The Architecture and Programming of a Fine-Grain Multicomputer

Thesis by

Jakov N. Seizović

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1994

(Submitted August 20, 1993)

Acknowledgments

Many thanks to all the great teachers I've had, in particular: To Chuck Seitz, my research advisor, for knowing when to be a friend and when a tough guy, for his willingness to be both a student and a teacher, for pointing out the obvious and insisting on simplicity. To Jan van de Snepscheut, Mani Chandy, and Eric van de Velde, members of my thesis-defense committee, for teaching me, through their example, that one must never stop learning. To Milenko Cvetinović, for giving me all the right books to get excited about. To Ilija Stojanović, Dejan Živković, and Slobodan Ćuk, for believing in me before I did.

Many more thanks to my fellow students: To Nan Boden, for her dedicated support and critique, depending on what I needed on any particular day. To Wen-King Su, for the long hours we spent working together, and for sharing his insights into the wonderful world of programming. To the late Mike Pertel, for teaching me about stamina in times good and bad. To Bill Athas, Don Speck and Craig Steele, for helping me keep both feet on the ground. To Tony Lee, for being the best office-mate there is.

Thanks also go to my research sponsors: To the Advanced Research Projects Agency, whose program managers repeatedly demonstrated a remarkable ability to balance researchers' vanities against the need for inter-project cooperation. To IBM, for their orientation towards the future, embodied, in part, in numerous student-support programs.

To the entire Caltech Computer Science Department staff, and to Arlene DesJardins in particular, for humming busily in the background of my cocoon.

And to my family and friends who made sure that it wasn't "all work and no play . . . "

To Goga

Abstract

The research presented in this thesis was conducted in the context of the Mosaic C, an experimental, fine-grain multicomputer. The objective of the Mosaic experiment was to develop a concurrent-computing system with maximum performance per unit cost, while still retaining a general-purpose application span. A stipulation of the Mosaic project was that the complexity of a Mosaic node be limited by the silicon complexity available on a single VLSI chip.

The two most important original results reported in the thesis are:

- The design and implementation of C+-, a concurrent, object-oriented programming system.

  Syntactically, C+- is an extension of C++. The concurrent semantics of C+- are contained within the *process* concept. A C+- process is analogous to a C++ object, but it is also an autonomous computing agent, and a unit of potential concurrency. Atomic single-process updates that can be individually enabled and disabled are the execution units of the concurrent computation. The limited set of primitives that C+- provides is shown to be sufficient to express a variety of concurrent-programming problems concisely and efficiently.

  An important design requirement for C+- was that efficient implementations should exist on a variety of concurrent architectures, and, in particular, on the simple and inexpensive hardware of the Mosaic node. The Mosaic runtime system was written entirely in C+-.

- Pipeline synchronization, a novel, generally-applicable technique for hardware synchronization.

  This technique is a simple, low-cost, high-bandwidth, high-reliability solution to interfaces between synchronous and asynchronous systems, or between synchronous systems operating from different clocks.

The technique can sustain the full communication bandwidth and achieve an arbitrarily low, non-zero probability of synchronization failure, P_f, with the price in both latency and chip area being $\mathcal{O}(\log \frac{1}{P_f})$.

Pipeline synchronization has been successfully applied to the high-performance inter-computer communication in Mosaic node ensembles.

Contents

Chapter 1

Introduction

1.1 Concurrency and VLSI

Progress in microelectronics technology during the past four decades has been remarkable by any measure. Three major factors contributed to this progress: (1) a rapid and steady pace of improvements in processing technology to produce ever smaller, faster, and lower-power devices; (2) the development of design methodologies [62] and tools to manage design complexity; and (3) the exploitation of concurrency [75]. The first two factors are readily understood. The importance of concurrency to the performance/cost ratio of VLSI systems can be understood from results of VLSI-complexity theory [97], and has been demonstrated repeatedly in practice.

Special-purpose computing engines were the first to employ concurrent solutions, and continue to do so, highly successfully, to this day [50, 11]. Although various forms of concurrency (bit-level parallelism, pipelining, vectorization) are exploited regularly in general-purpose computing engines [37], applying concurrent solutions to general-purpose computing at the application level has been slower in gaining ground.

A considerable effort has been made to exploit the concurrency that is implicit in sequential programs. This effort has been successful in discovering and utilizing modest degrees of concurrency, but is now regarded almost universally as having approached its limits [49]. Applications with explicitly concurrent formulations are the driving force for a range of concurrent architectures, some of which are discussed in the following section.

1.2 Concurrent Architectures

Most of today's concurrent computers are representatives of one of the following three architectures [33]:

- Computers with a single instruction stream and multiple data streams (SIMD).
- Two variants of computers with multiple instruction streams and multiple data streams (MIMD):
  - multiprocessors, which have one global address space, and
  - multicomputers, which have multiple local address spaces.

Early concurrent-computer implementations closely followed this classification: SIMD computers employed multiple computing units to which instructions were broadcast [40]; multiprocessors utilized buses and/or switches to connect multiple processors to the global memory [69, 24]; multicomputers featured independent processor-memory pairs interacting through a message-passing network [76].

The differences between the more recent representatives of these three architectures [96, 12, 53, 1, 64, 79] are blurred: When observed from a point that is sufficiently close to the hardware, or from a point that is sufficiently far away from the hardware, these three architectures are remarkably similar. Each consists of a communication network connecting a collection of computing nodes. Each node consists of one to several instruction-interpreting processors, a local memory, and a network interface. All three architectures support some concept of processes — computing agents that execute concurrently, and that can communicate data and synchronize activities with each other. What were once architectural distinctions became differences in programming style: data-parallel [70], shared-memory [18], and message-passing (Chapter 2) programming abstractions. Depending on the emphasis on support for one of these abstractions, additional architectural support is provided for global synchronization [96], for efficient non-local memory access [53, 1], or for low-latency, low-context-switch-overhead message handling [64, 79].

In this thesis, we shall focus principally on the multicomputer variety of MIMD computers, but shall indicate also how our results apply to multiprocessors. Multiprocessors provide hardware support for a global address space, and implement inter-process communication through shared-memory access, whereas multicomputers provide a generic message-exchange mechanism, and implement shared-memory access through inter-process communication.

Given the similarity between these two architectures, one might expect that, given a typical problem-set load of the computer, it would be easy to test and decide which architecture and which machine should be employed. Yet, the reality of concurrent computers and concurrent-programming systems is somber: programs are most often written in notations tailor-made for a particular computer architecture, sometimes even for a particular machine. The cost-effectiveness of program execution on concurrent machines is their main advantage over their sequential counterparts [75]. Striving to maximize this cost-effectiveness, however, emphasizes the concurrent computer's main disadvantage: the complexity of writing programs.

1.3 Concurrent Programming

There are two, typically conflicting, driving forces shaping the developments in concurrent programming: increasing efficiency and increasing expressivity.

The efficiency-conscious programming systems are typically the products of design teams also involved with the design of concurrent machines, and often reflect the underlying architecture. Shared-memory programming and explicit-message-passing programming are representatives of this class.

The expressivity-conscious programming systems are often produced by the frustrated users of the products of the former groups, and are typically architecture-independent (Section 1.3.3).

1.3.1 Shared-Memory Programming

The first developments in concurrent programming were motivated by the advent of multiprogramming and multiuser operating systems. It should not, therefore, be surprising that the first concurrent-programming systems supported concurrent processes that communicated and synchronized through the memory of the machine on which they were executing. The development of the Parallel RAM (PRAM) model, a theoretical framework on which much of the work in concurrent algorithms is based, also promoted the popularity of this programming style, which is still the predominant form of concurrent programming.

From the early stages on, shared-memory programming has been plagued by various incarnations of the mutual-exclusion problem. This problem is due primarily to the discrepancy in access granularity between the data structures and the memory units used to represent these data structures. A number of remedies were introduced: atomic test-and-set and/or fetch-and-add instructions [35], and semaphores [28]. One of the most significant efforts was the work of Per Brinch Hansen on Concurrent Pascal, and the development of *monitors* [36, 41]. Monitors encapsulate data with the (mutually-exclusive) operations defined on the data in programmer-defined, compiler-and-runtime-system-managed units. This work forms a foundation on which many of the recent developments in object-oriented concurrent programming are based, including the programming system described in this thesis.

1.3.2 Explicit Message Passing

Communication and synchronization through explicit message passing is a programming paradigm whose roots are as old as computers themselves, stemming from the need for inter-computer information exchange. This programming paradigm was adopted and adapted for programming multicomputers [42, 77]. Starting with the Cosmic Cube [76] and its commercial descendents [78, 54], the mainstream representatives of the multicomputer architecture employ off-the-shelf processor, memory, and compiler technology. Programming systems for these machines are based on a variety of sequential programming languages for specifying individual process behavior, wherein communication and synchronization between processes is achieved through a set of library routines.

There are two problems that are the curse of this programming style. First, although modular organization of data structures can be achieved within a process, this modularity does not extend readily to collections of processes. Second, the off-the-shelf technology often brought the off-the-shelf notion of process granularity; heavy, UNIX-style processes impose an unacceptably high software overhead to process communication and synchronization.

1.3.3 Architecture-Independent Programming

A number of programming models and notations have been devised to provide a uniform view to the programmer of concurrent computers, and to map computations onto either of the architectures described above. The advantages that these programming systems offer in reducing programming effort are remarkable; preserving the cost-effectiveness of concurrent computers running such programs, however, has yet to be demonstrated. The assembly programming of conventional, sequential computers has been all but eliminated by higher-level notations through large improvements in program-writing efficiency, with small degradations of program-execution efficiency. The same has yet to happen to tailor-made concurrent-programming notations.

Functional Programming and Dataflow

In its pure form [6], functional programming provides a method for defining functions in terms of other, more-primitive functions. The value of a function is determined only by the value of its arguments, and is not history-sensitive. Since there are no side effects, functional-programming notations are implicitly concurrent, and sub-expressions, including function arguments, can be evaluated independently of each other.

The introduction of side-effects into functional-programming notations enables them to model history-sensitive behavior, but it also opens them up to the full set of problems associated with imperative-programming notations. Extending pure functional programming with single-assignment variables and streams, as introduced by dataflow researchers, represents an important intermediate point. This extension relaxes the no-side-effects requirement into the *monotonicity* requirement: A variable starts up uninitialized, and an assignment bounds it to a value (multiple assignments are disallowed). A stream consists of a (possibly-infinite) sequence of variables that can only be read and appended. Using single-assignment variables for communication and synchronization is also used extensively in compositional programming [21, 19], and in concurrent logic programming, described next.

Concurrent Logic Programming

The programming model typically associated with sequential logic programming is that of proving an existentially quantified statement given a program that consists of a set of axioms [90]. Implementations of this model involve backtracking that could, in principle, be replaced by concurrent examination of all the alternatives. However, for efficiency reasons, and because of the need to better model input/output behavior [93, 83], concurrent logic programming makes a significant departure from this model: There is no backtracking; once a (non-deterministic) choice is made, no alternatives are examined.

A concurrent logic program consists of a set of guarded clauses, and each clause represents a recursive specification of process structures. To program in a concurrent logic programming notation is to specify tasks as unordered, concurrent sets of subtasks. Tasks communicate and synchronize with each other by binding single-assignment variables, and waiting for variables to become bound.

Restrictions on the expressivity of clause guards, to improve efficiency, lead to a family of *flat* concurrent-logic notations [93]. A minimalist approach to concurrent logic programming of Ian Foster and Stephen Taylor resulted in Strand, a streamlined and efficient concurrent-programming system [34], without giving up much of the expressive power.

UNITY

UNITY, developed by K. Mani Chandy and Jayadev Misra [20], is a computation model and a programming notation, with an associated proof methodology. A UNITY program consists of a set of guarded multiple assignments. These assignments are executed in arbitrary order: The focus of programming in UNITY is on *what*, *i.e.,* on data transformations, as opposed to *when.* A particular execution order can be enforced only through data dependencies. A computation terminates when it reaches a fixed point, *i.e.,* when no assignment in the program modifies any variables.

An interesting related research has been reported by Craig S. Steele [89]. In this work, a programming model and a corresponding notation are developed, in which program actions are associated with data objects through a programmer-specified triggering mechanism. An efficient multicomputer implementation of this UNITY-like programming system is demonstrated.

Actors

The Actors model of computation was first proposed by Carl Hewitt and Henry Baker [38, 39], and was later formalized by William D. Clinger [22] and Gul Agha [3]. In this model, the unit of concurrent computation is an *actor*, an independent computing agent that is activated in response to messages sent to it. Each actor has a unique address, an associated message queue, and a specified behavior. In a response to a message, an actor can: *send* messages, create *new* actors, and *become* a new actor by specifying its replacement behavior.

Because of its simplicity, potential efficiency, and straight-forward implementation on distributed architectures, the Actors model is the basis for numerous concurrent-programming systems. The reactive-process programming model, described next, and its associated notation, described in Chapter 2, are based in part on the Actors model of computation.

1.4 The Reactive-Process Programming Model

The reactive-process programming model is a variant of the Actors programming model. Computation in this model is performed by a set of *processes*, independent computing agents. A process is normally at rest, and starts executing in response to a *message* (including the initial, creation message). In the course of its execution, a process can send messages, create new processes, and modify its state, including self-termination. Message order is preserved for each pair of processes in direct communication. Each message is marked with a *tag* that specifies which of the process's compile-time-fixed set of *entry points* should be invoked. Each entry point runs to completion, and is therefore an atomic update of its process's state. A process can affect the order of execution of its entry points by enabling and disabling them selectively, at run time; all entry points are initially enabled. A message tagged for a disabled entry point is delivered when (and if) that entry point is active again.

This model is extended to include the *remote procedure call* (RPC). An entry point of a process can be specified to return a value to the message sender. When a message is sent and tagged for such an entry point, the sender is suspended until the message with the returned value arrives.

Background

The reactive-process programming model is a result of the work in our research group over the last decade. Interestingly, a comparison with the early work of C. R. Lang on a concurrent version of Simula [51] reveals that our group's ideas seem to have come almost full circle. The ideas of C. R. Lang, and the preceeding work of Per Brinch Hansen, were far-sighted and out-of-sync with the multicomputer technology of their time. In retrospect, it is as if much of what our research group has been doing was tracking and driving the necessary communication, processor, memory, and compiler technology to approach this target.

Starting with the development of the Cosmic Cube, our group embraced the explicit message-passing programming style. The design of an experimental fine-grain multicomputer, Mosaic C, and the similarity of our approach to the Actor model of computation, provided additional motivation; this effort culminated with the work of W. J. Dally on Concurrent Smalltalk [26]; of W. C. Athas and N. J. Boden on Cantor, a minimalist Actor-based notation [4, 9]; and of W.-K. Su on Reactive-C and distributed event-driven simulation [91]. The work on the Cosmic Environment [91] and the Reactive Kernel [82] shifted our focus from organizing computations around processes to organizing computations around messages, and the reactivity became an essential part of the programming model.

1.5 The Mosaic C Project

The work described in this thesis attempts to make a contribution to the un-stately condition of concurrent programming today. Our work is based on the following principles:

- Concurrency must come cheap: Concurrent machines must be extensible in small and inexpensive chunks.
- Concurrency must come cheap: Only those high-level programming constructs for concurrency that can be implemented efficiently by the (small and inexpensive) hardware can be used.
- Concurrency must come cheap: Expressing concurrency in programs must be simple.

Starting from these simple and restrictive requirements, our research group has been conducting a computer-architecture experiment to design a concurrent-computing system with as high a performance/cost ratio as possible, while still retaining the (not so well-defined) "general-purpose" application span. In the course of this experiment, we have built a testbed consisting of: a fine-grain multicomputer, the Mosaic C; a concurrent-programming notation, C+-; and a distributed runtime system, MADRE.

What computer architecture is all about is bridging the gap between a programming model and a technology [37, 77]. Designers of all of today's computers divide this complex spanning task into a set of smaller, more manageable subtasks. A particular choice of subtasks is illustrated in Figure 1.1. Well-defined anchor points

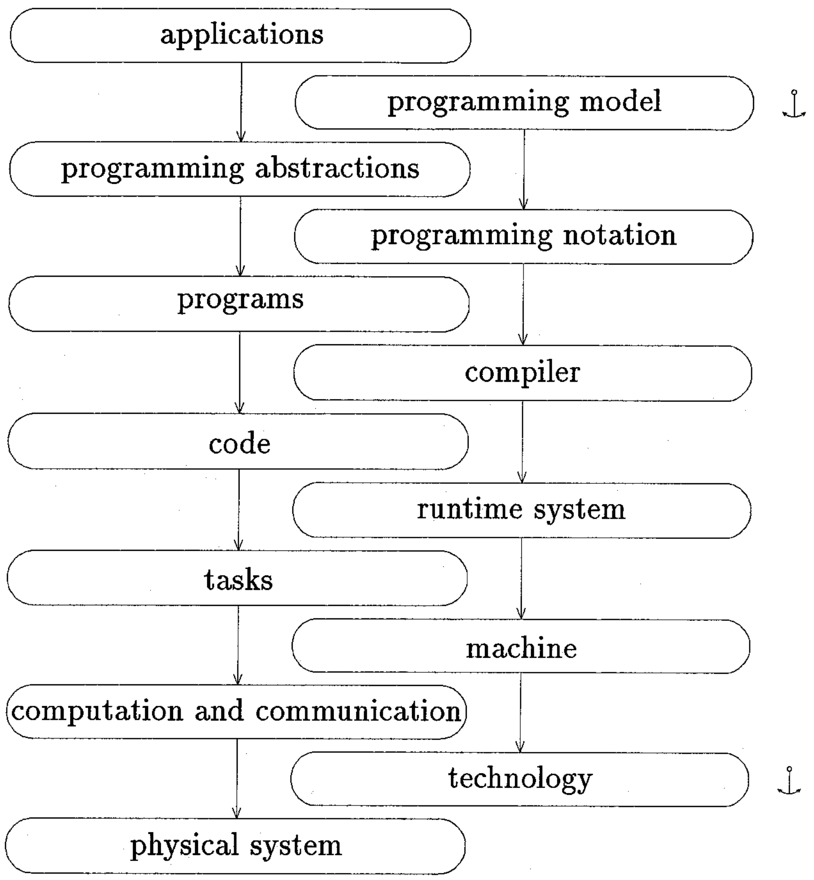


Figure 1.1: Computer architecture

simplify the implementation of the subtasks; however, if those anchor points are too numerous and/or too rigid, the design space may become severely restricted.

For the Mosaic experiment, the fairly rigid anchor points have been:

- the scalable CMOS VLSI technology, because that was the best technology that we both had access to and understood well, and,
- the reactive-process programming model, because we believed we could implement it efficiently.

For each of the remaining intermediate points, we have been able to identify opportunities for what appeared to be significant improvements over existing concurrent-computing systems. In particular:

- The *programming notation* was to be a derivative of a widely-accepted object-oriented programming notation, trying to leverage off of the advances that object-oriented programming brought to sequential-programming systems.
- The *compiler* technology was mature enough that we believed a compiler could perform much of the checking traditionally done at run time, if at all.
- The *runtime system* was to be fully distributed, and utilize the distributed-queue algorithm described in [10].
- The *machine* was to benefit from the results in router technology [81], dRAM technology [87], synchronization methods (Chapter 5), packaging technology [79], and self-testing possibilities [79].

Our computer-architecture experiment culminated in designing and building the Mosaic concurrent-computing system. Parts of the Mosaic system are described in [79, 81, 87, 10], and in this thesis.

In as closely-coupled a research team as ours, it is difficult to properly grant every bit of credit, but, to the first order, the following is the list of contributions to the Mosaic project. Most of the work has been done by five people: Nanette J. Boden, Charles L. Seitz, Don Speck, Wen-King Su, and the author of this thesis. Nan has been involved principally with the runtime system, but was also invaluable to the development of the programming model and of the notation, and was the most understanding and thorough tester of the compiler and of the machine. Chuck did most of the overall-machine architecture, the processor architecture, the system integration and packaging, and the router design. Chuck also had important contributions to every other aspect of the project, and the commitment to see it all the way through. Don designed the dRAM and ROM, and taught all of us the value of patience and thoroughness. Wen-King designed most of the processor and router, the workstation interfaces, and worked on all aspects of verification and packaging. Wen was also a principal contributor to the programming model, wrote a low-level-but-works-reliably-the-very-first-time programming system, and re-targeted the Gnu C and C++ compiler to the Mosaic. The author of the thesis feels he deserves credit only for saying: "Why don't we get this thing finished!", and for then going off to fill in the missing links: routing-network interface, chip-level system integration and verification, the programming notation, and the compiler. Numerous other people have been associated with the Mosaic project, most notably: William C. Athas, who contributed to the programming model and to the processor architecture, and Michael J. Pertel, who contributed to the choice of the routing network and to its performance evaluation.

1.6 Overview of the Thesis

In Chapter 2 we introduce C+-, a concurrent object-oriented notation based on C++. Chapter 3 defines the C+- runtime-system interface, illustrates how C+- can be customized, and explains the C+--compilation process. In Chapter 4, a brief description of the architecture of the Mosaic multicomputer is presented, with emphasis on architectural support for C+-. Chapter 5 presents a novel, generally applicable, synchronization technique, along with a proof of its correctness, and its application to the Mosaic. In Chapter 6, we compare the results of our work with the related research, and suggest possible future research directions.

Chapter 2

C+−

2.1 Introduction

2.1.1 Object-Oriented Programming *vs.* Concurrency

Programming notations that support object-oriented programming techniques are the notations of choice for a rapidly growing number of complex applications. Indeed, not since the introduction of structured programming [25] has there been such a degree of unanimity in the programming community. This unanimity is even more remarkable considering that, just as was the case with structured programming [29], the power of object-oriented techniques is difficult to convey to readers through short, example programs in books or articles. When observed in isolation, none of these techniques is new or revolutionary. It is only when one approaches a large-scale programming task armed with the full set of techniques that their power becomes evident.

Structured-programming techniques advocate structuring of program control flow in a top-down, compositional fashion. Object-oriented programming techniques promote data organization in a bottom-up, standard-parts fashion. Both paradigms emphasize modularity, but, whereas the former is focusing principally on modularity of control structures, the latter does a better job of encapsulating data structures with the operations defined on these structures.

Object-oriented programming came about through attempts to make large, sequential programs more manageable. Techniques such as data encapsulation and access protection, inheritance, and guaranteed initialization, all emerge from the goal of helping programmers help themselves.

By our view, much of what the techniques of object-oriented programming are really helping to manage is *concurrency.* Events are concurrent if they are unordered, *i.e.,* if they can occur in any order, or in parallel. Mutual exclusion is an example of an issue most often associated with concurrent programming, but the problems that result from a disregard for mutual exclusion also occur regularly in large sequential programs. With uncontrolled access to global variables, it is impossible to keep track of all of the places in the code where a certain variable is accessed, and of all the invocations of such code. Non-deterministic execution is another issue most often associated with concurrent programming. For a fixed set of inputs, the execution of a sequential program will always result in the same ordering of state changes, yet, with side effects on global variables, it is often far from obvious what all the inputs to a program are.

Whereas sequential programming brings out the worst in us only in the large, concurrent programming will do that already in the small. It should not be surprising, then, that in the hope of reaping some of the benefits that object-oriented techniques brought to sequential programming, we are witnessing a proliferation of programming systems trying to amend a particular object-oriented notation with concurrent semantics.

2.1.2 Concurrent Object-Oriented Languages

The three-way design tradeoffs illustrated in Figure 2.1 are typical of design of any programming system, not only those attempting to harness concurrency. However, all three requirements are more pronounced, and the balance more difficult to achieve, for a concurrent-programming system:

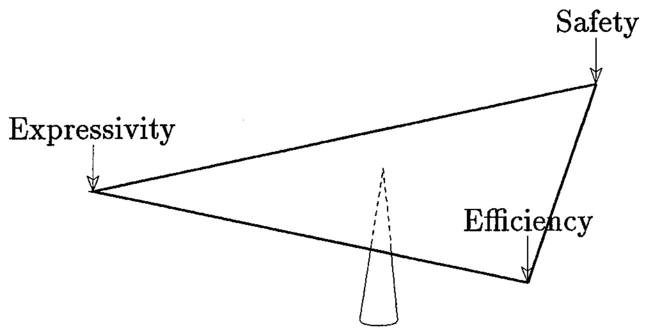


Figure 2.1: Design tradeoffs for concurrent programming systems

- *Efficiency* — One of the major reasons to employ concurrent solutions in the first place is to get more performance, and programming-system overheads are less likely to be tolerated by users.

- *Expressivity* — Moving from a single to many threads of control, and the requirement that threads can communicate and synchronize their activities, place additional demands on expressivity.

- *Safety* — In addition to mutual exclusion and possible non-determinism mentioned in the previous section, issues such as deadlock and livelock have to be dealt with. Simple semantics that aid correctness proofs are essential.

It is likely that some readers will find what we consider a balanced design to be biased in favor of efficiency, then expressivity, and then safety. Our argument about the increased importance of efficiency in a concurrent-programming environment is sometimes disputed on grounds that, because concurrent systems offer better performance/cost than their sequential counterparts, one can afford more inefficiencies at the operating/runtime system level. The consequence of this view on concurrent architectures is that machines with pathetic process-creation and communication overheads are being designed and built. The major goals of the work described in this thesis are to show that this pitfall can be avoided, and to demonstrate that fine-grain concurrency can be efficiently exploited.

Extensions of C++

C++ is an object-oriented notation that is in widespread use due to its efficiency, availability, and upward compatibility with C. C++ is the starting point for numerous programming systems that attempt to amend C++ with concurrent semantics, including the system described in this thesis.

A comparison of our work to related concurrent-programming systems can be found in Section 6.1.

C+-

C+- is the result of an experiment to express reactive-process, concurrent programs (Section 1.4) in an object-oriented programming notation. Although C+- is an extension[1] of C++, the objective of the C+- project has *not* been to be able to execute arbitrary C++ programs efficiently on the Mosaic. The emphasis of C+- is on providing efficient support for the simple abstractions fundamental to the reactive-process computational model: process creation and communication. C+- strives not to impose higher-level policies on synchronization, communication protocols, or process placement.

Although the C+- programming system is portable across a wide range of architectures, the Mosaic has been both the driving force and the reality test behind this effort. Design decisions have consistently been made to avoid compromising the performance of C+- programs on the Mosaic. Higher-level programming systems may be layered on top of C+-, but C+- is intended to serve as the Mosaic's lowest-level, *workhorse* programming system, suitable both for operating-system and application programming.

The remaining sections of this chapter are devoted to teaching the reader about C+-. Familiarity with the basic concepts of object-oriented programming and of C++ in particular is assumed: classes, inheritance, access rules, operator overloading. Keywords are underlined in programming examples. Although an effort has been

[1]C+- is not a superset of C++ because it imposes restrictions on global variables, as discussed in Section 2.3.

made to steer clear of the idiosyncrasies of C++, some of them were essential, and they are explained as they are encountered. The reader is cautioned, however, that C+- is by no measure a minimalist, toy-example-writing notation; some of the more advanced examples are likely to present difficulties to those not familiar with C++. Our hope is that this difficulty is the result of C+-'s completeness, rather than of poor design choices.

2.2 The Process Concept

The C++ *object* concept is carried over intact to C+-: `class` is a user-defined type; an object created according to a class definition is a collection of data items, a set of operations defined on them, and a set of access rules (Program 1). Class member functions have the usual, sequential semantics.

```
class   C
{
private:
        int     data;
public:
                C()             { data = 0; }           // initialization
        void    write(int i)    { data = i; }           // update
        int     read()          { return(data); }       // retrieve
};
```

Program 1: A Class Definition

The *process* concept is the only extension that C+- introduces to C++. The `processdef` keyword parallels the `class` keyword syntactically (Program 2). Access rules are associated with data members and functions of a process definition, and process definitions can be derived from other process definitions (Section 2.4.1).

However, a process created according to a process definition is more than a collection of data items:

Specification 1 A process is an independent computing agent, and a unit of potential concurrency. Its public interface consists of a set of atomic actions.

```
processdef      P
{
private:
        int     data;
public:
atomic          P()             { data = 0; }           // initialization
atomic  void    write(int i)    { data = i; }           // update
atomic  int     read()          { return(data); }       // retrieve
};
```

Program 2: A Process Definition

At creation time, the process *constructor*[2] is executed if it is defined. After the constructor completes, the process is at rest. The invocation of an atomic action of a C+- process is decoupled from its execution. Conceptually, there is an infinite queue of incoming requests for each process; the invocation of an atomic action places a request into this queue. Process execution consists of servicing these requests, with each atomic action running to completion.

Creating a process is no different from creating an object (Program 3). In most cases, processes are created dynamically (`pp = new P;`), and persist until they are explicitly destroyed (`delete pp;`). One can also create a temporary process as a local variable, just as with any other type (`P p;`). This temporary process is destroyed implicitly when execution leaves its scope.

A C+- computation is initiated by a runtime system that, concurrently with initialization of global processes, creates an instance of `root` (Program 4), the constructor of which is defined by the user.

Specification 2 A process can affect the order of execution of its atomic actions by enabling and disabling them selectively, at run time. All atomic actions are initially enabled; execution of a disabled action is postponed until the action is enabled again.

For example, let us assume that the rules for accessing a process of type `P` in Program 2 are such that it may be updated only once; every subsequent `write` request

[2]A process constructor is an atomic action with the same name as that of the process definition. The constructor may not return any value.

```
{
    int  i;                 // declaring an integer
    P*  pp;                 // declaring a process pointer

    pp = new P;             // creating a persistent process
    i = pp->read();         // retrieving a value
    pp->write(i+1);         // updating
    delete pp;              // explicitly destroying the persistent process

    {
        P  p;               // declaring a temporary process

        i = p.read();       // retrieving a value
        p.write(i+1);       // updating
    }                       // implicitly destroying the temporary process
}
```

Program 3: Programming with Processes

```
processdef      root
{
public:
atomic          root(int argc, char** argv);
};
```

Program 4: The `root` process

should be tagged as an error. Furthermore, all `read` requests occurring before the first `write` should be serviced only after the first update occurs. The process definition for this version of `P` is listed in Program 5.

Processes communicate and synchronize with each other through atomic actions. Thus far, we have discussed only the behavior of processes as servers — how they deal with incoming requests (invocations of their atomic actions). We shall now define the behavior of processes as clients — how they request services from other processes:

Specification 3 When invoking an atomic action that does not return a value (returns a `void`), or if the returned value is not used, the caller continues execution independently of the callee. The order of invocations is preserved for each pair of processes in direct communication. If the value returned by an atomic action is used, the caller may be suspended until the returned value is available.

```
processdef      P
{
private:
        int     initialized;
        int     data;
public:
atomic          P();
atomic  void    write(int);
atomic  int     read();
};

atomic          P::P()
{
    initialized = 0;
    passive read;
}

atomic  void    P::write(int i)
{
    if ( initialized )
    {
        report_error();
    }
    else
    {
        data = i;
        initialized = 1;
        active read;
    }
}

atomic  int     P::read()
{
    return(data);
}
```

Program 5: Enabling and Disabling Atomic Actions

Invoking an atomic action that returns a value does not, in itself, imply that the requesting process will be suspended until the requested value is available. It is only when this value is *used* that a thread of activity must be suspended. For example, the Program 6 uses a divide-and-conquer approach to compute the n^{th} Fibonacci number. Both sub-computations are initiated, and the process will suspend only if it attempts to add the two partial results before they are available.

It is sometimes desirable to enforce the sequential order of execution of

```
processdef      fib
{
public:
atomic  int     compute (int n)
                {
                    switch (n)
                    {
                        case    0:   return  0;
                        case    1:   return  1;
                        default:     fib  f1, f2;
                                     return  (f1.compute(n-1) + f2.compute(n-2));
                    }
                }
};
```

Program 6: Divide And Conquer

subcomputations. In such cases, the C+- `await` construct should be used. For example, `return (await(f1.compute(n-1)) + f2.compute(n-2));` ensures that the first subcomputation is complete before the second one is initiated.

Programming systems differ considerably in what constitutes use of unresolved variables, also called *futures*. The most aggressive systems allow futures to be exchanged between processes, and suspend a thread only when a value is needed for a hardware-implemented expression evaluation. Support for futures is the central issue for numerous concurrent-programming systems [44, 86, 99]. C+- is not one of these systems, and is not very aggressive in trying to discover and utilize this type of concurrency. *In C+-, assigning an unresolved value to any programmer-defined variable constitutes use of that future, and will cause the thread to be suspended. C+- guarantees only that a thread will not be suspended unnecessarily within an expression evaluation.* C+- semantics allow any additional compiler/runtime system optimization, but only within the body of a function or an atomic action. Unresolved variables must be resolved before they can be passed as arguments.

The reason for C+-'s non-aggressive utilization of futures is that we want to encourage a programming style in which the concurrent behavior is generated explicitly, as opposed to trying to utilize the concurrency that is implicit in sequential formulation. Synchronization on an unresolved future is inherently more expensive than, for example, synchronization using the active/passive semantics, because the process state that must be saved when blocking on a future is much larger. For notations that have stack-based implementations of the regular function-call abstraction, such as C+-, this state includes the stack.

2.3 Managing Concurrency

All concurrency-related issues in the C+- programming system are encapsulated into the process concept. The following syntactic restrictions enforce this requirement:

- Only atomic actions can be `public` members of a process definition.[3]
- Only values, process pointers, and process references[4] can be arguments to atomic actions.
- Processes are the only global[5] variables allowed.
- Process definitions can have no `friends`.[6]

As specified in Section 2.2, a process is a unit of potential concurrency. Processes communicate and synchronize with each other through atomic actions. The remainder of this section will be devoted to examples illustrating how some of the well-known concurrent-programming paradigms can be implemented in terms of C+- processes.

2.3.1 Remote Procedure Call

The remote procedure call (RPC) is a common form of interaction between threads of activity. As illustrated in Program 7 and in Figure 2.2, a client requests a service from a server and suspends its execution until the request has been attended to.

[3]The C++ `static` member functions can be public members of a process definition, since their semantics do not allow them to access process members anyway.

[4]The difference between pointers and references is a subtle idiosyncrasy of C++, and, for the purposes of this thesis, the two can be considered equivalent.

[5]This includes both global and `static` C++ variables, *i.e.,* all variables with file scope.

[6]The `friend` construct in C++ allows non-member functions to have full access to `private` class members.

The semantics of the RPC are identical to those of an ordinary procedure call. The implementations of the two types of procedure calls, however, are typically different, because the client and the server may be operating in different address spaces. A better name for the RPC might be "interprocess procedure call."

```
processdef      server
{
public:
atomic  int     request (int);
};


processdef      client
{
public:
atomic          client (server* s)
                {
                    int  i = s->request(123);
                }
};
```

Program 7: Remote Procedure Call

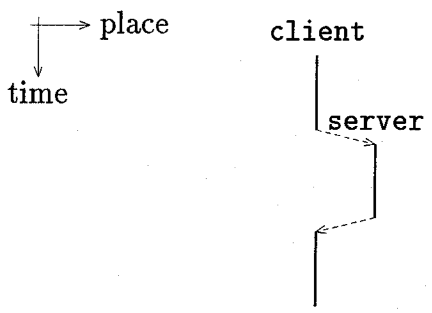


Figure 2.2: Remote Procedure Call

During a remote procedure call, the calling process is nominally suspended until the returned value is available, so no concurrency is introduced. However, as discussed in Section 2.2, with the use of futures, the semantics of the RPC can be extended so that several requests can be issued concurrently, and the calling process is suspended until all the requests have been serviced (Program 6 and Figure 2.3).

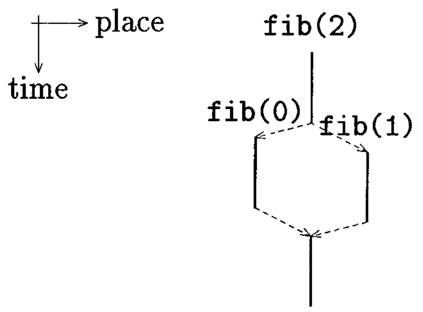


Figure 2.3: Divide And Conquer

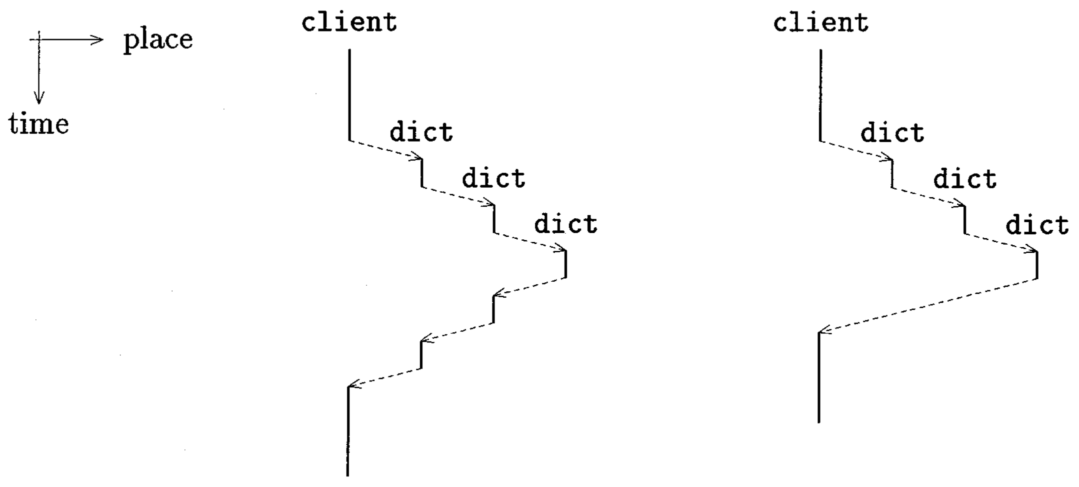
2.3.2 Call Forwarding

Call forwarding is a paradigm associated with message-based object-oriented programming systems, and is similar to tail recursion. As an example, consider the sequential search of a singly-linked list of dictionary processes in Program 8.

```
processdef      dict
{
private:
        dict*   next;
        int     index;
        int     data;
public:
atomic  int     find (int i)
                {
                    if ( i == index )
                        return  data;
                    else
                        return  next->find(i);    // can be replaced by:
                //      forward  next->find(i);
                }
};
```

Program 8: A Sequential Search

When the value returned from an atomic action is itself obtained by an atomic action invocation, programmer may choose to use the `forward` statement instead. With the `return` statement, a request is issued, the process is suspended until the value is available, and then reply is sent to the calling process. The effect of call forwarding is to defer servicing of the request to another process. Two sequential search examples, one using the `return`, and another the `forward` statement, are illustrated in Figures 2.4 (a) and (b), respectively. In addition to reducing the number



(a) (b)

Figure 2.4: A Sequential Search with RPC (a), and with Call Forwarding (b)

of replies, call forwarding enables the list of processes that form a dictionary to process multiple requests in a pipeline fashion. At any point in time, each search request is being worked on by at most one dictionary process.

2.3.3 Fork-Join

The remote-procedure-call mechanism with limited support for futures, as provided by C+-, offers a convenient and easy-to-understand programming paradigm for an important class of problems. A more flexible, fork-join mechanism for process synchronization in C+- is offered through the combination of non-suspending, atomic-action invocation and active/passive semantics.

There are two paradigms that C+- programmers can use to generate concurrent activities:

- *Creating new processes,* whether persistent or temporary. The parent process continues execution independently[7] of the child.

[7]When a pointer to a newly created process is used in a subsequent computation, this may or may not require suspending the parent, depending on the implementation. However, the parent continues execution concurrently with child's constructor.

- Upon *invoking an atomic action that does not return a value,* or when the returned value is not used, the caller continues executing without waiting for the callee.

The synchronization barriers can be expressed using active/passive semantics. Suppose that an FFT computation is implemented as illustrated in Figure 2.5 [65]. The expressions along the edges of the graph are coefficients. Multiple inputs to a

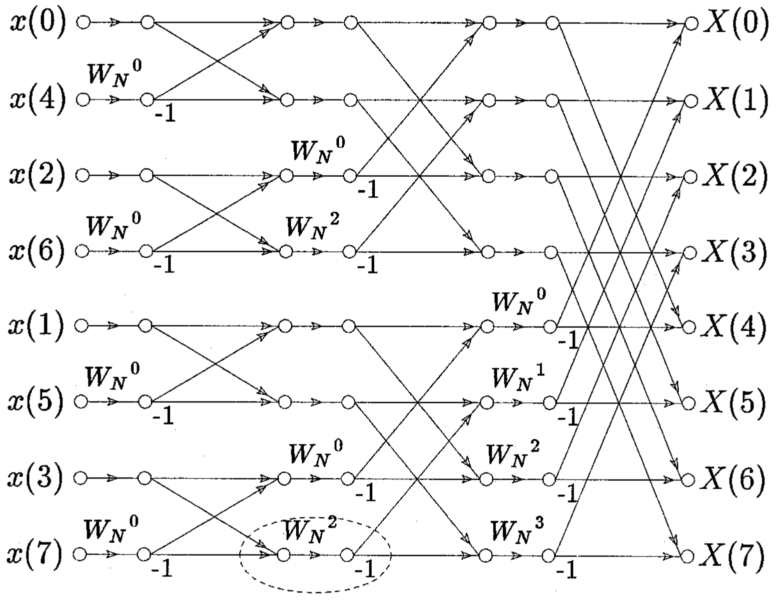


Figure 2.5: An 8-Point FFT Computation. ($W_N = e^{-i\frac{2\pi}{N}}, N = 8$)

node imply addition, and multiple outputs imply replication of the result.

A concurrent program for N-point FFT computation could employ N processes, and compute the result in $\mathcal{O}(\log N)$ steps. Each step would consist of: getting two requests along the input edges; adding the two input values; multiplying by the coefficient; and producing two output values.

A version of this program could similarly employ $N \log N$ processes in a pipeline regime, achieving the same $\mathcal{O}(\log N)$ latency, but a new result would be computed on every step.

In either approach, though, a process (circled in Figure 2.5) must get one data item along each of its input edges to be able to compute and emit one data item along each of its output edges. A process that might be used as part of the FFT-computation pipeline is listed in Program 9.

```
processdef      fft
{
private:
        Complex W, first;
        fft     *out_up, *out_dn;
        void    output(Complex in)
                {
                    Complex  result = (first + in) * W;
                    out_up->up(result);
                    out_dn->dn(-result);
                }
public:
atomic          fft(fft* u, fft* d, Complex r)
                {
                    W = r;
                    out_up = u;
                    out_dn = d;
                }
atomic  void    up(Complex in)
                {
                    if ( passive(dn) )          // upon receiving both requests
                    {                           // produce the output
                        active dn;
                        output(in);
                    }
                    else                        // if you only have one request
                    {                           // await the second one
                        passive up;
                        first = in;
                    }
                }
atomic  void    dn(Complex in)
                {
                    if ( passive(up) )          // upon receiving both requests
                    {                           // produce the output
                        active up;
                        output(in);
                    }
                    else                        // if you only have one request
                    {                           // await the second one
                        passive dn;
                        first = in;
                    }
                }
};
```

Program 9: An FFT-Computing Process

2.3.4 Semaphores

First introduced by E. W. Dijkstra [28], semaphores are low-level primitives for process synchronization. A semaphore is typically used to control access to a shared data structure, with an N-ary semaphore allowing access to at most $N-1$ processes at any point in time. Two operations are defined on semaphores: *acquire* and *release*. In general, an implementation of an N-ary semaphore must guarantee that the number of acquire operations minus the number of release operations is at most $N-1$, and at least 0. A C+- implementation of an N-ary semaphore is presented in Program 10.

```
processdef      semaphore
{
private:
        int     count;                  // number or processes inside
                                        //  the critical section
        int     max;                    // the maximum number allowed
public:
atomic          semaphore(int N)        // initially, there are no
                {                       //  processes inside the critical
                    max = N - 1;        /   section
                    count = 0;
                    passive release;
                }
atomic  int     acquire()
                {
                    count++;            // one more inside
                    active release;     // at least one can release
                    if ( count == max ) // if the maximum is reached,
                        passive acquire; //  no one can get in
                    return 1;
                }
atomic  int     release()
                {
                    count--;            // one less inside
                    active acquire;     // at least one can acquire
                    if ( count == 0 )   // no one is in, so
                        passive release; //  no one can exit
                    return 1;
                }
};
```

Program 10: N-ary Semaphore

An often-used special case for $N = 2$, the binary semaphore, is illustrated in Program 11.

```
processdef      semaphore
{
public:
atomic          semaphore()
                {
                    passive release;
                }
atomic  int     acquire()
                {
                    active release;
                    passive acquire;
                    return 1;
                }
atomic  int     release()
                {
                    active acquire;
                    passive release;
                    return 1;
                }
};
```

Program 11: Binary Semaphore

2.3.5 Monitors

Of all of the concurrent-programming paradigms, semantics of C+- processes are closest to those of monitors [36]. Just as with monitors, C+- processes encapsulate a set of data items and offer mutually exclusive access to a set of routines operating on this data. C+- processes also share some of the problems associated with monitors, as both are non-reentrant. The invocation of an atomic action of a C+- process is, unlike an invocation of a monitor function, decoupled from its execution: conceptually, there is an infinite buffer of incoming requests for each process. This decoupling enables processes to be active computing agents, able to affect the order of execution of their atomic actions.

2.3.6 Recursion

In the examples shown so far, the requirement that all the `public` member functions of a process be atomic actions has been helpful in expressing interactions between concurrent threads of activity. From the point of view of C+- programmers, the most significant repercussion of the atomicity of interprocess activities is that, since at most one execution thread can be associated with a process, atomic actions that return values are not reentrant. For example, in Program 12, the `private` member function `fac` has ordinary, sequential, reentrant semantics. However, the `public` member function `FAC` must be an atomic action. An invocation of `FAC` will, therefore, result in deadlock.

```
processdef      bad
{
private:
        int     fac(int n)
                {
                    if ( n == 0 )
                        return  1;
                    else
                        return  n * fac(n-1);   // OK: functions are reentrant
                }
public:
atomic  int     FAC(int n)
                {
                    if ( n == 0 )
                        return  1;
                    else
                        return  n * FAC(n-1);   // ERROR: atomic actions are
                }                               // not reentrant
atomic  int     Fac(int n)
                {
                    return  fac(n);             // OK: atomic-action interface
                }                               // to a function
};
```

Program 12: Recursive Functions and Non-Recursive Atomic Actions

In the world of non-reentrant atomic actions, processes are the medium used to express recursive behavior (Program 13).

2.3.7 Message Passing

Invoking an atomic action of a process is equivalent to wrapping up the argument list and sending it in a message. According to Specification 3, the atomic-action invocation does not imply blocking (waiting for the reply does), so it is equivalent to a non-blocking message send.

```
processdef      fac
{
private:
        int     output;
public:
atomic          fac(int input)
                {
                    if ( input == 0 )
                        output = 1;
                    else
                    {
                        fac  child(input-1);
                        output = input * child.result();
                    }
                }
atomic  int     result()
                {
                    return  output;
                }
};

// or

processdef      fac
{
private:
        int     input;
        fac*    parent;
public:
atomic          fac(int i, fac* p)
                {
                    if ( i == 0 )
                    {
                        p->result(1);
                        delete this;
                    }
                    else
                    {
                        input  = i;
                        parent = p;
                        new fac(i-1,this);
                    }
                }
atomic  void    result(int r)
                {
                    parent->result(input*r);
                    delete this;
                }
};
```

Program 13: Recursive Processes

Message receiving has two forms:

- *explicit*, associated with the behavior of processes as clients, which receive a value that is returned from a call to an atomic action; and
- *implicit*, associated with the behavior of processes as servers, which receive an argument list as part of a request to execute an atomic action.

The two forms of receive, explicit and implicit, cover the two extremes of the spectrum of possible mechanisms for message discretion: explicit receive accepts only a particular message from a particular process; implicit receive accepts any message from any process. The `active/passive` semantics provide a more general selective-receive mechanism: atomic actions of a process represent incoming communication channels, and the process can, at run time, select the communication channels over which it is ready to accept a message.

2.3.8 Single-Assignment Variables

Single-assignment variables are a safe form of futures (Section 2.2). Requesting a read access on an uninitialized, single-assignment variable causes the requesting process to be suspended until the variable is assigned to. Since there can be at most one assignment to a single-assignment variable, these variables can be effectively cached. Processes of type `P` in Program 5 are an example of a possible C+- implementation of single-assignment variables.

2.3.9 Process Aggregates

Thus far, we have described processes as independent entities, and have emphasized the code-execution aspects of processes. In this section, we shall show how processes can be treated as instances of a restricted data form, one that can be accessed only through a set of mutually exclusive, atomic actions.

As illustrated in Program 14, C+- programmers can treat processes as variables of any other type. Whether a process is a local variable, member of an object or of

```
processdef      P
{
    // ...
};


class   C                   // an object of class C contains:
{
public:
        P       p;          // a process
        P*      pp;         // and a process pointer
};


{
    P  p1, p2;              // declare two processes

    p1 = p2;                // process assignment

    P  p[10];               // declare a process array
}
```

Program 14: Treating Processes As Data

another process, element of an array, or used in any other way in which a variable can be used in C++, the process semantics are the same. According to the syntactic restrictions described in Section 2.3, the only operations allowed on a process are to take its address and to access its public members (all of which are atomic actions).[8] The various process usages determine only when a process is created and when it is destroyed. For non-process data types, variable usage also implies what the memory layout is. When accessing processes, one cannot assume, for example, that a process declared as a local variable resides on the stack; nor can one assume that a process that is a member of a class is placed in memory next to the other data members. In Section 3.1.1, we shall discuss how programmers can affect process-placement strategy.

The semantics of C+- are defined such that efficient implementations exist for both mainstream variants of MIMD computers: multiprocessors, which have one global address space, and multicomputers, which have multiple local address spaces. In C+-,

[8]Process assignment is an atomic action invocation, equivalent to issuing a request to the source process to send a copy of itself to the destination process (Section 3.1.5). Passing processes as arguments is a form of assignment.

regardless of the underlying architecture, a pointer to a process can be dereferenced globally, since it contains sufficient information to uniquely identify the process it points to.

An important advantage that multiprocessors have over multicomputers is that they can employ most of the data-layout strategies developed for sequential computers. There are additional performance considerations guiding the design decisions on the data layout, as discussed in [46]. If, for the time being, we neglect such performance considerations, a vector of C+- processes could, on a multiprocessor, be laid out in memory in the same way as a vector of elements of any simple data type. Elements with successive indices would reside at memory addresses that differ by a stride equal to the size of the process. This approach would allow the programmer to compute the address of any process in the vector given the address of any other process in the same vector, and the two corresponding indices.

On a multicomputer, using the above layout strategy for vectors of processes is unacceptable for two reasons: first, the address space of a multicomputer is contiguous only within each multicomputer node, so the maximum size of a process vector would be limited by the size of node memory; and second, although the computation model allows elements of a process vector to operate concurrently, that concurrency could not be used to a performance advantage, because the elements would all reside on the same node.

This example is but an instance of a more general problem of naming constituent elements of distributed objects [26, 17]. There are two issues that are central to the solution of this problem. The first issue is that there should exist a single name (address) of a distributed object, and a way of addressing constituents given this name. The second issue is that the programmer should be able to compute on references, not just store them at process-creation time and fetch them when they need to be used.

A simple solution that takes only the first issue into the account could employ an address-manager process. The manager's address would represent the address of the distributed process as a whole. All the requests would be directed to this process, and then forwarded to appropriate constituent processes. This solution obviously introduces an access bottleneck, but may be acceptable for element processes that exhibit a large ratio of computation/communication.

We consider this problem to be too important to be left to *ad hoc* approaches, particularly for such often-used paradigms as arrays of processes. Accordingly, C+- offers a runtime-system-supported mechanism for address management that preserves the C++ address-computation semantics.

The example in Program 15 shows that the creation of a process array consist of

```
{
    processdef  P       { };

    P*  p = new P[123];         // is equivalent to:

    {
        P*  p = unique__CPM(123,sizeof(P));
        for (int i=0; i<123; i++)
            new @(p+i) P;
    }
}
```

Program 15: Creating A Vector of Processes

two stages. First, a set of unique references is allocated by invoking the `unique__CPM` function, with arguments specifying how many references are required, and what the stride between the adjacent references should be. This function returns a pointer of the generic process-pointer type, `pointer_t`, analogous to `void*` in C++. Next, the actual process creation is requested, specifying that each new element process be placed in such a manner that it can be located through the given pointer. A description of various flavors of process creation is presented in Section 3.1.1. A set of algorithms that provide efficient support for process placement and lookup is described in [10].

2.3.10 Summary

The programming examples in Section 2.3 illustrate that a small set of mechanisms supported by C+- is sufficient to express a variety of concurrent-programming paradigms. This set consists of: process creation, asynchronous request, synchronous request (remote procedure call), and selective servicing of requests (active/passive mechanism). In Chapter 3, we shall present an implementation framework for this set of mechanisms.

2.4 Managing Program Complexity

In the introductory section of this chapter, we discussed how object-oriented programming techniques came about through efforts to aid programmers in managing program complexity. All of the object-oriented techniques supported by C++ are extended to managing processes in C+-. The interested reader may consult the wealth of available literature on C++, including, but not limited to [32].

In the remainder of this section, for completeness, we shall mention briefly two of those techniques: inheritance and virtual functions. We shall then discuss the techniques that are specific to C+- and concurrent programming: process layering, process libraries, and customizing of the data exchange.

2.4.1 Class Inheritance

Class inheritance is the C++ mechanism that enables user-defined types to be *derived* from more basic types, inheriting data members and functions from the base type, possibly adding new ones and/or overriding old ones. Access rights are associated with each class member. For example, in Program 16, `private` members of the base class `shape` can be accessed only by member functions of `shape`; `protected` members of `shape` can, in addition, be accessed by member functions of any class derived from `shape` (for example, `circle`); and `public` members of `shape` can be accessed by any piece of code anywhere in the program. The `class circle` is

```
class   shape
{
private:
        int     origin;
        void    modify_origin();
protected:
        int     color;
        void    modify_color();
public:
        void    draw();
};


class   circle : shape
{
private:
        int     radius;
public:
        void    modify_radius();
        void    draw();
};
```

Program 16: Class Inheritance

derived from `class shape` by adding a data member (`radius`) and a member function (`modify_radius()`), and by overriding the member function `draw()`.

A typical memory layout for the two classes is shown in Figure 2.6. The point to

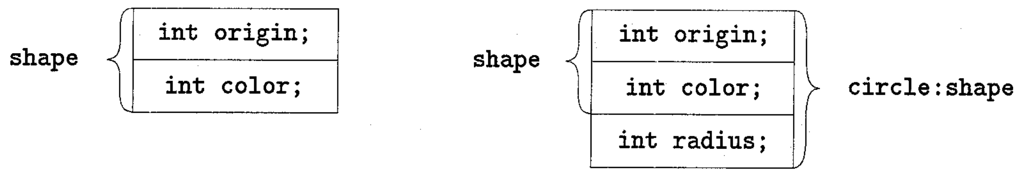


Figure 2.6: Class Inheritance *vs.* Memory Layout

be remembered is that C++ class inheritance is a compile-time rather than a runtime mechanism.[9] Every instance of `class circle` contains a part corresponding to an instance of `class shape`; it is the definition of `class shape` that is shared, not any particular instance of it.

The C++ class-inheritance mechanism is mimicked by process definitions in

[9]Neglecting, for the time being, such C++ features as multiple inheritance and virtual functions.

C+–; they too can be specified through their similarities with and differences from previously-defined process definitions.

2.4.2 Virtual Functions

The virtual-function mechanism supported by C++ is a mechanism that enables programmers to separate the design of member-function interfaces from the design of member functions themselves.

For example, in Program 16, given a `shape* sp`, and a `circle* cp`, the invocation of `sp->draw()` and `cp->draw()` will result in calling `shape::draw()` and `circle::draw()`, respectively. The compiler decides which call to generate based on the type of pointer through which the function has been called.

Had the two `draw()` functions been `virtual`, the invocation of `sp->draw()` could have invoked either of the two functions, depending on what the pointer `sp` pointed to. In this case, the compiler generates an indirect call through the class-specific table.

2.4.3 Process Layering

The standard C++ inheritance mechanism allows one to describe process definitions hierarchically. However, once a process is created, it is an independent entity. The hierarchy is reflected in its structure, *not* in its relationship with other processes.

There are important applications where, in addition to *hierarchy in structure*, it is useful to have runtime-exercised *hierarchy in control.* For example, in operating or runtime systems [10], user processes are created and managed by system processes. In simulators [91], processes that model the behavior of physical elements are managed by time- or event-driven schedulers.

The mechanism that C+– uses to support such applications is *process layering*, also called *dynamic process inheritance.* As illustrated in Program 17 and Figure 2.7, every instance of `processdef gate` is managed by an instance of `processdef scheduler`. The details of process layering will be discussed in Section 3.1, which

```
processdef      scheduler
{
private:
        int     time;
};


processdef      gate : dynamic scheduler
{
protected:
        gate*   output;
};


processdef      two_input_gate : gate
{
private:
        int     state;
atomic  void    input1(int);
atomic  void    input2(int);
};
```

Program 17: Process Layering

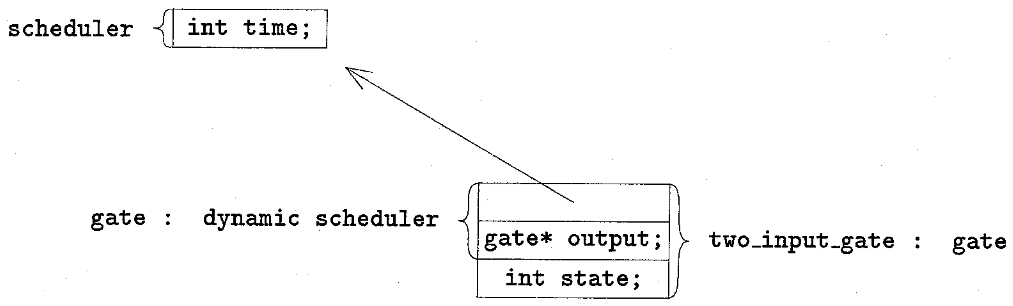


Figure 2.7: Process Layering *vs.* Memory Layout

describes the C+- runtime-system interface. The relationship between the manager process and the managed process is established at the creation time of the managed process. The manager provides a set of services to all processes that it manages, with the same access protection that is offered through the class-inheritance mechanism. The manager decides when an atomic action of any of the processes managed by it is executed (as opposed to invoked), while conforming to the definitions of process behavior as specified in Section 2.2.

2.4.4 Process Libraries

Libraries of C+- processes can be organized in the same way as libraries of data structures in C++. In most cases, the remote procedure calls to atomic actions of processes form a suitable interface, and these calls replace the class member-function interfaces. In these cases, it is sufficient that programs include header files that contain interface-process definitions.

There are cases, however, in which imposing the RPC interface would overly serialize computations that are otherwise concurrent. For example, a process library might initialize a set of processes for FFT computation, as illustrated in Section 2.3.3, employing several input and several output data streams. A stream of input values can be represented by a sequence of non-blocking atomic-action invocations. If a stream of output values were represented as a sequence of replies obtained through the RPC mechanism, just as in the sequential-search example of Section 2.3.2, the computation could not be pipelined. However, unlike in this search example, this problem could not be resolved with call forwarding.

The mechanism typically used for C+- libraries with multiple input and output streams is as follows: an input stream is represented by a sequence of non-blocking atomic-actions invocations of an input-interface process; an output stream is, similarly, a sequence of non-blocking atomic-actions invocations of a process provided by the library user. In this arrangement, the library-user process must be derived from the output-interface process of the library it uses (Section 2.5). When a process uses multiple libraries, multiple inheritance is employed to derive such a process from all of the output-interface processes from which it requires results.

2.4.5 Data Exchange

The designers of C++ made a commendable effort to provide an overloading mechanism that enables programmers to pass arguments by value, even when these arguments are arbitrarily-complicated, linked, data structures. This mechanism is not sufficient for concurrent-programming systems, which must take into account some additional considerations. On multicomputers, object pointers have local meaning. Also, concurrent computers may be heterogeneous ensembles comprised of machines with different data layout, alignment, size, or representation.

C+- addresses all of these potential problems at the inter-process-communication level (invocations of atomic actions) with mechanisms that are described in the remainder of this section. The communication specifications are *declarative*, as opposed to *imperative:* the programmer specifies what special actions should be taken when a data item of certain type is communicated; the compiler guarantees that actions thus specified will be invoked on every occurrence of communication.

Communicating Arbitrarily-Complex Data Structures by Value

One of the premises of fine-grain concurrent programming is that large data structures are implemented in terms of many small, cooperating processes, so it is tempting to claim that process pointers that can be globally dereferenced are all that programmers might possibly want. However, an important use for pointers in C++ is for data structures that are only partially specified at compile time: linked data structures and arrays of variable size. If proper support and clean semantics for this feature were not offered, users would have resorted to *ad hoc* solutions.

The mechanism supported by C+- enables the programmer to specify what *extra* actions should be taken when communicating an object of some class by value. In its most common form, it amounts to flattening the linked data structure before sending, and relinking it upon receiving. As will be illustrated in Section 3.1, variants of this mechanism can also be used to express more intricate (but sometimes much more efficient) communication protocols.

Suppose that the data type of choice is a singly-linked list of elements of type `list`, each of which contains a pointer to the next element in the list, a pointer to a vector of integers, and a field specifying the size of the integer vector. Figure 2.8 illustrates what is required to pass a data item of type `list` by value. Part (a) shows a data item scattered around in memory. Part (b) shows the flattened data structure, with the dashed parts corresponding to other arguments that may be sent in the same communication. If the concurrent computer at hand is a shared-memory multiprocessor, and if the flattened argument list is in the shared address space, the task is completed. Now suppose that passing arguments moves them from one address space to another, as typically happens on a multicomputer. When the message that encapsulates the argument list is received, all the pointers are off by a constant (c), and have to be re-linked, as in (d).

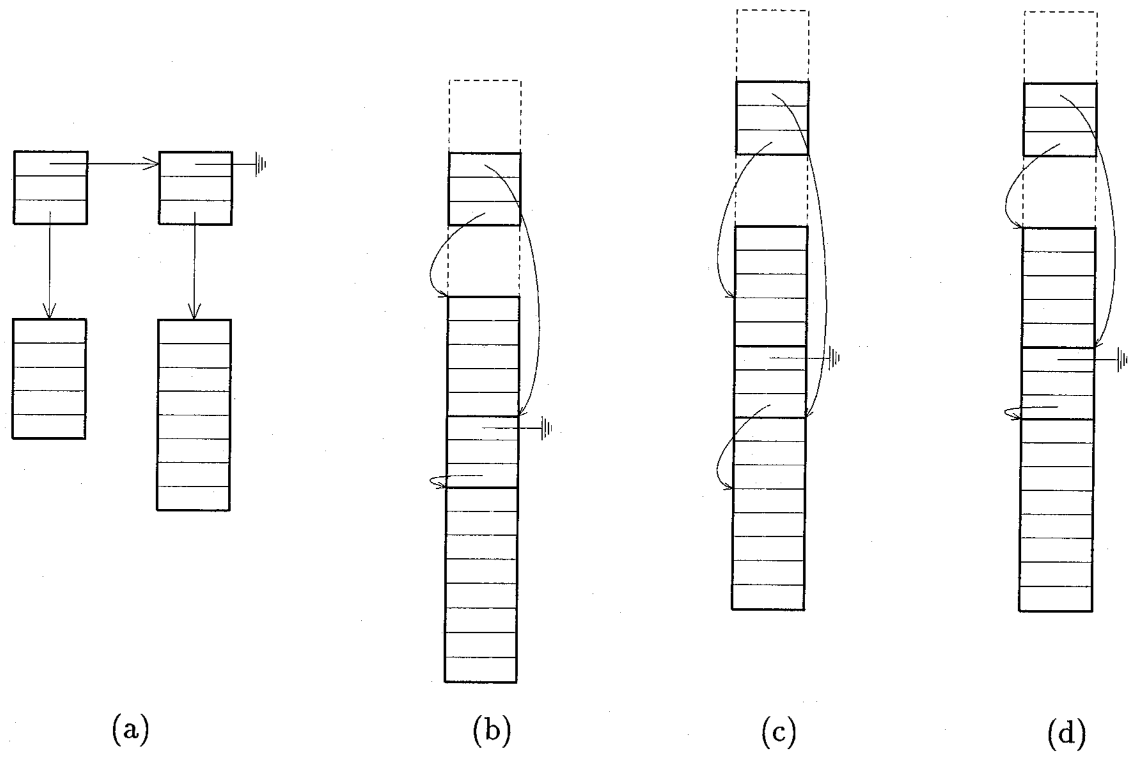


Figure 2.8: Flattening Linked Data Structures

Program 18 is the specification of the flattening and re-linking tasks: The `operator space` computes how much extra space is needed in the argument list when an instance of `list` is passed as an argument to an atomic action. The `operator send` specifies that, in addition to this instance of `list`, a vector of integers and the remaining part of the list should be passed along. The `operator recv` requests that the vector of integers (`data`) and the rest of the list `next` be re-linked in place on the receive side.

This special handling will be invoked not only for instances of `list`, but also for

```
class   list
{
private:
        int     size;           // number of integers "data" points to
        int*    data;
        list*   next;           // a pointer to the next of kin

public:
        size_t  operator space ()
                {
                    size_t  s = space(data,size);  // space for size integers
                    if (next)  s += space(next);   // space for the rest
                    return  s;                     //  of the list
                }

        void*   operator send (void* v)
                {
                    v = send(v,data,size);         // send size integers
                    if (next)  v = send(v,next);   // send the rest
                    return  v;                     //  of the list
                }

        void    operator recv ()
                {
                    recv(data);                    // re-link int*
                    if (next)  recv(next);         // re-link the rest
                }                                  //  of the list
};
```

Program 18: Passing Linked Data Structures By Value

all objects derived from `list`, and for all objects that contain instances of `list` as members. C+- data-structure libraries can, accordingly, be built in a way that allows library users to be indifferent about the details of the implementation.

This example illustrates how arbitrarily complex, linked, data structures can be passed by value. However, to avoid copying, and when sharing of data structures between processes is needed, structures must consist of linked processes, not of linked objects.

Communicating Across Heterogeneous Machine Boundaries

The C+- compiler assembles all messages (argument lists to atomic actions), and initiates all instances of communication (invocations of atomic actions). This information enables the compiler to handle the size and alignment of the basic data types (integers, floating-point numbers, *etc.*) for a programmer-specified set of machines that may be involved in direct communication.

The example in Program 19 specifies that, in addition to the local-machine type,

```
machine I286
{
    char,       1,      1;
    short,      2,      1;
    int,        2,      1;
    long,       4,      1;
};

machine Sparc
{
    char,       1,      1;
    short,      2,      2,      send_lib,       recv_lib;
    int,        4,      4;
    long,       4,      4;
};
```

Program 19: Machine Descriptions

communication may be established with machines of types I286 and Sparc (arbitrary, user-specified names). The entries within each machine description correspond to the data size and alignment (measured in units of size equal to the minimum-addressable memory unit on the machine running this program), and any special treatment that may be required for a particular basic data type.[10] For example, for a machine of type Sparc, short integers are of size 2 and have to be positioned on addresses divisible by 2. When sending a short integer to a process residing on a machine of type Sparc, the data item has to be converted using the user-supplied and user-named function send_lib; when receiving a short integer from such a process, the data item has to be converted using the function recv_lib.

The compiler implicitly generates type machine_t, defined as:

```
enum  machine_t  { local__CPM, I286, Sparc };
```

[10]The following is the complete list of C+- basic data types: char, short, int, long, float, double, long double, signed char, unsigned char, unsigned short, unsigned int, unsigned long, void*, entry_t, and pointer_t.

and the user is obliged to define the function

```
machine_t  machine__CPM (pointer_t);
```

that maps process pointers into machine types.

2.5 Putting It All Together

The examples of C+- programs shown so far were chosen to illustrate programming techniques. We have deliberately chosen clarity over completeness, and, indeed, some of these examples require the addition of forward declarations to be accepted by the compiler.

In this section, we shall show an example of a complete program that computes the N-point FFT, as illustrated in Figure 2.5. Our concurrent program will closely match this data-dependency graph, with one addition: We shall introduce a column of nodes whose purpose is to rearrange the input values from the standard, linear ordering of indices to the bit-reversed ordering required at the input of the FFT-computing graph. Figure 2.9 shows the modified graph, with circled parts corresponding to sub-computations performed by individual processes.

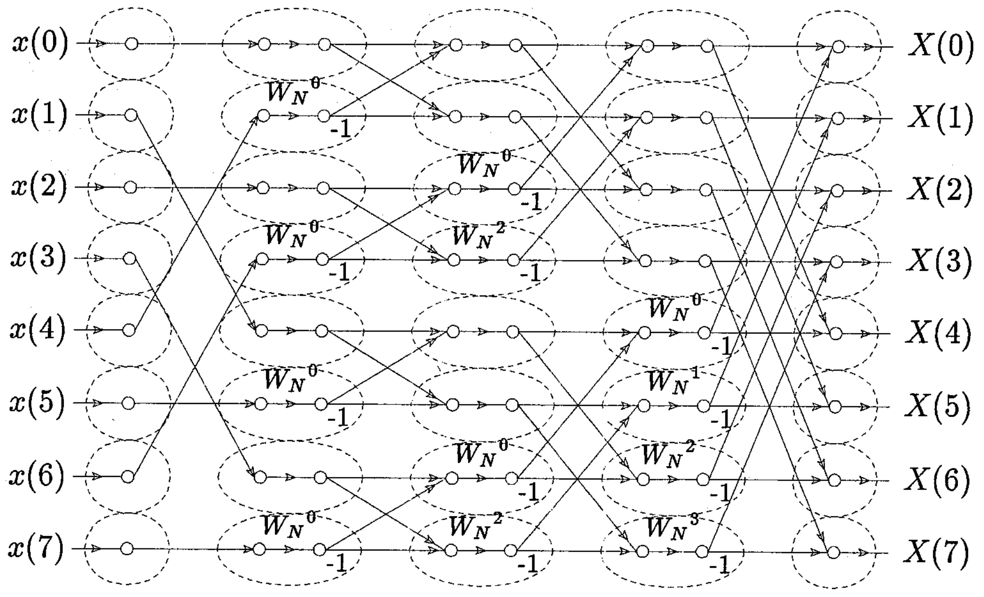


Figure 2.9: An 8-Point FFT Computation, with the Processes Circled.

Typically, writing C+- programs consists of four stages:

- Choosing a concurrent algorithm;
- Designing an input/output interface;
- Designing the process hierarchy; and,
- Describing process behavior.

We shall organize the program as a library package. Figure 2.10 illustrates the user-level view of this library. Input values are to be sent to processes of type `fft`, and output values will be delivered to processes of the same type. For an N-point FFT computation, there are N input and N output processes, all of which have to be derived from `fft`. The set of pointers to N input processes could be represented in a variety of ways, but it is often most intuitive to represent these processes as members of a process vector, as described in Section 2.3.9. The same is true for the set of pointers to N output processes.

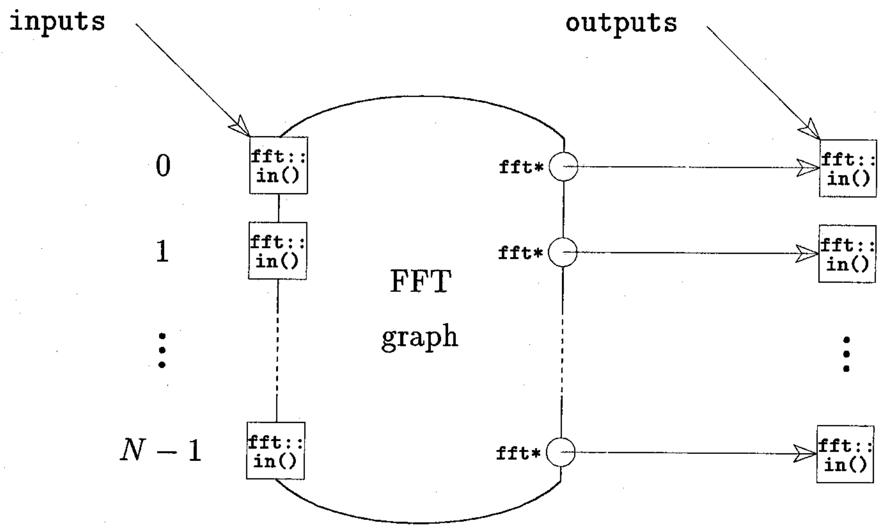


Figure 2.10: User View of the FFT-Library

Program 20 is the header file that the user must include to access the library. A user program might look like Program 21: Since the library sends the output values to the vector of `fft` processes, the `consumer` processes are derived from `fft`, and have to be created using the distributed-process mechanism. The `producer` processes, on the other hand, don't have to be elements of any vector unless some other part of the user code needs to treat them so.

```
// fft.h


#include        <c+-.h>                          // The runtime-system header file
#include        <Complex.h>                      // The complex-arithmetic package


processdef      fft : public CPM                 // The runtime system requires that
{                                                // every process be derived from CPM
public:
atomic  virtual void    connect(fft*) = 0;       // The '= 0' syntax in C++ denotes
atomic  virtual void    in(Complex) = 0;         // that this is the specification
};                                               // of an interface, leaving it to
                                                 // the derived processes to specify
                                                 // how the requests are serviced

processdef      fft_graph : public CPM           // This process represents the
{                                                // whole graph
private:
        fft*    inputs;                          // The pointer to the first input
        int     order;                           // Size of the FFT graph
public:
atomic           fft_graph(int, fft*);           // Creating the fft process graph
atomic          ~fft_graph();                    // Deleting the fft process graph
atomic  fft*    input(int);                      // Finding out the address of a
                                                 // particular input
};


Complex W(int N, int i);                         // A function that computes
                                                 // complex roots of 1

int     bit_reverse(int N, int i);               // A bit-reversing function
```

Program 20: The FFT-Library Header File

```
#include        <fft.h>


processdef      consumer : public fft
{
public:
atomic  virtual void    in(Complex);        // Do something with the result
};


processdef      producer : public CPM
{
public:
atomic                  producer(fft*);     // Produce input values
};


const   int  N = 32;


root::root (int argc, char** argv)
{
    fft*  outputs = new consumer[N];        // Create the vector of consumers

    fft_graph*  g = new fft_graph(N,outputs);
                                            // Create the computation graph

    fft*  inputs = g->input(0);             // Get the reference to the inputs

    for (int i=0; i<N; i++)                 // Create N producers
        new producer(inputs+i);
}
```

Program 21: An Example of FFT-Library Usage

Figure 2.11 shows the process-specification hierarchy that we chose to implement, and Programs 22 and 23 specify this hierarchy.

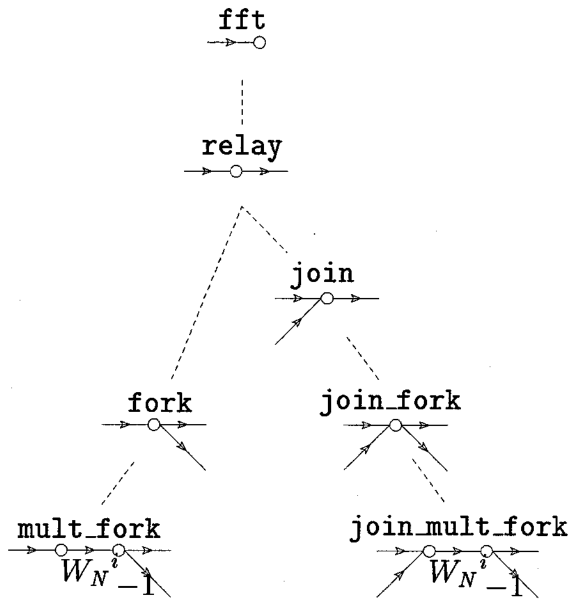


Figure 2.11: Process-Specification Hierarchy

The `fft` process definition is just an interface specification, and does not describe any computation. The remaining process definitions specify that the process activity consists of four distinct stages:

- Establishing a connection, *ie,* obtaining output references;
- Getting one or two input values;
- Computing the result, which may involve an addition and a multiplication; and,
- Outputting one or two output values.

The common parts of the code are shared between different process definitions through the process-inheritance mechanism. Using multiple inheritance (whereby process definitions can be derived from more than one process definition) would have resulted in better code reuse. Nevertheless, we felt that, in the examples in this thesis, multiple inheritance would not have contributed to reader's understanding of C+-.

```
// fft1.h


#include        "fft.h"


processdef      relay : public fft
{
protected:
                fft*    out;                // Output reference
                Complex result;             // The result
        virtual void    compute(Complex);   // How to compute the result
        virtual void    output();           // How to generate the output
public:
atomic  virtual void    in(Complex);
atomic  virtual void    connect(fft*);
atomic                  relay()
                        { passive(in); }
};


processdef      join;


processdef      fork : public relay
{
protected:
                join*   out1;               // Fork adds an output reference,
        virtual void    output();           // and produces two output values
public:
atomic  virtual void    connect(fft*, join*);
};


processdef      mult_fork : public fork     // Mult_fork also needs to multiply
{
protected:
                Complex W;                  // so here is the multiplicand
        virtual void    compute(Complex);   // and how to compute
        virtual void    output();           // It must generate the +- output
public:
atomic  virtual void    connect(fft*, join*, Complex);
};
```

Program 22: Process Hierarchy for FFT Computation, Part 1

```
// fft2.h


#include        "fft1.h"


processdef      join : public relay         // Join has two distinct inputs
{
protected:
        virtual void    compute(Complex);   // How to compute the result
public:
atomic  virtual void    in (Complex);
atomic  virtual void    in1(Complex);
atomic                  join()
                        { passive(in); passive(in1); }
};


processdef      join_fork : public join             // The same modifications
{                                                   // as from relay to fork
protected:
        join*   out1;
        virtual void    output();
public:
atomic  virtual void    connect(fft*, join*);
};


processdef      join_mult_fork : public join_fork   // The same modifications
{                                                   // as from fork to mult_fork
protected:
                Complex W;
        virtual void    compute(Complex);
        virtual void    output();
public:
atomic  virtual void    connect(fft*, join*, Complex);
};
```

Program 23: Process Hierarchy for FFT Computation, Part 2

The behavior of various process types is specified in Programs 24, 25 and 26.

```
// fft0.cpm

#include        "fft2.h"

atomic
void
relay::connect (fft* f)
{
    out = f;
    active;             // make all atomic function active
}

atomic
void
fork::connect (fft* f, join* j)
{
    out  = f;
    out1 = j;
    active;
}

atomic
void
mult_fork::connect (fft* f, join* j, Complex c)
{
    out  = f;
    out1 = j;
    W    = c;
    active;
}

atomic
void
join_fork::connect (fft* f, join* j)
{
    out  = f;
    out1 = j;
    active;
}

atomic
void
join_mult_fork::connect (fft* f, join* j, Complex c)
{
    out  = f;
    out1 = j;
    W    = c;
    active;
}
```

Program 24: The FFT Computation, Part 1

```
// fft1.cpm

#include        "fft2.h"

atomic
void
relay::in (Complex c)
{
    compute(c);
    output();
}

void
relay::compute (Complex c)
{
    result = c;
}

void
mult_fork::compute (Complex c)
{
    result = W * c;
}

void
relay::output ()
{
    out->in(result);
}

void
fork::output ()
{
    out->in(result);  out1->in1(result);
}

void
mult_fork::output ()
{
    out->in(-result);  out1->in1(result);
}
```

Program 25: The FFT Computation, Part 2

```
// fft2.cpm

#include        "fft2.h"

atomic
void
join::in (Complex c)
{
    if ( passive(in1) )
        {  compute(c);  output();  active(in1);  }
    else
        {  result = c;             passive(in);  }
}

atomic
void
join::in1 (Complex c)
{
    if ( passive(in) )
        {  compute(c);  output();  active(in);    }
    else
        {  result = c;             passive(in1);  }
}

void
join::compute (Complex c)
{
    result += c;
}

void
join_mult_fork::compute (Complex c)
{
    result = (result + c) * W;
}

void
join_fork::output ()
{
    out->in(result);  out1->in1(result);
}

void
join_mult_fork::output ()
{
    out->in(-result);  out1->in1(result);
}
```

Program 26: The FFT Computation, Part 3

Finally, Programs 27, 28 and 29 contain the code used to build the N-point FFT process graph. Depending on how time-critical this creation task is, solutions range from entirely sequential, taking $\mathcal{O}(N \log N)$ steps, to maximally concurrent, taking just $\mathcal{O}(\log N)$ steps. Our solution follows an intermediate approach, in which the process creation is concurrent and takes $\mathcal{O}(\log N)$ steps, whereas passing references around is sequential for each process column, and takes $\mathcal{O}(N)$ steps.

```
// fft3.h


#include        "fft2.h"


processdef      build_top_fft : public CPM
{
public:
atomic          build_top_fft(int, join*, int, int, fft*);
};


processdef      build_btm_fft : public CPM
{
public:
atomic          build_btm_fft(int, join*, int, int, fft*);
};
```

Program 27: Building the FFT Graph, Part 1

```
// fft3.cpm


#include        "fft3.h"


fft_graph::fft_graph (int N, fft* outs)
{
    order   = N;
    inputs  = new relay[N];

    if ( N > 1 )
    {
        join*  j = new join[N];
        new build_top_fft(N, j, 0, N/2-1, inputs);
        new build_btm_fft(N, j, N/2, N-1, inputs);
        for (int i=0; i<N; i++)
            (j+i)->connect(outs+i);
    }
    else
    {
        inputs->connect(outs);
    }
}
```

Program 28: Building the FFT Graph, Part 2

```
// fft4.cpm


#include        "fft3.h"


build_top_fft::build_top_fft (int N, join* outs, int from, int to, fft* inputs)
{
    int  n = to - from + 1;

    if ( n > 1 )
    {
        join_fork*  f = new join_fork[n];
        new build_top_fft(N, f, 0, n/2-1, inputs);
        new build_btm_fft(N, f, n/2, n-1, inputs);
        for (int i=0; i<n; i++)
            f[i].connect(outs+i,outs+n+i);
    }
    else
    {
        fork*  f = new fork;
        f->connect(outs,outs+1);
        (inputs+bit_reverse(N,from))->connect(f);
    }
}


build_btm_fft::build_btm_fft (int N, join* outs, int from, int to, fft* inputs)
{
    int  n = to - from + 1;

    if ( n > 1 )
    {
        join_mult_fork*  f = new join_mult_fork[n];
        new build_top_fft(N, f, 0, n/2-1, inputs);
        new build_btm_fft(N, f, n/2, n-1, inputs);
        for (int i=0; i<n; i++)
            f[i].connect(outs+n+i,outs+i,W(N,from+i));
    }
    else
    {
        mult_fork*  f = new mult_fork;
        f->connect(outs+1,outs,W(N,from));
        (inputs+bit_reverse(N,from))->connect(f);
    }
}
```

Program 29: Building the FFT Graph, Part 3

Chapter 3

Implementation Issues

There are two major components to the C+- programming system: the translator from C+- to C++, and the C+- runtime system. This programming system is currently supported on the Mosaic, and on all systems that support the Cosmic Environment/Reactive Kernel (CE/RK) [80] message-passing primitives, which includes sequential computers, networks of workstations, and a variety of commercial multicomputers and multiprocessors.

The translator is written in C++, and is both compile-machine- and target-machine-independent. Most of the runtime-system code is portable as well, with the exception of a small set of C+- library functions that are illustrated in Section 3.1.

3.1 The Runtime-System Framework

The relationship between the C+- programming notation and the C+- runtime systems is symbiotic: Programs written in C+- require runtime-system support; C+- runtime systems are typically written in C+-.

Although most of the runtime-system code is portable, the resource-allocation requirements on various machines are quite different. Given a sufficiently large node memory, the amount of runtime-system support that C+- programs require is minimal. The runtime systems for C+- implementations on computers with workstation-size nodes typically consist of less than a thousand lines of C+- code. The

Mosaic fine-grain multicomputer consists of nodes with severely restricted memory resources; hence, the runtime system for the Mosaic employs much more sophisticated runtime mechanisms. Various configurations of MADRE, the MosAic Distributed Runtime systEm, range from two to ten thousand lines of C+- code. MADRE was written by Nanette J. Boden, and its design and the distributed algorithms it employs are described in detail in her Ph.D. thesis [10]. This work demonstrates that the complexity of runtime systems for fine-grain multicomputers need not result in large penalties in speed, nor does it imply large chunks of node-resident code that reduce the available node memory even further. MADRE is itself a concurrent program that employs distributed solutions to manage distributed resources [10].

The mutual dependence of the C+- programming notation and the C+- runtime systems is only apparent. In fact, the runtime system is just a pre-written part of any user program — a part that includes an interface to the resource-allocation and communication capabilities of the machine it is running on. The C+- programming model and programming notation supply only the framework for implementing process management and data communication, striving not to restrict the spectrum of possible runtime-system implementations. The remainder of this section describes this framework. Since the primary target for executing C+- programs is the Mosaic, the names and default semantics of functions that we use correspond to message-passing communication primitives. This does not, however, imply that these primitives are the only ones that can be used; shared-memory communication primitives, for example, are equally suitable for implementing the necessary low-level routines.

3.1.1 Process Creation

An example of how process creation may be implemented in C+- is given in Program 30. In general, process creation consists of the following three stages:

- *Choosing a manager,* by invoking the `manager__CPM` function[1] corresponding to the type of the process being created. This function must return a pointer to the process that will be asked to instantiate the new process. It is possible to define multiple versions of this function, some of which may take arguments. For example, different versions may correspond to different process-placement strategies.

- *Requesting the creation* from the chosen manager by invoking the manager's `create__CPM` atomic action. The two arguments[2] correspond to the size of the process and the address of the constructor to be invoked. If the constructor takes arguments, those are passed as well. Various flavors of process creation can coexist in the system, with one of them selected at creation time.

- *Instantiating the process* is done by a manager process, not necessarily the one originally chosen: The creation can be delegated to other potential manager processes, and is eventually done in the consenting manager's address space [10].

3.1.2 Runtime Services

All of the `protected` and `public` members of a manager can be accessed by the processes it manages. This access is handled transparently by the compiler. The programmer need not be concerned whether some service is provided through regular inheritance or through dynamic inheritance, with the latter requiring one or more levels of indirection (Program 31).

[1]This function must be declared `static`, which is a C++ feature that makes a member function generic, associated with a certain class definition, not with any particular instance of that class.

[2]The `size_t` is a C++-defined integer type that can represent the size of the largest possible object (or process). The `entry_t` type is introduced by C+−, and will be described in Section 3.1.4.

```
processdef      Manager
{
public:
atomic  P*      create__CPM (size_t, entry_t, ...);
};

processdef      P : dynamic Manager
{
public:
static  Manager*  manager__CPM();
atomic            P();
atomic            P(int);
};

{
    new P;               // is equivalent to:
    P::manager__CPM()->create__CPM(sizeof(P),&P::P());

    new P(123);          // is equivalent to:
    P::manager__CPM()->create__CPM(sizeof(P),&P::P(int),123);
}
```

Program 30: Process Creation

3.1.3 Process Dispatch

A problem that emerges in the design of all operating and runtime systems is that of specifying an interface for invoking user programs. This task is typically done in an *ad hoc* way. For example, user programs written in C and run under UNIX must have a function called `main`, which is the user-code entry point. However, this approach does not enable the operating system code to merely call this function, since the address of `main` is not known at the operating-system linking time. The typical solution is to require that `main` always be at the same address, or to find its address at loading time.

Every C+– process has a fixed number of entry points, corresponding to its atomic actions, each of which could take different numbers and types of arguments, and return values of different types. If the runtime system itself is to be expressed in C+–, there must be a way of dispatching to any atomic action of any process, or of any process in some predefined set. In the remainder of this section, we describe the C+– atomic-action dispatch mechanism.

```
processdef      Manager                 // runtime-system code
{
protected:
        int     i;
        void    f();
};


processdef      P : dynamic Manager     // user code
{
private:
        int     j;
        void    g();
public:
atomic          P()
                {
                    j = 0;      // accessing local data
                    i = 0;      // accessing manager's data
                    g();        // calling local function
                    f();        // calling manager's function
                }
};
```

Program 31: Accessing Runtime Services

As illustrated in Figure 3.1, every process `P` is a node of a process tree, with its path toward the root of a tree leading through its manager `M`, its manager's manager `MM`, *etc.* Several such trees may coexist on each physical node. Every `processdef M` that could be used as a `dynamic` base for some process definition, which means that an instance of `M` could be a manager of some process, must have a special atomic action defined, `atomic @M(entry_t)`, called the *dispatcher.* A generic dispatcher, `atomic @(entry_t)`, also has to be defined; its job is to dispatch to root processes of process trees.

The `entry_t` is a type introduced by the compiler, corresponding to any and all types of entry points of processes that *could* be defined with `M` as their dynamic base. A variable of this type can be used like a regular C++ member-function pointer, with one important distinction: one need not know the interfacing details of all atomic actions that a variable of type `entry_t` may be used to invoke. How arguments are passed to anonymous atomic actions is discussed at the end of this section. How values are returned from atomic action is presented in Section 3.1.9.

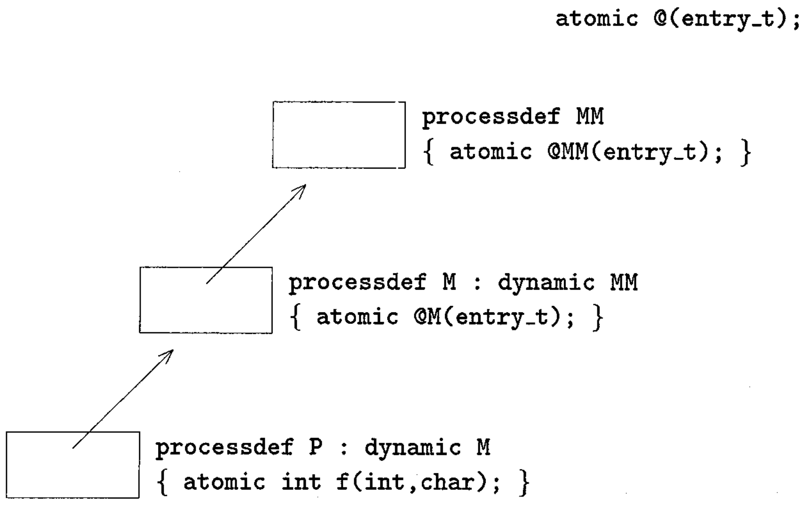


Figure 3.1: Process Dispatch

Specification 4 An execution of an atomic action of a process can be requested only from the body of its manager's dispatcher atomic action.

For the process hierarchy in Figure 3.1, this specification means that the execution of an atomic action of `processdef P`, say `P::f`, consists of executing the generic dispatcher `@`, which calls `MM::@MM`, which calls `M::@M`, which calls `P::f`. It is this layered execution that enables "managers" to manage other processes: *The semantics of atomic-action executions can be changed by modifying the runtime-system code.* As stated in The Annotated C++ Reference Manual: "...this opens vast opportunities for generalization and language extension in the general area of: What is a function and how can I call it?" [32]. This feature could strike the reader as intolerably under-specified and inviting of hacking and abuse. However, the safety properties of this mechanism are not as weak as they may appear to be. The runtime-system-specified mechanisms cannot be changed by users — the manager *always* gets to run before dispatching to the managed process. We have come to believe that the support for some mechanism of this kind is essential for a notation that is intended for expressing operating and/or runtime systems.

Another way of thinking about this layered dispatch mechanism is that every process provides a set of services (its atomic actions), and an *escape* mechanism to which it can defer the execution if it cannot handle the requested service itself.

Arguments to Atomic Actions

The memory layout of the arguments to atomic actions is the same as that for regular functions in C++, with additional arguments being passed to the dispatcher actions of the manager processes (Figure 3.2).

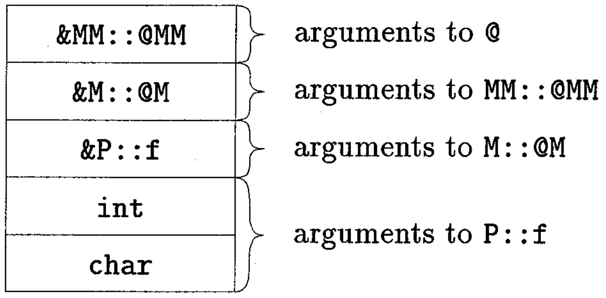


Figure 3.2: Atomic-Actions Arguments Layout

These additional arguments are, by default, generated by the compiler, but, as discussed in Section 3.1.7, this default behavior can be replaced by one defined by the programmer.

An additional feature is that the arguments are assumed to be members of the compiler-introduced structure `args_t`, and can be accessed as a unit through a pointer variable `args_t* args;` (similar to the `this` variable in C++).

3.1.4 The `pointer_t` and the `entry_t` Types

In the programming examples we made use of `pointer_t` and `entry_t` types, always referring to them as "introduced by the compiler." These two types are actually defined by the runtime-system in a file that has to be included by every C+- program (`<c+-.h>`). The C+- translator makes the structure of every process pointer the same as that of `pointer_t`, and the structure of every pointer to a member of a process the same as that of `entry_t`.

3.1.5 Process State

As discussed in the previous sections, the state of a C+- process consists of its:

- data members,
- active/passive set, and
- a pointer to the manager process.

What are the semantics of process assignment in the context of processes with the state defined above? The default C+- semantics for process assignment are bit-wise copying of data members and of the representation of the active-passive set; the pointer to the manager process is left untouched. The example in Program 32 shows process assignment as equivalent to sending a request to the source process to send a copy of itself to the specified destination process.

```
processdef      P
{
private:
        int     i;
public:
atomic  int     copy__CPM (P* pp)
                {
                    forward  pp->copy__CPM(*this);
                }
atomic  int     copy__CPM (P p)
                {
                    *this = p;
                    return  1;
                }
};


{
    P  p1, p2;

    p1 = p2;    // is equivalent to:

    await ( p2.copy__CPM(&p1) );
}
```

Program 32: Process Assignment

3.1.6 Process Migration

No notion of process migration is supported directly in C+-. A process pointer typically contains an absolute address of a piece of memory representing the state of a process. However, the example in Program 32 shows how simple it is to copy the state of a process. Furthermore, with the ability of the runtime system to define the structure of process pointers (Section 3.1.4), the runtime-system framework described in this chapter was sufficient to implement distributed processes (Section 2.3.9). The support for distributed processes requires the same indirection mechanism that might be used for process migration. The work reported in [10] is a first step towards a thorough examination of the issues involved in process migration. The results presented in this work establish conditions under which, for example, process state can be shipped to where the atomic-action code is located just as readily as code can be cached where the process state is located.

3.1.7 Invoking Atomic Actions

As illustrated in Program 34, an atomic-action invocation consists of three stages:

- *Introductory Stage* — Upon calling `operator space` to determine the size of the argument list, the `operator head` is invoked to build the dispatcher list. Given a data type `TYPE` and a process type `PROCESS`, the default operator semantics are as follows:

```
size_t  operator space(TYPE t)
        {
            return  sizeof(t);
        }

static
void*   PROCESS::operator head(void* v, pointer_t p, entry_t e, size_t s)
        {
            return  operator send(v,e);
        }
```

- *Main Stage* — For each element in the argument list, the `operator send` is invoked. The default operator semantics are bit-wise copy:

```
void*    operator send(void* v, TYPE t)
         {
             TYPE*  tp = v;
             *tp = t;
             return  tp+1;
         }
```

- *Final Stage* — The `operator tail` is invoked, with no-op default semantics:

```
static
void     PROCESS::operator tail(void*, void*)
         { }
```

At the time of atomic-action execution, `operator recv` is invoked for each element in the argument list. The default semantics for this operator are a no-op (Program 33).

```
void     operator recv(TYPE t)
         { }
```

Program 33: Default `operator recv`

The set of operators described above provides runtime-system programmers with a powerful tool that they can use to define how process communication is actually implemented in terms of lower-level routines. The same set of operators is available to users. An example of an application that might benefit significantly from the ability to exercise total control is a program that implements communication-network protocols. The general usability of the above mechanism, however, is highly questionable: Once the compiler relinquishes control over data layout to a naive user, obscure problems abound. For a great majority of applications, the efficiency of the data-exchange mechanisms described in Section 2.4.5 is sufficient.

```
processdef      MM
{
public:
atomic          @MM(entry_t);
};

processdef      M : dynamic MM
{
public:
atomic          @M(entry_t);
};

processdef      P : dynamic M
{
public:
atomic  void    f (int, char);
};

{
    P*   p;
    int  i;
    char c;

    p->f(i,c);          // atomic action invocation is equivalent to

    {
        size_t  size = operator space(&MM::@MM)         // assuming there are
                     + operator space(&M::@M)           // no alignment problems
                     + operator space(&P::f)
                     + operator space(i)
                     + operator space(c);
        void  *b, *v;
        pointer_t  pp = p;

        b = v =     operator head (   pp, &MM::@MM, size);
            v = MM::operator head (v, pp, &M::@M,   size);
            v =  M::operator head (v, pp, &P::f,    size);

        v = operator send(v,i);
        v = operator send(v,c);

        v =  M::operator tail (   v, pp, &P::f,    size);
        v = MM::operator tail (   v, pp, &M::@M,   size);
                operator tail (b, v, pp, &MM::@MM, size);
    }
}
```

Program 34: Atomic-Action Invocation

3.1.8 Active/Passive

The active/passive mechanism, because of its simplicity and efficiency (Section 4.3), is the C+- synchronization mechanism of choice. The runtime-system interface for this mechanism is presented in Program 35. If a different synchronization mechanism is required, it can be implemented following the same approach.

```
processdef      P
{
public:
atomic  void    f();
atomic  int     g();
};


atomic
void
P::f ()
{
    active  f;          // is equivalent to:
    P::active__CPM(&P::f);

    passive  g;         // is equivalent to:
    P::passive__CPM(&P::g);
}
```

Program 35: Active/Passive Implementation

3.1.9 Remote Procedure Call

When invoking an atomic action that returns a value, the sequence of events is identical to that described in Section 3.1.7, except that an extra argument is passed. This extra argument is the pointer to the currently-running process — the process that expects the reply. This pointer is obtained by calling the runtime-system-defined function `current__CPM()`.[3] The NULL extra argument implies that the returned value is not required.

[3]Note that it was not possible to use the `this` variable, because a process might be suspended while executing a non-member function.

Values Returned From Atomic Actions

Inside an atomic action, the extra argument is called `reply__CPM`. As illustrated in Program 36, returning a value from an atomic action is equivalent to invoking the `return__CPM(...)` atomic action of the process pointed to by the `reply__CPM` pointer.

```
processdef      P
{
public:
atomic  int     f();
};


atomic
int
P::f ()
{
    return  123;        // is equivalent to

    {
        if (reply__CPM)
            reply__CPM->return__CPM(123);
        return;
    }
}
```

Program 36: Atomic Actions Returning Values

Suspending A Process

Whenever a returned value is expected from an atomic action, the compiler introduces a placeholder for that value, and the runtime system is passed a pointer to this placeholder through the `wait__CPM(void*)` function. Multiple placeholders can be active at any time, as discussed in Section 2.2. When the process attempts to access the placeholder and finds it uninitialized, it suspends itself by invoking the `suspend__CPM()` function.

3.2 From C+- to C++

There are a number of reasons for translating from C+- to C++ instead of compiling from C+- directly to Mosaic code. First, this was a faster way to build a running system. Second, the wide availability of C++ compilers guaranteed machine-independence. Third, we had good experience in re-targeting the Gnu C++ compiler to produce excellent code for the Mosaic processor. And fourth, since C+- is syntactically so similar to C++, C++ debugging tools and other programming-support tools can be used with few or no modifications. One disadvantage of the translation approach is that the compile time increases, because programs must be parsed twice. A possible disadvantage is that some optimization opportunities may be lost when using C++ as an intermediate target notation. However, we have identified no such lost opportunities so far.

3.2.1 Parsing

The translator is a C++ program built within the framework of a Bison-produced parser [30]. Practically every person who has ever worked on a project that involved parsing of C++ has already expressed their distaste that C++ syntax cannot be described by an LALR(1) grammar. Nevertheless, we feel that our own distaste should be on record, too. We acknowledge that it is not the compiler writer, but the language user, who should be the ultimate judge of the value and style of a programming notation. However, if syntactic issues are subtle enough to be difficult for a compiler, what hope does a user have of not making obscure mistakes writing programs using that syntax? Fortunately, beginners tend to use a small set of basic language constructs, whereas experienced users tend to develop their own programming style from a subset of the rich C++ offering. In our experience, the complexity of handling the few special cases in parsing C++ is comparable to the complexity of all of the remaining issues of translating C+- into C++. Suffice it to say that we are looking forward to the ANSI standard for C++ syntax.

In our implementation of the translator, each grammar rule corresponds to a class definition. For example, given the grammar rule in Program 37, three class definitions

```
expression      :       assignment_expression
                |       expression  ,  assignment_expression
                ;
```

Program 37: An Example of a Grammar Rule

have to be written, as shown in Program 38. Parsing a C+- program generates a

```
class   expression
{
        void    output() = 0;
};

class   expression0 : public expression
{
        assignment_expression*  member0;
public:
        void    output()
                {
                    member0->output();
                }
};

class   expression1 : public expression
{
        expression*             member0;
        assignment_expression*  member1;
public:
        void    output()
                {
                    member0->output();
                    member1->output();
                }
};
```

Program 38: A Part of the Definition of the Parse Tree

parse tree that consists of nodes that are instances of classes such as these illustrated in Program 38. We developed a program that, given an input grammar such as the one illustrated in Program 37, generates the default class definitions (similar to those described in Program 38), the code that builds the parse tree, and the default definitions of `output()` functions. The resulting program code is a parsing specification for Bison, which can be used to produce a default parser. When a source program is fed to this default parser, the parser builds the parse tree. It then invokes the `output()` function at the topmost level of the tree, thereby causing the entire source program to be produced as the output. This default behavior can be modified by defining additional elements of class definitions, by specifying extra actions to be taken while building the parse tree, and by providing customized versions of the `output()` routine for any class definition. This simple tool for developing programs for source-to-source transformation, a program of less than two thousand lines of C++ code, has been crucial to our ability to experiment with numerous versions of C+- syntax. This tool generates about two-thirds of the approximately 60,000 lines of C++ code of a complete C+- translator.

3.2.2 Code Generation

Once the hurdle of parsing C+- is overcome, the translation from C+- to C++ is a fairly simple task. The description of the runtime-system framework in Section 3.1 also specifies this translation task. Since the process concept is the only extension that C+- introduces to C++, the focus of the translator is on keeping track of processes and various other process-related types. The translator considers each segment of a source program to be a type transformation. For example, a process-pointer type, when dereferenced, is transformed into a process type, and a function call transforms a list of argument types into the type of the returned value. Since the translator keeps track of all of the type transformations in a program text, operations on processes are detected, and the replacement code, as illustrated in Section 3.1, is generated.

3.2.3 Code Splitting

In addition to the transformations described in Section 3.1, there is one more requirement on the translator. Since the Mosaic, a machine with limited node-memory resources, is the most important target machine for executing C+- programs, the C+- translator must provide support for code splitting. Pieces of code are cached in each node by the runtime system, and invoked through the indirect-function-call mechanism. A design decision had to be made on what the code-splitting target should be.

The default object-code unit provided by the regular C++ compilers is a piece of code produced by the compilation of one source file. We considered this default setup to be unacceptable. Programmers would have to organize their code according to the code-splitting policy rather than according to the programming-abstraction requirements of the application. This setup would unavoidably lead to loss of portability, whereby the source code would have to be rearranged and split into smaller pieces when moving to a machine with less node memory.

Given that the default code-splitting policy was deemed unusable, we identified three well-defined code-splitting targets. These three targets, with increasing granularity, are to split the code so that each piece corresponds to:

- an *atomic action* of a process,
- a *function* and/or an atomic action of a process, or
- a *block of code* within a function, with strictly sequential execution (no conditional execution).

The next-higher-granularity target would be equivalent to turning the runtime system into a pseudo-code interpreter.

If the block of code with strictly sequential execution is the code-splitting target, only code that is certain to be executed is ever brought to the code cache. However, this implies more frequent code-cache updates.

If the code corresponding to a function or an atomic action is the code-splitting target, there is no unnecessary code duplication, as every named piece of code is a stand-alone unit. In this case, an indirect-call overhead has to be paid for each function call.

Even though each of these options could be supported by the C+- translator, we decided to split the code into pieces that correspond to atomic actions of processes.

This was the least-complicated and the best-understood approach, and it still allowed us to provide an experimental testbed that can be used to determine the effect of code-splitting granularity on the machine performance. Code of a function is linked with every atomic action that invokes it. Some of the runtime-system services, such as sending messages and creating new processes, are accessed by virtually every user process, and replicating that code would be equivalent to including a large fraction of the runtime system in the code of each user-process atomic action. Access to these services is through the indirect-function-call mechanism, but its specification is left entirely to the runtime-system implementation [10]. We consider this an acceptable compromise, particularly because any efficient code-caching policy must distinguish such often-used code anyway.

Chapter 4

The Mosaic C

4.1 Multicomputer Architecture

In its evolution away from the sequential computer organization, the multicomputer variety of MIMD concurrent computers has, from the very beginning, acknowledged the importance of locality in VLSI systems [75]. A multicomputer consists of a collection of computing nodes connected with a communication network (Figure 4.1). Each node is, typically, a sequential computer, with its own processor and memory.

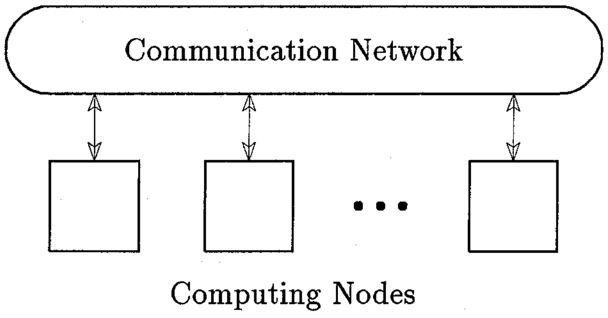


Figure 4.1: Multicomputer Architecture

These computers communicate data and synchronize their activities by exchanging messages through the communication network. Distinct mechanisms are used for processor-memory and inter-processor communication, the first optimized for minimum latency, the second for maximum bandwidth. Unlike inter-process communication latency, which can be covered up by excess concurrency (with multiple processes per node), there is no way to compensate for a lack of communication bandwidth [2, 48]. In multicomputers, the components involved in the latency-sensitive communication — processor and memory, are placed physically close — within the node. The use of these distinct communication mechanisms is why the multicomputer architecture has proven to be so cost-effective and scalable [5, 77]. Even though the question of the preferred programming model is the subject of much dispute, the majority of contemporary concurrent computers, regardless of their primary programming model, are built either as pure multicomputers, or as multicomputers with some additional hardware support [96, 12, 53, 1, 64, 79].

The communication bandwidth and latency are important figures of merit of a multicomputer. The bandwidth is limited by the communication network itself, or by the node's network interface. The latency consists of two components: the network latency — the time required for a message to traverse the network, and the software overhead — the time that the processor takes to launch a message into the network, and to accept a message arriving from the network. Since the communication network, typically, operates concurrently with the computing nodes, the network component of the message latency can be covered up with excess concurrency; the software overhead cannot. What is more, the software overhead consumes processor cycles that might have been devoted to the user's computation.

Background

The most common representatives of multicomputers, and arguably the most powerful multicomputers in existence, are computer networks. However, even when the logistics associated with using such collections of computers are taken care of, these systems are cost-effective today only for loosely-coupled concurrent computations [8]. This ineffectiveness is due to the inadequate communication bandwidth of existing networks, and to the software overheads of concurrent-programming systems that access these communication capabilities.

The history of multicomputer-design efforts is the history of attempts to increase the available communication bandwidth, and to reduce the communication latency, thereby enlarging the application span to include more tightly-coupled concurrent computations. While some of these attempts have been successful, we believe that most of them have been half-hearted, which has contributed to the widespread belief that "multicomputers are harder to program than multiprocessors". One aim of this thesis is to make the case that the programming-model issue can be separated from the machine-architecture issue.

In Chapter 2, we described a programming model with support for both shared-variable and message-passing programming paradigms. In the remainder of this chapter, we shall present the architecture of a multicomputer, the Mosaic. This architecture, although very simple, is a platform on which an efficient implementation of this programming model can be built.

4.2 The Mosaic Node

Most contemporary multicomputers adopt a node complexity that requires a circuit board per node (medium-grain nodes) [78, 54]. A stipulation of the Mosaic project was that the complexity of a Mosaic node be determined by the silicon complexity available on a single chip with reasonable ($\approx$ 50%) yield. This requirement was motivated primarily by the large disparity in performance and density between on-chip and inter-chip interconnection technology [7]. Nodes of even finer granularity are feasible, but, at the present state of fabrication technology, smaller nodes quickly reach a point where they are too small to hold even their own program code. An additional consideration was that we perceived single-chip-node multicomputers to represent a not-sufficiently-explored point in the design space of multicomputers [79]. The ultimate motivation for single-chip nodes came from realization that fine-grain multicomputers could have a much larger application span than their medium-grain counterparts [79]: they can deliver cost-effective computing in small, embedded configurations, as well as in large ensembles.

The organization of the Mosaic node, as shown in Figure 4.2, is centered around the memory bus. This bus connects the dynamic RAM (dRAM) and bootstrap

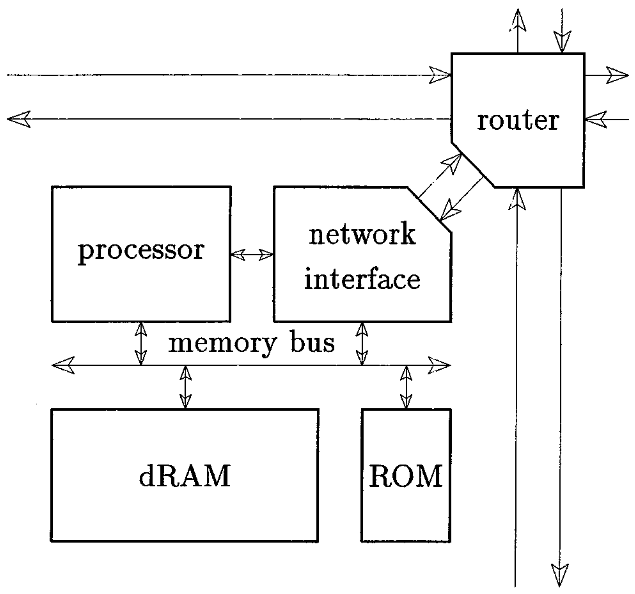


Figure 4.2: The Mosaic Node

ROM on one side with the instruction-interpreting processor and the communication-network interface on the other side. The node router, although logically this node's part of the communication network, is part of the Mosaic chip. A plot of the layout of the Mosaic chip is shown in Figure 4.3.

A more complete description of the Mosaic node and of other Mosaic assemblies can be found in [79]. In this thesis, we shall focus on those architectural issues that are fundamental to our programming model, and, in particular, on achieving low hardware and software communication overhead.

4.2.1 The Mosaic Router

The communication network of the Mosaic multicomputer is a two-dimensional, bidirectional mesh. The analysis of the network performance and arguments for employing this particular network can be found in [67, 68].

Each communication channel is an asynchronous, byte-wide link with a throughput of $60\frac{MB}{s}$. The router provides packet communication and deadlock-free, dimension-order, cut-through routing. The detailed description of the Mosaic router is provided in [81].

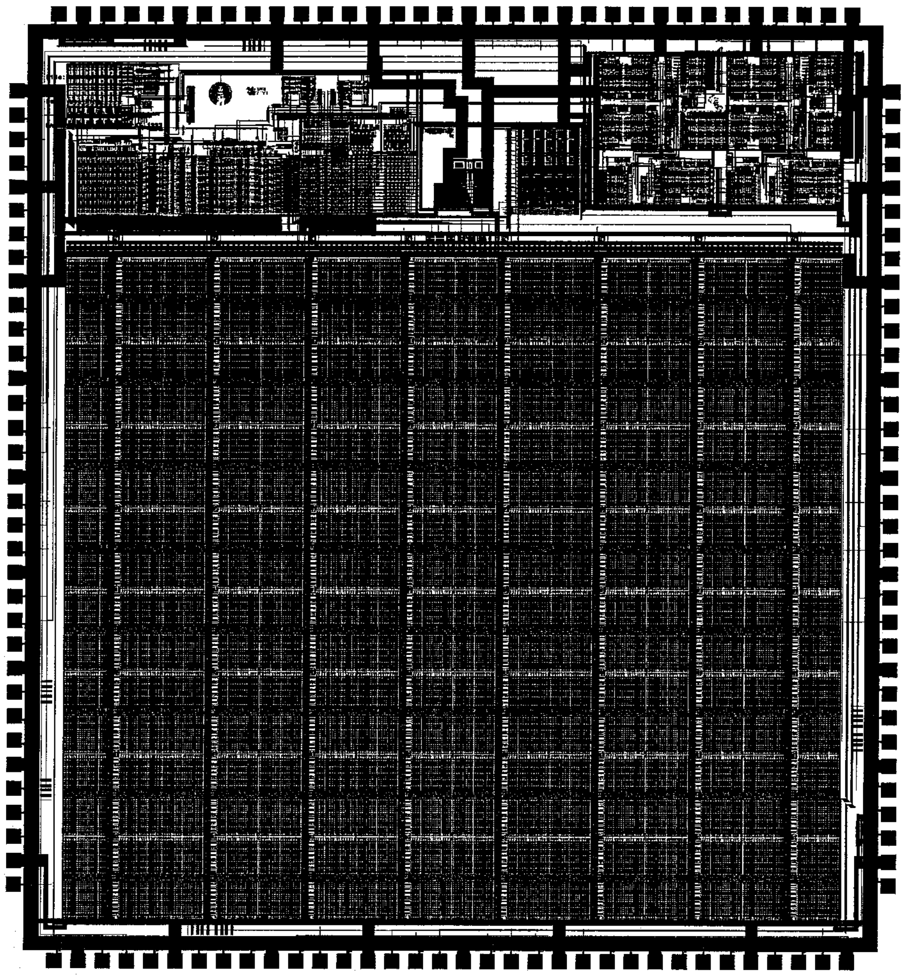


Figure 4.3: The Mosaic-C Chip: 9.25mm $\times$ 10.00mm, 1.2μm SCMOS technology, 0.5W at 5V and 30MHz

The largest configuration supported with the current version of the router is $128 \times 128 = 2^{14}$ nodes. Larger ensembles can be supported, but messages would have to be relayed in software when the distance traveled along any dimension exceeds 127. From the point of view of the rest of the computing node, the network is a bidirectional communication link with all other nodes. In a non-congested network, this communication link provides $60\frac{MB}{s}$ bandwidth in and out of the node, with the communication-establishing latency of 30ns per hop.

4.2.2 The Dynamic RAM

The Mosaic dRAM is by far the largest part of the Mosaic chip (63%), and the most precious resource of a Mosaic node. The dRAM has been described in detail in [87]. From the standpoint of the rest of the node, this memory is a single-clock-cycle dynamic RAM, operating in a pipeline mode (Figure 4.4). The memory access is allocated on a per-clock-cycle basis to one of the four independent address sources competing for the memory access: the processor, the send and the receive parts of the network interface, and the memory-refresh mechanism.

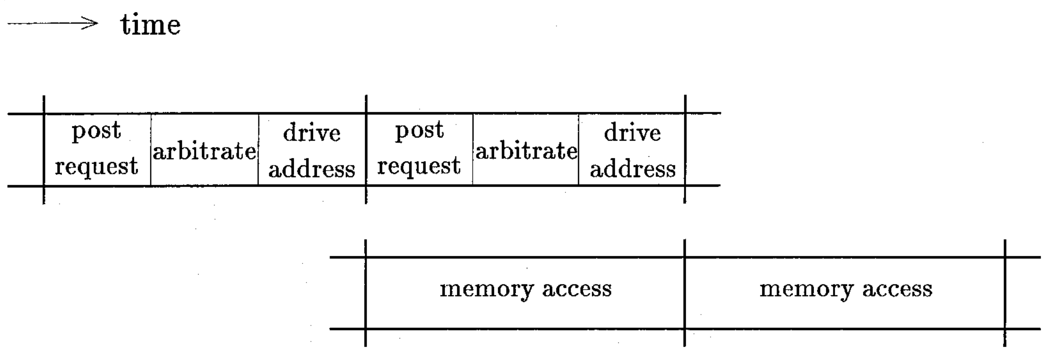


Figure 4.4: Memory-Access Pipeline

4.2.3 The Processor and the Network Interface

The Mosaic processor is a 16-bit, microprogrammed engine, with one-, two-, and three-word instructions, and with an average of approximately three clock cycles per instruction. The network-interface is a direct-memory-access device that transfers data and performs reliable synchronization between the asynchronous router and the synchronous memory (Chapter 5).

The prominent features of the Mosaic node architecture, those that make it particularly appropriate for a multicomputer node, are the interaction between the processor and the network interface. Although low-latency handling of messages was imperative for the Mosaic, message-handling capabilities had to be sufficiently general to allow experiments with different message-handling strategies. These two, often contradicting goals were achieved in part through the two-context-processor architecture, and in part by providing a set of highly-efficient low-level message primitives.

Two-Context Architecture

The most unusual feature of the processor is its two-context architecture: Each context has its own program counter, status register, and eight general registers. There are eight additional general registers that are shared between the two contexts.

Typically, one context is used for running programs, and the other for message handling under interrupts. In this regime of operation, the interrupt context can be thought of as an extension of the network interface.

A context switch is performed only between instructions, so it may be postponed for several cycles. However, once initiated, a context switch takes zero time, since no processor state needs to be saved.

Messages and Interrupts

The message-handling and interrupt-handling operations are centered around a small set of dedicated registers. All of these registers can be accessed both by the processor and by the network interface. For example:

- To *send* a message, processor must specify where the message is located in memory, and which node to send it to. This send operation is performed by writing into the following three registers:
  - Message Send Pointer (MSP) — that points to the location of the first word of the message;
  - Message Send Limit (MSL) — that points to the location of the last word of the message; and
  - Destination Register (DXDY) — that contains the address of the destination node encoded as the relative distance in X and Y dimensions of the mesh network.

Writing into DXDY triggers the network interface. The network interface then starts transferring the data from the memory, increments MSP, and continues until the MSP exceeds MSL.

- The message *receive* operation is initiated by the network interface. The processor must specify where the message is to be written by setting:
  - Message Receive Pointer (MRP) — that points to the first available memory location; and
  - Message Receive Limit (MRL) — that points to the last available memory location.
- The network interface generates three distinct *interrupts*, corresponding to the following conditions: when a complete message has been sent, when a complete message has been received, and when the receive buffer has been exhausted. The interrupts are handled by accessing two additional registers:
  - Interrupt Status Register (ISR) — that contains three bits that correspond to the three sources of interrupts. These bits are set by the network interface and cleared by the processor.
  - Interrupt Mask Register (IMR) — that contains three bits that correspond to the three interrupt sources. Modifying these bits enables or disables their corresponding interrupts.

Program 39 shows an excerpt that the runtime system might use as an interrupt-dispatch routine. All the registers can be accessed directly from C+- through a feature provided by the Gnu C++ compiler [94].

4.3 Software Overhead of Communications

As discussed in Section 4.1, the software overhead associated with message send and receive operations is the communication bottleneck of most programming systems for

```
void    (*interrupt_table[8])() =        // a jump table filled with the
{                                        // names of routines that correspond
    software_int,                        // to various interrupt conditions
    __________recv_int,
    _____send______int,
    _____send_recv_int,
    buff___________int,
    buff______recv_int,
    buff_send______int,
    buff_send_recv_int
};


while (1)
{
    int  pending = ISR;                  // get the pending interrupts

    (*interrupt_table[pending]) ();      // dispatch to an interrupt handler

    ISR = pending;                       // writing back acknowledges all the
                                         // interrupts that have just been
                                         // serviced

    asm ("PUNT");                        // assembly instruction to return to
                                         // the other context

                                         // when the next interrupt arrives,
}                                        // we shall start here
```

Program 39: An Interrupt-Dispatch Routine

multicomputers. The software overhead for C+- programs running on the Mosaic, under the MADRE runtime system [10], is analyzed in this section. The overhead shown represents a typical case, measured in Mosaic assembly instructions. The complexity of Mosaic assembly instructions is comparable to that of a typical, load-store, RISC processor, but the number of clock cycles per instruction is approximately three. The communication support of the Mosaic processor is minimal (Section 4.2.3), and the incorporation of such support into a typical RISC processor core is arguably relatively simple. Experiments with compiling the same code for contemporary RISC processors exhibit comparable instruction counts.

We want to emphasize that all the numbers were obtained from the compiled code (our C+- to C++ translator with the Gnu C++ compiler targeted for the Mosaic).

Both user programs and runtime-system code were written in C+-, and the only programmer-specified optimization was using inline functions for critical runtime-system code. Excerpts from the source code and the produced assembly code are presented in Appendix A. In our experience, the compiled code was typically only ten percent less efficient than the best hand-coded assembly we could produce, not nearly sufficient to justify such an effort.

Message Sending

Figure 4.5 illustrates the activity of two processes, placed on two different nodes, in direct communication. The producer process sends an infinite stream of empty messages to the consumer process.

The send overhead, in the typical case, consists of 43 Mosaic instructions. A major portion of the send overhead are the 17 instructions that allocate the space and update the send queue. Since the send queue is guaranteed to be used in the FIFO regime, it is implemented as a simple circular buffer. One way of reducing the overhead associated with the send queue is to have the processor write the message contents directly into the network instead of into the memory [64]. However, this approach would introduce too strong a coupling between the processor and the communication network. For example, an interrupt during the message-send operation would block the network; similarly, the blocked network would prevent the processor from doing other useful work, or would result in more-frequent context-switching.

Another big contributor to the software overhead on the sending side is the DXDY conversion. The routing hardware requires the first word of any message to consist of two bytes of sign-and-magnitude-encoded distance in the x and y routing dimensions. Extending the Mosaic instruction set with an instruction that would compute this relative distance would reduce the sender overhead with a negligible price in chip area. In general, the approach adopted by the Mosaic design team has been minimalist, avoiding such special instructions. However, the discrepancy between the byte-size sign-and-magnitude encoding required by the router and the

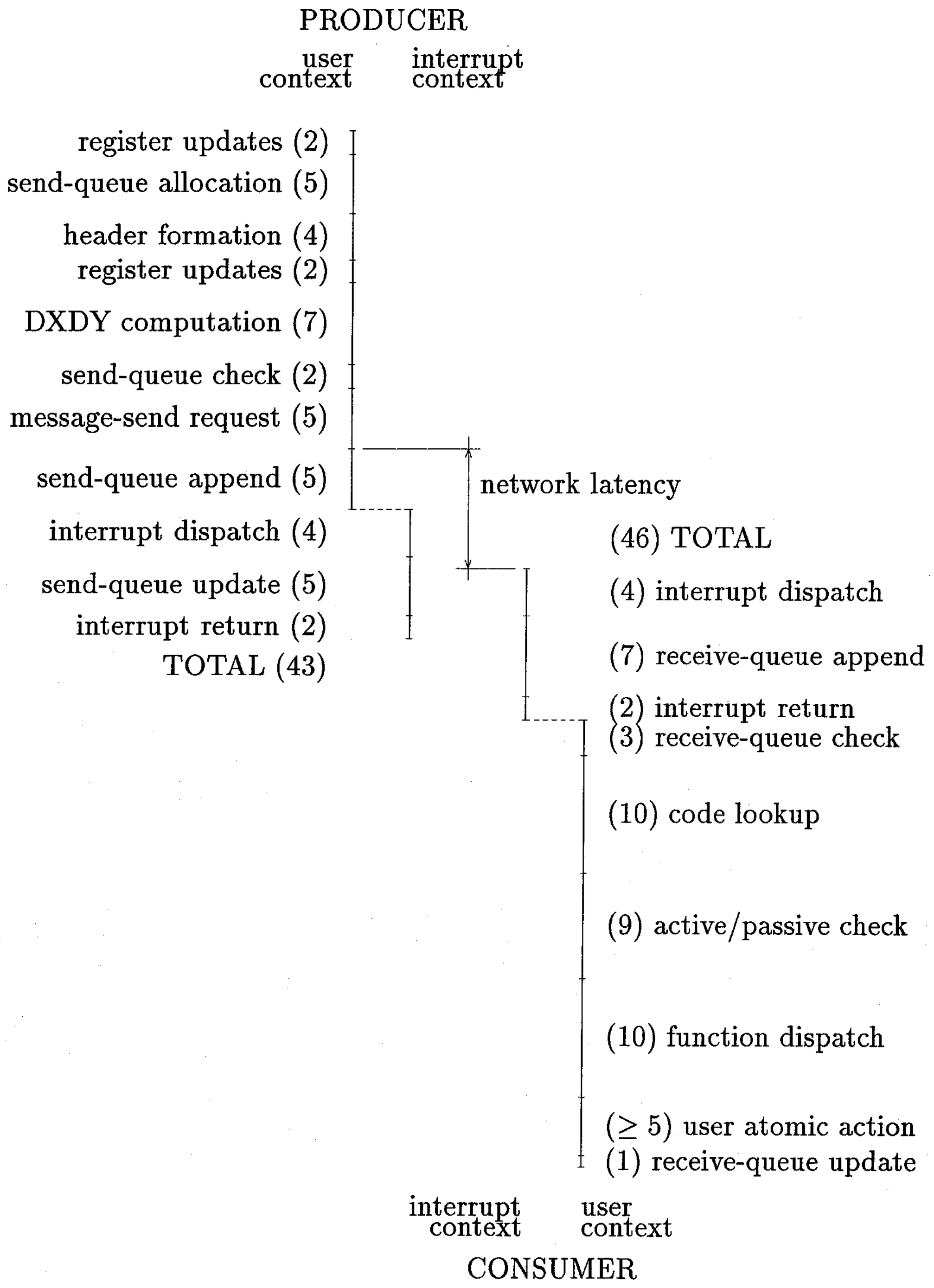


Figure 4.5: Components of the Software Overhead of a Communication

word-size two's-complement arithmetic capabilities of the Mosaic processor must be regarded as a design oversight.

The message header in this example consists of the size of the message, the node number and the address of the destination process, and the identifier of the atomic action to be invoked.

The four instructions designated as miscellaneous register updates could not be associated with any particular message-handling operation. These instructions are the necessary glue associated with register allocation between message-sending components.

The code could be optimized for case of short messages and lightly-loaded network by checking whether the message has been launched into the network to avoid the interrupt dispatch and return.

Message Receiving

The typical receive overhead totals 46 Mosaic instructions, about a quarter of which is spent in receive-queue management. Unlike with the send queue, consumption of the receive queue depends on user programs. Messages can be consumed out of order either when using the active/passive mechanism or when suspending while waiting for the RPC reply. In our experience, programming models and notations that do not allow such message discretion [4, 9] merely dump the burden of buffer management on the programmer. In the case of programs with regular communication patterns, this requirement is not too demanding [31]. However, in the case of highly-dynamic communication patterns, the buffer management becomes a significant part of the programming effort. This problem is exacerbated on fine-grain multicomputers, where local node resources may not be sufficient to absorb the receive-queue fluctuations. One may be tempted to deal with receive-queue overflow by blocking the incoming message traffic, which will eventually block the message source. Such an approach introduces negative feedback to equalize the communication rates of the producer and the consumer. Unfortunately, this approach violates the consumption assumption that routing networks typically require to guarantee freedom from deadlock [27]. The approach that the MADRE runtime system takes in dealing with the receive queue overflow is to export messages to other nodes, and retrieve them later. For a detailed description, see [10].

Once the decision is made that the messages can be consumed out of order, the runtime system must provide a general implementation of memory allocation. The performance of receive-queue management depends on the communication pattern of the application at hand. We have experimented with two approaches to receive-queue memory allocation:

- *Optimistic,* designed for minimum latency, and optimized for the case when the majority of messages are consumed in order. A circular buffer can be used, and the general memory allocator need only be invoked to allocate space in which to copy the messages not consumed in order. The software overheads of Figure 4.5 are obtained with this approach.

- *Pessimistic,* designed for maximum throughput, and optimized for the case when there is enough irregularity in the message consumption that the copying costs of the optimistic approach outweigh the advantages of simple memory allocation. According to this approach, implemented in MADRE, the header of a message is received into a small, dedicated buffer, and, upon buffer-full interrupt, the buffer of correct size is allocated for the rest of the message. This approach adds approximately twenty instructions to the receive overhead in the typical case, but the messages are never copied between memory buffers.

The code-lookup overhead depends on the algorithm used, and the ten instructions shown represent the overhead typical of a successful lookup operation. If the code is not present, it is located and brought from another node.

Chapter 5

Pipeline Synchronization

5.1 Introduction

The design method and tools for VLSI are oriented almost exclusively towards the design of clocked, synchronous systems. Except for a few notable examples [59], contemporary processor and memory technology is designed and used exclusively within the synchronous framework. The one area in which the asynchronous design style has been successful in upsetting the dominance of the synchronous circuitry is in high-performance data communication and routing [81]. Even though there are mechanisms for minimizing clock skew [7, 63], the performance penalty for maintaining clock coherency in physically large systems is prohibitive.

One of the premises of the Mosaic project was that the machine be (at least in principle) arbitrarily extensible, so the communication network of the Mosaic node is asynchronous. The rest of the Mosaic node, however, is a synchronous design. The Mosaic network interface performs data transfer between the $60\frac{MB}{s}$ asynchronous communication link and the $60\frac{MB}{s}$ synchronous memory bus. A large Mosaic ensemble with 2^{14} nodes has a worst case of 10^{12} synchronization events per second, or almost 10^{20} synchronization events per year. Just to be able to reduce the rate of synchronization failure [74] to once per year, the best synchronizers we know how to build in $1.2\mu m$ CMOS technology require about half of the available clock period. We were clearly very close to a point where small, unexpected process variations could result in nasty surprises. To deal with this difficult synchronization problem, in which synchronization and data rates are similar, we developed a technique that can sustain the full bandwidth and achieve arbitrarily-low, non-zero probability of failure P_f, with the price in both latency and chip area of $\mathcal{O}(\log \frac{1}{P_f})$.

5.2 Problem Specification

Given the required rate of data transfer of E events per second between an asynchronous and a synchronous system, with each event delivering W bits of information, design an interface that will guarantee that the probability of synchronization failure be less than a given $P_f > 0$.

The assumption is that the flow control is implemented as either a two-phase or four-phase signaling protocol with bundled data [74].

5.3 Existing Solutions

The standard approach to interfacing asynchronous and synchronous systems is to use a *synchronizer* at each synchronous-system input. A synchronizer is a circuit that attempts to solve one of the following two, equivalent, decision problems characteristic of digital systems: given an input signal and a time reference, decide whether the input signal makes a transition before or after the reference; or, given an input signal and a voltage reference, decide whether the input voltage is higher or lower than the reference. As shown by the theoretical work [56, 98], and by a wealth of experimental evidence [16, 66, 71], any system attempting to solve one of these two problems is of limited reliability: In addition to two stable states, corresponding to two decision points, the system has a metastable state. One cannot put a bound on how long the system may require to exit this metastable state. However, a number of simple synchronizer implementations have been demonstrated [74, 72] that can guarantee that the probability that the metastable state will last longer than t_m decreases exponentially with t_m:

$$P_f = e^{-\frac{t_m}{\tau_0}}, \tag{5.1}$$

where τ_0 is a characteristic of the implementation. Therefore, to achieve a sufficiently small probability of synchronization failure of a single asynchronous input, all that is required is to allow a sufficiently long time for the synchronizer to exit the metastable state.

Let us apply this single-synchronizer approach to problem of Section 5.2 (Figure 5.1). We shall use a synchronizer that compares the arrival time of

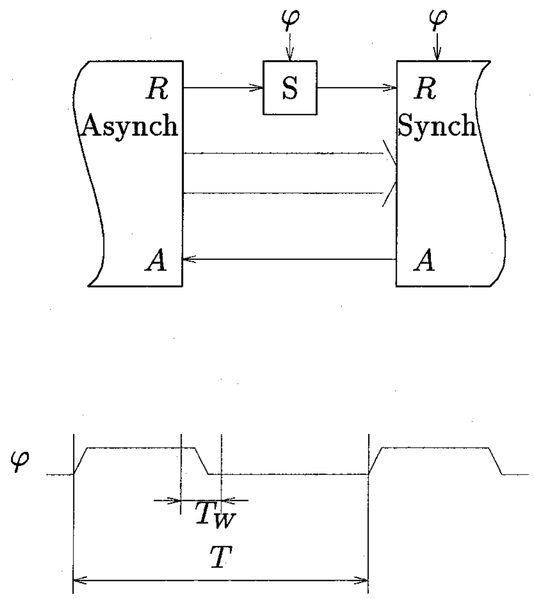


Figure 5.1: Interfacing synchronous and asynchronous systems

its asynchronous input with the time reference defined by the down-going edge of its clock input. When the asynchronous input changes state within T_W around the down-going edge of the clock input, the synchronizer will enter the metastable state, with the exit probability determined by Equation 5.1. Regardless of the signaling protocol used, for the synchronous system with clock period T,

$$E \leq \frac{1}{T} \tag{5.2}$$

to be able to sustain the required throughput of E events per second. From now on, we shall assume that E is equal to $\frac{1}{T}$, the maximum attainable throughput. We shall assume that the implementation of the signaling protocol on the synchronous and asynchronous side imposes a total overhead of T_{oh} per transferred data item. This assumption implies that we can allow at most $T - T_{oh}$ for the synchronizer to exit the metastable state, and that there is a lower bound on the probability of synchronization failure that this simple approach can achieve, $P_f \geq e^{-\frac{T-T_{oh}}{\tau_0}}$.

Widening the Data Path

One possible approach to improve on this simple solution is to change the data representation: instead of transferring W bits every $\frac{1}{E}$ seconds, we can transfer kW bits every $\frac{k}{E}$ seconds in order to allow k times as much time for synchronization (Figure 5.2). A simple variant of this solution requires that all communications consist of multiples of k data units. A less-restrictive solution is equivalent to the solution presented next.

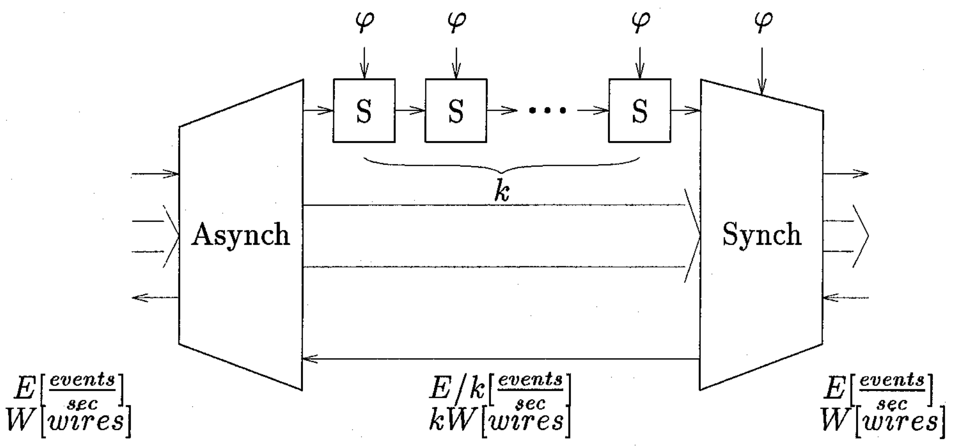


Figure 5.2: Widening the data path

Deriving Signals With Less Than $E[\frac{events}{sec}]$

An alternative approach is to change the control representation: instead of using a request signal that changes state for every data item transferred, we can derive a request signal that denotes that there are at least k data items to be transferred (Figure 5.3).

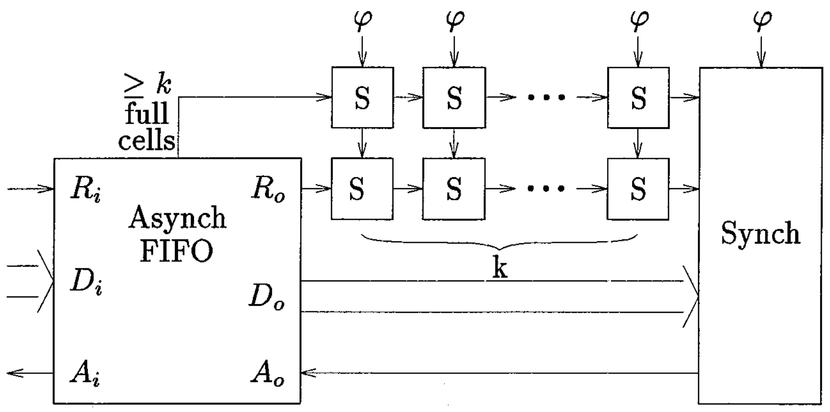


Figure 5.3: Deriving signals with less then $E[\frac{events}{sec}]$

Stretchable Clocks

The solution illustrated in this section does not, strictly speaking, conform to the problem specification, but is presented here for completeness. This solution achieves a P_f of exactly 0, but it does not maintain the required bandwidth. The synchronizer must be able to detect that it is in the metastable condition, and it stretches the clock cycle of the synchronous system until the metastability has been resolved. Instead of synchronizing asynchronous input to the clock, the clock is synchronized to the asynchronous input [74, 66, 72]. When there is more than one asynchronous input, the clock must be stretched until all the synchronizers have exited the metastable state.

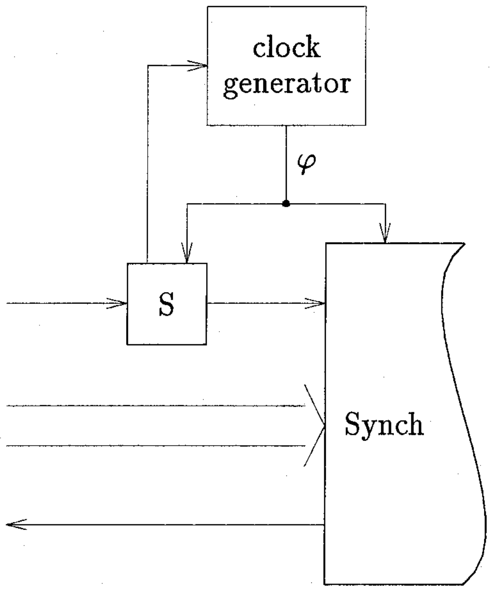


Figure 5.4: Stretchable clock

5.4 Pipeline Synchronization

A common denominator for all the solutions presented in Section 5.3 is that they treat synchronization as a "one-shot" process: signal events (transitions) are either asynchronous or synchronous. In this section, we shall characterize signal S with the *probability distribution of signal-event arrival time*, $p_S(t)$, with respect to time as measured at the synchronous system to which S is an input. In some cases, we shall be interested only in the arrival time of positive or negative edges, and shall use symbols $p_{S\uparrow}(t)$ and $p_{S\downarrow}(t)$, respectively. The two graphs in Figure 5.5 represent two typical cases for $p_S(t)$, one for a synchronous and one for an asynchronous signal. The parameter T is equal to the clock period of the synchronous system, and $p_S(t)$ is a periodic function with period T:

$$\forall t_0 : \int_{t_0}^{t_0+T} p_S(t)dt = 1. \tag{5.3}$$

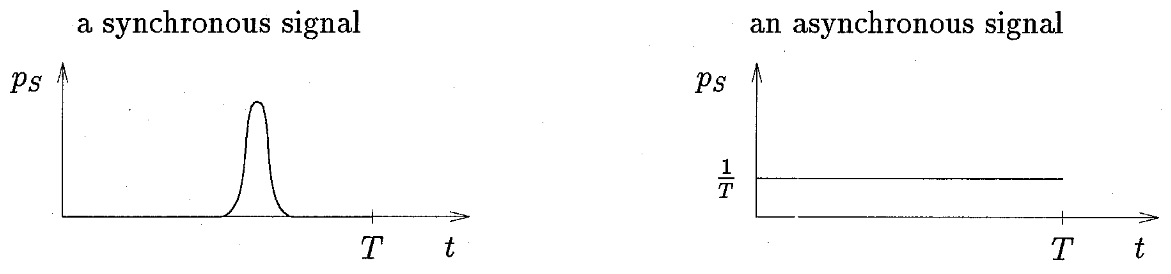


Figure 5.5: Probability distribution of signal-event arrival time

What makes a signal synchronous with respect to some clock is that events on that signal satisfy some setup and/or hold time [74] with respect to the clock. Relating to Figure 5.5, we use the following definition:

Definition 1 *Signal S is* synchronous *with respect to some clock if there exists a non-empty time segment $[t_s, t_h]$ in which $p_S(t) = 0$.*

The usual assumption for asynchronous signals is that each arrival time is equally probable (Figure 5.5), or that we have no knowledge of the probability distribution. We define the asynchronous signals as all non-synchronous ones:

Definition 2 *Signal S is* asynchronous *with respect to some clock if there is no non-empty time segment in which $p_S(t) = 0$.*

The probability distribution for a signal contains more information than is often necessary, so we shall also introduce a simpler metric to characterize "how asynchronous a signal is."

Definition 3 *For a signal S, and a given time window, T_W, $0 < T_W \leq T$, the* asynchronicity *is defined as:*

$$\mathcal{A}_S(T_W) = \min_{0 \leq t_0 < T} \int_{t_0}^{t_0+T_W} p_S(t)dt. \tag{5.4}$$

The intuitive meaning of asynchronicity is that given a signal S and a synchronous sampling device D with setup time T_{setup} and hold time T_{hold}, $\mathcal{A}_S(T_{setup} + T_{hold})$ is the lowest probability of metastable behavior that can be achieved when sampling S with the device D. For example:

- For synchronous signals, according to Definition 1, we can find $T_{max} > 0$ such that $\forall T_W \in (0, T_{max}], \mathcal{A}_S(T_W) = 0$. Therefore, if we can build a sampling device with $T_{setup} + T_{hold} < T_{max}$, it can sample S without ever exhibiting metastable behavior.

- For asynchronous signals with uniform probability distribution (Figure 5.5), $\mathcal{A}_S(T_W) = \frac{T_W}{T}$. From Equation 5.3, it is easy to show that this is the worst case for asynchronicity.

- An asynchronicity of 1 corresponds to a hypothetical, "malicious," asynchronous signal: no matter how we position the sampling window, and no matter how small we make it, the probability distribution will be a unit-size delta function positioned within our chosen time window. This case has to be assumed when interfacing to a signal of unknown probability distribution.

In the remainder of this chapter we shall show how we can build circuits that transform the arrival-time probability distribution of signals in a way that reduces their asynchronicity, and how to use these circuits to build pipeline synchronizers.

5.4.1 The Mutual-Exclusion Element

The mutual-exclusion (ME) element is a variant of synchronizer. The ME element compares the signal-arrival time at the two asynchronous requests, and generates mutually exclusive acknowledge signals. If the ME element enters the metastable state, no acknowledge is granted until the ME element exits this state. Figure 5.6 shows the symbol we use for the mutual-exclusion element, and a CMOS implementation that was first introduced by Seitz [74, 73].

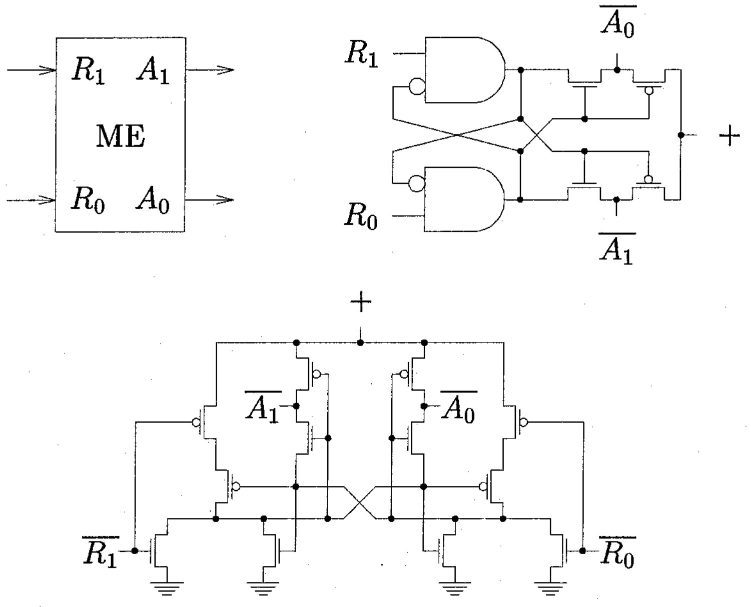


Figure 5.6: Mutual-exclusion element

Let $t_{S\uparrow}^{(j)}$ and $t_{S\downarrow}^{(j)}$ denote the time of the j^{th} up-going edge, and the time of the j^{th} down-going edge, respectively, of signal S. Let $\tau_{E_1E_2}$ denote a causal delay from one event (signal edge), E_1, to another, E_2. Then Equations 5.5 and 5.6 are the requirements that any implementation of the ME element must satisfy:

$$\begin{aligned} \infty > t_{A_i\uparrow}^{(j)} &\geq t_{R_i\uparrow}^{(j)} + \tau_{R_i\uparrow A_i\uparrow}, \quad j \geq 0, \quad i = 0,1 \\ t_{A_i\downarrow}^{(j)} &= t_{R_i\downarrow}^{(j)} + \tau_{R_i\downarrow A_i\downarrow}, \quad j \geq 0, \quad i = 0,1 \end{aligned} \tag{5.5}$$

$$A_0 \wedge A_1 = 0. \tag{5.6}$$

$\tau_{R_i\uparrow A_i\uparrow}$ and $\tau_{R_i\downarrow A_i\downarrow}$ depend on the implementation, and are typically approximately equal. Without loss of generality, we shall assume that $\tau_{R_i\uparrow A_i\uparrow} = \tau_{R_i\downarrow A_i\downarrow} = \tau_{RA}$.

The environment of the ME element is obliged to behave according to the following specification:

$$\begin{array}{rcll} t_{R_i\uparrow}^{(j)} & > & \begin{cases} 0, & j=0, \\ t_{A_i\downarrow}^{(j-1)}, & j>0, \end{cases} & i=0,1 \\ \infty > t_{R_i\downarrow}^{(j)} & > & t_{A_i\uparrow}^{(j)}, \quad j \geq 0, & i=0,1. \end{array} \tag{5.7}$$

Let us now present an example illustrating how an ME element can be used to reduce the asynchronicity of signals. We shall examine what happens when one of the request inputs of the ME element is connected to a periodic signal, φ, and the other input to an asynchronous signal with a uniform probability distribution of up-going edges (Figure 5.7).

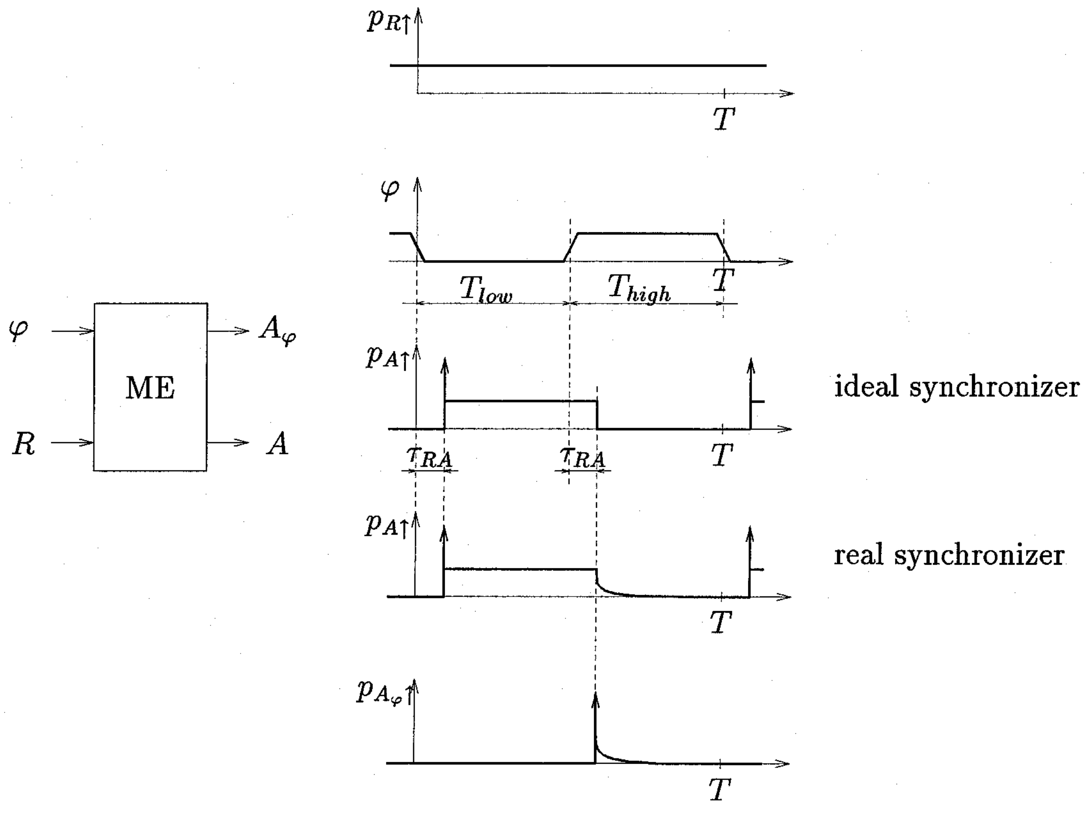


Figure 5.7: ME element as a synchronizer

If an asynchronous event arrives while $\varphi = 0$, there is no contest in the ME element, and, after τ_{RA}, the acknowledge will be granted. If the arrival time is during the period in which $\varphi = 1$, the acknowledge will be postponed until $\varphi = 0$ again. The behavior in this idealized case, which does not take metastability into account, is what we would like synchronizers to achieve. The delta function corresponds to all the requests that occur during the $\varphi = 1$ period, and are acknowledged after $\varphi = 0$ again. The area of the delta function is such that Equation 5.3 holds.

As discussed in Section 5.3, any implementation of ME element will exhibit metastable behavior if an input event occurs within some narrow time window, T_W, around the time when φ makes the transition from low to high. The probability that the ME element will remain in a metastable state longer than t_m decays exponentially with t_m (Equation 5.1). Upon exiting the metastable state, the ME element generates an acknowledge either on the A or A_φ output. Therefore, the ME element transforms an input signal of asynchronicity $\mathcal{A}_{R\uparrow}(T_W) = \frac{T_W}{T}$ into the output signal of asynchronicity

$$\mathcal{A}_{A\uparrow}(T_W) = \frac{T_W}{2T}(e^{-\frac{T_{high}-T_W}{\tau_0}} - e^{-\frac{T_{high}}{\tau_0}}) = \frac{T_W}{2T}e^{-\frac{T_{high}}{\tau_0}}(e^{\frac{T_W}{\tau_0}} - 1). \tag{5.8}$$

For a typical implementation, $T_W \ll \tau_0$, and

$$\mathcal{A}_A(T_W) \approx \frac{T_W}{2T}\frac{T_W}{\tau_0}e^{-\frac{T_{high}}{\tau_0}}, \tag{5.9}$$

so the ME element reduces the asynchronicity of the input signal by a factor that is exponential in the time allowed for synchronization.

Since the other request input is connected to a clock, the metastable state cannot last longer than T_{high}, but the A signal is still asynchronous, according to Definition 2. The ME element attempts to do the synchronization in the allotted time, T_{high}, and if it doesn't succeed, it is forced out of the metastable state. We trade one uncertainty (whether the input made a transition), for another (whether the ME element is in a metastable state). Connecting a request input to a clock violates the requirements that the ME element imposes on its environment (Equation 5.7). If the ME element is leaving the metastable state and generating A_φ just as the clock makes the down-going transition, this violation can result in a glitch on the A_φ output. Therefore, A_φ must not be used as an input to any circuit, and need not even be generated. As long as R does not violate the ME element requirements, there will be no spurious signals on the A output.

Synchronizers

We shall postulate existence of three types of synchronizers, as shown in Figure 5.8, depending on whether they synchronize only the up-going, only the down-going, or both edges of the input signal. CMOS implementations of these synchronizers are presented in Section 5.5.

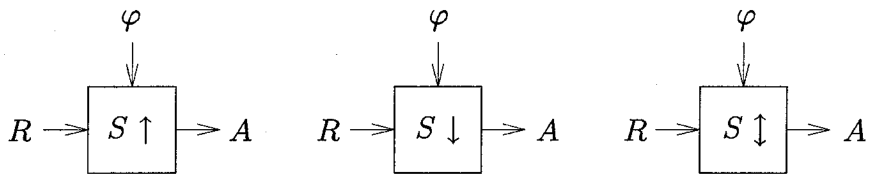


Figure 5.8: Synchronizers

These synchronizers will be characterized with the time window, T_W, centered around the *down-going edge* of the clock. Unless otherwise specified, the phrase "coincident with the down-going edge" of some clock, will mean "within T_W around the down-going edge" of that clock.

If the input-event arrival time is of uniform distribution (where the input event is an up-going edge, a down-going edge, or a signal transition), the output-event distribution is as shown in Figure 5.9, with $\tau_S = \tau_{RA}$.

The input-output behavior of each synchronizer clocked with a periodic signal of period $T = T_{low} + T_{high}$ can be modeled as a (non-deterministic) variable delay τ_V, where

$$\tau_S \leq \tau_V \leq T_{low} + \tau_S. \tag{5.10}$$

5.4.2 Two-Phase-Protocol FIFO

Figure 5.10 shows the symbol we use to represent a two-phase-protocol FIFO element.

A number of different implementations of this FIFO cell can be found in [92, 81, 14]. For the purposes of this thesis, we shall not concern ourselves with

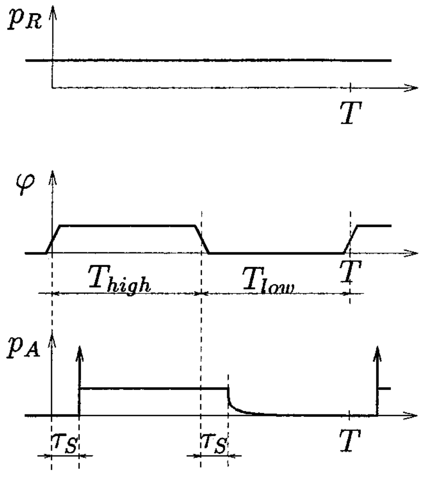


Figure 5.9: Synchronizer input-output specification

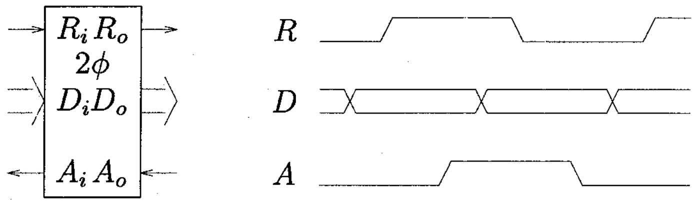


Figure 5.10: Two-phase-protocol FIFO element

details of any particular implementation. To be a valid implementation, the circuit has to behave according to the following behavioral specification [57]:

$$* \, [[R_i]; A_i, R_o; [A_o]]. \tag{5.11}$$

The operational description of the above specification is to repeat forever ($*[]$) the following sequence (;) of actions: wait ([]) for an event (signal transition) on R_i; generate concurrent events (,) on A_i and R_o; and wait for an event on A_o.

Each signal's events are numbered from zero. A formal definition corresponding to the specification in Equation 5.11 is as follows [14]:

$$t_{A_i}^{(j)} = \begin{cases} t_{R_i}^{(0)} + \tau_{R_i A_i}, & j = 0 \\ \max(t_{R_i}^{(j)} + \tau_{R_i A_i}, t_{A_o}^{(j-1)} + \tau_{A_o A_i}), & j > 0 \end{cases}, \tag{5.12}$$

$$t_{R_o}^{(j)} = \begin{cases} t_{R_i}^{(0)} + \tau_{R_i R_o}, & j = 0 \\ \max(t_{R_i}^{(j)} + \tau_{R_i R_o}, t_{A_o}^{(j-1)} + \tau_{A_o R_o}), & j > 0 \end{cases} . \tag{5.13}$$

The environment is obliged to behave according to the following specification:

$$t_{R_i}^{(j)} > \begin{cases} 0, & j = 0 \\ t_{A_i}^{(j-1)}, & j > 0 \end{cases} , \tag{5.14}$$

$$t_{A_o}^{(j)} > t_{R_o}^{(j)}, \quad j \geq 0 \; . \tag{5.15}$$

The specification does not mention the data-handling requirements, because they do not affect the synchronization issues. Since there is a well-defined time relationship between the data and the control signals, the only source of uncertainty is the time when the input signals (R_i and A_o) make transitions.

Figure 5.11 shows the dependency graph for a two-phase-protocol FIFO element. Nodes of the graph represent signal events, and edges represent the dependencies between the events. The τ values associated with solid edges are delays characteristic of a particular FIFO implementation. The dashed edges correspond to event dependencies that are maintained by the environment; the only property that can be assumed about the delays associated with these dependencies is that they are positive.

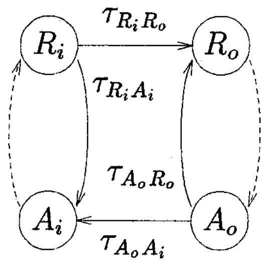


Figure 5.11: The dependency graph of a two-phase-protocol FIFO element

Let $p_{R_i}(t)$ and $p_{A_o}(t)$ represent the probability distributions of the signal-event arrival times for the two inputs of a two-phase-protocol FIFO element, and, for the moment, disregard that these two distributions are not independent (Equations 5.12 through 5.15). The upper bound for the output-event probability distributions is:

$$\begin{aligned} p_{R_o}(t) &\leq p_{R_i}(t-\tau_{R_iR_o}) + p_{A_o}(t-\tau_{A_oR_o}), \\ p_{A_i}(t) &\leq p_{R_i}(t-\tau_{R_iA_i}) + p_{A_o}(t-\tau_{A_oA_i}). \end{aligned} \tag{5.16}$$

5.4.3 Pipeline Synchronizer

The block diagram of a pipeline synchronizer that can be used to interface an asynchronous input data stream to a synchronous system is shown in Figure 5.12.

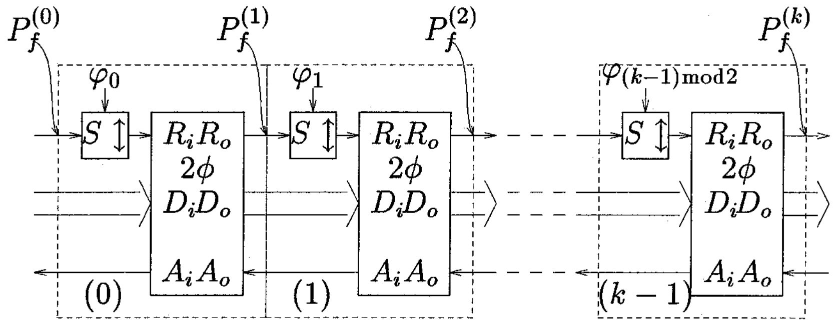


Figure 5.12: Asynchronous-input, synchronous-output pipeline synchronizer

φ_0 and φ_1 (Figure 5.13) are two-phase, non-overlapping, clock signals often used for internal clocking of CMOS chips. For simplicity, we shall assume that:

$$\begin{aligned} T_0 &= T_1 > 0 \\ T_{01} &= T_{10} > 0 \end{aligned} . \tag{5.17}$$

One can show that this simplification does not qualitatively affect the results that we shall present in this section. The FIFO elements are two-phase-protocol asynchronous FIFO elements described in Section 5.4.2. The synchronizers are the symmetrical elements described in Section 5.4.1 and Section 5.5. The wires connecting the two have zero delay (the wire delay is absorbed into the FIFO elements).

The principal claim is that *the data-stream synchronization can be done in stages, along with the data flow.* In the following section we shall prove that the probability of synchronization failure at the synchronous end of the structure in Figure 5.12

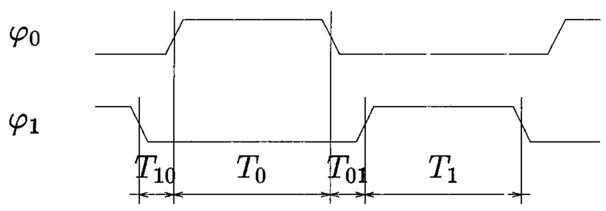


Figure 5.13: Two-phase non-overlapping clocks

decreases exponentially with the number of stages:

$$P_f^{(k)} = P_f^{(0)} e^{-\frac{k(\frac{T}{2}-T_{oh})}{\tau_0}}, \tag{5.18}$$

where T_{oh} is the implementation-dependent overhead. Therefore, to achieve the desired P_f at the synchronous end, one needs at least

$$\left\lceil \frac{\tau_0}{\frac{T}{2} - T_{oh}} \log \frac{P_f{}^{(0)}}{P_f} \right\rceil \tag{5.19}$$

stages. The area complexity of this solution is $\mathcal{O}(\log \frac{1}{P_f})$. Both theoretical work [56, 98] and experimental evidence [16, 66, 71] show that it is not possible to synchronize asynchronous signals with a latency less then $\mathcal{O}(\log \frac{1}{P_f})$, so this solution is latency-optimal, $\Omega(\log \frac{1}{P_f})$.

It is often the case that the output sections of asynchronous data sources themselves consist of a series of FIFO elements. In these cases, it is possible to insert the synchronizer elements into the output section, and perform a part of or the entire synchronization there. The latency and area will still be $\mathcal{O}(\log \frac{1}{P_f})$, but it may be achieved with a smaller T_{oh}.

5.4.4 Correctness Proof

We shall first show that the structure in Figure 5.12 behaves as an asynchronous FIFO. Then, we shall find implementation requirements for the FIFO elements and synchronizers that are sufficient to guarantee that Equation 5.18 holds, and that the maximum throughput can be sustained. Finally, we shall find the upper bound for latency.

Proper Functional Behavior

As discussed in Section 5.4.1, the input/output behavior of a synchronizer whose clock input is connected to a periodic signal is equivalent to that of a variable delay (Equation 5.10). The introduction of a bounded (albeit variable) delay to any signal of a speed-independent implementation of the FIFO element does not affect the correctness of the circuit operation.

Probability of Metastability Failure

Figure 5.14 shows a two-stage segment of the pipeline synchronizer from Section 5.4.3.

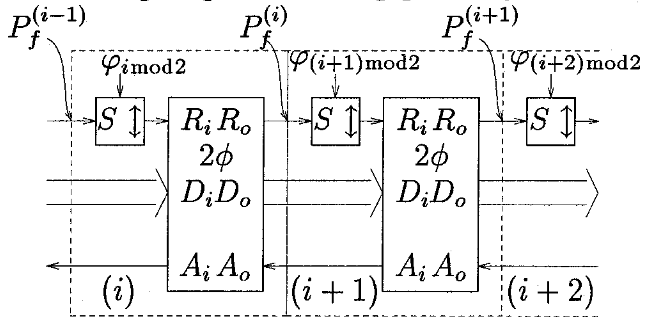


Figure 5.14: A segment of a pipeline synchronizer chain

The j^{th} event on $R_o^{(i)}$ can occur only at time:

$$t^{(j)}_{R_o^{(i)}} = t^{(j)}_{R_i^{(i)}} + \tau_{R_iR_o}, \tag{5.20}$$

or at time

$$t^{(j)}_{R_o^{(i)}} = t^{(j-1)}_{A_o^{(i)}} + \tau_{A_oR_o}. \tag{5.21}$$

This event can cause metastable behavior of the $(i+1)^{st}$ synchronizer (Section 5.4.1) only if it occurs coincidently with the down-going transition of $\varphi_{(i+1)\mathrm{mod}2}$. Therefore,

$$P_f^{(i)} \leq P_f^{(i)}(R_i) + P_f^{(i)}(A_o). \tag{5.22}$$

The first element of the sum in Equation 5.22 corresponds to the probability that an event on R_i occurs $\tau_{R_iR_o}$ before the end of the $\varphi_{(i+1)\mathrm{mod}2}$, and the second element to the probability that an event on A_o occurs $\tau_{A_oR_o}$ before the end of the $\varphi_{(i+1)\mathrm{mod}2}$.

We shall first examine the case corresponding to $P_f^{(i)}(R_i)$. If the implementation of the FIFO element satisfies the requirement

$$\tau_S + \tau_{R_iR_o} < T/2, \tag{5.23}$$

the i^{th} synchronizer will guarantee that this can occur only if it was in a metastable state for at least

$$t_m = T/2 - \tau_S - \tau_{R_iR_o}. \tag{5.24}$$

Therefore,

$$P_f^{(i)}(R_i) \leq P_f^{(i-1)} e^{-\frac{T/2-\tau_S-\tau_{R_iR_o}}{\tau_0}}. \tag{5.25}$$

If we could show that $P_f^{(i)}(A_o) = 0$, then:

$$P_f^{(k)} \leq P_f^{(0)} e^{-\frac{k(\frac{T}{2}-T_{oh})}{\tau_0}}, \tag{5.26}$$

$$T_{oh} = \tau_S + \tau_{R_iR_o}. \tag{5.27}$$

However, we shall show that the $P_f^{(i)}(A_o) = 0$ requirement is overly restrictive. In the remaining part of this section, we shall devise a set of criteria that are sufficient to guarantee that, even if the synchronizer enters the metastable regime caused by an event on the A_o input of a FIFO element, the properties of the FIFO chain will guarantee that this particular metastability results in a benign behavior at the synchronous end.

To prove this part, we shall first define the *first-event metastability* (FEM), and the *second-event metastability* (SEM), corresponding to two particular propagation modes of the FIFO in Figure 5.12. Next, we shall show that an A_o event can only cause SEM, and that the SEM can itself only cause SEM. Finally, we shall show that the SEM is benign at the synchronous end.

Let us observe one synchronizer element S from Figure 5.12, with its clock input connected to clock phase φ. S will exhibit metastable behavior only if its input event is coincident with the down-going edge of φ.

Definition 4 *When the input of a synchronizer element S, clocked with φ, changes state coincident with an arbitrary, j^{th}, down-going edge of φ, and there were no prior input events between the $(j-1)^{st}$ and the j^{th} down-going edge of φ, we shall say that S has entered first-event metastability.*

Definition 5 *When the input of a synchronizer element S, clocked with φ, changes state coincident with an arbitrary, j^{th} down-going edge of φ, and there was at least one prior input event between the $(j-1)^{st}$ and the j^{th} down-going edge of φ, we shall say that S has entered second-event metastability.*[1]

Let us now pick up the thread of the proof again by going back to Equation 5.22 and analyzing what happens when the j^{th} event on $R_o^{(i)}$ is caused by the $(j-1)^{st}$ event on $A_o^{(i)}$:

$$t^{(j-1)}_{A_i^{(i+1)}} \equiv t^{(j-1)}_{A_o^{(i)}} = t^{(j)}_{R_o^{(i)}} - \tau_{A_oR_o}. \tag{5.28}$$

According to Equations 5.12 and 5.13:

$$\begin{array}{l} t^{(j)}_{R_o^{(i)}} \leq t^{(j-1)}_{A_i^{(i+1)}} + \max(\tau_{R_iR_o} - \tau_{R_iA_i}, \tau_{A_oR_o} - \tau_{A_oA_i}) \\ t^{(j)}_{R_o^{(i)}} \geq t^{(j-1)}_{A_i^{(i+1)}} - \max(\tau_{R_iA_i} - \tau_{R_iR_o}, \tau_{A_oA_i} - \tau_{A_oR_o}) \end{array}. \tag{5.29}$$

For a typical FIFO implementation, $\tau_{R_iR_o} \approx \tau_{R_iA_i}$ and $\tau_{A_oR_o} \approx \tau_{A_oA_i}$, so we shall assume that $\forall i, j : t^{(j)}_{R_o{}^{(i)}} \approx t^{(j)}_{A_i{}^{(i)}}$. This simplification does not qualitatively change the results we shall obtain; it only improves the readability of our arguments.

If the metastability at the $(i+1)^{st}$ synchronizer is a result of the $(j-1)^{st}$ transition on $A_o^{(i)}$, this can, in turn, cause metastability in the $(i+2)^{nd}$ synchronizer. However, we can assure that the latter is always SEM, if:

$$t^{(j)}_{R_o^{(i)}} - T/2 < t^{(j-1)}_{R_o^{(i+1)}} < t^{(j)}_{R_o^{(i)}} + T/2. \tag{5.30}$$

From Equations 5.28 and 5.30, we can derive the following implementation

[1]The distinction between the first- and second-event metastability has no physical basis; it is defined only for purposes of the proof.

requirement:

$$\tau_{A_oR_o} < T/2. \tag{5.31}$$

We shall focus next on showing that SEM can only cause SEM. Suppose that the j^{th} event on $R_o^{(i)}$ is SEM. Then, according to Definition 5:

$$t^{(j)}_{R_o^{(i)}} = t^{(k)}_{\varphi(i+1)\mathrm{mod}\,2\downarrow}, \tag{5.32}$$

$$t^{(k-1)}_{\varphi(i+1)\mathrm{mod}\,2\downarrow} < t^{(j-1)}_{R_o^{(i)}} < t^{(k)}_{\varphi(i+1)\mathrm{mod}\,2\downarrow}, \tag{5.33}$$

and therefore:

$$t^{(j)}_{R_o^{(i)}} - T_0 + \tau_S \leq t^{(j-1)}_{R_i^{(i+1)}} \leq t^{(j)}_{R_o^{(i)}} + \tau_S, \tag{5.34}$$

$$t^{(j-1)}_{R_o^{(i+1)}} \geq t^{(j)}_{R_o^{(i)}} - T_0 + \tau_S + \tau_{R_iR_o}, \tag{5.35}$$

which implies that

$$t^{(j-1)}_{R_o^{(i+1)}} > t^{(j)}_{R_o^{(i)}} - T/2. \tag{5.36}$$

Since

$$t^{(j-1)}_{R_o^{(i+1)}} \approx t^{(j-1)}_{A_i^{(i+1)}} \leq t^{(j)}_{R_o^{(i)}} - \tau A_o R_o, \tag{5.37}$$

and according to Equation 5.36:

$$t^{(j)}_{R_o^{(i)}} - T/2 < t^{(j-1)}_{R_o^{(i+1)}} < t^{(j)}_{R_o^{(i)}} + T/2. \tag{5.38}$$

Therefore, SEM at the i^{th} synchronizer can only cause SEM at the $(i+1)^{st}$ synchronizer.

Finally, with reference to Figure 5.15, it should be obvious to the reader that one can design a synchronous circuit whose correct operation will not be affected by the SEM. The required throughput is equal to one event per clock cycle, so the synchronous circuit must, after the first event within a clock cycle is observed (and this event is guaranteed by Definition 5 not to cause synchronization failure), acknowledge the first event, and disregard any input change until the following clock cycle.

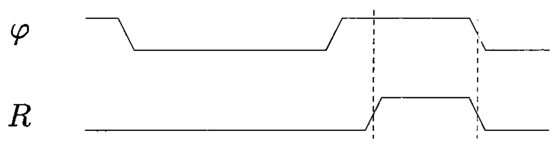


Figure 5.15: Second-event metastability

Sustaining the Throughput

The maximum data throughput of the pipeline synchronizer in Figure 5.12 is throttled by the synchronous side, and is equal to one data item per clock cycle. In the remainder of this section, we shall find implementation requirements that are sufficient to guarantee that the pipeline synchronizer can sustain that throughput.

Let us first consider an infinitely-long pipeline-synchronizer chain, and assume that it is in the steady state. Figure 5.16 illustrates this condition, with the arcs between events describing causal dependencies. For all even-numbered FIFO cells, the events

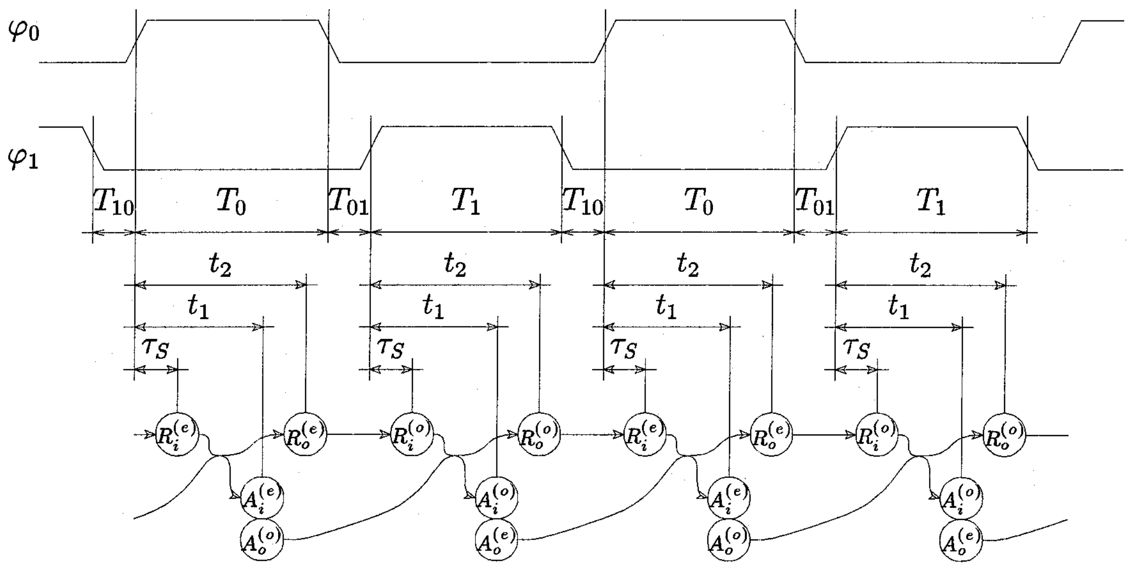


Figure 5.16: Steady-state operation of the pipeline synchronizer

on R_i occur simultaneously, τ_S after φ_0 becomes high. For all odd-numbered FIFO cells, the events on R_i occur simultaneously, τ_S after φ_1 becomes high. In this regime of operation, no synchronizer enters the metastable state, and:

$$\begin{aligned} t_1 &= \max(\tau_S + \tau_{R_iA_i}, t_1 + \tau_{A_oA_i} - \tfrac{T}{2}) \\ t_2 &= \max(\tau_S + \tau_{R_iR_o}, t_1 + \tau_{A_oR_o} - \tfrac{T}{2}) \end{aligned} . \tag{5.39}$$

For $\tau_{A_oA_i} < \frac{T}{2}$,

$$\begin{aligned} t_1 &= \tau_S + \tau_{R_iA_i} \\ t_2 &= \max(\tau_S + \tau_{R_iR_o}, \tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} - \tfrac{T}{2}) \end{aligned} . \tag{5.40}$$

For the steady state in Figure 5.16 to be possible, t_2 must be less than the half-period. The following conditions on the implementation of the synchronizer and of the FIFO element are, therefore, sufficient to guarantee that the infinite chain of pipeline synchronizers can sustain the maximum throughput while operating in this particular regime:

$$\begin{aligned} \tau_{A_oA_i} &< T/2 \\ \tau_S + \tau_{R_iR_o} &< T/2 \\ \tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} &< T \end{aligned} . \tag{5.41}$$

To examine the case of a finite-length pipeline synchronizer, we shall use an analogy with transmission lines. A piece of transmission line that is terminated with its characteristic impedance behaves as if it were of infinite length. The pipeline synchronizer, similarly, has to be terminated with the circuitry that satisfies the above conditions if the full throughput is to be maintained.

The requirement at the synchronous end is:

$$\tau_{RA} + \tau_{A_oR_o} < T, \tag{5.42}$$

where τ_{RA} is the delay from the time when one data transfer is requested until it is acknowledged. In absence of metastability at the synchronous end, this condition is trivially achieved.

On the asynchronous end, to terminate our pipeline synchronizer properly to maintain the bandwidth, the outside circuitry has to satisfy the condition:

$$\tau_S + \tau_{R_iA_i} + \tau_{AR} < T, \tag{5.43}$$

where τ_{AR} is the delay from the time when one data transfer is acknowledged until the time when the next data transfer is requested. Of course, the definition of asynchronous signals prevents us from imposing such requirements. What we shall prove in the following section is that, with a properly terminated sink, if a particular asynchronous data transfer at the source is initiated at any point in time, and if, from then on, Equation 5.43 is satisfied, the pipeline synchronizer will with bounded latency enter the steady state illustrated in Figure 5.16.

Latency

Let us observe the behavior of a pipeline synchronizer of length k (Figure 5.12), with the properly-terminated sink (Equation 5.42). We shall assume that the synchronizer is in the state equivalent to the state at $t = 0$, that is, all FIFO cells are empty. Starting in this initial state, we shall number each signal's transitions starting from zero.

1) If the first asynchronous request occurs while $\varphi_0 = 0$, between the $(n-1)^{st}$ down-going edge and the n^{th} up-going edge of φ_0, the first event at $R_i^{(0)}$ will occur after $\varphi_0 = 1$ again, at time

$$t^{(0)}_{R_i^{(0)}} = t^{(n)}_{\varphi_0\uparrow} + \tau_S. \tag{5.44}$$

Given that the conditions of Equation 5.41 are satisfied, the first input transitions of the FIFO stages occur at times:

$$t^{(0)}_{R_i^{(i)}} = t^{(n+\lfloor i/2 \rfloor)}_{\varphi_{i \bmod 2}\uparrow} + \tau_S, \quad 0 \le i < k. \tag{5.45}$$

Using double induction, it is simple to show that, if the throughput-sustaining conditions are satisfied, every transition will occur, at the latest, at the time corresponding to the steady state illustrated in Figure 5.16:

$$\begin{array}{ll} t^{(j)}_{R_i^{(i)}} \leq t^{(n+j+\lfloor i/2 \rfloor)}_{\varphi_{i \bmod 2}\uparrow} + \tau_S, & 0 \leq i < k \\ t^{(j)}_{R_o^{(i)}} \leq t^{(n+j+\lfloor i/2 \rfloor)}_{\varphi_{i \bmod 2}\uparrow} + \max(\tau_S + \tau_{R_iR_o}, \tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} - T/2), & 0 \leq i < k \end{array} . \qquad (5.46)$$

The initial step of the double induction is satisfied by Equation 5.45. The recursive step establishes that, with

$$\tau_{A_oR_o} < T/2, \qquad (5.47)$$

if the condition in Equation 5.46 is satisfied for $t^{(j)}_{R_i^{(i)}}$ and for $t^{(j-1)}_{R_o^{(i+1)}}$, then it is also satisfied for $t^{(j)}_{R_o^{(i)}}$, and for $t^{(j)}_{R_i^{(i+1)}}$.

Therefore, the latency of the pipeline synchronizer is, in this case, $T_l \leq \frac{(k+1)T}{2}$.

2) Let us assume that the first asynchronous request occurs while $\varphi_0 = 1$, between the nth up-going edge and the nth down-going edge of φ_0, at some time $t^{(n)}_{\varphi_0\uparrow} + \Delta^{(0)}_0$. For the ith pipeline-synchronizer element, the notation $\Delta^{(j)}_i$ will denote how far from the steady state is the jth transition of its R_i input. $\Delta^{(j)}_k$ will denote how far from the steady state is the jth transition of the synchronous-system input, $R_o^{(k-1)}$.

With the throughput-sustaining conditions of Equation 5.41 satisfied, the first transitions on the inputs of the FIFO elements and on the input of the synchronous system occur at times:

$$\begin{array}{lcl} t^{(0)}_{R_i^{(i)}} & = & t^{(n+\lfloor i/2 \rfloor)}_{\varphi_{i \bmod 2}\uparrow} + \tau_S + \Delta^{(0)}_i, \quad 0 \leq i < k \\ t^{(0)}_{R_o^{(k-1)}} & = & t^{(0)}_{R_i^{(k-1)}} + \tau_{R_iR_o} \end{array} , \qquad (5.48)$$

where

$$\begin{array}{lcl} \Delta^{(0)}_i & = & \max(0, \Delta^{(0)}_{i-1} + \tau_S + \tau_{R_iR_o} - T/2), \quad 1 \leq i \leq k \\ \Delta^{(0)}_k & = & \Delta^{(0)}_{k-1} \end{array} . \qquad (5.49)$$

Since $\tau_S + \tau_{R_iR_o} < \frac{T}{2}$ (Equation 5.23), the first output transition of each subsequent stage is closer to the steady-state condition ($\forall i, j : \Delta^{(j)}_i = 0$).

Using double induction, one can show that the subsequent transitions occur at times:

$$\begin{array}{lcll} t^{(j)}_{R_i^{(i)}} & \leq & t^{(n+j+\lfloor i/2 \rfloor)}_{\varphi_{i \bmod 2}\uparrow} + \tau_S + \Delta_i^{(j)}, & 0 \leq i < k \\ t^{(j)}_{R_o^{(k-1)}} & \leq & t^{(n+j+\lfloor (k-1)/2 \rfloor)}_{\varphi_{k \bmod 2}\uparrow} + \tau_S + \tau_{R_i R_o} + \Delta_k^{(j)}, & 0 \leq i < k \end{array}, \tag{5.50}$$

where:

$$\Delta_i^{(j)} = \begin{cases} \max(\Delta_1^{(j-1)} + \tau_{AR} - T/2) & i = 0 \\ \max(\Delta_{i-1}^{(j)} + \tau_S + \tau_{R_i R_o} - T/2, \Delta_{i+1}^{(j-1)} + \tau_{A_o R_o} - T/2), & 1 \leq i < k \\ \max(\Delta_{k-1}^{(j)}, \Delta_k^{(j-1)} + \tau_{RA} + \tau_{A_o R_o} - T), & i = k \end{cases}. \tag{5.51}$$

Therefore, if the following conditions are satisfied:

$$\begin{array}{lcl} \tau_{AR} & < & T/2 \\ \tau_S + \tau_{R_i R_o} & < & T/2 \\ \tau_{A_o R_o} & < & T/2 \\ \tau_{RA} + \tau_{A_o R_o} & < & T \end{array}, \tag{5.52}$$

successive transitions on $R_o^{(k-1)}$ (the input of the synchronous circuit) always occur less than $\Delta_0^{(0)} < T_0$ after the successive clock edges. Latency in this case is $T_l \leq \frac{kT}{2}$.

3) In this subsection, we shall find the bound on latency in case of metastability caused by the input asynchronous event. Since the synchronous side throttles the throughput, and since we cannot assume correct operation of the synchronous system under synchronization failure, this part of the proof will consider only the case when the metastability does not reach the synchronous end.

If the first asynchronous request occurs coincident with the n^{th} down-going edge of φ_0, the first synchronizer will enter metastable state, and, depending on when it exits this state, we shall distinguish between the three cases described next.

3a) If the metastable state lasts less for t_m, and $t_m < T/2 - \tau_S - \tau_{R_i R_o}$, then

$$\begin{array}{lcl} \Delta_0^{(0)} & = & T_0 \\ \Delta_1^{(0)} & = & t_m + \tau_S + \tau_{R_i R_o} - T_{01} \end{array}, \tag{5.53}$$

and all the subsequent transitions will follow Equation 5.50, as analyzed in **2**).

3b) If the metastable state lasts for t_m, and $t_m > T/2 - \tau_S - \tau_{R_iR_o}$, then:

$$t^{(n)}_{\varphi_1\downarrow} < t^{(0)}_{R_o^{(0)}} < t^{(n+1)}_{\varphi_0\uparrow} + \tau_S + \tau_{R_iR_o}, \tag{5.54}$$

$$t^{(0)}_{R_o^{(i)}} = t^{(n+1+\lfloor i/2 \rfloor)}_{\varphi_1\uparrow} + \tau_S + \tau_{R_iR_o}, \quad 1 \le i < k \ , \tag{5.55}$$

and the modified version of Equation 5.46, with n replaced by $n+1$, will hold for all pipeline-synchronizer signals. The latency will be $T_l \le \frac{(k+2)T}{2}$.

3c) Finally, if the metastable state lasts for exactly $T/2 - \tau_S - \tau_{R_iR_o}$, it will cause metastability in the subsequent synchronizer. Let us assume that the first m synchronizers enter metastable state as a result of their first input transition, and that remaining $(k-m)$ synchronizers do not. Then the first output transitions of the first m synchronizers occur at times:

$$t^{(0)}_{R_o^{(i)}} = t^{(n+\lfloor i/2 \rfloor)}_{\varphi_{i \bmod 2}\downarrow}, \quad 0 \le i < m-1 \ , \tag{5.56}$$

and, after the first transition, they follow Equation 5.50. The remaining $(k-m)$ synchronizers behave as analyzed in **3a**) or **3b**). The latency is $T_l \le \frac{(k+2)T}{2}$.

Conclusion

In this section we have found a set of requirements that are sufficient to guarantee that a particular implementation of a synchronizer and of a FIFO element can be used for pipeline synchronization. These requirements are listed in Equations 5.23, 5.31, 5.41, 5.42, 5.43, and 5.52. The union of all these requirements is:

$$\begin{array}{lll}
\tau_S + \tau_{R_iR_o} & < & T/2 \\
\tau_{A_oR_o} & < & T/2 \\
\tau_{AR} & < & T/2 \\
\tau_{A_oA_i} & < & T/2\ . \\
\tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} & < & T \\
\tau_S + \tau_{R_iA_i} + \tau_{AR} & < & T \\
\tau_{RA} + \tau_{A_oR_o} & < & T
\end{array} \tag{5.57}$$

5.4.5 Variations On the Theme

In Section 5.4.3 we showed how pipeline synchronization can be used to interface input data streams that follow the two-phase asynchronous protocol to synchronous systems, along with the proof of its correct operation. In this section, we shall show that the same technique is applicable for interfacing to output streams that follow the two-phase asynchronous protocol, as well as for synchronization of input and output streams that operate under the four-phase asynchronous protocol. The same proof techniques can be applied, so only the implementations will be presented.

Reversing the Direction of Data Flow

A block diagram of a pipeline synchronizer that can be used to interface an asynchronous output data stream to a synchronous system is shown in Figure 5.17.

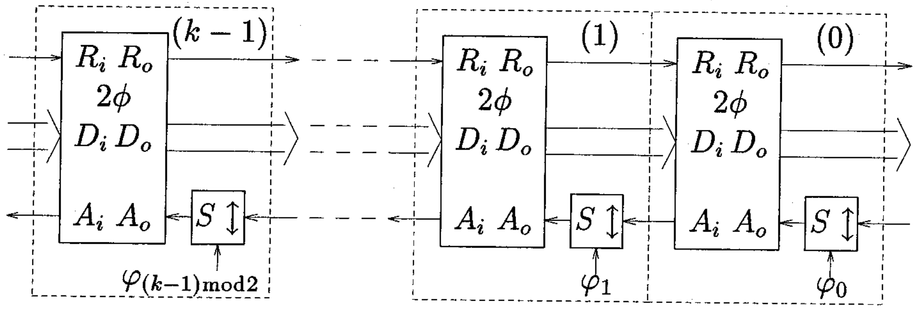


Figure 5.17: Synchronous-input, asynchronous-output pipeline synchronizer

The implementation of synchronizer and of FIFO elements must satisfy the following requirements:

$$\begin{array}{lcl} \tau_S + \tau_{A_oA_i} & < & T/2 \\ \tau_{R_iA_i} & < & T/2 \\ \tau_{RA} & < & T/2 \\ \tau_{R_iR_o} & < & T/2\,. \\ \tau_S + \tau_{A_oR_o} + \tau_{R_iA_i} & < & T \\ \tau_S + \tau_{A_oR_o} + \tau_{RA} & < & T \\ \tau_{AR} + \tau_{R_iA_i} & < & T \end{array} \tag{5.58}$$

Using Four-Phase-Protocol FIFO Elements

When using the four-phase signaling protocol, the spectrum of valid implementations of FIFO elements is much larger. For an exhaustive study, see [14]. One possible implementation is:

$$* \, [[R_i]; A_i \uparrow; [\overline{R_i}]; A_i \downarrow, R_o \uparrow; [A_o]; R_o \downarrow; [\overline{A_o}];], \tag{5.59}$$

with the dependency graph as shown in Figure 5.18.

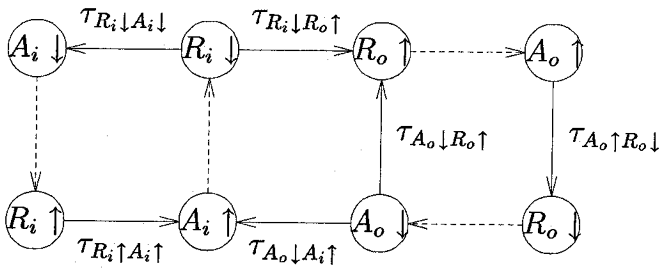

$\tau_{R_i\downarrow A_i\downarrow}$
$\tau_{R_i\downarrow R_o\uparrow}$
$A_i \downarrow$
$R_i \downarrow$
$R_o \uparrow$
$A_o \uparrow$
$\tau_{A_o\downarrow R_o\uparrow}$
$\tau_{A_o\uparrow R_o\downarrow}$
$R_i \uparrow$
$A_i \uparrow$
$A_o \downarrow$
$R_o \downarrow$
$\tau_{R_i\uparrow A_i\uparrow}$
$\tau_{A_o\downarrow A_i\uparrow}$

Figure 5.18: The dependency graph of one form of four-phase-protocol FIFO element

When using the four-phase protocol, there are two transitions for every data item transferred, and we shall pick only one of the edges to represent an event (either an up- or down-going edge). Let us assume that we have made the choice such that the $R_i \uparrow$, $A_i \downarrow$, $R_o \uparrow$, and $A_o \downarrow$ represent events, *i.e.*, their arrival time is uncertain.[2] Then, the pipeline synchronization chain that we have used for synchronization of the

[2]In case of this particular, arbitrary choice, when the FIFO elements with the behavior as specified in Equation 5.59 form a FIFO chain, there is no uncertainty about the arrival time of

two-phase protocol in Figures 5.12 and 5.17 can be used with the four-phase protocol when modified in the following way: FIFO elements are replaced with the four-phase-protocol version, and symmetric synchronizers are replaced with an asymmetric version that synchronizes only the edges that have been chosen to represent signal events.

The requirements in Equations 5.57 and 5.58 still apply, with:

$$\begin{aligned} \tau_{R_iA_i} &= \tau_{R_i\uparrow A_i\uparrow} + \tau_{A_o\uparrow R_o\downarrow} + \tau_{R_i\downarrow A_i\downarrow} \\ \tau_{R_iR_o} &= \tau_{R_i\uparrow A_i\uparrow} + \tau_{A_o\uparrow R_o\downarrow} + \tau_{R_i\downarrow R_o\uparrow} \\ \tau_{A_oA_i} &= \tau_{A_o\downarrow A_i\uparrow} + \tau_{A_o\uparrow R_o\downarrow} + \tau_{R_i\downarrow A_i\downarrow} \\ \tau_{A_oR_o} &= \tau_{A_o\downarrow R_o\uparrow} \end{aligned} . \qquad (5.61)$$

5.5 A CMOS Implementation

All four versions of pipeline synchronizers described in this chapter (with two- and four-phase protocol, with synchronous and asynchronous input stream) have been implemented and used in the Mosaic over the past three years. In a set of experiments reported by Cohen *et al.* [23], a local-area network was implemented using Mosaic components, and used to transmit and receive more than 10^{15} bits without a single error.

In Section 5.4.1, we showed the ME element that we use to build asymmetric synchronizers, which synchronize only the up-going or only the down-going transitions. To build a symmetric synchronizer, we utilize two ME elements in the circuit illustrated in Figure 5.19. The circuit around the two ME elements implements the

the complementary edges:

$$\forall j \geq 0 : \begin{cases} t^{(j)}_{R_i\downarrow} = t^{(j)}_{A_i\downarrow} - \tau_{R_i\downarrow A_i\downarrow} \\ t^{(j)}_{A_i\uparrow} = t^{(j)}_{A_i\downarrow} - \tau_{R_i\downarrow A_i\downarrow} - \tau_{A_o\uparrow R_o\downarrow} \\ t^{(j)}_{R_o\downarrow} = t^{(j)}_{A_o\downarrow} - \tau_{R_i\downarrow A_i\downarrow} \\ t^{(j)}_{A_o\uparrow} = t^{(j)}_{A_o\downarrow} - \tau_{R_i\downarrow A_i\downarrow} - \tau_{A_o\uparrow R_o\downarrow} \end{cases} . \qquad (5.60)$$

following specification:

$$* [[R_i]; R_1 \uparrow; [A_1]; R_o \uparrow, R_1 \downarrow; [\overline{A_1}]; [\overline{R_i}]; R_2 \uparrow; [A_2]; R_o \downarrow, R_2 \downarrow; [\overline{A_2}];]. \quad (5.62)$$

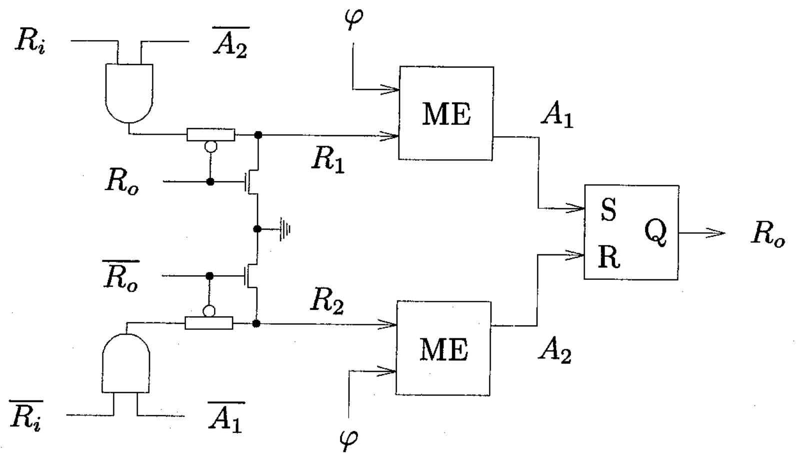


Figure 5.19: A symmetric synchronizer

The symmetric synchronizer, and the four-phase-protocol FIFO elements were designed using a pre-release version of the asynchronous-design tools described in [58].

5.6 Conclusions

Even though the motivation for pipeline synchronization came from a specific problem — interfacing Mosaic's asynchronous router and synchronous memory —, the applications of this technique are much broader. Pipeline synchronization is a simple, low-cost, high-bandwidth, high-reliability solution to interfaces between synchronous and asynchronous systems, or between synchronous systems operating from different clocks, where the data rate is too high for the single-synchronizer approach.

The power and simplicity of this technique allow the designer to break away from the traditional divisions in the design of asynchronous and synchronous circuits: clocked systems deal with synchronous events; self-timed systems deal with asynchronous events; and interfacing between the two is done with synchronizers of limited reliability. With pipeline synchronization, the designer can achieve arbitrarily low failure rates in exchange for latency rather than a reduction in bandwidth.

In developing pipeline synchronization, instead of reasoning about signal events, we reasoned about probability distributions of signal-event arrival times, and introduced a metric to characterize "how asynchronous a signal is." Along the way, we used some simple yet unconventional techniques: we can "partially synchronize" a signal event, use it to perform some work using self-timed circuits, and then repeat the process until the desired reliability is achieved.

The probability model that we used was crucial to our understanding of synchronization techniques, and to our ability to devise a novel approach to data-stream synchronization. However, we were not successful in using this model to prove all the circuit properties that interested us; hence, we had to resort to rather elaborate proof techniques. This is a topic that we believe deserves additional investigation.

Chapter 6

Conclusions

6.1 Comparison With Related Work

A variety of concurrent machines have been built in effort to apply concurrent approaches to "general-purpose" computing at the application level. In this section, we shall compare the Mosaic architecture and the C+- programming system with other contemporary architectures and programming systems. At the architecture level, we shall focus on communication bandwidth and latency, and on complexity of implementation. At the programming level, we shall discuss the relative importance of efficiency, expressivity, and safety compared to C+-.

6.1.1 Medium-Grain Multicomputers

The antecedent of all modern-day multicomputers is the Cosmic Cube [76]. A number of commercial developments followed this effort, with similar multicomputers manufactured by Intel, nCUBE and Ametek. With the exception of nCUBE's custom designs of integrated processor and network interface, these machines employed off-the-shelf processor, memory, and compiler technology.

The raw hardware performance of these machines is quite impressive in terms of peak instruction rates, and, in some cases, in terms of available communication bandwidth. The nodes are of complexity comparable to that of workstations, with hardware floating-point capabilities, top-of-the-line single-chip processors, and megabytes of node memory.

Standard programming systems for these machines are based on sequential programming notations for specifying individual process behavior, with library routines for inter-process communication and synchronization. The expressive power of these programming systems is comparable to that of C+-, but because programs that specify individual process behavior are, unlike in C+-, lexically separate, compilers cannot perform many safety-improving checks typical of object-oriented programming that C+- performs including, for example, type matching of messages between the sender and the receiver. Since the message scope in these systems is not well-defined, it is difficult, if not impossible, for the compiler to assist the programmer in message passing of runtime-specified data structures and/or message exchange in heterogeneous environments. An important difference in programming emphasis is due to C+-'s blurring the distinction between processes as computing agents and processes as data, deliberately implying that process creation is inexpensive.

Standard programming systems for commercial multicomputers regularly fall short of utilizing the hardware capabilities to their fullest, mainly because of the large software overhead of communications, typically on the order of a thousand of processor instructions. Recent work on Active Messages [31] clearly demonstrates how the incompatibility of the hardware mechanisms with the programming model results in large software overheads. The Mosaic design team's approach to reducing the software overhead anticipated that advocated by research on Active Messages. A relatively small amount of hardware support is devoted to message handling, allowing for software optimizations of special cases. The message-handling layer of the MADRE runtime system [10] is triggered by message reception, and message handlers run to completion. Two contexts, one for message handling, the other for regular computation, are used to eliminate the context-switch overhead. Where Active Messages diverges from our approach is that it expects the user to handle the message-buffer management, something we consider too large a burden except for applications with highly regular communication patterns. For the Mosaic, a machine with scarce node-memory resources and so much potential concurrency to necessitate runtime-system-managed process placement, message-buffer management is particularly demanding.

6.1.2 Fine-Grain Multicomputers

In this section, we shall compare the Mosaic architecture to two representatives of fine-grain multicomputers: the Transputer, a well-established commercial family of chips manufactured by INMOS [45], and the J-Machine, developed by the research team led by William J. Dally, at MIT [64, 63]. Just as with the Mosaic, both the Transputer and the J-Machine have been developed in conjunction with, and influenced by, their respective programming systems.

Transputer and Occam

Occam [61] and its subsequent variants [13] are based on work of C. A. R. Hoare on Communicating Sequential Processes (CSP) [42]. Occam is an explicit-message-passing notation, with processes interacting through synchronous communication channels. In Occam programs, all processes and all channels must be known at compile time, and Occam compilers use this information to completely eliminate the need for run-time resource management. This requirement, together with typed channels, makes Occam a very safe notation, to the point that there are rules for semantics-preserving program transformations. However, Occam does not support dynamic process creation, and it supports only a limited form of iteration and a non-recursive procedure call.

The Transputer approach is to offer pre-packaged computing solutions in hardware, including the Occam process model and scheduling. The presence of floating-point hardware on chip, unlike in Mosaic or J-Machine, is the consequence of INMOS being driven by its markets, whereas the other two projects are testbeds for concurrent-programming experiments. The Transputer communication mechanism is limited, with nearest-neighbor communication. It requires either software-assisted, store-and-forward routing, or additional, external communication components to provide a general communication capability. The newest-generation Transputer improves the routing generality and performance through use of virtual channels.

When used as intended, for running Occam programs, the Transputer provides an excellent computing platform. However, its limited communication capabilities and the hard-wired process notion typically present difficulties for implementation of non-Occam-like concurrent-programming systems.

J-Machine and CST

Concurrent Smalltalk (CST) [26, 44, 43] is a concurrent, object-oriented, programming notation, and, in spite of vastly different syntax, is in many important aspects similar to C+-. The main semantic difference between C+- and CST is that, whereas the state of a C+- process can be accessed only through a set of mutually-exclusive, atomic actions, methods of a CST object can access the object concurrently. The atomicity of CST methods must be managed explicitly, using locks. The primary mechanism for generating concurrency in CST is issuing concurrent remote procedure calls and synchronizing through futures. Although this mechanism is supported in C+-, the primary mechanisms for generating concurrency in C+- are process creation and non-blocking message sends. Some recently-reported results [64] on the performance evaluation of the J-Machine have been obtained with programs written in the J language, an extension of C with a small number of constructs for communication and synchronization, not unlike the C+- approach.

Unlike the Transputer, the Message-Driven Processor of the J-Machine does not impose its preferred process model on the user. The emphasis of J-Machine is on implementing in hardware the instruction sequences that CST programs use often: a simple hashing policy with the two-way set-associative cache for name translation, detecting access to uninitialized variables for synchronization, scheduling off of the message-receive queue, injecting data into the routing network, and computing the relative distance of the message destination.

Because there are fewer pre-conceived notions built into the hardware than with the Transputer, the J-Machine more readily accommodates programming systems for which it had not originally been designed [60, 88].

The emphasis of Mosaic is on providing high-performance routing and memory, with a seamless interface between the two. One can argue that a C+- implementation on the Mosaic would benefit from most, if not all, of the hardware-supported communication and synchronization mechanisms of the J-Machine. However, the reduction in the number of cycles and in the size of the code has to be compared against the additional price in chip area and the loss of flexibility. Given the large discrepancy in access time between on-chip and off-chip memory [64], the advantages of nominally time-saving, but area-expensive, mechanisms is less than obvious. The importance of preserving the flexibility to as late a design stage as possible cannot be overstated. For example, the `xlate` instruction on the J-Machine implements a simple hashing policy and a 512-entry two-way set-associative cache, and executes in three cycles. An equivalent of this instruction takes 11 Mosaic instructions, but it can be modified trivially to work with different hash functions, and different cache structure and/or size.

The J-Machine and the Mosaic projects share many of the same underlying principles and motivations, many of which trace back to the time when William J. Dally was a Ph.D. student in our research group [26], and the rest is due to our relatively frequent mutual progress updates. The necessity to limit the scopes of the respective projects made the Mosaic team focus on efforts that were more aggressive technologically (single-chip nodes with internal dRAM, pipeline synchronization, advanced packaging), whereas the focus of the J-Machine team's efforts was primarily on mechanisms (synchronization, name translation, scheduling).

6.1.3 Multiprocessors

As discussed in Sections 1.2 and 1.3, the shared memory programming model is a predominant concurrent-programming paradigm, mostly because many programmers find it to be a natural extension of the sequential programming model. Traditional shared-memory programming systems [85] extend sequential notations with process creation and synchronization primitives. Data communication is done through shared-memory access. Some recently-developed shared-memory programming systems, such as COOL [18], move towards concurrent, object-oriented programming, similar to C+-, although for a different set of reasons. While the goal of C+- is to increase expressivity of explicit message-passing notations by providing a global name space, COOL offers monitor-like data-encapsulation as a safe alternative to the all-powerful access of a shared-memory word.

C+- programs often look remarkably similar to programs written in notations designed with multiprocessors as their primary targets. Let us compare the run-time behavior of concurrent object-oriented programs on a cache-coherent multiprocessor and on the Mosaic.

Assume that a concurrent computation consists of a partially-ordered set of actions; each action modifies the state of an object (`o`), by executing a function (`f`), with an argument list (`a`). One can write:

```
o.f(a)
```

Assume also that the state of the object, the function code, and the argument list are each stored in their respective pieces of memory somewhere in the concurrent machine.

A typical multiprocessor is *processor* centered. A processor is assigned to perform this action, possibly by directly accessing the task queue of the assigned processor, or by appending the global task queue responsible for scheduling the entire machine. Once the action is scheduled, using that processor's cache-update mechanism the (pieces of) object state, function code, and argument list are requested and brought to the processor. After the required computations are performed, the object's state is eventually updated in the memory through the cache-coherence protocol. Most communications are *demand* driven, initiated by the cache-update requests, with the exception of appending the task queue(s).

The Mosaic is *memory* centered. There is only one processor that can access the object's state — the one controlling the memory where the object's state is stored.

The argument list will be put into the same local memory by a message. The function code can, in principle, be sent along with the argument list, but because the code is read-only, it can be effectively software-cached in the same memory as the object it operates on. The communication is, therefore, mostly *supply* driven, with the exception of updating the software code cache.

Due to the inherent complexity of coherent data caching [15, 95], a typical multiprocessor employs as much as two orders of magnitude fewer nodes than a Mosaic multicomputer of the same price [79]. On the other hand, the higher up-front node cost enables one to incorporate a more powerful processor. The emphasis of programming effort is, therefore, radically different. Whereas the effort in programming a multiprocessor will typically be in keeping the expensive nodes busy with better load-balancing strategies [18], the effort in programming the Mosaic is in generating as much concurrency as possible.

For problems with limited concurrency, a multiprocessor clearly holds the performance edge; for problems with abundant concurrency, the Mosaic is the machine of choice. For applications which belong to neither of these two extremes, the performance will depend on how effective the coherent data caching is. If the shared data is mostly of the single-writer, multiple-readers variety — including when who the writer is changes relatively infrequently —, coherent data caching is effective [47]. For highly contested, multiple-writers, multiple-readers, shared data, numerous updating and invalidating requests render caching useless.

6.2 Summary

A computer-architecture experiment is a complex task, indeed. To make such an experiment a successful one is to achieve perfection in a balancing act. Many an experiment has failed to fulfill its promise due to less-than-perfect solutions for such "mundane" details as: a few-bits worth of addressing space, I/O capabilities, mechanical assemblies, *etc.*

One of the great pitfalls, so forcefully exposed by the developers of the RISC processor architecture [37], is to restrict oneself into a well-defined box, no matter how nice the box looked from the inside: It was after processor architects had poked into the compiler-designer's turf that the possibility for dramatic improvement in processor performance was revealed.

Accordingly, the single most important design requirement, the requirement that turned into the cornerstone of our testbed, is that we must not assume that we understand all the possible ways in which the system will be used. As best as we could, we tried to provide mechanisms, not policies [84]. This effort should be obvious from the simplicity of our hardware communication primitives, and should be even more obvious to readers who had the stamina to follow through the examples describing the runtime-system interfaces. We made sure that the default operations of our software and hardware systems were as simple and intuitive as possible, and strived not to hide anything from an inquisitive user. The resulting, non-assuming architecture of the Mosaic is suitable for implementation of number of concurrent-programming systems [10, 55, 52], as well as for special-purpose, embedded, applications [23].

We have started with the one-chip-computer stipulation, and tried to push the application-span envelope as far as we could. The performance potential of fine-grain concurrent computers has long been recognized [75]. What is lacking is a powerful programming system to exploit that performance potential, not for a handful of carefully-crafted applications, but for a much larger application span. C+- approaches this problem from the bottom up. Programming paradigms that can be implemented efficiently on a simple and inexpensive hardware — atomic updates that can be enabled or disabled — are mapped into a simple extension of a popular, object-oriented notation.

What if we had an arbitrary amount of processing power dispersed around the memory? What if copying of a memory segment from the local memory into the remote memory was actually faster than the local copy operation? What if we sent the code and the argument list to where the data is, instead of having all three of those join at the all-important processor? Or perhaps we could send code and data to where the argument list is? How much caching can we efficiently do in software?

What if concurrency was so abundant that we did not have to worry about utilization of individual processors at all, but rather how to extract more concurrency from an application?

We do not know answers to any of these questions today. What we have provided is a testbed that could help in answering some of these questions in a *quantitative* way, and some results have already been established ([10]).

Appendix A

Example Products of C+− Compilation

In the interest of making it possible for other researchers to understand the structure of the software overhead of communications reported in Section 4.3, we shall present a few simple examples of the Mosaic assembly code produced by translation from C+− to C++ followed by compilation using the Gnu C++ targeted for the Mosaic processor.

The Mosaic register set is described in Section 4.2.3. The assembly instructions of interest are:

- `op` *src, dst* — where `op` is one of the standard arithmetic instructions (add, subtract ...), *src* and *dst* are general-purpose registers, and the semantics are *dst* := *dst* `op` *src.*

- `mov` *src, dst* — with the semantics of *dst* := *src,* where *src* and *dst* can be general-purpose registers, special registers, or memory operands with addresses determined by:

  - `(r)` — a register,
  - `(r++)` — a register, post-incremented,
  - `(--r)` — a pre-decremented register,
  - `(r+const)` — a register plus a constant, or

- `(const)` — a constant.

- `jmp` *dst* — jump to destination *dst*, determined by one of the addressing modes described above.

- `call` *dst* — actually two instructions, one to save the return address on the stack, and another to jump as above.

- `j`*cc label* — where *cc* is a condition under which to jump to the address specified by the *label*.

Program 40 defines the basic data structures used by the runtime system.

```
typedef unsigned        node_t;         // multicomputer node
                                        //   identifier

struct  pointer_t                       // a process pointer
{                                       // consists of:
        node_t  node;                   //   a node number
        void*   ptr;                    //   and an address
};

typedef unsigned        entry_t;        // a unique identifier
                                        // of an atomic action

struct  RTS_msg                         // a message
{
        RTS_msg*        next;           // queuing information
        int             size;           // >>
        pointer_t       ptr;            // >> message header
        entry_t         e;              // >>
};
```

Program 40: Runtime-System Data Structures

The code in Program 41 allows one to access the machine registers from the source code. Note that the pointers used in managing the send and the receive queue are kept in the general registers that are shared between the two contexts.

```
register  void*     MSP      asm ("msp");   // message receive pointer
register  void*     MSL      asm ("msl");   // message receive limit
register  void*     MRP      asm ("mrp");   // message send pointer
register  void*     MRL      asm ("mrl");   // message send limit
register  int       DXDY     asm ("dxdy");  // destination register
register  int       IMR      asm ("imr");   // interrupt mask register
register  int       ISR      asm ("isr");   // interrupt status register
register  RTS_msg*  SQ_HEAD  asm ("r15");   // send queue head
register  RTS_msg*  SQ_TAIL  asm ("r14");   // send queue tail
register  void*     SQ_END   asm ("r13");   // end of the send buffer
register  RTS_msg*  RQ_HEAD  asm ("r12");   // receive queue head
register  RTS_msg*  RQ_TAIL  asm ("r11");   // receive queue tail
register  void*     RQ_END   asm ("r10");   // end of the receive buffer
```

Program 41: Accessing Machine Registers from the Source Code

As shown in Program 42, the routing word Δx, Δy computation uses two tables. If the storage cost is prohibitive, it can be done with no extra storage in approximately twice the time.

```
int     dxtable[256];
int     dytable[256];

inline
int
dxdy_conversion (node_t dest)
{
    unsigned  dx = dest >> 8;
    unsigned  dy = dest & 0x00FF;
    return  dxtable[dx] | dytable[dy];
}
```

Program 42: The Δx, Δy Computation

As described in Section 3.1.7, a message-sending operation consists of calling the `operator head`, building the arguments list, and calling the `operator tail`. Program 43 presents an example definition of these two operators.

```
inline
void*
send_queue_alloc (int size)
{
    if ( (void*)SQ_TAIL + size >= SQ_END )
        return  send_queue_wraparound(size);
    else
        return  (&SQ_TAIL->size);
}

inline
void *
operator head(pointer_t p, entry_t e, int size)
{
    void*  v = send_queue_alloc(size);
    v = operator send(v,size);                // build message header
    v = operator send(v,p);
    v = operator send(v,e);
    return  v;
}

inline
void*
operator tail(void* begin, void* end, pointer_t p, entry_t e, int size)
{
    if ( SQ_HEAD == SQ_TAIL )
    {
        int  dxdy = dxdy_conversion(p.node);
        IMR = 0;                              // disable interrupts
        MSP = &SQ_TAIL->size;                 // send the message
        MSL = end-1;                          //
        DXDY = dxdy;                          //
        SQ_TAIL = SQ_TAIL->next = end;        // update the send queue
        SQ_TAIL->next = 0;                    //
        IMR = 7;                              // enable interrupts
    }
    else
        send_queue_append(begin,end,p,e,size);
}
```

Program 43: Building the Message

By now we have all the code we need to compile a message-sending example. Let Program 44 be the user code.

```
class   C                           // a user-defined class
{
public:
        void    foo();
};

void                                // and its member function
C::foo ()
{
}

void
f (C* c)
{
    while (1)                       // an endless sequence of
        c->foo();                   // member-function invocations
}


processdef     P                    // a user-defined process
{
public:
atomic  void    foo();
};

atomic
void                                // and its atomic action
P::foo ()
{
}

void
f (P* p)
{
    while (1)                       // an endless sequence of
        p->foo();                   // message send operations
}
```

Program 44: A Message-Send Example

Programs 45 and 46 are the result of translation of the user code to C++ followed by the compilation to Mosaic assembly code.

```
.globl  _f__FP1C                      | f(char*)
_f__FP1C:
                                      | FUNC PROLOGUE BGN
        mov bp,(--sp)                 | save frame ptr
        mov sp,bp                     | new frame ptr
                                      | Save regs used
        mov r3,(--sp)
                                      | FUNC PROLOGUE END
        mov (bp+2),r3
L68:
        mov r3,(--sp)
        call _foo__1C
        inc sp,sp
        jmp L68
                                      | FUNC EPILOGUE BGN
                                      | Restore regs used
        mov (sp++),r3
        mov bp,sp                     | restore stack ptr
        mov (sp++),bp                 | restore frame ptr
        rtn
                                      | FUNC EPILOGUE END


.globl  _f__FG9P_cpm_ptr              | f(P*)
_f__FG9P_cpm_ptr:
                                      | FUNC PROLOGUE BGN
        mov bp,(--sp)                 | save frame ptr
        mov sp,bp                     | new frame ptr
                                      | Save regs used
        mov r3,(--sp)
        mov r4,(--sp)
        mov r5,(--sp)
        mov r6,(--sp)
                                      | FUNC PROLOGUE END
        mov #4,r5
L72:
        mov (bp+2),r3
        mov (bp+3),r4
        mov r14,r0
        add #4,r0
        cmp r13,r0
        jltu L76
        mov r5,(--sp)
```

Program 45: A Message-Send Example, Compiled Code, Part 1

```
        call _send_queue_wraparound__Fi
        inc sp,sp
        jmp L75
L76:
        inc r14,r0
L75:
        mov #4,(r0++)
        mov r3,(r0++)
        mov r4,(r0++)
        mov #1,(r0++)
        mov r0,r4
        mov (bp+2),r3
        cmp r14,r15
        jne L81
        rnr r3,r0               | lsr by #8
        rnr r0,r0
        and #255,r0
        and #255,r3
        mov (r0+_dxtable),r0
        mov (r3+_dytable),r3
        mov #0,imr
        inc r14,r6
        mov r6,msp
        dec r4,r6
        mov r6,msl
        or r3,r0
        mov r0,dxdy
        mov r4,(r14)
        mov r4,r14
        mov #0,(r14)
        mov #7,imr
        jmp L72
L81:
        mov r4,(--sp)
        call _send_queue_append__FPv
        inc sp,sp
        jmp L72
                                | FUNC EPILOGUE BGN
                                | Restore regs used
        mov (sp++),r6
        mov (sp++),r5
        mov (sp++),r4
        mov (sp++),r3
        mov bp,sp               | restore stack ptr
        mov (sp++),bp           | restore frame ptr
        rtn
                                | FUNC EPILOGUE END
```

Program 46: A Message-Send Example, Compiled Code, Part 2

Program 47 is a program that can be used for user-level dispatch, and Programs 48 and 49 its compiled version.

```
struct  RTS_code                                // a code piece
{
        int     offset;                         // info on how to find the
        int     mask;                           //   active/passive bit
                                                //   within process state
        int     code[0];                        // the code itself
};

typedef void    (*FP)(void*,void*);             // a generic atomic action pointer

inline
atomic
void
@ (int size, pointer_t p, entry_t e)            // generic dispatcher
{
    RTS_code*  c = lookup_code(e);              // find the code piece

    void*  p = p.ptr;                           // the process pointer
    FP     f = (FP)(c->code);                   // the code pointer
    void*  a = args+1;                          // the arguments pointer

    int  offset = c->offset;
    int    mask = c->mask;
    if ( *(((int*)p)+offset) & mask )           // check the active/passive bit
        (*f)(p,a);                              // dispatch to user code
    else
        message_refused(args);
}

void
user_dispatch ()                                // this code runs continuously
{                                               // in the user context
    while (1)
    {
        while ( RQ_HEAD->next == 0 );           // polling the receive queue
        @(&RQ_HEAD->size);                      // call the generic dispatcher
        RQ_HEAD = RQ_HEAD->next;                // updating the receive queue
    }
}
```

Program 47: User-Level Dispatch

```
.globl  _user_dispatch__Fv
_user_dispatch__Fv:
                                |**  FUNC PROLOGUE BGN
        mov bp,(--sp)           | save frame ptr
        mov sp,bp               | new frame ptr
                                | Save regs used
        mov r3,(--sp)
        mov r4,(--sp)
        mov r5,(--sp)
        mov r6,(--sp)
        mov r7,(--sp)
                                |** FUNC PROLOGUE END
```

Program 48: User-Level Dispatch, Compiled Code, Part 1

```
L33:
L35:
        mov (r12),r0
        cmp #0,r0
        jeq L35
        mov (r12+4),(--sp)
        call _lookup_code__FUi
        mov (r12+3),r4
        mov r0,r6
        add #2,r6
        mov r12,r5
        add #5,r5
        mov (r0),r3
        mov (r0+1),r0
        mov r4,r7
        add r3,r7
        mov r7,r3
        mov (r3),r3
        and r3,r0
        inc sp,sp
        cmp #0,r0
        jeq L38
        mov r5,(--sp)
        mov r4,(--sp)
        call r6
        add #2,sp
        jmp L37
L38:
        mov r12,(--sp)
        call _message_refused__FP7RTS_msg
        inc sp,sp
L37:
        mov (r12),r12
        jmp L33
                                |** FUNC EPILOGUE BGN
                                | Restore regs used
        mov (sp++),r7
        mov (sp++),r6
        mov (sp++),r5
        mov (sp++),r4
        mov (sp++),r3
        mov bp,sp               | restore stack ptr
        mov (sp++),bp           | restore frame ptr
        rtn
                                |** FUNC EPILOGUE END
```

Program 49: User-Level Dispatch, Compiled Code, Part 2

Program 50 is the interrupt-dispatch loop, and Programs 51 and 52 are the compiled code.

```
void    (*interrupt_table[8])() =        // a jump table filled with the
{                                        // names of routines that correspond
    software_int,                        // to various interrupt conditions
    __________recv_int,
    _____send______int,
    _____send_recv_int,
    buff___________int,
    buff______recv_int,
    buff_send______int,
    buff_send_recv_int
};

void
interrupt_dispatch ()
{
    while (1)
    {
        int  pending = ISR;              // get the pending interrupts

        (*interrupt_table[pending]) (); // dispatch to an interrupt handler

        ISR = pending;                   // writing back acknowledges all the
                                         // interrupts that have just been
                                         // serviced

        asm ("PUNT");                    // assembly instruction to return to
                                         // the other context

    }                                    // when the next interrupt arrives,
}                                        // we shall start here

void
__________recv_int ()                    // on receive interrupt,
{                                        // the newly-arrived message is
    RTS_msg*  m = MRP;                   // linked into the receive queue
    m->next = 0;
    RQ_TAIL = RQ_TAIL->next = m;
    MRP = &m->size;
}

void
_____send______int ()                    // on send interrupt,
{                                        // the send queue is updated and
    SQ_HEAD = SQ_HEAD->next;             // checked for additional messages
    if ( SQ_HEAD->next )
        send_next_message();
}
```

Program 50: Programming the Interrupt-Driven Context

```
.data
.globl  _interrupt_table
_interrupt_table:
        .word _software_int__Fv
        .word ___________recv_int__Fv
        .word ______send______int__Fv
        .word ______send_recv_int__Fv
        .word _buff___________int__Fv
        .word _buff______recv_int__Fv
        .word _buff_send______int__Fv
        .word _buff_send_recv_int__Fv
```

Program 51: Programming the Interrupt-Driven Context, Compiled Code, Part 1

```
.text
.globl  _interrupt_dispatch__Fv
_interrupt_dispatch__Fv:
                               |**  FUNC PROLOGUE BGN
        mov bp,(--sp)          | save frame ptr
        mov sp,bp              | new frame ptr
                               | Save regs used
        mov r3,(--sp)
                               |** FUNC PROLOGUE END
L2:
        mov isr,r3
        call (r3+_interrupt_table)
        mov r3,isr
#APP
        PUNT
#NO_APP
        jmp L2
                               | Restore regs used
        mov (sp++),r3
        mov bp,sp              | restore stack ptr
        mov (sp++),bp          | restore frame ptr
        rtn
                               |** FUNC EPILOGUE END

.globl  ___________recv_int__Fv
___________recv_int__Fv:
        mov mrp,r0
        mov #0,(r0)
        mov r0,(r11)
        mov r0,r11
        inc r0,r0
        mov r0,mrp
        rtn

.globl  ______send______int__Fv
______send______int__Fv:
        mov (r15),r15
        mov (r15),r0
        cmp #0,r0
        jeq L9
        call _send_next_message__Fv
        rtn
```

Program 52: Programming the Interrupt-Driven Context, Compiled Code, Part 2

Bibliography

[1] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. MIT/LCS/TM-454.b, MIT, November 1991.

[2] Alok Aggarwal, Ashok K.Chandra, Marc Snir. On Communication Latency in PRAM Computations. in *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures,* 1989.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[4] William C. Athas. Fine-Grain Concurrent Computation. Caltech Computer Science Technical Report 5242:TR:87 (Ph.D. thesis), 1987.

[5] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* 21(8): 9–24, August 1988.

[6] J. Backus. Can programming be liberated from the Von Neumann style?, A functional style and its algebra of processes. *CACM,* 21(8): 613–641, August 1978.

[7] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison-Wesley, 1990.

[8] Gordon Bell. Ultracomputers: A Teraflop Before Its Time. *CACM,* 35(8): 26–47, August 1992.

[9] Nanette J. Boden. A Study of Fine-Grain Programming Using Cantor. Caltech-CS-TR-88-11, 1988.

[10] Nanette J. Boden. Runtime Systems for Fine-Grain Multicomputers. Caltech-CS-TR-92-10, 1993.

[11] Robert W. Brodersen. Evolution of VLSI Signal-Processing Circuits. in *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference,* edited by Charles L. Seitz, MIT Press, 1989.

[12] Henry Burkhardt III, Steven Frank, Bruce Knobe, and James Rothnie. Overview of the KSR1 Computer System. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[13] Alan Burns. *Programming in occam 2.* Addison-Wesley, 1988.

[14] Steve M. Burns. Performance Analysis and Optimization of Asynchronous Circuits. Caltech-CS-TR-91-01, 1991.

[15] David Chaiken, John Kubiatowicz, Anant Agarwal. Limitless Directories: A Scalable Cache Coherence Scheme. MIT/LCS/TM-448, MIT, June 1991.

[16] Thomas J. Chaney and Charles E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers,* pp. 421–422, April 1973.

[17] Andrew A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[18] Rohit Chandra, Anoop Gupta, John L. Hennessy. Integrating Concurrency and Data Abstraction in a Parallel Programming Language. Stanford University Technical Report No. CSL-TR-92-511, February 1992.

[19] K. Mani Chandy, Carl Kesselman. Compositional C++: Compositional Parallel Programming. Caltech-CS-TR-92-13, 1992.

[20] K. Mani Chandy, Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[21] K. Mani Chandy, Stephen Taylor. A Primer for Program Composition Notation. Caltech-CS-TR-90-10, 1990.

[22] William Douglas Clinger. Foundations of Actor Semantics. MIT AI Lab Technical Report AI-TR-633, May 1981.

[23] Cohen, D., Finn, G., Felderman, R., DeSchon, A. ATOMIC: A High-Speed, Low-Cost, Local Area Network. USC/ISI Technical Report ISI/RR-92-291, Sept 1992.

[24] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance Measurements on a 128-Node Butterfly Parallel Processor. in *Proceedings of the 1985 International Conference on Parallel Processing,* pp. 531–540, 1985.

[25] Ole-Johan Dahl, Edsger W. Dijkstra and Charles A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[26] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, Norwell MA, 1987.

[27] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnections Networks. *IEEE Transactions on Computers,* C-36, 5: 547–553, May 1987.

[28] Edsger W. Dijkstra. Cooperating Sequential Processes. in *Programming Languages*, edited by F. Genuys, Academic Press, 1968.

[29] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[30] Charles Donnelly and Richard Stallman. BISON: The YACC-compatible Parser Generator. The Free Software Foundation, June 1992.

[31] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. University of California, Berkeley Report No. UCB/CSD 92/675, March 1992.

[32] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[33] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE,* 54:12 1901–1909, December 1966.

[34] Ian Foster and Stephen Taylor. *STRAND: New Concepts in Parallel Programming.* Prentice Hall, 1990.

[35] A. Gotlieb *et al.* The NYU Ultracomputer — Designing and MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers,* 175–189, February 1983.

[36] Per Brinch Hansen. Structured Multiprogramming. *CACM,* 15(7): 574–578, July 1972.

[37] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[38] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. in *Artificial Intelligence: An MIT Perspective,* edited by Winston and Brown, MIT Press, 1979.

[39] Carl Hewitt and Henry Baker. Laws for Communicating Parallel Processes. IFIP-77, Toronto, August 1977, pp. 987–992.

[40] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[41] C. A. R. Hoare. Monitors: an Operating System Structuring Concept. *CACM,* 17(10): 549–557, October 1974.

[42] C. A. R. Hoare. Communicating Sequential Processes. *CACM,* 21(8): 666–677, August 1978.

[43] Waldemar Horwat. Concurrent Smalltalk on the Message-Driven Processor. MS Thesis, MIT, Department of Electrical Engineering and Computer Science, May 1989.

[44] Waldemar Horwat, Andrew A. Chien, William J. Dally. Experience with CST: Programming and Implementation. in *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* 1989.

[45] INMOS Limited. *Transputer Reference Manual* Prentice Hall, 1988.

[46] Kirk Johnson, Anant Agarwal. The Impact of Communication Locality on Large-Scale Multiprocessor Performance. MIT/LCS/TM-463, MIT, February 1992.

[47] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. MIT/LCS/TM-473, MIT, October 1992.

[48] Clyde P. Kruskal, Marc Snir. Cost-Bandwidth Tradeoffs for Communication Networks. in *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures,* 1989.

[49] David J. Kuck *et al.* Measurements of Parallelism in Ordinary FORTRAN Programs. *IEEE Computer* 7(1): 37–46, January 1974.

[50] H. T. Kung and Charles E. Leiserson. Algorithms for VLSI Processor Arrays. Chapter 8.3 in *Introduction to VLSI Systems,* by Carver Mead and Lynn Conway, Addison-Wesley, 1980.

[51] Charles R. Lang, Jr. The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture. Caltech Computer Science Technical Report 5014:TR:82, 1982.

[52] K. Rustan M. Leino. Multicomputer Programming with Modula-3D. Caltech-CS-TR-93-15, 1993.

[53] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz and Monica Lam. Design of the Stanford DASH Multiprocessor. Stanford University Technical Report No. CSL-TR-89-403, December 1989.

[54] Sigurd L. Lillevik. The Touchstone 30 Gigaflop DELTA Prototype. IEEE 0-8186-2290-3/91/0000/0671/$01.00 671–677, March 1991.

[55] Johan J. Lukkien and Jan L. A. van de Snepscheut. A Tutorial Introduction to Mosaic Pascal. Caltech-CS-TR-92-26, 1992.

[56] L. R. Marino. The Effect of Asynchronous Inputs on Sequential Network Reliability. *IEEE Transactions on Computers,* C-26: 1082–1090, November 1977.

[57] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. Chapter one in *Developments in Concurrency and Communication,* edited by C. A. R. Hoare, Addison-Wesley, 1990.

[58] Alain J. Martin, Drazen Borkovic, Marcel van der Goot, Tony Lee, José Tierno. CAST, Caltech Asynchronous Synthesis Tools: The First Release. in Caltech-CS-TR-93-11, 1993.

[59] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic,Pieter J. Hazewindus. The Design of an Asynchronous Microprocessor in *Advanced Research in VLSI: Proceedings of the 1989 Decennial Caltech Conference,* edited by Charles L. Seitz, MIT Press, 1989.

[60] Daniel Maskit, Stephen Taylor. Experiences in Programming the J-Machine. Caltech-CS-TR-93-11, 1993.

[61] David May. Occam and the Transputer. Chapter two in *Developments in Concurrency and Communication,* edited by C. A. R. Hoare, Addison-Wesley, 1990.

[62] Carver Mead, Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[63] Michael Noakes, William J. Dally. System Design of the J-Machine. *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI,* MIT Press, 1990.

[64] Michael D. Noakes, Deborah A. Wallach and William J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. pp. 224–234 in *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, California, May 16-19, 1993.*

[65] Alan V. Oppenheim and Ronald W. Schafer. *Digital Signal Processing,* pp. 294–299, Prentice-Hall, 1975.

[66] Miroslav Pěchouček. Anomalous Response Times of Input Synchronizers. *IEEE Transactions on Computers,* C-25: 133–139, February 1976.

[67] Michael J. Pertel. A Simple Network Simulator. Caltech-CS-TR-92-04, 1992.

[68] Michael J. Pertel. A Critique of Adaptive Routing. Caltech-CS-TR-92-06, 1992.

[69] G. F. Pfister *et al.* The IBM Research Parallel Processor (RP3): Introduction and Architecture. in *Proceedings of the 1985 International Conference on Parallel Processing,* pp. 764–771, 1985.

[70] John R. Rose, Guy L. Steele Jr. C*: An Extended Language for Data Parallel Programming. Thinking Machines Technical Report PL-87.5, March 1987.

[71] Fred Rosenberger and Thomas J. Chaney. Flip-Flop Resolving Time Test Circuit. *IEEE Journal of Solid-State Circuits,* SC-17: 731–738, August 1982.

[72] Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney and Ting-Pien Fang. Q-Modules: Internally Clocked Delay-Insensitive Modules. *IEEE Transactions on Computers,* 37(9): 1005–1018, September 1988.

[73] Charles L. Seitz. Ideas About Arbiters. *LAMBDA,* 10–14, First Quarter 1980.

[74] Charles L. Seitz. System Timing. Chapter seven in *Introduction to VLSI Systems,* by Carver Mead and Lynn Conway, Addison-Wesley, 1980.

[75] Charles L. Seitz. Concurrent VLSI Architectures. *IEEE Transactions on Computers,* C-33: 1247–1265, December 1984.

[76] Charles L. Seitz. The Cosmic Cube. *CACM,* 28(1): 22–33, January 1985.

[77] Charles L. Seitz. Multicomputers. Chapter five in *Developments in Concurrency and Communication,* edited by C. A. R. Hoare, Addison-Wesley, 1990.

[78] Charles L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizovic, C. S. Steele and W-K. Su. The Architecture and Programming of the Ametek Series 2010 Multicomputer. in *The Third Conference on Hypercube Concurrent Computers and Applications,* Pasadena, California, 1988.

[79] Charles L. Seitz, Nanette J. Boden, Jakov Seizovic and Wen-King Su. The Design of the Caltech Mosaic C Multicomputer. in *Research on Integrated Systems: Proceedings of the 1993 Symposium,* edited by Gaetano Borriello and Carl Ebeling, MIT Press, 1993.

[80] Charles L. Seitz, Jakov Seizovic, Wen-King Su. The C Programmer's Abbreviated Guide to Multicomputer Programming. Caltech-CS-TR-88-1, 1988.

[81] Charles L. Seitz and Wen-King Su. A family of routing and communication chips based on the Mosaic. in *Research on Integrated Systems: Proceedings of the 1993 Symposium,* edited by Gaetano Borriello and Carl Ebeling, MIT Press, 1993.

[82] Jakov Seizovic. The Reactive Kernel. Caltech-CS-TR-88-12, 1988.

[83] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers.* MIT Press, 1987.

[84] Ray Simar. Personal Communication. 1992.

[85] Jaswinder Pal Singh, Wolf-Dietrich Weber, Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Stanford University Technical Report No. CSL-TR-92-526, June 1992.

[86] K. Stuart Smith, Arunodaya Chatterjee. A C++ Environment for Distributed Application Execution. MCC Technical Report ACT-ESP-275-90, 1990.

[87] Don Speck. Fast 512K Scalable CMOS dRAM. *Advanced Research in VLSI 1991: UC Santa Cruz,* MIT Press, 1991.

[88] E. Spertus. Execution of Dataflow Programs on General-Purpose Hardware. MS Thesis, MIT, Department of Electrical Engineering and Computer Science, August 1992.

[89] Craig S. Steele. Affinity: A Concurrent Programming System for Multicomputers. Caltech-CS-TR-92-08, 1992.

[90] Leon Sterling, Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[91] Wen-King Su. Reactive-Process Programming and Distributed Discrete-Event Simulation. Caltech-CS-TR-89-11, 1989.

[92] Ivan E. Sutherland. Micropipelines, Turing Award Lecture. *CACM,* 32(6): 720–738, June 1989.

[93] Stephen Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, 1989.

[94] Michael D. Teimann. User's Guide to GNU C++. The Free Software Foundation, 1990.

[95] Manu Thapar. Cache Coherence for Scalable Shared Memory Multiprocessors. Stanford University Technical Report No. CSL-TR-92-522, May 1992.

[96] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary. Thinking Machines Corporation, January 1992.

[97] Clark D. Thompson. Area-Time Complexity for VLSI. in *Caltech Conference on Very Large Scale Integration,* edited by Charles L. Seitz, pp. 495–508, 1979.

[98] Harry J. M. Veendrick. The Behavior of Flip-Flops Used as Synchronizers and Prediction of Their Failure Rate. *IEEE Journal of Solid-State Circuits,* SC-15: 169–176, April 1980.

[99] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1987.