SUBMICRON SYSTEMS ARCHITECTURE

Department of Computer Science
California Institute of Technology

1. Overview and Summary

1.1 Scope of this Report

This report is a summary of research activities and results for the nine-month period, 1 July 1992 to 31 March 1992, under the Advanced Research Projects Agency (ARPA) Submicron Systems Architecture Project. Previous semiannual technical reports and other 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 multicomputers (message-passing concurrent computers), and includes related efforts in concurrent computation and VLSI design.

1.3 Highlights

- A 256-node Mosaic C multicomputer composed of four 8×8 boards was assembled in September 1992. After the replacement of several Mosaic chips that failed within their first 500 hours of operation, this machine has been operating reliably. Extensive testing and use of this system show that it completely meets our design objectives. Six additional, prototype, 8×8 boards are currently in the manufacturing pipeline from chips that came out of fabrication in December 1992, and fabrication of one production lot of 24 6″ wafers per week commenced at the beginning of March 1993. (See section 2 and the first attachment.)
- The Modula-3D programming system, an extension of Modula-3, has been implemented on the Mosaic. (See section 3.1.)
- A family of communication, routing, and interface chips based on the Mosaic have been developed. (See section 4.1 and the second attachment.)
- The Caltech Asynchronous Synthesis Tools (CAST) are now being distributed. (See section 4.2.)

2. The Mosaic Project

The first attachment to this report, a paper titled "The Design of the Caltech Mosaic C Multicomputer," appeared in the March 1993 proceedings of the University of Washington Symposium on Integrated Systems. This paper describes the architecture, design, and programming of the Mosaic C multicomputer, and the status of the project as of December 1992.

The following sections supplement the detailed information in this paper with reports on other and subsequent Mosaic-project activities and results. In addition, research efforts that are using the prototype Mosaic multicomputers for programming experiments are described in sections 3.1, 3.2, and 3.4.

2.1 Mosaic C 8×8-board Production

Chuck Seitz, Wen-King Su, Arlene DesJardins

After sending the Mosaic wafer-test program to HP-Corvallis in the Fall 1991, and getting good results with the Mosaic C v1.1 chips; after the fabrication of a lot of Mosaic C v1.2 chips with 31% yield in April 1992; and after the delivery of the 8×8-board test system to MCC in May 1992; we expected smooth sailing for the production of Mosaic 8×8 boards. We had hoped that, having left the manufacturing to the experts, we would by now be concentrating on other efforts that are more interesting to us. Although the prototype 8×8 boards operate flawlessly in a 256-node Mosaic system that is being used for programming-system and application experiments, a series of production problems in scaling up to large Mosaic C systems has taxed our time and patience.

The first four 8×8 boards built with the first batch of Mosaic C v1.2 chips – the first to include the Elko router – were assembled into a 16×16, 256-node machine in September 1992. All four boards worked correctly initially; however, several chips failed within the first 500 hours of powered operation. Specifically, single memory bits gradually failed over a period of several hours in a mode that, in one case, a test program was able to monitor as it was occurring. This infant-mortality problem is probably confined to chips from one or two wafers that, evidently, were fabricated in a way that made them prone to gate-oxide failures in the transistors that are used as storage capacitors in the dynamic RAM. We were able to rule out drift in the dRAM timing or sense-amplifier characteristics as possible failure mechanisms. It was, however, necessary to go through a cycle of testing and rework to replace chips that failed. As a precaution, we also replaced several chips that, due to their exhibiting relatively short refresh-period requirements, we believed might fail later.

Once these failed or potentially failing chips were replaced, this 256-node, 2.7GIPS, 16MB Mosaic was placed in service for programming-system and application-programming experiments. This machine has continued to function completely reliably – no failures, and no memory or communication errors – over the past six months, and demonstrates that we have met all of our design objectives for the Mosaic C hardware.

After receiving the wafer-test results for the chips from which these four prototype 8×8 boards were built, we changed the production of Mosaic C v1.2 chips to HP-Corvallis's 6″-wafer fabrication line in hopes of getting higher yields and slightly lower cost per fabricated chip. However, the first lot of 24 6″ wafers showed very low yield in wafer test; the lot was scrapped after a subsequent physical examination by HP-Corvallis QA people indicated that there was a fabrication problem that compromised the gate-oxide layer. The first replacement lot of wafers was scrapped half-way through fabrication. The second replacement lot of chips came out of fabrication in December 1992, but also exhibited a lower yield than we had previously experienced, only 742 chips usable (24%) of the 3144 chips fabricated on 24 wafers. (Wafers yielding below a certain limit are classified as "below ship limit" (BSL), and possibly marginal, so the few tested-good chips on these wafers are lost when the wafer is scrapped.) As shown in the following tabulations, there was a large variation in yield from wafer to wafer.

TESTER S15C   DATE 22-DEC-1992   TIME 15:29:06

	As tested with BSLs	Adjusted excluding BSLs
TOTAL TESTED DIE	3144	2096
TESTED GOOD DIE	834	742
YIELD	26.5%	35.4%
TOTAL WAFERS TESTED	24	16
NET DIE/WAFER	34.8	46.4

WAFER #	GOOD PARTS	OPEN	SHORT	NOM FUNC	POST STRES	HV FUNC	LV FUNC	IDD DYN	PAD LEAK	OTHER PARAM	IDD STAT
1 BSL	0	3	10	118	0	0	0	0	0	0	0
2	55	6	3	67	0	0	0	0	0	0	0
3 BSL	14	3	5	108	0	1	0	0	0	0	0
4	63	3	1	64	0	0	0	0	0	0	0
5	63	3	2	62	0	1	0	0	0	0	0
6	37	8	4	80	0	2	0	0	0	0	0
7 BSL	17	3	1	108	0	2	0	0	0	0	0
8	54	3	1	72	0	1	0	0	0	0	0
9	46	2	1	79	0	3	0	0	0	0	0
10	39	3	3	85	0	1	0	0	0	0	0
11 BSL	19	2	10	99	0	1	0	0	0	0	0
12 BSL	13	1	3	113	0	1	0	0	0	0	0
13	42	1	4	84	0	0	0	0	0	0	0
14	42	2	4	81	0	2	0	0	0	0	0
15	31	2	4	93	0	1	0	0	0	0	0
16 BSL	17	2	8	104	0	0	0	0	0	0	0
17	60	3	2	65	0	1	0	0	0	0	0
18	42	3	2	83	0	1	0	0	0	0	0
19 BSL	0	2	6	123	0	0	0	0	0	0	0
20 BSL	12	3	6	110	0	0	0	0	0	0	0
21	21	3	8	99	0	0	0	0	0	0	0
22	58	3	3	64	0	3	0	0	0	0	0
23	55	6	2	67	0	1	0	0	0	0	0
24	34	8	5	84	0	0	0	0	0	0	0

As expected, most of the chips that fail are rejected in the "nominal-functional" test. In response to the problems we had found earlier with dRAM bits failing or requiring a short refresh period, we had added additional tests to the test sequence, and increased the speed of other tests. There were some anomalies in two tests that persuaded us to have the HP people turn these suspect tests off in order to achieve any yield on this lot. As we have done before when we have suspected testing problems, we also had HP send us a wafer (#21, the lowest-yielding non-BSL wafer) for additional testing, leaving, then, 721 chips for board production. We had MOSIS arrange to have this wafer diced, and the tested-good and a representative sample of the tested-bad parts packaged in PGA packages. Testing of these parts confirmed that the tested-good parts functioned correctly in systems, and that the tested-bad parts were, indeed, faulty.

After some additional experiments at Caltech and at HP, we concluded that there were limitations in the probe attachments in the wafer test that cause functional chips to fail certain parts of the wafer test. These chips will, however, pass these tests with the probe attachments used in the TAB test. These parts of the test program are, accordingly, omitted from the wafer test, at the cost of having to package a few chips, $\approx$1%, that will fail the TAB test.

The lot that came out of fabrication in December 1992 did provide enough good chips to start building 10 more 8$\times$8 boards. However, when the packaged chips were re-tested after the TAB inner-lead-bonding and encapsulation, only 429 of the 721 tested-good chips (60%) passed the TAB test. Most of the failures were detected as opens, shorts, and in a "dynamic-current" test. None of these tests had previously been such an important factor in the yield. By de-encapsulating the chips and studying them with a SEM, the HP packaging people concluded in February that the inner-lead bonding had physically damaged the chips; this problem was a combination of operator error and fixture-tolerance problems. By this time, there were only enough chips to build 6 8$\times$8 boards.

The last of the sequence of problems that has been delaying our scaling up the Mosaic to larger systems is that HP-Corvallis would not honor our reservation for four wafer-lot starts scheduled for January 1993. Wafer fabrication was on allocation at this time, and we were displaced by higher priority customers. After much wrangling, we finally got our slots reinstated, so that four wafer lots were started in March. We are scheduled to maintain production at the rate of four wafer-lot starts per month for the next several months.

2.2 The Mosaic C Chip

Wen-King Su, Chuck Seitz

We have been engaged in a nearly continuous effort to understand the variation in yield of the Mosaic C v1.2 chips.

One approach that we have used to search for design marginality is to correlate the variation in wafer yield across a lot with the variations in parameters from the parametric-test structures on each wafer. One such correlation was found in the September 1992 lot of 6″ wafers that was subsequently scrapped. The distribution of parameters and yield was so bimodal on this run that two important parameters, the small-transistor, n-channel threshold and current, were clearly correlated with yield. There were other correlations, such as with breakdown voltage, that is apparently indicative of a variation of n-diffusion doping, but is inconsequential in circuit behavior.

From this variation in n-channel threshold and current information, we applied Hspice circuit analysis to a number of critical circuits under variation of these parameters. The prime candidate for a circuit whose correct operation might be sensitive to small variations in these parameters is the dRAM timing generator, which provides a particular current waveform to the sense amplifiers. We have not, however, been able to simulate a sensitivity of operation of the chip to variation in these parameters. Subsequent analyses from the several earlier 4″-wafer CMOS34 runs showed no correlation between yield and process parameters.

Another approach that we have used to study yield problems is to perform diagnostic tests, followed by microscope examinations, of HP chips that fail the wafer test, or of MOSIS chips that fail our functional tests. For example, of 48 Mosaic C v1.4 chips recently received from MOSIS, only 4 chips functioned completely correctly. Of the rest:

- 1 chip passed the ROM self-tests, but failed the downloaded memory-refresh-period test.
- 27 chips failed only the ROM-resident memory test.
- 2 chips failed the ROM-resident memory test, and had bad router channels.
- 4 chips failed the ROM-resident memory test, and sometimes failed to respond to probe messages at all.
- 9 chips failed to respond to all probe messages, but the router worked.
- 1 chip failed completely, and got hot.

It is no surprise that the dRAM, which accounts for more than 90% of the active devices, shows the largest number of fabrication faults. Of the 44/48 chips that fail, at least 34 had some kind of diagnosable memory failure, and 10 had more massive failures.

Of these 34 chips, 27 worked well enough to hold test programs somewhere in their memory, so that they could provide a map of memory faults. Of the 27 chips, 12 had severe problems, visible under the microscope, that eliminated rows or columns in a bank, or worse. The remaining 15 chips contained errors in individual memory cells; a tabulation of the number of faulty bits follows:

chip #	hard-1	soft-1	hard-0	soft-0
1	2	0	0	0
2	8	2	0	0
3	1	1	0	0
4	4	3	0	0
5	2	0	0	0
6	5	4	0	0
7	2	0	4	0
8	5	0	0	0
9	6	2	0	0
10	4	3	0	0
11	7	0	0	0
12	10	1	0	0
13	8	2	0	1
14	8	0	0	0
15	2	2	0	0
total	74	20	4	1

Hard-1 or Soft-1 indicates that the bit value charges to an incorrect state; Hard-0 or Soft-0 indicates that the bit value discharges to an incorrect state. Hard errors occur immediately (within 0.2μs). The check for soft errors occurs after 10ms.

Our guess is that the predominant hard-1 mode is a gate-oxide failure that connects the n-diffusion side of the dRAM storage capacitor to the poly gate, which is at V_{dd}. The soft-1 errors could be due to leakage through the pass transistor between the storage capacitor and the bit line. The only case of a soft-0 error is curious in that the same cell also exhibits soft-1 errors.

Although our tests indicate that the Mosaic C yield problem is the result of fabrication problems rather than design marginality, we have been doing diagnostic tests and experimenting with possible improvements to the Mosaic C dRAM through Mosaic C v1.3 and v1.4 chips fabricated on two MOSIS runs. We have used these same runs to verify minor packet-interface and router refinements that had previously been verified in memoryless Mosaic chips. In this way, if we find that we can improve the yield significantly, we can include the design refinements safely in a new mask set. The only evidence of design marginality that we have discovered in extensive Hspice simulations of models of the dRAM have been what might be a marginally sized, p-channel, bitline-pullup transistor. Increasing the size of these transistors did not, however, produce a measurable improvement.

2.3 Mosaic Sbus host-interface boards

Wen-King Su, Arlene DesJardins

A small production run of the Mosaic Sbus host-interface boards – pictured on page 14 of the first paper attached to this report – was delivered to USC/ISI for use in the ATOMIC LAN.

The latest iteration of the memoryless Mosaic chip, MM3.9y, is able to receive odd-length packets, and the router will silently ignore head flits that are tagged as tails. Both of these refinements involve situations that would not occur in correctly operating Mosaic arrays, and are aimed at simplifying the use of these host-interface boards in LAN applications.

2.4 Mosaic C Supporting Software

Craig Steele

The C+− translator emits C++ source code for compilation by a GNU C compiler re-targeted to generate Mosaic assembly code. In the interest of expediency, the first version of the GNU cross-compiler restricted basic arithmetic data types to 16-bit integers. The version 2 GNU C/C++ cross-compiler supports 32-bit long integer operations inline, and IEEE-format floating-point operations by calling an assembly-language runtime library.

2.5 MADRE: The Mosaic Runtime System

Nan Boden, Chuck Seitz

The prototype MADRE (**M**os**A**ic **D**istributed **R**untim**E**) system has been completed and is functional on all Mosaic ensembles.

The modular design of the MADRE system enables system programmers to tailor the runtime system by modifying the set of processes, or handlers, included in the program that defines the runtime system. Practical Mosaic ensemble configurations range from as few as one Mosaic node to many thousands. Application programs designed to run on those ensembles span a similarly large range, with some programs requiring very little runtime system support while others rely heavily on the ability of the runtime system to distribute resource demands across the machine. Thus, matching the performance of the runtime system to the class of target ensembles and applications is particularly important for the Mosaic architecture.

The set of handlers written for the MADRE prototype system include:

- *Code handler.* The user program is partitioned into pieces that are distributed across the nodes of the ensemble. Code handlers provide access to the code pieces by retrieving the requested code piece, or by remotely executing the code.
- *Exported-Message handler.* This handler provides the capability for exporting buffered messages from a node that has exhausted the memory space available for its received-message queue. This capability is fundamental to ensuring that messages can continue to be received at the node and, thus, that the network does not deadlock.
- *Termination-detection handler.*
- *Reply handler.* This handler implements a two-trip message-passing protocol in which each user message is acknowledged by the receiving node before another user message can be sent by the sending node. The purpose of this handler is to experiment in software with different message-passing protocols.
- *User-message handler.* By varying the definition of this handler, the handling of incoming user messages can be modified to support various user-process execution strategies. For example, messages can be queued for later consumption, or can be delivered immediately to their destination process.

Implementation. The program that defines the MADRE system is written in C+−. Currently, the MADRE system loads and executes C+− user programs. The MADRE kernel occupies between 5K and 10K 16-bit words, depending on the set of handler processes that is specified.

Runtime-System Experiments. Using the prototype MADRE system, the C+− programming system, and the Mosaic ensembles, we have been conducting experiments concerning runtime-system algorithms and performance.

Process Placement. Since the MADRE system automatically assigns user processes to ensemble nodes, various algorithms for process placement have been explored. We have used three basic algorithms for selecting nodes for process placement.

- *Random.* A node in the ensemble is selected at random by computing a random number modulo the number of nodes in the ensemble. No locality between the parent and child processes is preserved using this algorithm. Placing processes purely at random provides a useful base case for the class of randomized process-placement algorithms.
- *Walk.* One of the four neighbors of the node where the parent process resides is selected at random. This algorithm in effect executes a random walk of the machine. In contrast to purely random placement, this algorithm represents the other endpoint of the locality-versus-dispersal spectrum.
- *k-biased.* A distribution on a variable k, the distance from the parent node to the selected node, is input to the runtime system at initialization. A distance d is selected according to this distribution. Currently, the distribution can be either uniform, normal, or Poisson. The mean and variance of the normal and Poisson distributions can also be varied.

We can use different algorithms for initial process placement and for process-placement failures, so that we can experiment with having MADRE place processes predominantly with one algorithm when the machine is lightly loaded, and revert to another algorithm when resources become more scarce. We have executed a variety of user programs with the runtime system employing each combination of these algorithms. The general results of these experiments indicate that the k-biased strategies perform nearly as well as purely random methods, while providing a tunable degree of locality. The complete results of these experiments will appear in Nan Boden's PhD thesis, Caltech-CS-TR-92-10.

Exporting User Messages. The capability for exporting user messages from congested nodes has been shown to be particularly important when processes were placed using methods that emphasize locality between the parent and child process. Using these methods, process-creation requests and user messages can easily exhaust the memory resources available on a node for incoming messages. By exporting user messages, the MADRE system can execute user programs to completion if memory resources can be found elsewhere in the ensemble.

3. Concurrent Computation

3.1 The Modula-3D Programming System

K. Rustan M. Leino, Jan L.A. van de Snepscheut

Modula-3D is an effort to explore the possibilities and limitations of safe, high-level programming notations on fine-grained multicomputers. Most programming of multicomputers has been done in languages to which send and receive operations were added, forcing the programmer to deal with details of each communication.

Object-oriented notations generally facilitate a higher level of abstraction and expressiveness from which the multicomputer programmer can benefit. However, present object-oriented notations do not provide the right set of constructs to allow for an efficient implementation, because there is no way to express locality of data.

We have based our experiments on Modula-3 because it is a language that already supports object-oriented and concurrent programming. This allows us to focus on the aspect of distribution of data rather than starting from scratch. Furthermore, Modula-3 is geared to allow implementations to feature a garbage collector, which we deem essential since memory management is an important issue in the programming of fine-grain multicomputers. Finally, the semantics of Modula-3 are well-defined, its type system is safe (with unsafe modules allowed), and the conciseness of its defining report compares favorably with that of other languages.

Language Extension. In Modula-3, (traced) object types form a hierarchy descending from the built-in object type `ROOT`. We extend the language with a new object type called `NETWORK`, a subtype of `ROOT`. When a new object is created, it can be created on any node of the multicomputer if it is a subtype of `NETWORK`, and on the local node otherwise.

Methods may be invoked on any object and their semantics are independent of the where the object resides. However, to allow for efficient implementation on multicomputers, fields of network objects can be accessed via methods only. Furthermore, parameters or return values of network object methods should not contain any references other than network objects (that is, no pointers, no other objects).

Implementation. Given the above extension and restrictions, calls of network-object methods can efficiently be implemented via remote procedure calls. Therefore, the implementation maintains for every network object type two method suites instead of one: the regular one that is used in case the object resides locally, and another one that implements the communication protocol for remote procedure calls.

We have implemented Modula-3D on the Mosaic multicomputer. The Modula-3D compiler has been derived from the Modula-3 compiler provided in the public domain by DEC SRC. This is a machine-independent compiler that compiles Modula-3 into C. Our implementation has first been changed to (drastically) reduce the size of the object code generated by the compiler. A completely new runtime system had to be written. Next, the language was extended with network objects, and additions to the runtime system were implemented. In particular, this includes the protocol for remote procedure call, and a distributed garbage collector.

Most of the implementation, including the runtime system, has been written in Modula-3D. Four functions were written in C and 30 were written in Mosaic assembly language; roughly half of these are for reporting runtime errors, and are three instructions long. The present system has a few implementation restrictions (such as a limit on the size of the return value of a method) but is otherwise completely functional. We have written and run some test programs, including a LISP interpreter in Modula-3D, and we are currently analyzing performance statistics. We need more experience with the system to comment on its expressivity.

3.2 Program transformation

Jan L.A. van de Snepscheut

Developing programs through transformation is a programming method that has been widely advocated, but not so widely practiced. The idea is to start with an "obviously correct" but probably inefficient program, and then transform it by stepwise application of semantics-preserving transformation rules. Although it is attractive in principle, it has not been attractive in practice for two reasons:

1. It is a lot of work because of the large number of steps required.
2. Most transformation rules are valid only under certain conditions that may not be easy to check.

Our goal is to use the computational power offered by modern (multi)computers to help solve both problems. We are developing an editor that maintains the current program text. It keeps a history of the program's development and carries out the transformation steps. Checking the validity of transformation steps is a restricted form of theorem proving. We aim at developing a system in which these restrictions are such that no interaction with the programmer is needed for verifying the conditions, much like the type checking done by a compiler. It is mainly in this area where we expect to benefit from a lot of computational power.

The present prototype supports transformation of functional programs and of proofs. Checking of applicability conditions is still far from complete. We are presently expanding the program and experimenting with different sets of transformation rules.

3.3 Refinement of Concurrent Programs

H. Peter Hofstee, Jan L.A. van de Snepscheut

At present, properties of communicating sequential processes (CSP) are proved either by proving properties of corresponding nondeterministic sequential programs, or by using a process algebra. Using nondeterministic sequential programs has the disadvantage that one needs to reason about atomicity of the statements, even when the processes that are described do not share data. Process algebras have the disadvantage that there are many laws, whereas no model is given in which these laws can be proven.

The present work attempts to give a model that is an extension of trace theory. The model describes choice and sequential and parallel composition. The basic statement is a communication action. The space of concurrent programs combined with a refinement relation for this model forms a lattice. Iteration is defined as a fixpoint of a monotonic function. The theory is compositional, that is, if a component A refines a component B, then any component in which B occurs may be implemented by a component in which B is replaced by A. The theory gives rise to a process algebra, but its laws can be proven within the model. Present efforts concentrate on extending the processes with state.

3.4 mcc on the Mosaic

Marcel van der Goot, Alain Martin

We have created a version of the mcc compiler that generates code for a Mosaic multicomputer. The compiler generates C code for each of the Mosaic nodes, as well as for the Sparc host. The C program for the Mosaic is then further compiled with the gcc version described in an earlier report.

Since, until now, the mcc compiler had been targeted only for medium-grain multicomputers, porting mcc to the Mosaic provided an interesting test to see whether it was feasible to use mcc on a fine-grain multicomputer. As far as the size of the code is concerned, the answer is positive: The runtime system needed by mcc takes approximately 3500 words (about 800 words of which are standard functions that are not part of the mcc system proper); that is roughly 11% of the total memory of a Mosaic C node. The mcc language itself has some advantages for fine-grain computers, such as light-weight processes, and, in particular, rendezvous-style communications, which limit message-buffering requirements. At this moment, we cannot yet say how the speed of the generated programs compares with programs written in other languages for the Mosaic.

The Mosaic processor is rather different from other processors for which mcc has been installed. Hence, this experiment provided a good test of the portability of the compiler. In particular, there are many differences between data representations on the Mosaic and on the Sparc host. These differences are important for the compiler, because, in mcc, all nodes are treated uniformly: messages can be sent from a process on the host to a process on a Mosaic node, and any process can instantiate other processes on the host. As a result, most of the difficulties in porting the compiler were related to the conversions between data representations. In the process of porting the compiler, a number of hidden assumptions about the data representation were found. These assumptions had, until then, gone undiscovered because of similarities between nodes. (Several problems were related to the difference between the smallest addressable units: 16-bit words on the Mosaic, and 8-bit bytes on the Sparc.) As a result, the portability of the compiler has been improved. At the same time, better documentation of the installation process was written.

The Sparc version of the mcc compiler has also been used by the students of a class on concurrent computing, which provided a useful test of its robustness. It withstood this test quite well, and we expect to make the (portable) compiler available for distribution in June.

4. VLSI Design

4.1 The Routing-Chip Project

Wen-King Su, Chuck Seitz

After gaining confidence in the new Elko mesh router in the Mosaic and memoryless Mosaic chips, we began making separately packaged routers available to people who like to use our routers in their projects. The most common mesh-routing chip is the EMRC-2D8, a 2D, 8-bit configuration similar to the router used in the Mosaic. The Elko router's modular layout style and internal signaling conventions made it simple for us to create variations of the standard router. The variants currently available are:

- *EMRC-2D9* has one extra data bit in each channel, and is packaged in a 160-pin QFP. It was designed initially for the MCC high-performance modules, which required an extra bit for parity.
- *EMRC-1D16-DASH* is a 1D, 16-bit router that is pin-compatible with the "Frontier" routers in the Stanford DASH multiprocessor, and is packaged in the MOSIS 132-pin PGA package. These chips are normally used in pairs to implement a 2D-mesh router in which the 1D routers strip the head flit before passing a packet to the next 1D router or to the node. These routers were tested in the Stanford DASH multiprocessor. In spite of the improved setup- and hold-time margins in comparison with the Frontier routers, these chips exhibit communication errors in the DASH. We believe that these errors would be eliminated by additional Vdd/GND pins and/or better packaging.
- *EMRC-1D16-CHPC* is another 1D, 16-bit router, but is packaged in a 160-pin QFP, and does not strip the head flit before passing a packet to the next router or to the node. Instead, a 2D router is implemented by swapping the left and right bytes at the router outputs, and word alignment is retained in the packet output. This router was designed to fit the requirements of the WPI Center for High-Performance Computing GalacticaNet multiprocessor.
- *EMRC-SP* uses an EMRC-2D8 core together with pipeline synchronizers on the *p* channels and error-detecting circuits on the request/acknowledge links of the *news* channels. This router was designed for a commercial licensee that is using this router in a database multicomputer, in which continuous operation in the presence of network faults is crucial.
- *ERIC-8* (Elko Router Interface Chip) is not a router, but is assembled from modules lifted directly out of the Elko router. It provides a FIFO-buffered, synchronous interface between the Elko router channel and an 8-byte (72-bit) microprocessor bus. The first ERIC-8 chips went to MCC for use in their high-performance modules.

The same factors that simplify the creation of variants also made it easy for us to improve the original EMRC-2D8 design by replacing modules with more advanced modules without re-wiring a large portion of the router. One example is the replacement of the module that generates arbitration requests, such that, if the first flit of the packet happens to have the tail bit set, the packet is silently absorbed. This module was developed and checked in the EMRC router in the memoryless Mosaic chip, and this design improvement migrated to the other variants the next time they happen to have been fabricated.

The specifications, signaling protocols, and internal design of the EMRC-2D8 and of the other chips in this family are described in an attached paper.

4.2 CAST, Caltech Asynchronous Synthesis Tools: The First Release

Alain Martin, Drazen Borkovic, Marcel van der Goot, Tony Lee, José Tierno

The first stage of the project on asynchronous VLSI circuit synthesis is now ending. We have developed a complete method for the synthesis of asynchronous and delay-insensitive circuits. The method produces designs that are both correct by construction and efficient. In fact, we claim that most of the synthesized designs are more efficient than equivalent circuits produced by seat-of-the-pants approaches.

We have demonstrated the method by designing a large suite of circuits, the most interesting of which are a complete asynchronous microprocessor (the first one ever designed) in CMOS and in GaAs, a memory-management-unit, several multipliers, and several asynchronous static RAMs (also the first ones ever designed). We are now working on a data cache to be incorporated in the next version of the microprocessor.

Another important result of the method is the introduction of a whole set of standard building blocks for asynchronous circuits that are now widely used, in particular the Q-element for sequencing, the synchronizer, the various dual-rail registers, left-right buffers, a standard dual-rail adder, completion trees, isochronic forks, *etc.*

The method relies on the concept of synthesis by program transformations, and is therefore begging for automatic compilation tools. We designed the first automatic compiler in 1987, based on the notion of syntax-directed compilation, *ie,* it provided a standard implementation for each construct of the language. Although the results were encouraging, we became convinced very soon that an entirely automatic compiler would not easily produce circuits as efficient as the ones obtained when the method is applied "by hand."

Hand-designed circuits are optimized using two main techniques that are difficult to incorporate into an automatic compiler: (1) by using global semantic properties of the program (describing the circuit) and of the environment, and (2) by searching through the state space of possible solutions for an optimal solution. Such a search requires that several solutions be generated at each stage of the compilation, and that backtracking be possible, since a solution that may be considered the best at one level of the compilation may turn out not to be the best at the next level.

We therefore switched from automatic compilation to what we called *Designer-Assisted Compilation.* The designer is provided with two sets of tools, one for synthesis, and one for performance analysis and optimization. Each synthesis tool corresponds to one step of the synthesis procedure. Each tool can, however, be applied independently, and can be used in such a way as to produce all possible solutions. The solutions can then be compared by applying the analysis and optimization tools. One or several solutions can be selected for the next step of the compilation. The designer may also decide to bypass one step and provide a solution for that step that has been compiled by hand — for instance, with the use of information not available to the compiler. At some point, the designer may decide to abandon one solution that was so far considered the best (according to some figure of merit related, for example, to speed or power consumption), and backtrack to a solution rejected at a previous stage.

This approach requires more expertise from the designer, of course, but produces excellent circuits. CAST is the suite of tools supporting this approach. Although the suite of tools is not complete — we are missing a good layout program for datapaths, and the top transformations down to the so-called handshake level still have to be done by hand — we decided that it was time to make this first generation of CAST available to the designers in the ARPA community that are already familiar with the method, or are willing to learn the method and design some wonderful chips!

The diagram of Figure 1 gives an overview of the relationships between the different tools of CAST. The programs marked with an asterisk are not part of CAST but should be available as they are used together with the tools in the manner indicated on the figure. All programs with names of the form $x2y$ are conversion programs transforming data in format x to data in format y.

We have written a User's manual for CAST. The manual is available as a Caltech Computer Science Technical Report CS-TR-93-09, which is available by ftp. The attached paper *The Design of a Delay-Insensitive Multiply-Accumulate Unit,* presented at the last Hawaii International Conference on System Sciences in January 1993, and to be republished in a special issue of *Integration*, describes how the tools have been used in a medium-size design.

The following is a short description of CAST in its present state.

- prgen is a logic-synthesis program that takes a handshaking expansion of a straightline program (no if-statements) and produces a set of production rules. The program automatically inserts state variables when necessary, and generates production rules that are minimal in terms of the length of the longest transistor chain.

- prif is similar to prgen but allows if-statements. However, the handshaking expansion has to be *safe,ie,* state variables are not needed to identify each state uniquely. State-variable introduction, when necessary, must be done by hand or with prgen prior to the use of prif. Reset circuitry is automatically inserted.

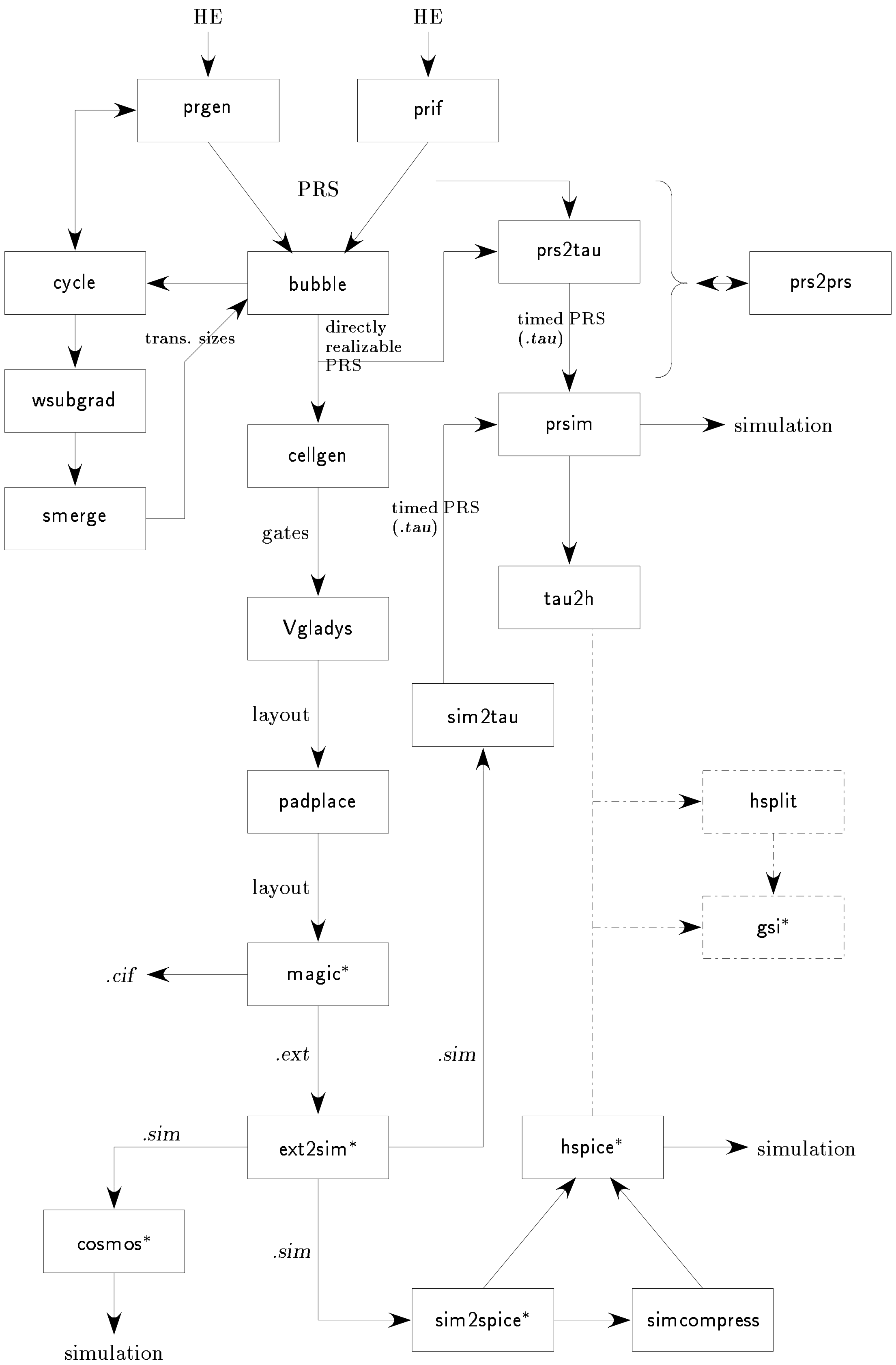
- bubble is also a logic-synthesis tool. It optimizes the placement of inverters in the production-rule set generated by one of the previous programs in particular in connection with the isochronic forks present in the circuit, and with the requirements of CMOS complementary logic. It also uses the performance-optimization tools to generate sizes for the transistors.

- cellgen, Vgladys, and padplace are used for layout synthesis. cellgen produces a CMOS layout for each gate or state-holding element corresponding to the set of production rules that set and reset a given variable—the output of the gate. Vgladys assembles the generated cells and adds wires for the interconnections. The length of the longest wire can be minimized by a simulated annealing technique. padplace generates a padframe and routes to the pads. This form of layout generation is excellent for control circuits, but doesn't exploit the regularities present in datapath layouts. The layout produced by the programs is in the format used by magic.

- cycle, wsubgrad, and smerge are used for performance analysis and optimization. cycle inputs a closed set of handshaking expansions or production rules, and computes its period under one of the three available delay models. wsubgrad optimizes the delay in the critical cycle as a function of the transistor sizes. The result is a set of transistor sizes. The program smerge includes the transistor sizes into the production rules to generate a set of sized production rules.

- prsim is an event-driven simulator that executes a set of timed production rules, under one of different timing models, producing a list of timed firings.

A version of CAST for GaAs is internally available and will be released later.



–Figure 1: The CAST tools and their interactions–

4.3 Performance Analysis and Optimization of Disjunctive Systems

Tony Lee, Alain Martin

One of the major advantages of CAST is the ability to perform timing analysis at different stages of the synthesis. Such an analysis requires that a choice be made about the run-time behavior of the computation. At the moment, such a requirement makes it difficult to analyze disjunctive systems, *ie,* systems where the different occurrences of a transition may be "caused" by different sets of events. These systems arise from the compilation of programs containing if-statements, or programs where some transitions occur more frequently than others (*eg,* a toggle).

We have been investigating methods to extend our tools to analyze the performance of disjunctive systems in a straightforward manner. Our approach is to require that a fixed environment be given for each system, convert the system into a non-disjunctive one for which our current tools can be applied, and perform the timing analysis for the new system. The result of the analysis is then an indication of the performance of the original system for the given environment.

We have developed an efficient algorithm for converting a disjunctive system into a non-disjunctive one. Currently, we are writing codes for the implementation of this algorithm and its incorporation into CAST.

4.4 Gallium Arsenide Asynchronous Microprocessor

Jose Tierno, Alain Martin

Our previous semi-annual technical report described the design of the first asynchronous GaAs processor. The design was intentionally very conservative, and, as a result, the performance was "only" 70MIFS (million instruction fetches per second). During this nine-month period, the GaAs microprocessor was redesigned, laid-out, and fabricated. The logical structure was mostly preserved; however, the technology mapping is completely different. We used our experience with the GaAs chips that we fabricated and tested previously to select the circuits that perform best, in terms of power and speed, for each of the different parts.

In terms of the type of circuits used, the microprocessor is divided in three parts: control, datapath, and register file. The datapath is, from the electrical point of view, the simplest. Most cells are purely combinational, and a few are registers. The main concern while designing the datapath was keeping the power dissipation low. This was achieved with not much of a penalty in speed.

The control circuitry includes instruction decoding, synchronization of the pipeline, condition-code calculations, register decoding, etc. The functions to be implemented are generally more complex. The design emphasis is on the correct construction of these functions, and tries to minimize gate delay. As for power consumption, the fraction of the total power dissipated by the control is relatively small, about 20% of the total; some extra power was spent to increase speed.

The register file incorporates a novel sense-amplifier design, and a very compact layout for the cell. These two improvements make it possible to include, in the same area as the previously fabricated processor, twice the number of registers dissipating half the power. A slice of this register file has already been successfully fabricated and tested.

This new design was submitted to the December 1992 HGAAS3 run. Because of the change of technology from HGAAS2 to HGAAS3, some of the circuits in the processor had not been tested in the new technology, only simulated. The processor and a small memory were simulated together using Hspice. The results for this simulation give, taking into account process variation, a speed between 200 and 270MIFS. Most instructions are one word; branch and immediate instructions are two words.

Power dissipation is as follows: The core of the processor takes 2W, the padframe plus the external pull-down resistors take another 2W. The padframe was designed to be compatible with ECL parts, to make testing and prototyping easier. This choice is costly in terms of speed and power dissipation, and requires a very delicate adjustment of the pad drivers and receivers. With the change in technology, new pads had to be designed, and adjusted using Hspice. A direct connection using GaAs levels would be much preferable, in terms of speed, power, and reliability, but very few standard parts are available with GaAs levels, and off-chip level conversion is prohibitively expensive.

A first batch of these chips was packaged and tested, and found non-functional. The measured power dissipation for this batch was almost twice that predicted by Hspice. The problem was traced to a fabrication flaw. Apparently, the wafers used for this batch were different from the wafers Vitesse normally employs, resulting in a out-of-spec threshold voltage for transistors. This accounts for the increased power dissipation and the non-functionality. A second, corrected batch is ready now, and will be packaged this week. Testing results will be available shortly thereafter.

4.5 Semantic Issues in VLSI Synthesis

Marcel van der Goot, Alain Martin

According to Carver Mead, a design goes to fabrication after extensive simulation with an average of one hundred errors left. It takes an average of eight runs to uncover and correct most of the errors, and the product is shipped to the customer with an average of four errors left! Most of these errors are related to timing. One doesn't need to be a great expert to realize the enormous costs to the society such an approach causes, since, in the end, the customer pays for these mistakes. It also stifles the creativity of the engineers by discouraging them from departing significantly from their previous, painstakingly almost-debugged, designs.

The possibility to design circuits that are correct by construction has therefore enormous economical consequences, and is not just a justification for theoretical investigation.

The basis of our synthesis method for VLSI design is the notion "semantics-preserving transformations." An initial CSP program, the correctness of which is relatively easy to establish because of its succinctness, is transformed into a circuit by applying a series of transformations. Each transformation produces a program semantically equivalent to the previous one. The transformations occur in several phases: some phases involve conversion from one language to another; others involve optimizations without changing the program notation. Although many working circuits have been built using the synthesis method, relatively little work has been done on proving formally that the transformations are, indeed, semantics-preserving. This project, which is still in its initial stage, is an attempt to formalize the transformations and to prove their correctness.

As a first step, the notion "one program implements another program" must be defined. "Implements" differs from normal program equivalence in two respects: First, it is a weaker condition, in that the implementation only needs to retain certain "relevant" aspects of the original program. Second, the programs may be in different languages, each with their own semantics, so that comparison of the programs is not straightforward. To formalize what the relevant aspects of a program are, we introduce the notion of an interface between a program and its environment. Program and environment can only observe each other's behavior (and hence interact) by observing the interface. Hence, the meaning of a program is the way in which the program changes the interface, and an implementation must change the interface in a compatible manner. This description in terms of interfaces is suitable for both sequential and parallel compositions.

We have chosen operational semantics to define the programming languages we use. (A common method to do so is through G. Plotkin's Structural Operational Semantics.) In operational terms, an interface can be seen as a data structure that is shared between environment and program. In order to compare the interfaces of a program and its implementation, we introduce interpretation functions that relate the values of the interface data structure to values appropriate to the programming language. For instance, a high voltage can be interpreted as a boolean true. Interpretation functions do not solve all problems related to the use of different languages. Even with interpretation functions, the descriptions of changes to the interface are not necessarily comparable. Currently, we are studying methods to give more uniform semantic descriptions of the languages, so that comparison becomes possible. One method might be to use P. Mosses's Action Semantics, a variant of operational semantics.

4.6 A Program to Check Stability and Non-Interference

James Cook, Alain Martin

Our synthesis method produces asynchronous circuits that are hazard-free by construction. Absence of hazard is guaranteed by the fact that the production rules are *stable* and *non-interfering*. The programs prgen and prif produce production-rule sets that are stable and non-interfering by construction.

However, we have mentioned that we leave the possibility open for the designer to include their own set of production rules if so desired. In that case, it may be interesting and prudent to provide a tool for the designer to check whether the proposed production rules are indeed stable and non-interfering.

The non-interference property of these circuits states that during normal operation a correct circuit should never simultaneously enable both up and down transitions for any variable (wire). Stability states that whenever a transition becomes enabled, it must stay enabled until the transition is complete (and the output wire has reached logic zero or one). Although such properties can be checked by simulation, simulations must utilize some sort of model of the delays in the circuit's implementation. The timing assumptions in such models can hide design flaws that would invalidate these properties. We have built an early prototype program to verify these properties without assumptions about timing, and we are now developing the program for eventual incorporation into CAST.