theorem 1, we derive the merge result with two arbitrary input streams. The remaining task is to show that the merge result is correct. Given a process with two inputs, a legal input combination is an input pair such that the input elements of the pair are originated from the same output of a common ancestor along the two input paths. An input path is a path containing the current process, one of the two input links, and tracing back to any ancestor of the current process. There are many such paths. If a process is shared by any two input paths, in which each contains one different input link, then only the inputs which are derived by the same output of that process can be combined. Notice that such a common ancestor is marked as a Sync generator. Therefore, by observing the Sync sequences of the two input streams, we can determine all the common ancestors which affect the merge result along the two input paths.

Theorem 2 shows that the merge result in Theorem 1 indeed contains all the legal input combinations.

Theorem 2. *The result of the merge algorithm contains all the legal input combinations.*

Proof: Supposed that the two input streams in Theorem 1 have n common Sync signals, i.e., $|R_A \cap R_B| = n$, we need to prove that all the inputs that are derived from the same outputs generated by the n Sync generators are combined. Let P_i be the Sync generator that generates a Sync signal S_i. Then, each output generated by P_i is separated by a pair of S_i's. By propagating the output stream of P_i throughout the data flow graph, the syntactic structure of the output stream may or may not be changed. If the syntactic structure of the output stream is not changed, any result derived by the k-th output of P_i is appeared in the same data segment enclosed by the corresponding pair of S_i's. When the syntactic structure of the output stream is changed by merge operations or the generation of new Syncs, S_i may be further nested into other Sync signals. In this case, the results derived by the k-th output of P_i are divided into several data segments and spread into different locations. Generally speaking, with the input stream A in Theorem 1, the inputs that are derived by the k-th output of process P_i are the union of all the data segments with the j-th element of its id vector being k, where j is the position that S_i is placed in the Sync

sequence R_A.

$$\bigcup_{\substack{\forall v_a: \\ v_a[j]=k \wedge R_A[j]=i}} D^a_{v_a}$$

where $\bigcup_{\forall v_a: v_a[j]=k \wedge R_A[j]=i}$ is used as an abbreviation for a sequence of unions with the index v_a satisfing the condition specified in the subscript of $\bigcup$.

Assume there are n common Syncs, $S_{i_1}, S_{i_2}, \ldots, S_{i_n}$, in the two input streams. We will show that the merge result in the case the Sync generator P_{i_j} generating the t_j-th output, for all j, $1 \leq j \leq n$, is the Cartesian Product of the portions of the two input streams under the same condition. Let k_j, l_j and m_j be the locations where S_{i_j} appears in R_a, R_B and R_C, i.e., $R_A[k_j] = R_B[l_j] = R_C[m_j] = i_j$, for $1 \leq j \leq n$. Then the above relation can be formulated as follows:

$$\bigcup_{\substack{\forall v_c: \\ (\forall j:1\leq j\leq n: \\ v_c[m_j]=t_j)}} D^c_{v_c} = \bigcup_{\substack{\forall v_a: \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=t_j)}} D^a_{v_a} \times \bigcup_{\substack{\forall v_b: \\ (\forall j:1\leq j\leq n: \\ v_b[l_j]=t_j)}} D^b_{v_b}. \tag{5}$$

Eq. (4) can be derived from Eq. (3) easily. First add a big union $\bigcup_{\forall v_c:(\forall j:1\leq j\leq n:v_c[m_j]=t_j)}$ to both sides of (3). Then divide the unions at the right hand side into two independent sets of unions and then move the unions inside the CP and associate the first set of unions to D^a and the second set of unions to D^b.

$$\begin{aligned}
\bigcup_{\substack{\forall v_c: \\ (\forall j:1\leq j\leq n: \\ v_c[m_j]=t_j)}} D^c_{v_c} &= \bigcup_{\substack{\forall v_c: \\ (\forall j:1\leq j\leq n: \\ v_c[m_j]=t_j)}} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\substack{\forall (v_a \sqcup v_b): \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=v_b[l_j]=t_j)}} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\substack{\forall v_a: \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=t_j)}} \bigcup_{\substack{\forall v_b: \\ (\forall j:1\leq j\leq n: \\ v_b[l_j]=t_j)}} (D^a_{v_a} \times D^b_{v_b}) \\
&= \bigcup_{\substack{\forall v_a: \\ (\forall j:1\leq j\leq n: \\ v_a[k_j]=t_j)}} D^a_{v_a} \times \bigcup_{\substack{\forall v_b: \\ (\forall j:1\leq j\leq n: \\ v_b[l_j]=t_j)}} D^b_{v_b}.
\end{aligned}$$

Therefore, we can conclude the merge algorithm gives all the legal input combinations. ∎

With the above theorems, we can show that the Sync Model is complete, i.e., the Sync Model generates all the solutions for a given program.

From Kowalski [4], we know that each successful computation of an initial goal can be represented as a subtree of the AND/OR tree, i.e., the process tree in our Model. Such a subtree

starts from the root, expands by including exactly one descendant OR process for each of its AND process and all the descendant AND processes for each of its OR process, and ends with leaf nodes that successfully terminates.

Since any successful computation can be mapped onto a subtree in the Sync Model, if we can prove that such subtree generates the same solution as this successful computation, then the Sync Model is proved to be complete.

Theorem 3. *The Sync Model is complete.*

Proof: We first prove that a subtree that represents a successful computation generates the same solution as this computation. Let's first choose any OR process in a subtree that corresponds to a successful computation. Assume this OR process contains a goal g and a clause "$g1 :- p_1, p_2, \ldots, p_n$.". Let $X_1, X_2, \ldots, X_m$ be the variables within this clause and the successful computation gives a unique solution to these variables, i.e., $t_1, t_2, \ldots, t_m$. Moreover, let each p_i contains a set of input variables and a set of output variables. The input-variable set and the output-variable set of any p_i are disjoint and both of them are subsets of $(X_1, \ldots, X_m)$.

Let's assume that the subtree under each p_i produces the correct solutions for the output variables of p_i if the input variables are bound to the correct values. Here, the correct solution of a variable X_i is meant to be t_i. Therefore, any process p_i that has no input variables will generate the correct solutions to its output variables. Furthermore, any p_i with nonempty input-variable set will produce the correct solutions to its output variables if the producers of its input variables generate the correct solutions. The above statement is obviously true if p_i has only one input variable. It is also true if p_i has more than one input variable because the merge algorithm in p_i always generates the correct input combinations from Theorem 2. Therefore, the OR process generates the correct solution for its goal g assuming the subtrees under each p_i are correct. Furthermore, if in the subtree corresponding to a successful computation, there is an OR process which contains a unit clause. This OR process is always a leaf node and it generates the correct solutions to the output variables of the goal in the process. Thus, by induction, the subtree corresponding to a successful computation will generate the correct solution for that computation.

From the other direction, we shall also prove that any minimal subtree which produces an answer corresponds to a successful computation. A minimal subtree is a subtree which contains no failure nodes. The proof is similar to the proof above and thus omitted here.

Since any successful computation can be mapped onto a subtree in the Sync Model and each subtree generates the correct solution for the corresponding computation, we conclude that the Sync Model generates all the solutions for a given program and therefore it is complete. ∎

5 Conclusion

We have presented a model for the parallel execution of logic programming on a message-passing multiprocessor system. AND parallelism is carried out by constructing an efficient data flow graph dynamically The mechanism that is used to synchronize the multiple partial solution flows in the data flow graph makes it possible to realize both AND parallelism and OR parallelism without any form of backtracking.

Our model is complete. It handles both deterministic and non-deterministic programs, and it is particularly good for non-deterministic programs with multiple solutions. It is able to handle a pure logic program as well as an extended logic program with variable annotations and guarded clauses.

In our model, the AND/OR tree is searched in both breadth-first and depth-first manner. Consider two sibling AND processes that share a common variable. The subtree under the producer of the variable will be searched first and then the search for the consumer and its subtree can be started. If the producer produces multiple solutions to the variable, the execution of the two sibling AND processes are pipelined. Although this approach seems to be less parallel than purely breadth-first search of the AND/OR tree, our model is in fact more efficient because we avoid unnecessary computations in the consumer process. In a purely breadth-first search, invalid bindings of the shared variable are sent to the consumer and later found invalid by a process in the subtree of the producer.

We believe that any form of backtracking – "naive" or "intelligent" – should be totally eliminated from an OR-parallel model of logic programming. Backtracking simply means complicated control and high overhead. The synchronization mechanism proposed in the Sync Model is clean and simple. Although we need extra Synchronization signals, we don't need to send the complete set of bindings and thus, the overhead is actually lower.

Our Model can be modified to handle stream parallelism as well. Extended with tail recursion optimization [6], our model becomes an efficient parallel model that exploits all kinds of parallelism

inherent in a logic program. The mapping from the Sync Model onto the Sneptree, which is chosen as the target machine for our Model, is found to have minimal mapping cost in terms of load balancing and communication overhead. Therefore, it is feasible to construct a message-passing multiprocess system based on the Sneptree architecture to implement the Sync Model effectively.

Acknowledgements

The authors would like to thank Kevin Van Horn for his valuable comments and the members of the "Thursday Meeting Group" for their carefully reading on an earlier draft.

Reference

[1] Chang, J.H., and D. DeGroot, *AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis,* Dept. of Electrical Engineering and Computer Science, Univ. of Calif, Berkeley. Sep, 1984

[2] Conery, John S., *The AND/OR Process Model for Parallel Interpretation of Logic Programs,* Ph.D. Dissertation, TR204, University of California, Irvine, June 1983

[3] DeGroot, Doug, *Alternate Graph Expressions for Restricted AND-Parallelism.* Compcom 85, Spring, pp206-210, Feb. 1985.

[4] Kowalski, R.A., *Logic for Problem Solving,* Elsevier Nother Holland Inc., 1979.

[5] Li, P., and A.J. Martin, *The Sneptree - A Versatile Interconnection Network,* 5194:TR:85, Computer Science, Caltech, 1985.

[6] Li, P., *A Parallel Execution Model for Logic Programming,* Ph.D. Dissertation, Computer Science, Caltech, 1986.

[7] Lindstrom, G., *OR-Parallelism on Applicative Architectures,* Lab. for Computer Science, Mass. Institute of Tech., Jan, 1984

[8] Martin, A.J. and J. van de Snepscheut, *Networks of Machines for Distributed Recursive Computations,* TR:84:5147, Caltech, Computer Science, 1984

[9] Nakagawa, Hiroshi, *AND Parallel PROLOG with Divided Assertion Set,* 1984 International Symposium on Logic Programming, pp22-28, Feb, 1984.

[10] Pereira, L.M., and Porto, A., *Selective Backtracking for Logic Programs,* Departamento de Informatica, CIUNL no. 1/80 University Nova de Lisboa.

[11] Ramakrishnan, R. and A. Silberschatz, *Annotations for Distributed Programming in Logic,* TR-85-15, Department of Computer Science, University of Texas at Austin, 1985.

[12] Saraswat, V.A., *Problems with Concurrent Prolog,* CMU-CS-86-100, Department of Computer Science, Carnegie-Mellon University, 1985.

[13] Shapiro, Ehud Y., *A Subset of Concurrent Prolog and Its Interpreter,* TR-003, ICOT-Institute for New Generation Computer Technology, Jan, 1983, Japan.

[14] Wise, Michael J., *A Parallel PROLOG: the construction of a data driven model,* Conference Proceeding of the Symposium on LISP and Functional Programming, pp 56-66, ACM, August, 1982.