Concurrent Scheme

Robert R. Kessler and Mark R. Swanson[1]

UUCS-90-014

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

August 29, 1990

Abstract

This paper describes an evolution of the Scheme language to support parallelism with tight coupling of control and data. Mechanisms are presented to address the difficult and related problems of mutual exclusion and data sharing which arise in concurrent language systems. The mechanisms are tailored to preserve Scheme semantics as much as possible while allowing for efficient implementation. Prototype implementations of the resulting language are described which have been completed. A third implementation is underway for the Mayfly, a distributed memory, twisted-torus communication topology, parallel processor, under development at the Hewlett-Packard Research Laboratories. The language model is particularly well suited for the Mayfly processor, as will be shown.

[1]Work supported by Hewlett-Packard Research Labs-Palo Alto.

CONCURRENT SCHEME

Robert R. Kessler and Mark R. Swanson*
Utah Portable A.I. Support Systems Project, Department of Computer Science
University of Utah, Salt Lake City, Utah 84112

Abstract

This paper describes an evolution of the Scheme language to support parallelism with tight coupling of control and data. Mechanisms are presented to address the difficult and related problems of mutual exclusion and data sharing which arise in concurrent language systems. The mechanisms are tailored to preserve Scheme semantics as much as possible while allowing for efficient implementation. Prototype implementations of the resulting language are described which have been completed. A third implementation is underway for the Mayfly, a distributed memory, twisted-torus communication topology, parallel processor, under development at the Hewlett-Packard Research Laboratories. The language model is particularly well suited for the Mayfly processor, as will be shown.

1 Introduction

The intent in developing Concurrent Scheme (CS) has been to provide an efficient concurrent Lisp for distributed memory multiprocessors, in particular for the Mayfly architecture, a descendent of the FAIM-1 Symbolic Multiprocessing System[DR85]. The approach adopted has been to minimize the addition of new syntax and mechanism and to limit

*Work supported by Hewlett-Packard Research Labs - Palo Alto.

changes in familiar Lisp semantics to a few well-defined areas. CS itself is based on Scheme, as defined in *The Revised[3] Report on the Algorithmic Language Scheme* (R3RS)[RC86].

The design of the language has been driven by three forces:

1. the expected nature of the computations to be supported;
2. the characteristics of the underlying architecture;
3. a pragmatic requirement of real, demonstrable performance through concurrency.

The structure of a computation in the original FAIM-1, and in the current Mayfly system, is an object-oriented one. The design envisions multiple objects, each completely encapsulating its state, communicating via messages. The design also envisions the existence of multiple threads of control and a sufficiently large population of objects, so that at any one time evaluation can be occurring within multiple objects. This model of computation was chosen both for its desirable programming characteristics of modularity, encapsulation, abstraction, etc. and for its promise of potential concurrency that arises from the relative independence of objects.

The design of the architecture and the design of the programming model are closely intertwined. The distributed nature of the Mayfly system constrains the universe of efficiently implementable mechanisms. For example, a monolithic shared address space across all processors is not practical. On the other hand, we shall see that message passing and its inherent data copying may be quite acceptable.

Finally, the Mayfly is more than an exercise in building a distributed multi-computer. A real goal exists of producing a system that outperforms existing sequential systems utilizing similar technology.[1] Scheme was the Lisp dialect selected because its compactness and constrained set of language features provide the potential for an efficient uniprocessor implementation and for a distributed implementation. Also to this end, certain language features have been included in or excluded from Concurrent Scheme largely because of their effects on global efficiency of the language implementation.

[1] Actually, the goal is to outperform uniprocessors using faster/more expensive circuit technology with a Mayfly system constructed from a large number of less-costly/slower components.

In the first section, Concurrent Scheme's reliance on closures is discussed. In the following section, new mechanisms are described and motivated and the deviations from standard Scheme semantics are likewise described and motivated. The next section discusses specification of concurrency and synchronization. The fourth section addresses the existing implementations, their particular strengths and weaknesses. In the final section, the Mayfly architecture and its suitability for supporting Concurrent Scheme are described.

2 Threads and Closures

CS provides for the creation of multiple threads of control; the procedure **make-thread**, given a function object as an argument, will create a new thread that will apply that function to the rest of its arguments. **Make-thread** is similar to the *future* construct[BH77, HJ85]. The new thread is not necessarily run immediately, so **make-thread** returns an object called a **placeholder**[Mil87] as soon as the new thread is created. The creating process can then continue, using the **placeholder** *in place of* the value that **make-thread** will eventually return. Unlike *future*, **make-thread** cannot be wrapped around any arbitrary form, but rather functions like **apply**. The reason for this difference will become clear later in this section.

Concurrent Scheme relies on the *closure* as its primitive for object-oriented programming. A closure exhibits those features of an object that we deem most important: pairing of computational methods with an associated instance of state (the closure's environment) and encapsulation of that state by the lexical scoping of the environment variables. These features do not present what is commonly thought of as a complete object system, but do provide the core upon which an object system can be built.

A further consideration in concentrating upon closures as a key mechanism in Concurrent Scheme is that closures would have to be supported in a consistent manner in any event. In a language such as Scheme, environments can be nested to arbitrary depths, with sharing occurring at different depths. The lifetimes of these environments is, however, are

determined dynamically rather than statically, due to the existence of closures. Introducing the ability to create new threads of control at arbitrary levels in these environments introduces an expanded need to preserve and share environments.

```
(define foo
  (lambda (n)
    (let ((m nil))
      (touch (future (set! m (bar n))))
      m)))
```

Consider the function **foo**; note that the **future** construct, rather than **make-thread** has been used. When **foo** creates a new thread to evaluate **(set! m (bar n))**, the environment defining **m** must be captured in order for the **set!** to produce the correct side effect, for **bar** to receive the proper argument, and for the function to return the correct value.[2] In effect, an implicit closure has been generated to capture this environment.

Concurrent Scheme limits the specification of concurrency to function application. A legal formulation of **foo** in CS would be:

```
(define foo
  (lambda (n)
    (let ((m nil))
      (set! m (touch (make-thread bar n)))
      m)))
```

No closure need be produced in this formulation, since the arguments to **make-thread** are evaluated in the existing environment before **make-thread** is invoked. Thus, the problem of environment preservation for concurrency is removed. Real programs are more complex than this simple example. Should the new thread need to share an environment with an existing thread, using explicitly created closures as the functional argument to **make-thread** can achieve the desired effect, as in the following version of **foo**[3].

[2] Touch is a system function that blocks the current thread until the thread created by **future** completes. The value of touch is the value of its argument.

[3] This example also results in deadlock in CS, as will be shown in the next section.

```
(define foo
  (lambda (n)
    (let ((m nil))
      (touch (make-thread (lambda () (set! m (bar n)))))
      m)))
```

We have taken *implicit* instances of closures and made them explicit. As the following section will show, considerable effort has been expended to rationalize closure behavior in a distributed, concurrent environment. Having spent this effort on an object-like mechanism, introducing and supporting a competing object mechanism would be both redundant and wasteful.

3 Domains

Any concurrent programming system with shared, mutable data must offer mechanisms for mutual exclusion. Earlier parallel Lisps, such as MultiLisp[HJ85] and MultiScheme[Mil87] depend on explicit specification and use of locks by the applications programmer. Qlisp [GM84] provides both locks and closures with queues. While locks provides the programmer with almost unlimited flexibility and precision in specifying mutual exclusion, three factors motivate against using this model for CS. First, the commonly used technique of sharing of structure within Lisp data objects is highly dynamic — so much so that Lisp programmers rarely think about it. This sharing can obscure the true boundaries of mutually exclusive operations and may make it impossible to actually utilize the precise control which user-specified locks appear to offer. Second, on general purpose architectures, even a highly tuned implementation of locks will result in an operation that is quite costly, relative to other constructs in the language such as function call or constructing a simple data object like a cons cell. Thus, use of locks to provide fine-grained mutual exclusion is likely to provide only the illusion of fine control. Third, we wished to provide a programming environment that was largely similar to the sequential environments pro-

grammers are familiar with. The need to specify locks around accesses to shared data is a direct violation of this goal.

3.1 Mutual Exclusion

A domain is an entity containing mutable data, specifically in the form of closure environments. It has the property that at most one thread of control can execute within ("occupy") it at any time. The name derives from the domain construct in Hybrid [Nie87]. The mechanism is also very similar to Hoare's monitor [Hoa74], but with modifications to address the needs of a highly dynamic language operating in a truly concurrent environment. It provides a guarantee of *mutual exclusion.* Other threads needing to execute within an *occupied* domain are queued outside the domain by the runtime system. In a CS program, *all* computation occurs within these monitor-like objects; i.e., mutual exclusion is the norm.

```
(define foo
  (lambda (n)
    (let ((m nil))
      (touch (make-thread (lambda () (set! m (bar n)))))
      m)))
```

This example from the previous section was noted as resulting in deadlock. The behavior responsible for this outcome is easily explained. The initial thread creates a new thread and proceeds immediately to wait for that thread's completion. In waiting, the initial thread continues to "occupy" the current domain. The new thread, which must also execute in that domain, cannot gain entry and can never begin execution, let alone complete.

A domain is created by calling the procedure **make-domain**. **Make-domain** requires at least one argument, a function object. It creates a new domain, *enters* that domain, and applies the function object to the remaining arguments, if any. The value of that application is the value returned by **make-domain**. Note in particular that domains are

not hierarchical; the new domain is in no sense "contained" within the domain that caused its creation.

This behavior is also similar to that of the **qlambda** construct of Qlisp, which creates a closure with an associated queue and a guarantee of mutual exclusion. A domain's specification of mutual exclusion is more complete than that of **qlambda**, because the mutual exclusion is a characteristic of entire environments contained within the domain rather than just being an attribute of a particular closure.

```
(define (foo x)
  (let ((state (list x)))
    (cons (qlambda t (arg)
            (set! state (cons arg state)))
          (qlambda t (arg)
            (set-car! state arg)))))
```

In this example[4], the pair of closures resulting from the **qlambda**'s each has its own queue, but since they both access the same environment variable, **state**, the consistency of that variable is not protected by the **qlambda** construct.

```
(define (foo x)
  (let ((state (list x)))
    (cons (lambda (arg)
            (set! state (cons arg state)))
          (lambda (arg)
            (set-car! state arg)))))
```

This version of **foo** is simply standard Scheme syntax. The domain discipline of CS guarantees mutual exclusion for both closures, since they *must* exist in the same domain and only one thread can execute in that domain at any given time.

[4] Qlisp is actually based on Common Lisp; the example uses Scheme syntax. The first argument of **qlambda** is a concurrency control device which can be ignored for our purposes here.

In addition, the domain "operationally" encapsulates data; data that "enters" or "leaves" a domain is *copied*, not passed by reference. When a domain is created by calling the procedure **make-domain**, the first argument, the function object, is applied in the new domain to *copies* of the remaining arguments, if any. A *copy* of the value of that application is the value returned by **make-domain**.

3.2 Closures

Domains are not first-class objects; references to them are indirect. A program can create domains and can invoke a closure existing within a domain. To maintain a domain as "reachable" or "live," a program must maintain a live pointer to a closure in that domain. This in turn implies that the function object supplied to **make-domain** must return a closure if the domain is to remain reachable.

```
(define (make-collection)
  (let ((state ()))
    (lambda (method-name . args)
      (cond ((eq? method-name 'add)
             (set! state (cons (car args) state))
             '())
            ((eq? method-name 'contents)
             state)))))

(set! a-collection
  (make-domain make-collection))
```

Here an invocation of **make-domain** returns a closure existing within a newly created domain. As long as **a-collection**, or some other reachable location, retains the value returned by **make-domain**, that new domain will remain "live." When no reachable location contains that object, the domain itself is no longer reachable and will be garbage collected.

This close interaction between closures and domains is entirely intentional. In CS, all closures exist within domains. A stronger condition exists: a closure's entire environment exists within some specific domain. An implication of this is that closures that share an environment, even if the sharing is partial, must exist in the same domain. This implication is transitive. The mutual exclusion provided by the domain is sufficient to provide exclusive access to environment variables shared by multiple closures. Since closures represent the "methods" for primitive "objects" in CS, this encapsulation of closures within domains makes invocation of methods in CS atomic by default.

3.3 Copying Semantics

CS's domain mechanism goes beyond simple monitors, `qlambda`'s, and Hybrid's domains in that domain boundaries are also the boundaries at which copying semantics is specified to hold. This congruence is reasonable, considering that mutual exclusion is normally specified to ensure consistent access to and modification of data objects. Exporting pointers to the structured data of a domain would void the utility of the mutual exclusion provided by the domain.

In the `collection` object of the example above, the `contents` method returns the list which contains the object's state. Were the expression `(a-collection 'contents)` evaluated immediately after the expression that created the `collection` object, the result of that expression would be a copy, in the current domain, of the list contained in the `collection` object.

This use of copying semantics for interactions between domains is the most far-reaching divergence of CS semantics from those of Scheme. A domain interacts with another domain by invoking, as a normal function call, a gateway object that corresponds to a closure in the other domain. The arguments passed to the gateway and the results returned through it are *copied* from one domain to the other. Unless otherwise specified, copying preserves structure sharing. Closures, however, are not really copied. Instead we create and pass gateway objects. There are three motivations for this use of copying semantics, one at the language level and the other an implementation consideration:

Encapsulation – At the language level, we view the closure as a form of object. An object here is simply some state, or data, with related behavior and an exported interface. One "virtue" of these objects is the encapsulation of the state of the object. This encapsulation is an artifact of the lexical scoping of the closure's environment variables. Complete encapsulation can only be guaranteed by ensuring that not only names, but the structure of the closure's component objects be contained. Placing these objects (closures) within domains ensures this complete encapsulation, since only *copies* of the state variables can leave the domain (and hence the closure).

Distributed Systems – Since our target architecture falls into the class of distributed systems, a choice arises between managing external data pointers or copying data between processors. For a variety of complexity and performance reasons, we have chosen to copy data between processors. Physical placement of objects onto particular processors may differ between different executions of an application. As a result, copying that was simply an artifact of the placement of interacting objects would potentially yield different results with different placement strategies. Hence we chose to impose copying semantics as a language feature at the domain level.

Predictability – Interactions between separate domains entail possible scheduling activities. The scheduling characteristics are generally not part of the language definition. It is not possible, in the general case, for a programmer to predict the order of evaluation of independent threads. This means that between the time the invocation of a closure in a different domain is initiated and the time that evaluation actually occurs, the state of the invoking domain could have changed. This could occur if the invocation was the result of `make-thread` or by use of `delegate` (see Section 3.8). If pointers to structured data were passed as arguments, this unpredicatbility of actual time of use of the arguments would, without explicit synchronization by the programmer, lead to unpredictable results. *Copying* the arguments preserves their values at the time of closure invocation. A similar argument can be made regarding return values.

Although we have introduced copying semantics at the inter-domain level, CS is not intended to be a purely functional language, even at that level. The objects themselves can be, and are, shared, albeit in a mutually exclusive manner. This sharing is accomplished through the exported interface, which in CS is defined in terms of closures. Thus, when a closure is transmitted from one domain to another, it cannot be copied, since this would be tantamount to copying the entire object and would preclude sharing of objects. Rather, the closure is transformed into a gateway, which is an executable object that implements monitor-like access to the shared closure. There is a further operational characteristic of gateways: in determining the domain in which to evaluate a form, they ensure that control travels to the location of data, so that control never acts upon remote data.

As a result of the imposition of copying semantics on inter-domain transactions, it can be guaranteed that no pointers to a domain's mutable structured data exist outside of that domain. Likewise, if a pointer to mutable data is found within a domain, it points to that domain's data. These "invariants" have significant impacts on other language features, notably global variables and storage management.

3.4 Global Values

Another major divergence from standard Lisp semantics involves CS's treatment of global values. The combined effects of copying semantics and a distributed implementation result in severe constraints on the use of globals.

As with all other structured data items, the values of globals must exist within some domain. A distinguished domain exists that contains all global values. The language supports only single assignment of global values.[5] The language does not *enforce* the single-assignment discipline; i.e., subsequent assignments can be performed, but the language does not define the results for such assignments. The language also specifies, and again does not enforce, that global values should not be destructively modified.

These restrictions are necessary to maintain the invariant stated at the end of the

[5] At the time the assignment is performed, the value is copied into the distinguished domain, maintaining the invariant of the last section.

previous section, that a domain completely encapsulates pointers to its mutable data. Since globals are not mutable, it is acceptable to have pointers to global data within a domain; likewise, the immutability of globals guarantees that a global value cannot be changed to contain a pointer to some domain's private data.

The distributed nature of CS also serves to motivate against multiple assignments to global variables. Current implementations employ broadcast of global values to replicated symbol tables on each of the physical processors. Thus, for shared, read-only data structures, fast local references are provided.[6] A correct implementation of multiple assignments would require serialization of "simultaneous" assignments to the same variable. This would require global synchronization mechanisms that the design of CS has purposely avoided.

3.5 Storage Allocation and Reclamation

A side effect of the complete encapsulation of mutable data is that storage allocation and garbage collection can be limited to the scope of individual domains. This reduces the size, scope, and latency of garbage collections, since collections become strictly local to the current domain. The roots of the collection are limited to the stacks of any threads that might contain pointers into the domain[7] and the list of potentially live closures that have been exported to other domains. Global synchronization for allocation and garbage collection of normal Scheme data structures is unnecessary.

Garbage collection must address exported closures, of course. Export information is kept externally to the domain and is managed on a reference count basis. Therefore, although garbage collection does not entail any global synchronization, it often results in message passing to maintain the reference count of other domains' exported objects.

[6] Read access time to globals in CS is identical to that in sequential Scheme.

[7] Note the use of the plural for threads; this apparent violation of the rule of mutual exclusion is addressed in Section 3.8.

3.6 Open Semantic Issues

Standard Scheme does not include dynamic, or "special," variable bindings. CS, with its previously mentioned constraints on global variables, provides no support whatever for dynamic variables.

One major facility that is not yet provided is fully general continuations. The distributed nature of CS implies that the representation of a continuation is distributed. At a minimum, the continuation includes the stack(s) associated with a thread, portions of which exist on each node visited by a thread. Full continuations, while not conceptually difficult, promises to be an expensive facility.

The current error handling facilities are rudimentary at best. Scheme specifies little in this respect, so we have turned to Common Lisp for a mechanism. Limited forms of *catch* and *throw* are supported. A special error value is provided to communicate an error termination of a remote procedure invocation. No facilities currently exist for pruning task trees descended from error-producing branches.

3.7 Concurrency

The potential for concurrency exists when multiple threads of control exist and those threads are executing in *different* domains. Other factors must be considered to determine whether physical concurrency will actually be realized. One factor is the physical placement of the domains on nodes within the system; this is discussed in Section 5. Another factor is the interaction of the various threads and their "footprint" within the space of domains. When a thread executing in one domain invokes a closure residing in another domain, the thread will execute in (and "occupy") that other domain if it is not currently occupied; otherwise it will wait to enter that domain. What about the invoking closure's domain? That domain will remain occupied while the thread is executing, and while it is waiting to execute, if that case occurs, in the other domain. The thread's "footprint" covers all of the domains that it simultaneously occupies.

```
(define (first-fn-maker)
  (let ((other-domain (make-domain second-fn-maker 'second-fn)))
    (lambda (x)
      (other-domain x))))

(define (second-fn-maker name)
  (lambda (x)
    (display name)
    (newline)
    (display x)
    (newline)))

(let ((first-fn (make-domain first-fn-maker)))
  (first-fn 'some-argument))
```

In this example, we start by creating a new domain that contains and returns just one closure. This closure is saved in the variable **first-fn**. Within that closure the variable **other-domain** is bound to another closure residing in another newly created domain. When **first-fn** is invoked, the current thread of control enters the domain containing **first-fn**. This domain is now *occupied*; until the call on **first-fn** returns, no other thread can enter that domain. Now, in the body of **first-fn**, the closure stored in **other-domain** is invoked. The current thread enters the domain in which this second closure resides, occupying it, too. After displaying the variable **name** and the argument **x**, the second closure returns. Its domain is now "unoccupied." The first closure likewise returns, rendering its domain unoccupied, too.

Pairs (or groups) of threads interacting in the same domains must take care to avoid deadlock. **Delegate** (see Section 3.8 is one way minimize this situation). A single thread executing in several domains generally can not cause deadlock; an exception is described in Section 6. The constraint that only a single thread may occupy a domain does not preclude a thread from re-entering a domain multiple times without intervening exits from the domain. At creation, a thread is assigned a unique, system-wide identifier. Using

this identifier, the runtime system determines whether a thread attempting to enter an occupied domain is actually the thread currently occupying it. In that case, the thread is allowed to enter, even if other threads are queued waiting to enter.

3.8 Delegation

There are times when maintaining the regimen of a thread occupying a domain from closure invocation until return from that closure is too restrictive. The function **delegate** provides a way for a thread to leave the current domain without exiting the procedure through the normal return-unwinding route. **Delegate** takes at least one argument, a function object, and applies it to the rest of the arguments, if any. Before the application occurs, however, the thread "leaves" the current domain, causing it to be unoccupied, so that some other thread may enter the domain. The thread then performs the application, observing the normal rules for entering new domains (or the original domain, if the function object specifies that domain); Section 4.1 describes how a thread's new location is determined. When the function object returns, the thread must re-enter the original domain; this re-entry once again observes the rules of exclusive access to the domain. The thread may well be queued waiting to re-enter the domain. The return value of **delegate** is the value returned by the function object.

```
(define (manager workers)
  (let ((jobcount 0))
    (do ((w workers (cdr w)))
        ((null? (cdr w)) (set! (cdr w) workers)))
    (lambda (job)
      (let ((worker (car workers))
            res)
        (set! workers (cdr workers))
        (set! res (delegate worker job jobcount))
        (set! jobcount (+ jobcount 1))
        res))))
```

The **manager** procedure, when called with a list of "worker" closures, circularizes the list and returns a closure that, given a "job" to be done, assigns that "job" in a round-robin fashion to the next worker. The "manager" uses **delegate** to make the assignment, so that it is immediately available to respond to another request to assign another job. Note that the manager has been constructed in such a way that its state is consistent at the point **delegate** is used, since another thread may enter the manager before the delegated one returns. Also note that **jobcount** is only incremented after the delegated thread re-enters the domain. Suppose that **worker** returns as its value its third argument, in this example **jobcount**. It is not necessarily the case that **res** and **jobcount** will be equal after the return from **delegate**, since other threads may have entered and completely transmitted the procedure in the meantime.

4 Specifying Concurrency and Synchronization

In CS, concurrency is *explicitly* specified by the programmer through the action of creating a new thread of control to evaluate some function. The related syntax is **(make-thread <procedure> . <args>)**, which is similar to **apply**, except that the application occurs in a new thread. **Make-thread** was introduced earlier, but its behavior is somewhat more complex than indicated by that simple exposition. The actual activities it performs are as follows:

- determine from the function object the initial domain of execution for the new thread;
- perform a structure-preserving copy of the arguments, if any, for the function object;
- initiate creation and subsequent scheduling of a new thread;
- create a placeholder to receive the value of the thread's evaluation and return that placeholder.

Each of these activities is explained in more detail in the following sections.

4.1 Initial Thread Location

The domain within which a newly created thread initially executes is determined by the nature of the function argument to **make-thread**. If the function is not a closure, then the thread will be started in the current domain. The new thread must obey the usual rule of exclusive access, of course, so, in the latter case, it will not start until the thread that created it causes the current domain to become unoccupied (either by returning out of it or by performing a **delegate**).

If the function is a closure, the new thread will start in the domain that contains the closure. Were every thread to start in the current domain, a new thread would not actually start until its creator left the current domain, severely limiting realizable concurrency.

```
(define (thread-maker closure-list)
  (let ((results #f))
    (do ((c closure-list (cdr c)))
        ((null? c))
      (set! results (cons (make-thread (car c)))))
    (do ((r results (cdr r))
         (ans 0))
        ((null? r) ans)
      (set! ans (+ ans (car r))))))
```

Assume that **closure-list** is a list of closures in domains other the one where **thread-maker** is invoked. Further assume that each of these closures will return some integer of interest. After creating the threads and collecting the resulting placeholders, **thread-maker** proceeds to sum the results. Since + is strict in its arguments, it will block if any of the threads have not completed and returned a value by the time + needs that value. Now consider what would happen if the threads did not start in the other domains, but rather started in the current domain. The computation would deadlock, since the main thread will not leave the **thread-maker** function (and therefore will obviously not leave **thread-maker**'s domain) until all of the new threads have finished. But the new threads would not start until the main thread left the domain.

4.2 Argument Copying

Make-thread always copies the arguments provided (beyond the first one, the function object). In the case where the function object is a closure residing in another domain, the arguments are copied into that domain in keeping with the copying semantics discussion in Section 3.3. In the case of a non-closure function object, or a closure that resides in the current domain, the arguments are still copied, even though there is no crossing of a domain boundary. This behavior ensures that when the thread starts, its arguments will be unchanged from their values at the time the thread was created. If the actions of the function are intended to produce side-effects to structured data shared with other threads, such data must be accessible in the function's environment rather than being passed as arguments.

4.3 Thread Creation and Scheduling

Make-thread initiates the creation of a new thread. This does not imply anything about when the thread will actually be started, or even when it will be created. It is not necessarily the case that the thread has been created when **make-thread** has returned, especially in a multi-node system where the thread may actually be created on a different physical node. There is, in fact, no guarantee that the threads created by sequential calls on **make-thread** will be created or scheduled in that same sequence. Occasionally, some guarantees about *relative* scheduling are useful. **Make-acked-thread** guarantees that the thread it creates will be scheduled before other threads created in the same target domain by subsequent calls to **make-thread** by the same originating thread. Suppose two threads called A and B, executing concurrently (necessarily in different domains), both start a number of threads in a third domain using **make-acked-thread**. The threads started by A will start in the same order that A created them; likewise for those created by B. Nothing, however, is implied or guaranteed about the order of A's children with respect to B's and vice versa.

4.4 Return Value

Since **make-thread** may return before the new thread has a chance to return a result, a placeholder is "attached" to a thread and this placeholder is returned by **make-thread**. An alternative procedure, **make-orphaned-thread**, is provided for use when no result is expected and the thread is created for the side-effects it will produce. The return value of **make-orphaned-thread** is simply **#f** and no placeholder is created. This form can be an important optimization tool. Not only are the costs associated with creation and later reclamation of the placeholder saved, but a return of control by the thread is avoided.

4.5 Placeholders

A **placeholder** is a first class object in CS that can be allocated independently of creation a new thread. The standard procedure **make-placeholder** will return a new, unresolved placeholder. An unresolved placeholder is one that has not yet been given a value. Attempting to use the value of an unresolved placeholder in a strict operation causes the current thread to block. The blocked thread will wait until the placeholder receives a value, at which time the thread will be allowed to proceed.

Placeholders can receive values in two ways. First, the placeholder associated with a thread by **make-thread** will receive a value when the thread completes execution. In this case the value of the placeholder is the value returned by the thread. Second, a placeholder may be explicitly given a value by the **determine** function, which takes two arguments, an unresolved placeholder and the value it is to be given. If the placeholder already has a value, **determine** signals an error. The programmer can test whether a placeholder has a value using the function **determined?**. This function takes a single argument, which should be a placeholder. It returns **#t** if the placeholder has a value or **#f** if it does not. Note that it is possible to use **determine** explicitly to set the value of a placeholder associated with a thread. In this case, the thread will cause an error when it completes and an attempt is made to place the return value in the placeholder.

Because they can be separately allocated and explicitly determined and tested, placeholders provide a mechanism for explicit synchronization in CS programs. There is an

additional function, **touch** which explicitly synchronizes on a placeholder. Given a single argument which is an unresolved placeholder, **touch** causes the current thread to block until the placeholder receives a value. If the argument is a resolved placeholder, **touch** returns the placeholder's value; if it is any other data type, **touch** simply returns its argument. **Touch** is analogous to Common Lisp's **identity** function, except that it is *strict* in its argument.

```
(define (make-worker obj go-ph done-ph)
  (lambda ()
     (touch go-ph)
     (munge obj)
     (determine done-ph 'done)))

(define (worker-mgr obj-lst)
  (let ((go-ph (make-placeholder))
        (done-phs '()))
    (do ((objs obj-lst (cdr objs)))
        ((null? objs) #f)
      (set! done-phs (cons (make-placeholder) done-phs))
      (make-thread
        (make-domain
          make-worker (car objs) go-ph (car done-phs))))
    (do-managerial-stuff obj-list)
    (determine go-ph 'go)
    (do ((done done-phs (cdr done)))
        ((null? done) #f)
      (touch (car done)))))
```

In this example, **make-worker** returns a closure that will perform some task after its creator frees it to do so. This delay is achieved by **touch**ing the argument **go-ph**, which should be an undetermined placeholder. After it performs the **munge** task, it signals

completion of the task by `(determine done-ph 'done)`. `Worker-mgr` creates one worker object for each object in the list passed to it. Each of these workers exists in its own domain (and so can potentially run concurrently with the other workers). In addition, it starts threads for running each of the worker objects. Then it performs some unspecified administrative tasks with the list of objects and finally frees the workers to perform their individual tasks by `(determine go-ph 'go)`. The manager function waits for all the workers to finish by touching the list of placeholders associated with the workers.

`Touch` has an optional second argument. If this second argument is missing or is `#f`, then `touch` behaves as previously described. If the second argument is given and is not `#f`, then `touch` will *leave* the current domain unoccupied if it is blocked because the first argument (the placeholder) is unresolved. The motivation for this behavior is similar to that for `delegate`: the programmer may wish to allow multiple threads to begin some potentially blocking activity within a domain, without excluding other threads. This is an exception to the mutual exclusion rule; hence the syntax requires explicit specification of the exception. As with *delegate*, the programmer is responsible for ensuring the consistency of shared data.

5 Nodes and Generators

CS is designed to run on systems of physically distributed processors or nodes. Several global variables are available to provide the program with information about the configuration of the system. `*Pe-self*` contains the zero-relative logical node number; it will, of course, differ depending on which node a thread is running on. `*Maxpes*` is the number of nodes in the system; it will always be greater than zero.

At system startup, each node has (at least) one domain. The function `ith-gen`, given a non-negative integer argument `n`, will return a closure residing in the initial domain on node `(mod n *maxpes*)`. This closure is a *generator*. Its purpose is to initiate evaluation on that other node. The generator closure takes its first argument, a function object, and applies it to the rest of the arguments. Its result is the value returned by the application.

Since the generator is a closure, it will enter the domain it resides in *on whatever node the domain exists* just as any other closure. It is usually the case that the function object passed to the generator should not be a closure, since invoking that closure within the generator will entail entering the closure's domain. This is not usually the intended use of a generator. Generators are usually used either to distribute "pure" computations or to create new domains on other nodes.

```
(let ((i (+ *pe-self* 1)))
  (set! a ((ith-gen i) fact 99)))
```

Assume that **fact** is not a closure, that ***maxpes*** is greater than 1, and that ***pe-self*** is 0. This sequence will result in the evaluation of **(fact 99)** on node 1 within the domain that contains node 1's generator closure. We have here an example of distribution of computation *with no concurrency*, since only one thread is involved. Note that the first assumption is crucial. If **fact** were a closure, then evaluation of **(fact 99)** within the generator (and its domain) would in fact entail entry into the domain containing **fact**, as noted above.

```
(set! i (+ *pe-self* 1))
(set! a (make-thread (ith-gen i) fact 99))
(do-something-else)
(print a)
```

With the same assumptions as before, this sequence not only distributes the computation but contains potential concurrency, since a new thread has been introduced. **Do-something-else** may execute concurrently with **fact** provided that node 1 is not busy, node 1's generator is not currently "occupied," and **do-something-else** takes long enough for **fact** to be invoked on node 1.

In normal practice, the function argument to the generator is often **make-domain**. The new domain created by **make-domain** resides on the node where **make-domain** is invoked. Thus, generators can be used to spread domains and the closures they contain across the available nodes.

```
(define (worker-mgr obj-lst)
  (let ((go-ph (make-placeholder))
        (done-phs '()))
    (do ((objs obj-lst (cdr objs))
         (node 0 (+ node 1)))
        ((null? objs) #f)
      (set! done-phs (cons (make-placeholder) done-phs))
      (make-thread
        ((ith-gen node)
           make-domain
           make-worker (car objs) go-ph (car done-phs))))
    (do-managerial-stuff obj-list)
    (determine go-ph 'go)
    (do ((done done-phs (cdr done)))
        ((null? done) #f)
      (touch (car done)))))
```

Worker-mgr is the same function used in Section 4.5 except that it now produces potential concurrency. The do-variable **node** is used in conjunction with **ith-gen** to spread the workers across the available nodes in a round-robin fashion. The sequence of events for each invocation of **make-thread** is as follows:

1. the argument to **make-thread** is evaluated; this results in the next four events;
2. **(ith-gen node)** returns a closure which is the generator for some node;
3. the generator closure, which resides in its own domain, applies **make-domain** to the arguments **make-worker**, **(car objs)**, etc.;
4. **make-domain** creates a new domain on the same node as the generator, enters that domain, and applies **make-worker** to the remaining arguments;
5. **make-worker** returns a closure to **make-domain**, which returns it to the generator which also returns it; this is the argument to **make-thread**;

6. **make-thread** creates a thread with initial domain determined by the its argument and which may well be on another node;

7. **make-thread** returns a placeholder.

6 Delay Queues

Although placeholders provide a powerful synchronization device, they are not always appropriate. For instance, it is sometimes desirable that the interacting objects not contain specific synchronization which would limit the generality of their use. Another mechanism, the delay queue[Nie87], is provided that allows methods within an object to control the availability of its methods. A delay queue is similar to a condition variable in the monitor construct[Hoa74]. The function **make-delay-queue**, given a closure, creates a delay queue and associates it with that closure. The delay queue can be either "open" or "closed." A thread invoking a delay-queue associated closure from outside the closure's domain can only proceed, i.e., enter the domain, if the delay queue is "open." This constraint is *in addition to* the mutual exclusion property of domains. If the delay queue is "closed," the thread will wait *outside the domain* until the delay queue is opened. The thread does not, therefore, occupy the domain, which would prevent other threads from entering it. This is necessary, since a closed delay queue can only be opened by some other closure (method) which contains the closed delay queue in its environment. That is, a thread must be able to enter a domain in order to open a delay queue belonging to that domain.

```
(define (bounded-buffer)
  (letrec
    ((buffer #f)
     (full? #f)
     (get-method
       (make-delay-queue
         (lambda ()
           (set! full? #f)
```

```
            (dqopen put-method)
            (dqclose get-method)
            buffer)))
      (put-method
        (make-delay-queue
          (lambda (val)
            (set! full? #t)
            (set! buffer val)
            (dqclose put-method)
            (dqopen get-method)))))

    (dqclose get-method)
    (dqopen put-method)
    (lambda (m)
      (if (eq? m 'get-method)
        get-method
        (if (eq? m 'put-method)
          put-method
          (error "bounded-buffer: no such method"))))))

(define (producer sink)
  (do () ()
    (sink (produce-a-datum))))

(define (consumer source)
  (do () ()
    (consume-a-datum (source))))

(define (manager)
  (let* ((buf (bounded-buffer))
         (p (make-thread
              (ith-gen 1) producer (buf 'put-method))))
```

```
(c (make-thread
    (ith-gen 2) consumer (buf 'get-method)))))))
```

In this example, the manager function creates a bounded buffer (which, for simplicity's sake is of size one) and two other objects, a producer and a consumer. The manager connects the producer and consumer by passing them the bounded buffer **put** and **get** methods, respectively. The buffer performs the necessary synchronization without any "knowledge" of this synchronization in either the producer or consumer. It uses **make-delay-queue** to cause the two methods (closures) that it exports to be associated with delay queues. It then opens and closes the delay queues as its internal state dictates.

6.1 Delay Queues and Domains

Delay queues are attributes of a domain; this has a number of implications for their use. Their scheduling characteristics are only enforced at domain boundaries. This means that a thread already executing within a domain can invoke a "closed" delay queue/closure and it will not block. Were it to block, the domain would be permanently deadlocked, since the occupying thread would be blocked and no other thread could enter the domain to open the delay queue. If it is necessary for such a thread to observe the delay queue regimen, it should use **delegate** to leave the domain before trying to enter the delay queue.

The delay queue primitives (those already mentioned, plus **dq-open?**) are only valid within the domain containing the closure and associated delay queue. Calling any of these functions with a closure residing in another domain is an error. One implication of this is that control of delay queues can only occur within the domain containing the associated closure. Further, it is not possible to determine the state of a delay queue from outside the domain it resides in; in fact, it is not even possible to find out if it is a delay queue.

7 A Short Example

The following simple example computes factorial in a distributed manner.

```
(define grain-size 10)

;;; Compute a partial range for the factorial.

(define (partial-fact start end product)
  (if (= start end)
    (* end product)
    (partial-fact (- start 1) end (* start product))))

;;; Portion out the work in grain-size chunks

(define fact-aux
  (lambda (low high)
    (if (> (- high low) grain-size)
     (let ((rest
             (make-thread (ith-gen (+ *pe-self* 1))
                          fact-aux low (- high grain-size))))
        (* (partial-fact high (- high (- grain-size 1)) 1)
           (touch rest #t)))
     (partial-fact high low 1))))

(define (fact n)
  (fact-aux 1 n))
```

The function **partial-fact** recursively computes the product of the integers in the range **start** through **end** (inclusive). **Fact-aux** is the actual agent of distribution. It partitions the range of (assumed positive) integers passed as arguments: one portion to be passed to partial fact *within the current thread* and the rest to be processed by a new thread on another node (by means of **ith-gen**). Note the use of **touch** with its optional second argument; for a sufficiently large range of integers, **ith-gen**'s modulo calculation

will eventually wrap around to a node already in use. To avoid deadlock, the domain is left unoccupied so that subsequent threads, whose values will be needed to complete earlier threads, will be available.

8 The Prototype Implementations

8.1 Architectural Support

The decision to avoid implementing external data pointers was based in part on the expectation that concurrent Lisp would always be run on general-purpose processors; that is, on processors with no integral support for runtime detection and resolution of external references. It was expected, however, that support *external to the processor* would be available to absorb some of the cost of communications. This, in fact, is the case with the Mayfly architecture and to a lesser extent with the BBN GP1000.

To date, three different implementations of CS have been produced. They vary in how they utilize the Mayfly model in which each processing element is really a shared memory parallel processor with an Evaluation Processor (EP) which executes the current task and a Message Processor (MP) which is responsible for task management, inter-node message traffic, and message preparation. The first of these is a uniprocessor implementation, which served as a testbed for the basic mechanisms of multiple threads, mutual exclusion, and copying semantics. It remains as a baseline implementation on which initial development and debugging of CS programs can be performed, free from the effects of "true" concurrency. The other two implementations, the BBN GP1000 multiprocessor version and the networked workstation version, both deliver true concurrency but at widely separated points in the spectrum of multiple/cooperating computers. Each implementation has made different contributions to the ongoing development of the CS model.

8.2 The GP1000 Implementation

The GP1000 is a shared memory multiprocessor of the NUMA (non-uniform memory access) variety. As a shared memory machine, it offered the opportunity to experiment with

the Mayfly model in which message transmission and reception time, including copying time necessitated by copying semantics, could be overlapped with evaluation of application code. Therefore, we implemented individual PEs as asymmetric pairs of GP1000 nodes, sharing memory. Message passing communication between the MPs was straightforward to implement using shared memory.

Using the GP1000, it has been possible to develop and test the kernel mechanisms for creating, managing, and scheduling threads and performing communications tasks on an MP while concurrently running application code on an EP sharing the same physical memory. We have a high degree of confidence that the large portion of CS support code comprising these mechanisms will perform correctly on the Mayfly architecture when it becomes available, which was the initial motivation for the GP1000 implementation.

While GP1000 is a NUMA machine, the Mayfly PE is an Uniform Memory Access (UMA) machine, and the CS support code is currently tailored for the UMA architecture. As a result, the performance of the current GP1000 implementation is poor and its primary value is as a prototype.

8.3 The Network Implementation

A networked implementation is used for debugging the CS runtime system and to develop parallel application programs in a truly concurrent environment. Although the majority of the software is identical for all of the implementations, the networked version differs markedly from the Mayfly model since we use only one physical processor for each PE. The single processor divides its time between MP tasks and applications code. Communication between PEs is implemented as point-to-point UDP links [FJSW85], with a minimal reliability protocol provided by the CS kernel.

The communication characteristics of the networked version differ from the Mayfly model. Not only is message overhead not offloaded to an MP, but message latency is much higher than in either the GP1000 or the Mayfly architecture. In addition to these direct effects, there are indirect effects. For instance, message passing activity involves a switch to the kernel's context, and potentially can result in another process being run

while CS waits. Such context switches will not occur on the Mayfly, and on the GP1000 will never occur *as a side-effect of message passing.* This difference in communications costs has an effect on the granularity of tasks that can be usefully run in parallel.

One useful capability available with the network implementation is the ability to start fully interactive Lisp sessions on each of the remote processors. This allows the application programmer to use the debugging tools provided by Lisp such as trace, backtrace, etc., on each node. Interaction is provided using optional "xterm"[8] windows which are "connected" to each Lisp session using TCP sockets. When xterm windows are not used, only the "root" node is interactive, and output from remote nodes is displayed by sending normal CS messages to the root node. In this way, the remote nodes function more like Mayfly processing elements, waiting for messages to arrive to initiate work. The value of the networked version is in prototyping applications. Its advantages are the common availability of networked workstations and the debugging environment of separate toploops for each physical processor.

9 The Mayfly Architecture

To provide high performance support of Concurrent Scheme, an architecture needs at least the following characteristics:

- low latency, high-bandwidth inter-node communications;
- fast (or overlapped) message preparation, transmission, and receipt;
- fast (or overlapped) task scheduling;
- fast (or overlapped) context switch;
- sufficient memory to hold a reasonable population of tasks;
- sufficient memory bandwidth to support concurrent evaluation and message passing activities.

[8] Xterm is the terminal emulator for the X Window System.

The Mayfly architecture[Dav89] is a distributed memory machine consisting of a number of nodes or PEs. Each PE is connected to six of its neighbors in a twisted torus via fast serial lines. Fully configured Mayfly "surfaces" come in a selection of sizes; the first version will be a nineteen PE surface. The Mayfly is scalable in terms of these surfaces; i.e., one could imagine tiling a plane with these nineteen PE surfaces. The Mayfly nodes and interconnect are designed to display the characteristics listed above. Each node is comprised of 9 subsystems:

1. a Post Office chip;
2. a message processor (MP);
3. an evaluation processor (EP);
4. a floating-point coprocessor;
5. a dual-ported data cache;
6. separate instruction caches for each of the two processors;
7. a moderately-sized (8 megabytes) main memory;
8. a custom context cache device.

The Post Office chip[SRD86] is the communications engine connecting each individual PE with six of its neighbors. Together with the topology, it provides the low latency and high-bandwidth necessary to support many distributed, message-passing objects. It provides packet buffering, flow control, and routing services, so the task of message-passing as seen by the rest of the PE consists largely of address calculation (which can be table driven) and packetization/de-packetization.

Each PE is actually an asymmetric shared memory parallel processor with an EP and an MP; both of which are one chip implementations of the HP Precision Architecture. The EP evaluates application code. A task on the EP runs to completion, times out, or requests a service from the MP, such as invocation of a gateway procedure. When the EP cannot continue execution of the current thread for any of these reasons, it performs a

context switch and starts another available thread from a queue of threads maintained by the MP. The MP performs system services such as task scheduling, message preparation (including copying) and reception, and driving packets to/from the Post Office chip.

Message preparation includes the overhead of copying values between domains. This is true even for intra-PE domain interactions, so that on a Mayfly system the costs of copying semantics will be absorbed by the MP through the overlap of its execution with that of the EP. The same approach is used to absorb the cost of scheduling.

Achieving fast context switch using general purpose processors is more difficult. Early designs of the Mayfly actually included a second EP; it was intended that the two EP's could alternate roles, one actively processing while the other performed a context switch. This design was considerably more complex, requiring a third instruction cache, a third port to the data cache, switching logic for FP the coprocessor, and inter-EP interrupt logic. The second EP was abandoned in favor of a *context cache*. The context cache (CC) is actually a separate memory module divided into a fixed number of caches. The CC also has a co-processor interface to each of the EP and MP. Through these interfaces, each of the EP and MP can select a particular module which will respond to memory requests lying within a pre-defined context cache address range. Values comprising a task descriptor are assigned addresses in this range. A context switch then reduces to selecting the number of a new context module (presumably from a queue of ready contexts) and making that the current context via the CC coprocessor interface. The CC also implements a cache for the top 128 entries of the control stack.

Adequate memory bandwidth is provided by the combination of caches: separate instruction caches for the EP and MP, the shared data cache, and the CC. In addition to speeding up context switches, the CC serves to diminish demands on data cache and main memory bandwidth, since references to values in the task descriptor and the control stack are serviced by the CC. Furthermore, since contexts need not be saved to/restored from main memory (except in the case of context cache overflow), main memory bandwidth demand is further reduced.

10 Recent Changes

Since the time the workshop was held, Concurrent Scheme has matured. One major area of change has been scheduling and synchronization. The introduction of delay queues was the primary development, with the addition of a version of `touch` that left the domain unoccupied was a pragmatic addition motivated by certain areas of application (notably non-object oriented applications). The other major development lay in coalescing the two major functions of domains. Originally two kinds of domains were specified: temporal and spatial, which, respectively, addressed the issues of mutual exclusion and copying semantics. Experience in developing applications indicated that copying semantics was generally not specified except in the presence of mutual exclusion; hence, the two kinds of domains are presented as one to users. At the implementation level, spatial domains remain a useful device.

11 Conclusion and Future Work

`Make-thread`'s similarity to *future* make it the least interesting mechanism added in creating Concurrent Scheme. The more pervasive change of copying semantics and the fundamental mechanisms of domains and gateways are the contributions that set Concurrent Scheme apart from previous efforts.

Our main parallel constructs, domains, are small, dynamically-created, monitor-like objects, which provide the basic mutual exclusion mechanism. We are currently tuning the existing implementations and preparing to transport the system to a two PE Mayfly in May 1990.

We have handled the problem of data in a distributed system by specifying mutual exclusion between threads and by copying data sent across domain boundaries. Aside from the major impact of copying semantics, the language supported is standard Scheme.

The most problematic part of our current system is that it requires an understanding of closures and an appreciation for the subtle issues of what syntax leads to creation of closures. It is easy for an application programmers to produce a closed procedure

unwittingly by macro expansion (our Scheme does have compile-time macros). Conversely, not all parallelism fits into our model; programmers needing to distribute computation of a non-closed function are forced either to use the generator functions, or to create closed functions based on unused environment's.[9] We are investigating making domains a first-class data type, but the problem remains open.

As noted earlier, the cost of fully-general, structure-preserving copying can be substantial. We are exploring methods to decrease this cost in two ways:

- by providing syntax to specify that structure preservation is not required;
- by providing copiers tailored to types of the arguments and/or result values specific to a particular gateway.

In the longer term, implementation of an object system is envisioned which implicitly uses the CS mechanisms, thereby hiding them from the programmer. Eventually, we plan to create a parallel Utah Common Lisp, but the size of Common Lisp persuades us not to divert our efforts in this direction at the current time.

References

[BH77] H. Baker and C. Hewitt. *The Incremental Garbage Collection of Processes.* AI Memo AIM-454, MIT AI Laboratory, Cambridge MA, December 1977.

[Dav89] A. Davis. *The Mayfly Parallel Processing System.* Technical Report HPL-SAL-89-22, Hewlett-Packard Research Laboratory, March 1989.

[DR85] A. L. Davis and S. V Robison. The Architecture of the FAIM-1 Symbolic Multiprocessing System. In *Proc. IJCAI-85*, pages 32–38, 1985.

[FJSW85] E.J. Feinler, O.J. Jacobsen, M.K. Stahl, and C.A. Ward. *DDN Protocol Handbook, Volume Two, DARPA Internet Protocols.* Sri International, 1985.

[GM84] R.P. Gabriel and J. McCarthy. Queue-based Multi-processing Lisp. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 25–44, August 1984.

[9] This practice is suspect, since compiler technology can sometimes optimize away the unneeded environments.

[HJ85] R.H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions of Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hoa74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Mil87] J. S. Miller. *MultiScheme, A Parallel Processing System Based on MIT Scheme.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, August 1987.

[Nie87] O. M. Nierstrasz. Active Objects in Hybrid. In *Object-Oriented Programming Systems, Languages, and Applications 1987 Conference Proceedings*, pages 243–253, 1987.

[RC86] J. Rees and W. Clinger. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[SRD86] K. Stevens, S. Robison, and A. L. Davis. The Post Office: Communications Support for Distributed Ensemble Architectures. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 160–166, May 1986.