International Journal of Parallel Programming, Vol. 17, No. 5, 1988

Implementing a Scheme-Based Parallel Processing System[1]

James S. Miller[2]

Received August 1988; Revised March 1989

The Scheme language can be converted into a parallel processing language by adding two new data types (**placeholders** and **weak pairs**), two processor synchronization primitives, and a task distribution mechanism. The mechanisms that support task creation, scheduling, and task synchronization are built using these extensions and features already present in the sequential language. Implementing the core of the parallel processing component in Scheme itself provides testbed for a variety of experiments and extensions.

MultiScheme, the system resulting from these extensions, supports Halstead's future construct as the simple model for parallelism. By revealing the underlying placeholders on top of which this construct is built, Multischeme supports a variety of additional parallel programming techniques. It supports speculative computation through a simple procedural interface and the automatic garbage collection of tasks. The qlet and qlambda constructs of the QLisp language are also easily implemented in MultiScheme, as are the more familiar fork and join constructs of imperative programming.

KEY WORDS: MultiScheme; parallel Lisp; implementation; future construct; placeholders.

1. INTRODUCTION

MultiScheme[(1)] is a fully operational parallel-programming system based on the Scheme dialect of Lisp. Like its Lisp ancestors, MultiScheme

[1] This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N00014-83-K-0125, N00014-84-K0099, N00014-86-K-0180, and MDA903-84-C-0033. Additional funds and resources were provided by BBN Advanced Computers Inc., and the Hewlett-Packard Corporation. The work was performed as part of the author's dissertation research at the Massachusetts Institute of Technology.

[2] Brandeis University, Department of Computer Science, Waltham, Massachusetts 02254.

0885-7458/88/1000-0367$06.00/0

828/17/5-1

provides a conducive environment for prototyping and testing new linguistic structures and programming methodologies. MultiScheme supports a diverse community of users who have a wide range of interests in parallel programming. MultiScheme's flexible support for system-based experiments in parallel processing has enabled it to serve as a development vehicle for university and industrial research. At the same time, MultiScheme is sufficiently robust, and supports a sufficiently wide range of parallel-processing applications, that it has become the base for a commercial product, the Butterfly Lisp System produced by BBN Advanced Computers, Inc.

MultiScheme, in the tradition of the Scheme language, is designed as a "minimalist" system. It provides a small but powerful set of constructs from which a researcher can build layers of language suited directly to a particular application. This paper describes the innermost core of the MultiScheme system, the procedures (written in Scheme) that implement the critical operations of the system. Collectively these procedures are referred to as "the scheduler," although they provide a greater range of services than this name implies.

1.1. Placeholders and the Future Construct

From a simple user's point of view, MultiScheme is just a Scheme system with one important addition: the future construct derived from Halstead's Multilisp.[2] This gives the programmer a way to annotate opportunities for parallelism. The special form future can be wrapped around any expression in the language, and indicates that the enclosed expression is permitted to run in parallel with the surrounding expression. MultiScheme requires the use of a specific construct to express opportunities for parallelism because Scheme permits side-effects, and thus a certain amount of control is desirable. Furthermore, the minimalist approach taken by the Scheme community argues in favor of a programmatic interface to parallelism in order to form a convenient base for experimentation in the design of automated tools for inserting parallelism. Initial exploration into building such tools has been undertaken by Gray[3] and Wang.[4]

As a simple example, the doubly recursive calculation of fibonacci can be conveniently described in MultiScheme:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (future (fib (- n 1)))
         (future (fib (- n 2))))))
```

In order to implement the future construct an important change must be made to Scheme—the introduction of a new data type called the **placeholder**. A placeholder is a data structure used to represent a value that has not yet been computed. When the actual value of a placeholder is required (in the predicate of an if expression, or as an argument to a strict primitive operation, for example), the computation waits for the value to be computed. Testing for the availability of a value is known as touching the placeholder. This provides a form of synchronization very similar to that found in dynamic data flow models of computation. In order to allow users to add synchronization points not automatically supplied by the data flow model, the primitive operation touch (a strict identity procedure) is available.

Placeholders, without parallelism or the future construct, are a powerful addition to Scheme. They are the basis for adding objects that behave like Prolog's logic variable, provide for controlled normal order evaluation, and allow the implementation of McCarthy's amb[5] operator, and a fair-merge operation. In addition, by separating the creation of tasks from the creation of placeholders (a possibility novel to MultiScheme), it is possible to implement the constructs of QLisp[6] as simple macros. All of this work is described in Ref. 1.

Given an implementation of placeholders it is straightforward to implement future:

1. Create a placeholder to represent the value of the embedded expression.
2. Create a task to compute the value of the embedded expression and store it in this placeholder. The act of storing a value into a placeholder is called **determining** its value.
3. Return this placeholder as the value of the future expression.

In fact, future in MultiScheme is a macro that expands as follows:

```
(future exp) ⇝ (spawn-task (lambda () exp) parent-gets-priority)
```

The procedure spawn-task is discussed in Section 5.1; its job is to accomplish the three steps previously described. The procedure parent-gets-priority, also discussed later, is the default scheduling policy to be used when new tasks are created—the task that executed the future expression retains control of the processor and the newly created task is scheduled for execution at another time.

1.2. Speculative Computation and Garbage Collection

One area of particular interest in the development of MultiScheme has been the support of **speculative computation**. That is, the ability to spin off in parallel multiple problem solving approaches in the hope that (at least) one of them will succeed. Most uses of parallelism strive to perform faster than the comparable sequential algorithm. By contrast, speculative parallelism attempts to perform a computation as fast as the fastest method that can solve the problem—even where there are multiple techniques to choose from and no fast way to choose among them. By extension, speculative computation also allows the predicate of a conditional expression to be computed in parallel with both the consequent and the alternative expressions. Both Katz[7] and Knight[8] have proposed architectures to support speculative computation in a Lisp with side-effects. Katz has, in fact, used MultiScheme as a base for construction of a simulator of his ParaTran architecture. While his work does not use MultiScheme's parallelism, it uses the underlying placeholder mechanism for recording references to variables and data structures. Katz's work requires the ability to modify the value of a placeholder even after it has been computed. For this reason, MultiScheme marks every placeholder as either having no value, having an immutable value (the usual case), or having a mutable value.

Support for speculative computation is available both directly and indirectly from MultiScheme. As described in Section 5.2, the scheduler for MultiScheme provides a procedure, disjoin, that allows a task to wait for any one of a number of values to be computed. Less directly, MultiScheme's data structures are designed to support the garbage collection of tasks that are computing a value that is no longer needed. The ability to garbage collect processes (proposed by Baker and Hewitt[9]) supports speculative computation by relieving the user of a pair of worries. The user need worry neither about forgetting to destroy a task nor about the consequences of prematurely destroying the task. Just as a user of Pascal must worry about creating dangling pointers, a user of a parallel processing system must worry about killing a process prematurely and creating deadlock. MultiScheme's garbage collector automates the removal of "useless" tasks, eliminating this worry.

The details of garbage collection are beyond the scope of this paper, but parts of the process are directly relevant. The job of the garbage collector is quite simple: it starts with the root set of objects and computes the transitive closure of this set under the data access operations. Any memory not in this set is garbage and is released for reuse. This is a very conservative approach to the definition of "garbage," and assures the programmer

that no accessible object will be released inadvertently. Sometimes, however, there is a need to construct data structures that don't force retension of their contents. In order to support placeholders, for example, each placeholder records the tasks that have suspended execution because they touched it before it had a value. When the placeholder's value is computed these tasks will resume execution. The fact that a task is suspended, however, is no reason to keep the task in existence indefinitely: it may have been created to (speculatively) compute a value that is already known. Thus, the data structure that records the suspended tasks must not cause the garbage collector to keep the tasks.

MultiScheme supports these data structures by supplying a **weak pair** data type[10] containing two parts. Unlike an ordinary pair the car (left half) of a weak pair reverts to a particular value after garbage collection if the object formerly stored there is no longer needed by any other data structures in the system. The implementation of weak pairs is quite simple, since MultiScheme uses a copying garbage collector. The cdrs (right half) of weak pairs are used during the transitive closure computation, but not the cars. An imperfect copy of the weak pair is made and the original storage is used to maintain a list of all weak pairs encountered during garbage collection. A post-pass updates the cars of the weak pairs after the transitive closure has been computed.

The garbage collector also provides one other very important service for MultiScheme. Since it must traverse all of the data structures in the system periodically, it can increase the efficiency of a program by replacing all placeholders by their values once the values are known. This process, called **splicing**, is actually performed in MultiScheme in two different ways: through the garbage collector, and whenever a variable is referenced. That is, every variable reference tests to see if the value of the variable is a placeholder. If so, and if the placeholder has a value, then the reference returns the value rather than the placeholder. The decision to place splicing in variable reference is questionable since it slows down all variable references. In MIT Scheme a test is needed at this point for other reasons, and the same test suffices to detect placeholders. Thus, in MIT Scheme there is no performance penalty for the test and it can improve performance when placeholders are heavily used.

Splicing, whether by the garbage collector or by variable reference, does introduce one problem. Since the scheduler is written in Scheme, any placeholder it manipulates is subject to splicing. This would make the coding of the scheduler quite difficult, so a special exception is made. Every placeholder contains a lock, and splicing only occurs when the placeholder has a value, the value is marked as immutable, and the placeholder isn't locked.

1.3. Assumptions and Extensions

There is a fine line between distributed and parallel computing, and MultiScheme deliberately explores only parallel computation. Thus MultiScheme is concerned with computation using multiple processing elements where performance issues related to the division of labor far outweigh those related to communication cost. As a result of this decision we assume that all of the processors use a single shared heap area, with roughly equal time required by each processor to access all items on the heap. We continue to adopt the Scheme philosophy that objects are, in general, heap allocated and that procedure call is always by transfer of references to objects (i.e. pointers) rather than by copying.

In addition to this assumption, MultiScheme requires four extensions to the existing MIT Scheme language. Each of these is described in detail later in this paper.

1.3.1. Placeholders

The single most extensive change is the addition of placeholder objects. Much of this paper is directly related to this extension, since the placeholder is the central data structure maintained and modified by the scheduler. In addition to the scheduler, however, there is some support required from the underlying machine. This support comes in two forms: detection of placeholders, and inter-processor locks.

The detection of placeholders is described in Section 6 along with the implementation of await-placeholder. Most strict primitive operations (whether compiled, interpreted, or in-line coded) must test for placeholders and either extract the value from the data structure or call the scheduler's await-placeholder procedure. Only primitive operations that deal with the placeholder data structure itself are exempt from this requirement. The same testing applies to the value of any expression appearing as an operator of a combination (procedure call) or the value of a predicate in a conditional expression.

The scheduler assumes the existence of four procedures to lock and unlock placeholders and tasks. Measurements have shown that collisions are extremely rare under normal conditions so we have implemented them as spin locks. Lock-placeholder! either immediately returns #F (i.e. false, if its argument is not a placeholder) or it waits until it is able to acquire the lock that is part of the placeholder data structure (see Figure 1) and returns a value of #T (i.e. true). Lock-task! is similar to lock-placeholder!. In MultiScheme, lock-placeholder! is provided as a primitive because of the complexity of writing the *correct* code to deal with a potential race between one processor setting the value of a placeholder and another processor attempting to lock it.

1.3.2. Task Distribution

The choice of a mechanism to distribute tasks to processors is frequently highly dependent on the particular hardware and communications technology. The scheduler abstracts away from this detail by assuming the existence of three procedures: put-work to release a task for (possibly) parallel execution, get-work to retrieve a task awaiting a free processor, and drain-work-pool to return a list (made from weak pairs) of the tasks currently awaiting free processors. This last operation is not guaranteed to be atomic, but a particular task will either be returned by a call to get-work or drain-work-pool, but not both.

1.3.3. Inter-processor Communication

Section 8 provides the motivation and description of MultiScheme's three procedures for coordinating the work of the processors within the system. MultiScheme users are encouraged to think in terms of logical processes (tasks) and leave the control of the processors to the system. These three procedures form the support required by the system: global-interrupt provides a means for initiating an interrupt sequence on all other processors; and the pair make-synchronizer and await-synchrony allow all the processors to perform a barrier synchronization (i.e. all processors must call await-synchrony with the same synchronizer object before any processor is allowed to proceed from its call to await-synchrony).

In addition to these extensions, MIT Scheme[10] itself contains a large number of extensions to the Scheme language.[11] Most of these extensions are not needed to implement the procedures described here. The code in this paper depends only on weak pairs (described earlier) and the following items:

(within-continuation continuation thunk)

This procedure restores the state of the machine from the continuation and then executes the thunk. If the thunk returns, its value is passed to the continuation. This provides a mechanism for changing the control state of the processor just prior to executing a piece of code—in implementation terms, it releases the current stack and restores the stack stored in continuation before calling thunk.

the-error-continuation

This is a continuation made when MIT Scheme is first started. It has a minimal control state, and can be used in conjunction with within-continuation to return to a control state that will retain a minimum of information after garbage collection.

1.3.4. Primitive Continuation Handlers

MIT Scheme has a number of primitive continuations that ordinarily cuase errors to occur. Users can supply procedures to be called in place of the default handlers for these continuations. In the case of MultiScheme, the primitive continuation used when a task completes its work is replaced by one that determines the corresponding placeholder and terminates the task.

2. OVERVIEW OF THE SCHEDULER

The MultiScheme scheduler provides a convenient interface, in the form of a package of procedures, between MultiScheme programs and the underlying machine. Some of the procedures are invoked by programs written in MultiScheme while others are invoked as part of the trap or interrupt handling of the machine. The scheduler is itself written in MultiScheme and is relatively small: 20 pages of code including utility routines. This has proven to be an important factor in the development of MultiScheme, providing a localized and flexible base for a number of experiments with the nature of parallel computing.

This paper discusses each of the major operations supported by the scheduler: task creation (Section 5), task suspension and task switch (Section 6), storing a value into a placeholder (Section 7), and transition from parallel processing to single task execution (Section 8.2). The rough outline of the scheduler—the services it supports and the interrelationship between these services—has proven quite robust over time. Even as the system grew to support more parallel programming styles, the core of the scheduler as described here has remained almost constant. The scheduler was originally intended to be, and remains, a highly flexible body of code. The scheduler described here is the "standard" scheduler as it currently exists. As new applications are developed, driving the system toward new modes of computation, the data structures of the scheduler are modified to accommodate the new requirements. Users are encouraged to examine and understand the scheduler, and feel free to modify it for their own needs. Naturally, such modifications must be undertaken with a good deal of care. But these modifications have proven useful in the past and have in some cases been formalized and added to the standard MultiScheme scheduler.

The presentation is roughly bottom up, describing the data structures in Section 3, general utility routines in Section 4, and then the user-visible routines. In order to avoid an overwhelming amount of detail the examples are simplified versions of the actual procedures in the scheduler. They present the important core of each procedure, and should be considered more closely related to pseudo-code than to fully worked out implementations.

In many cases the versions presented here will not work correctly in the actual implementation of MultiScheme. This comes primarily from two reasons: race conditions and name changes. The race conditions that exist here are easily resolved using standard programming techniques. The detail required to include the solutions merely obscures the core ideas of the scheduler. Readers interested in the complete versions of these procedures should contact the author for a current version of the scheduler code.

3. SCHEDULER DATA STRUCTURES

Much of the scheduler procedures' work revolves around the correct maintenance of the data structures that implement placeholders, tasks, and a pool of tasks that are ready to run. For this description, all access to data structures is assumed to be through mutators and selectors for each part of the structure. Thus, corresponding to the goal slot of a task data structure there are two procedures: task.goal returns the goal of a given task and set-task.goal! stores a new goal into the task data structure.

The design of these data structures is often motivated by a desire to garbage collect tasks[9] that are no longer computing useful values. These tasks arise largely from the use of speculative computation techniques. For a description of these techniques and the difficulties involved in garbage collecting tasks, see Refs. 1 and 12.

3.1. Placeholders

Placeholders are the primary vehicle connecting the scheduler, and hence programs written in MultiScheme, with the underlying support for parallel processing. Placeholders are created by a scheduler procedure, normally as part of the task creation process (see Section 5). Supplying a value for a placeholder (called **determining** the placeholder) is also supported by a scheduler procedure (determine!, see Section 7). Detection of placeholders and automatically forcing them is built into the primitive operations and the underlying machine itself, as described in Section 6.

When programmers rely on the future model of computation, none of these procedures are invoked directly by user code. Instead, they are called by code from the expansion of the future macro. As new programming styles have developed, however, each of these procedures has proven useful for implementing the support required for the new style.

The placeholder data structure is shown in Fig. 1. Each of the fields is described later.

Slot Name	Notes

Placeholder

Determined?	Yes, No, or Mutable
Lock	For primitive lock operations
Value or Waiting Set	*See text*
Motivated Task	Task computing this value

Task

Goal	Placeholder associated with this task
Lock	For primitive lock operations
Code	Work to be performed when task is next run
Status	*See text for details*
Original Code	For debugging purposes
Task-Private Data	*See text for details*
Waiting For	Placeholder(s) for which this task is waiting
Wake-up Value	*See text for details*

See text for complete description

Fig. 1. Task and Placeholder Data Structures.

3.1.1. Determined?

A tri-state flag that indicates whether the placeholder: (a) has no value yet; (b) has an immutable value; or (c) has a mutable value. This flag is used by the inderlying machine to test whether touching this placeholder should trap into the scheduler as discussed in Section 6 or extract the current value and continue.

3.1.2. Lock

A standard mutual exclusion lock used to indicate that the placeholder is currently being modified by MultiScheme code. This slot is used by the procedures lock-place-holder! and unlock-placeholder! described in the introduction.

3.1.3. Value

Stores the value of the placeholder when it is known. This slot is mutable, but the scheduler enforces a protocol that allows the slot to be treated either as a write-once location or a multiple writeable location. The future construct produces placeholders that receive a value exactly once.

3.1.4. Waiting set

A set of tasks currently waiting for this placeholder's value to be determined. This set is built using weak pairs since membership in this set does not constitute a reason for the task to continue computing.

3.1.5. Motivated task

The task that has the computation of a value for this placeholder as its goal.[12] As with any item other than a weak pair, the garbage collector *does* trace through this link. Thus it serves to retain the task that is computing the value of this placeholder as long as the placeholder itself is needed.

3.2. Tasks

The task data structure contains a variety of information, but is not directly referenced by the underlying machine. Tasks represent work that has been requested to be performed, and they are the objects that the scheduler has the underlying machine store in its work distribution pool.

In order to support garbage collection of no longer useful tasks, the root used by the garbage collection algorithm contains a particular set of tasks whose continued existence is required by the user interface to MultiScheme.[12] Other tasks are retained only if they can be reached either from this initial set of tasks or from the global environment. One task can be "reached from" another if the former task is waiting for the value of a placeholder and the latter task is the one that is responsible for calculating the placeholder's value. The **waiting for** and **motivated task** slots are responsible for recording this relationship between tasks.

The task data structure is shown in Fig. 1.

In addition to the slots shown in Fig. 1, it has long been expected that some information might be stored here for use by user applied scheduling policies. This information could indicate task priority or estimated time required for the value to be computed. Our on-going research includes implementing and studying these extensions.

3.2.1. Goal

The placeholder that is the goal for this task. When a task is actively computing, this placeholder is known as the **current placeholder** for the processor doing the computation. When a task executes the termination continuation (see Section 5.3) it stores the computed value into this placeholder.

3.2.2. Lock

A standard lock to serialize access to the task description. Used by the procedures lock-task! and unlock-task! mentioned in the introduction.

3.2.3. Code

The code to run in order to reactivate this task. If the object stored here is not applicable (i.e. neither a procedure nor a continuation) then either the task is already active or for some reason it cannot be reactivated. For example, the task may have finished computing but not yet been garbage collected.

3.2.4. Status

The current state of the task. This can be the following:

created	Task is newly created
delayed	See delay-policy, Section 5.2
determined	Task is finished
disjoin	Waiting for the first of several placeholders
paused	See Pause-everything, Fig. 11
runnable	Available for execution
running	Actually in possession of a processor
waiting	Waiting for a specific placeholder
within-task	Running, but another task has requested this task to execute a block of code. See within-task, Fig. 12.

3.2.5. Original code

For debugging purposes this contains the expression that the task was created to evaluate.

3.2.6. Task-private data

Used to implement **fluid variables**, a form of per-task data storage.[1, 10] This slot holds an association list mapping variables to values. It is used when a variable marked as **fluid** is referenced, using a runtime trap mechanism.

3.2.7. Waiting for

If the task has status **waiting** or **disjoin** this specifies the placeholder(s) for which it is waiting. This is an ordinary list, since the value of these placeholders is necessary for this task to continue its own computation. Hence this task represents a reason for the tasks that are computing the values of these placeholders to continue their computation.

3.2.8. Wake-up value

When the task is awakened, the **code** is passed this value as its argument. It is primarily used in the implementation of disjoin, the mechanism that supports speculative computation (see Sections 5.2 and 6).

3.3. Runnable Task Pool

The subject of task distribution is one on which a great deal of work has been done, both theoretical and practical. The choice of a mechanism, while important, is often intimately tied to details of the hardware or communications medium. Since the main thrust of the MultiScheme work is not related to these issues, we chose to isolate the scheduler from these issues by encapsulating the choice in the three primitive operations described in the introduction. Because the shared heap contains all of the task, placeholder, stack, and other structures needed for a computation, there is no requirement that a task be run on the same processor that released it with put-work. While providing this behavior may improve performance on some architectures, it is up to the primitives to either support this behavior or not.

The scheduler's use of these procedures is therefore simple. It announces that a task needs processing resources using put-work. When a processor needs more work to do, the scheduler will retrieve a task using get-work. When the system must retract work that has been declared to be available (such as during garbage collection initiation, see Section 8.2), it calls drain-work-pool. Synchronization and serialization are the responsilibity of the MultiScheme code, not the procedures themselves. Thus, it is possible for one task to drain the work pool while other tasks are still active and adding new entries to the pool. The results will be consistent although it may not represent an instantaneous snapshot of the internal data structures.

4. OVERALL CONCEPTS AND UTILITY ROUTINES

The scheduler is organized around the data structures described in Section 3 and two additional notions. The first, described in Section 4.1, is atomicity and critical sections of code. These are supported through a system of priority interrupts within a single processor and a set of data object locks between processors. The second is the task state and task switch operations, described in Section 4.2, supported through the use of Scheme's continuations.

4.1. Atomicity

As with any operating system scheduler, most of the routines in the scheduler must appear to occur without interruption. The fact that these routines are written in Scheme, however, does not permit them to run completely uninterrupted: the garbage collector cannot be suppressed for long

intervals without serious consequences. As a result, most of the operations are written to raise their own interrupt level to prohibit any kind of interrupt *except* garbage collection, and the garbage collection code guarantees that any task that is running at a raised interrupt level will continue to run after the garbage collection. This notion is embedded in the macros atomic and define-atomic that are used liberally throughout the scheduler implementation. To make the code more easily understood, however, these have been omitted from the simplified versions described here.

A second standard problem, exclusive access to certain data structures, also exists in the scheduler. The scheduler is built using two utility routines that are in turn based on the interlock routines described earlier. The operation lock-placeholder! is used to implement the more complicated of the two utility routines:

```
(define (With-Placeholder-Locked Placeholder Procedure)
  (atomic
   (if (lock-placeholder! Placeholder)
       (let ((result (Procedure #T)))
         (unlock-placeholder! Placeholder)
         result)
       (Procedure #F))))
```

As can be seen, with-placeholder-locked runs a procedure with a given placeholder locked. The procedure receives an argument that indicates whether the object is in fact a placeholder. This handles an important race condition: another processor might have supplied a value for the placeholder before we were able to lock it. The placeholder might then be spliced out by variable reference or garbage collection before we can acquire the lock. Once locked, however, a placeholder is no longer subject to splicing. Since MultiScheme does not provide any standard way of locking arbitrary objects, the object is only locked if it is indeed a placeholder.

A similar utility routine, With-Task-Locked is also supplied. Unlike With-Placeholder-Locked, the task *is* always locked when the procedure runs since the race condition that exists for placeholders is not a problem with tasks.

4.2. Task Switch

When a processor changes tasks it is really performing three separate operations. The first operation captures the current state of the task in a

way that allows it to be restarted later. The second chooses a new task for execution, and the third activates a chosen task. Task termination (as described in Section 5.3) is nothing more than performing the last two steps but not the first. Task creation (see Section 5.2) may include the first and third steps with a standard choice for the second.

The first operation, capturing the current state of a task, is done using Scheme's standard call-with-current-continuation procedure. Once a task is suspended it will be resumed only once, and then that state will later be suspended and so forth. Thus, unlike an ordinary continuation object, the object that denotes a suspended task state need not be able to be invoked multiple times. In implementation terms, this means that a certain amount of copying of continuation stack entries can be avoid with task suspensions. At one time a special variant of call-with-current-continuation that took advantage of this optimization was implemented. Performance measurements indicated that the overall effect was minimal, but these measurements were taken based on the MultiScheme interpreter and may not be indicative of the performance of a compiled system.

While acquiring a representation of the current state of the computation is simple, actually storing it in the task data structure is not as straightforward. In Section 8.3 an important user operation, within-task, will be introduced (the relevant code is shown in Fig. 12). Calling within-task allows a user to modify the operation of a task that is already running. It marks the task data structure to indicate the work that must be performed, and it is the responsibility of the task when it next saves its state away to arrange to perform that work when the task is subsequently activated.

The routine store-my-state, shown in Fig. 2 is provided to support this operation. It allows scheduler routines to specify the state to be used when the task normally regains control (the argument state), and additional work to be performed on the task data structure while it is locked (while-locked). It locks the current task data structure and then stores either the specified state or a procedure that first executes the work specified by a call to within-task as the work to be performed when the task is next activated. Notice that store-my-state tests whether the task needs to continue running; if not, the procedure while-locked is *not* executed, nor is the state of the task actually saved.

One other common way of capturing the state of a task is provided by the procedure release-task, shown in Fig. 3. In this case, the intention is to release the current task for parallel execution and then execute some other code while the processor is temporarily not performing any task. The reason for providing it with the thunk to be executed may not be obvious, but notice that the thunk is executed as part of the procedure called by

```
(define (store-my-state state while-locked)
  (let ((my-task (current-task)))
    (with-task-locked my-task
      (lambda (am-I-runnable?)                ; (1)
        (if am-I-runnable?
            (begin
              (set-task.code! my-task
                (if (eq? (task.status my-task) 'WITHIN-TASK)
                    (let ((within-task-code (task.code my-task)))
                      (lambda (wake-up)       ; (2)
                        (within-task-code wake-up)
                        (state wake-up)))
                    state))                   ; (3)
              (while-locked my-task)))))))    ; (4)
```

Notes:

1. Find and lock the current task data structure.
2. If the task is expected to continue running but has been marked for special handling by `within-task` (see Figure 12), then when the task next awakens it must first execute the code specified in the call to `within-task` and then continue on to its ordinary computation.
3. Under ordinary circumstances, the state to be stored is just the state specified by the caller.
4. If the task will continue to run, call the user-specified procedure while the task data structure is still locked.

Fig. 2. Saving state for future execution: Store-My-State.

call-with-current-continuation. Thus it is executed by the calling processor when release-task is called, but *not* when the task is resumed. Resuming the task occurs by calling the continuation my-state, thus effectively returning from this call to call-with-current-continuation. The processor's current task is set to 'STATE-SAVED both as a debugging aid and to reflect the fact that the processor is not currently executing a task. This is important if the processor subsequently needs to save its state since there is no task into which the state can be stored.

```
(define (release-task thunk)
  (call-with-current-continuation
   (lambda (my-state)
     (store-my-state my-state
       (lambda (my-task)
         (set-current-task! 'STATE-SAVED)
         (set-task.status! my-task 'RUNNABLE)
         (put-work my-task)))
     (thunk))))
```

Fig. 3. Relinquishing the processor: Release-Task.

The second operation, choosing a task to perform, is most often deferred to the underlying machine, using the primitive get-work to select the task:

```
(define (next)
  (Set-Current-Task! 'WAITING-FOR-WORK)
  (run (get-work)))    ; See run in Figure 4
```

The third operation, activating a chosen task, is the most complicated. This job is handled by the procedure run, shown in Fig. 4. It consists mostly of routine housekeeping activities. The task being activated is first locked and tested to see if it is actually runnable. If so, the status is changed to **running** and the code and wake-up value are extracted from the task data structure. The task is then unlocked, and either the code is activated with the appropriate wake-up value as its argument or if the task turned out not to be runnable an alternative task is chosen using next.

4.3. Other Utility Routines

There are a handful of other utility routines that are referenced later.

- A task data structure and its related placeholder can be created using (Make-Task), which makes the pair simultaneously and

```
(define (run task)
  (define what-to-actually-do
    (With-Task-Locked task
     (lambda (Still-Runnable?)
       (if Still-Runnable?                ; (1)
           (let ((code-for-new-task (task.code task))
                 (wake-up-value (task.wake-up-value task)))
             (set-task.status! task 'RUNNING)
             (set-task.wake-up-value! task '())
             (Set-Current-Task! task)
             (lambda ()                   ; (2)
               (code-for-new-task wakeup-value)))
           next))))                       ; (3)
  (what-to-actually-do))                  ; (4)
```

Notes:

1. Test the task to see if it is actually runnable.
2. If the task is runnable, this procedure will restart it.
3. If the task is not runnable, this procedure will select an alternate task and start it instead.
4. Actually call the procedure chosen in either step 2 or 3 above.

Fig. 4. Activating a chosen task: the run procedure.

```
(define (immutable? placeholder)
  (eq? (placeholder.determined? placeholder) #T))

(define (undetermined? placeholder)
  (eq? (placeholder.determined? placeholder) #F))

(define (determined? placeholder)
  (not (undetermined? placeholder)))

(define (mutable? placeholder)
  (and (determined? placeholder)
       (not (immutable? placeholder))))
```

Fig. 5. Tri-state Flag Representation.

supplies a standard set of default values for all of the information required.

- The three possible states of the **determined?** slot of a placeholder are: #**T** indicating that the placeholder has an immutable value; #**F** indicating that it has no value at all; and anything else indicates that the value is mutable. This is captured in the four procedures shown in Fig. 5. Notice that the placeholder must be locked in order to safely perform these operations.
- A task that has been waiting is activated using the procedure activate shown in Fig. 6. This procedure tests whether the task is still runnable and is in fact waiting for the condition that has occurred (as indicated by the test). It then updates the task data structure and releases it for distribution using the underlying machine operation put-work.
- The procedure Saving-State, shown in Fig. 7, allows a task to save its state away and then execute a selected piece of code, thunk. The code is run in a continuation that is part of the root of garbage collection and not as part of the task that called saving-state. This permits the garbage collector to reclaim the originating task if necessary. When the code finishes execution,

```
(define (activate task test wake-up-value)
  (With-Task-Locked task (lambda (task-runnable?)
    (if (and task-runnable? (test (task.status task)))
        (begin
          (set-task.waiting-for! task '())
          (set-task.status! task 'RUNNABLE)
          (set-task.wake-up-value! task wake-up-value)
          (put-work task))))))
```

Fig. 6. Activating a waiting task.

```
(define (saving-state thunk)
  (release-task (lambda ()                ; (1)
    (within-continuation                  ; (2)
     the-error-continuation
      (lambda () (thunk) (next))))))
```

Notes:

1. Release the current task for potential parallel execution.
2. Begin execution within `the-error-continuation` which was created at system boot time, and does not reference any other continuations.

Fig. 7. Code for Saving-State.

another task is selected for execution rather than returning to the original task. A good way of thinking about saving-state is that it performs a task switch into a non-existent task and executes the thunk in the new task.

- Two procedures, Current-Task and Set-Current-Task!, are provided to keep track of the task that is currently executing on this processor. These can either be implemented as primitive operations that access processor-private data or in Scheme using a primitive procedure that identifies the processor on which the task is currently running.
- Weak-list → list converts a list composed of weak pairs into one composed of ordinary pairs.
- Add-to-waiting-set! adds a task to the set of tasks waiting for the value of a particular placeholder. We have chosen a trivial implementation of sets since nothing depends on removal of duplicates:

```
(define (add-to-waiting-set! placeholder task)
  (set-placeholder.waiting-set! placeholder
    (cons task (placeholder.waiting-set placeholder))))
```

5. TASK CREATION AND TERMINATION

Creating a task in MultiScheme really has four steps: create a continuation, create a task data structure, create a placeholder, and schedule the running and newly created tasks for parallel execution. Each of these can be performed independently and then combined to provide specialized handling of unusual cases. Tasks are normally created, however, by using the future macro.

This macro has one required argument, the **expression** to be executed in parallel, and an optional **policy** used to schedule the parent and child tasks. Thus

```
(+ (future e1 policy) (future e2)) ⇝
(+ (spawn-task (lambda () e1) policy)
   (spawn-task (lambda () e2) parent-gets-priority))
```

The remainder of this section consists of a description of the Spawn-Task procedure (Section 5.1), alternatives to this standard method of task creation (Section 5.2), and finally the handling of task termination (Section 5.3).

5.1. Ordinary Task Creation

As described earlier, most of the work of creating a task is ordinarily carried out by Spawn-Task. This is merely a standard way of using the four steps previously mentioned.

```
(define (spawn-task code policy)
  (let ((the-new-task (make-task)))
    (let ((result (task.goal the-new-task)))
      (set-task.code! the-new-task
        (lambda (wake-up-argument)
          (new-task-continuation code)))
      (policy the-new-task)
      result)))
```

Part of the task data structure is the **code** it is to execute, and spawn-task uses a specially constructed new-task-continuation for this purpose (see the discussion below). The continuation expects to receive a procedure as argument (code in this case); it calls the argument and then calls a primitive continuation indicating the end of task with the result. The handling of this task termination continuation is described in Section 5.3.

Spawn-Task calls the user-supplied policy routine to schedule the current task and the newly created task. The default routine, parent-gets-priority (shown in Section 5.2), releases the new task for potentially parallel execution. The value returned by spawn-task to the task that called it is the newly created placeholder.

The decision to use a continuation rather than a procedure for the initial code of a task is not completely arbitrary. If a procedure is used the

task switch code in run (see Fig. 4) that activates the newly created task would be nothing more than a procedure call. But procedure call includes passing an implicit continuation for use when a value is returned, and this continuation will in some way reference the task that made the procedure call. This prevents the garbage collector from reclaiming that task as long as the newly created task is in existence.

In implementation terms, which may be easier to understand, a continuation is just a saved procedure call stack. If the initial code for a task were simply a procedure then the stack used for the new task when it first runs would be the same as the stack of the task that was relinquishing the processor. This works perfectly well but leads to a form of cactus stack implementation that has the garbage collection problem previously mentioned.

By explicitly building a continuation, however, task switch becomes the same as invoking a continuation that does *not* implicitly reference the old task. The continuation created when the task is created is an initial stack frame and task switch (i.e. invoking a continuation instead of an ordinary procedure) causes the stack to be switched as well.

5.2. Alternative Ways to Create a Task

The task creation code is modularized into the four steps described earlier. Actually, utilizing these individual components is unusual since the flexibility available using the policy argument to Spawn-Task is sufficient for most problems. To make this power easily available, two alternative policies are included in the scheduler package along with the default policy.

The standard policy, parent-gets-priority is very efficient:

```
(define (parent-gets-priority new-task)
  (put-work new-task)
  'CHILD-QUEUED-FOR-EXECUTION)
```

This policy gives processing priority to the parent task: the task that calls Spawn-Task continues to run, while the task which is created is scheduled for parallel execution. Since the task and its associated placeholder have been made and initialized by Spawn-Task, all that must be done is to make the new task available for computation. This is done using the underlying task distribution mechanism, implemented by put-work. In this, as in the other policies, the value returned by the policy is ignored by Spawn-Task but is useful in debugging the scheduler itself.

Halstead argues, in his overview of Multilisp,[2] that this standard

policy can lead to undesired performance characteristics as a system reaches saturation. He suggests a strategy in which the parent task is deferred while the child task immediately begins execution. This is implemented using the child-gets-priority policy:

```
(define (child-gets-priority new-task)
  (release-task (lambda () (run new-task))))
```

The third policy, delay-policy, marks the spawned task as **delayed** and does *not* release it for parallel execution:

```
(define (delay-policy new-task)
  (set-task.status! new-task 'DELAYED)
  'OK-I-DELAYED-IT)
```

Instead, the first task that touches the **goal** placeholder associated with the newly created task will release that task for execution (see Section 6).

In addition to these three policies, there is one other case that occurs sufficiently often to be provided standardized support. This is the ability to wait for the first of a number of placeholders to return a value. This ability, implemented by the procedures disjoin and await-first of the scheduler, is the key to implementing McCarthy's amb[5] and fair-merge procedures. The actual code for these procedures is complicated because it must deal with the possibility that one of the placeholders has already received a value before the operation has been completed, and because more than one of the placeholders may eventually receive a value. Figure 8 shows a much simpler version that does not deal with these problems; the footnotes to the figure explain the most impotant omissions. Notice that disjoin itself returns a placeholder rather than actually waiting for the value to be known.

This simplified version works by creating a task and its corresponding placeholder using Make-Task. The purpose of this new task is to propagate the value of the appropriate placeholder (the first one that receives a value) out to its own goal. The **code** to be performed when this new task is awakened is supplied as an explicit procedure. This is very similar to the processing of the normal case, except that the task has a list of placeholders for which it is waiting and it is added to the set of tasks waiting for each of these placeholders. The decision to represent the **code** as a procedure rather than a continuation here is somewhat arbitrary. It is easier to write as shown and the procedure will relinquish the processor almost instantly so that the garbage collection problem mentioned earlier is not an issue.

```
(define (disjoin . Placeholders)          ; (1)
  (let ((My-Task (Make-Task)))
    (let ((My-Placeholder (task.goal My-Task)))
      (set-task.status! My-Task 'DISJOIN)
      (set-task.waiting-for! My-Task Placeholders)
      (set-task.code! My-Task
        (lambda (awakened-value)          ; (2)
          (determine! My-Placeholder awakened-value)
          (next)))
      (for-each                           ; (3)
       (lambda (Placeholder)
         (add-to-waiting-set! Placeholder My-Task))
       Placeholders)
      My-Placeholder)))                   ; (4)
```

Notes:

1. As explained in the text, this code does not deal with a number of important possibilities.
2. Code to be run when this newly created task is activated (i.e. when one of the placeholder receives a value). This is one of two major race conditions that the complete version handles. If more than one task completes, `My-Placeholder` may already have a value when this code is run.
3. Add this task to the waiting-set of each of the placeholders. This is the second of the major race conditions. In the process of adding the task it may be discovered that one of the placeholders already has a value which must then be returned instantly.
4. The value returned by `disjoin` is the placeholder that will ultimately receive the value of the first computed placeholder.

Fig. 8. Simplified Code for disjoin.

When any of the placeholders for which this task is waiting receives a value (see Section 7), that placeholder is stored in this task's **wake-up value** slot and the task is made available for execution. When the task is activated, using the run procedure described in Section 4.2, the procedure stored in the **code** slot will be passed this wake-up value. The procedure will propagate it to the placeholder created by the call to disjoin, and then call next (see Section 4.2) to release the processor and find another task.

The use of a task to propagate the value of the appropriate disjunct may seem unusual, but it improves the modularity of the scheduler code. This work could have been made part of the determine! code, but this organization allows determine! to simply awaken tasks in a standard manner. Determine! is never required to do any specialized processing on behalf of the tasks it awakens.

5.3. Task Termination

As mentioned in Section 5.1, there is a primitive continuation that denotes the termination of a task. As with any continuation, it receives a

value; this continuation treats the value as the value for the **goal** placeholder of the task. To make modifications to the system simpler, the handling of this and many other primitive continuations is reflected back into the scheduler as a call to a procedure. The standard procedure is quite simple since it runs as part of the task that is terminating:

```
(define (end-of-computation-handler value)
  (determine! (task.goal (current-task)) value)
  (next))
```

This merely stores the final value into the **goal** and then locates and activates the next available task. Notice that by simply calling next without saving its own state, this task relinquishes the processor and will not be reactivated.

6. SUSPENDING A TASK

There are three ways in which a task can relinquish the processor. It can explicitly relinquish the processor using reschedule, a procedure supplied by the scheduler for this purpose. An interrupt can occur and cause the processor to be relinquished (e.g., the initiation of a garbage collection or a clock interrupt). Finally, and most commonly, the task can attempt to **touch** a placeholder that does not yet have a value.

With the utility procedures described earlier it should be easy to see how the first operation is performed:

```
(define (reschedule)
  (release-task next))
```

Garbage collection initiation will be described in Section 8.1. Timer interrupts are handled by calling reschedule as part of the interrupt handler.

The remainder of this section is devoted to the third problem, touching a placeholder. When placeholders were introduced it was stated that they "can be used to denote an object whose value is not yet known," and that the scheduler is responsible for handling an attempt to **touch** a placeholder which does not yet have a value. This mechanism has two parts, one implemented in the machine underlying the MultiScheme system, and the other as part of the scheduler.

The underlying machine is responsible for both detecting and handling the simple cases related to placeholder objects. There are three different ways in which a placeholder can be initially noticed:

1. The code that implements certain primitive operations (e.g. touch, eq?, and memq) explicitly touches objects that they manipulate. If

the object is not a placeholder, or the placeholder has a value, the code will retrieve the correct value and the operation proceeds unimpeded. The operations are all carefully written, however, so that if a placeholder is encountered that does *not* have a value the operation can be stopped and restarted at a later time. The operation gracefully backs out and returns the state of the system to what it was before the operation began. It then performs a call to the scheduler's await-placeholder operation (see Fig. 9). The result is as though the user had written a call to touch of the appropriate placeholder immediately prior to the call to the operation.

2. Many primitive operations normally type-check their arguments for validity before doing any processing. For those operations that do not permit an operand to be a placeholder (e.g., arithmetic operations restricted to numeric data types), the normal error handling mechanism of MIT Scheme would cause the primitive to gracefully back out just as in the previous case and then invoke an error handling procedure. In MultiScheme, the code for these error handlers tests for placeholders and restarts the primitive automatically (i.e. without any form of trap into Scheme code) if the erroneous argument is a placeholder that has a value. If the

```
(define (await-placeholder placeholder)
  (call-with-current-continuation
   (lambda (me)                                  ; (1)
    (With-Placeholder-Locked placeholder
      (lambda (waiting-for-a-placeholder?)
        (cond ((not waiting-for-a-placeholder?)
               (me 'RESUME-COMPUTATION))         ; (2)
              ((determined? placeholder)
               (unlock-placeholder! placeholder)
               (me 'RESUME-COMPUTATION)))        ; (3)
        (activate                                ; (4)
         (placeholder.motivated-task placeholder)
         (lambda (status)
           (or (eq? status 'DELAYED) (eq? status 'PAUSED)))
         'NO-RELEVANT-WAKE-UP-VALUE)
        (store-my-state me (lambda (My-Task) ; (5)
          (set-task.status! My-Task 'WAITING)
          (set-task.waiting-for! My-Task placeholder)
          (add-to-waiting-set! placeholder My-Task)))))
    (next))))                                    ; (6)
```

See the text of Section 6 for footnotes.

Fig. 9. Code for Await-Placeholder.

argument is a placeholder that does not yet have a value then instead of invoking one of the Scheme error handling procedures it invokes the scheduler's await-placeholder procedure.

3. Compiled code contains calls to the primitive operation touch whenever it must ensure that an object (argument to an in-line coded primitive, predicate of a conditional, or function to be applied) is not a placeholder and cannot, at compile time, deduce this. Touch, which is one of the operations described in case 1, merely tests its operand to see if it is a placeholder. If it is not, the operand is returned. If it is a placeholder with a value, that value is returned. The net result is that an attempt to use a placeholder whose value is already known will proceed unimpeded, but one whose value is still undetermined will cause a call to the await-placeholder procedure. The exact placement of these calls to touch is a topic for further investigation. Placing them earlier in the code can frequently make the code more efficient but it reduces the potential for parallelism.

In each case, the underlying machine handles placeholders that have already received a value but calls the await-placeholder procedure to handle placeholders that do not have a value. The job of await-placeholder, then, is to save the state of the current task if it needs to continue running and add this task to the **waiting-set** of the placeholder. It then releases the processor by calling next.

The code for await-placeholder is shown in Fig. 9. The following description of its operation is keyed to the numbers in the figure.

1. Create a continuation, me, that holds the state of the current task. Attempt to lock the placeholder for which the task is waiting.
2. If the placeholder couldn't be locked, just resume the current task. This can occur if the placeholder has received an immutable value prior to reaching this point in the code. The placeholder would be subject to the splicing operation during variable reference or garbage collection. After the lock is acquired this splicing will no longer occur.
3. If the placeholder has a value, unlock the placeholder and resume the current task. This can occur if the placeholder has received a mutable value before reaching this point in the code. In this case, the placeholder is not subject to splicing so it will have been locked, but there is no need to await the arrival of a value.
4. Activate the task that is calculating the value for the placeholder if it is inactive. This arises either because the placeholder was

created by the delay-policy and hence the task has status **delayed**, or because the task has been suspended by pause-everything (see Section 8.2) and has status **paused**.

5. At this point, the task will definitely be releasing control of the processor. The state of the task, me from step 1, is saved away in the current task data structure. Mark the task as **waiting** and add it to the **waiting-set** of this placeholder.
6. Unlock the placeholder (by exiting the with-placeholder-locked procedure). Find another task and start executing it. Notice that this call to next is *not* executed when control returns to the original task using the me continuation created in step 1 since it is part of the body of (lambda (me)...) (count the parentheses...).

7. STORING THE VALUE OF A PLACEHOLDER

Storing a value into a placeholder is a straightforward operation, although the details are somewhat complicated. The essential work is to store the value into the placeholder data structure and activate any tasks that may have been waiting for this value to appear. The detailed code is shown in Fig. 10. The following notes describe the fine details of its operation. They are geared to the numbers appearing in Fig. 10. Most of the complexity comes from the need to keep the placeholder locked for as short a time period as possible, and the possibility of a race if two tasks attempt to supply values to the placeholder nearly simultaneously.

1. The auxiliary procedure update-placeholder! makes the changes necessary to the placeholder data structure to reflect the fact that it now has a value.
2. What-to-do will contain one of three procedures to be performed after the placeholder is unlocked in step 7. The three procedures are (a) an error procedure from step 3; (b) awaken the waiting tasks from step 3; and (c) do nothing from step 6.
3. With the placeholder locked, test whether it already has an immutable value. If so, return a procedure that will cause an error in step 7.
4. If the placeholder did not previously have a value, remember the tasks that are waiting for the value of this placeholder and then update the placeholder.
5. Since the placeholder didn't have a value before, in step 7 we must activate each task that is waiting for this placeholder. The acti-

```
(define (determine! placeholder value allow-mutations?)
  (define (update-placeholder!)              ; (1)
    (set-task.status! (placeholder.motivated-task placeholder)
                      'DETERMINED)
    (set-placeholder.value! placeholder value)
    (set-placeholder.determined?! placeholder
      (if allow-mutations? 'MUTABLE #T)))
  (define what-to-do                         ; (2)
    (With-Placeholder-Locked placeholder
      (lambda (still-a-placeholder?)
        (cond ((or (not still-a-placeholder?)
                   (immutable? placeholder))
               (lambda ()                    ; (3)
                 (error "Immutable Placeholder" placeholder)))
              ((undetermined? placeholder)
               (let ((waiters
                      (placeholder.waiting-set placeholder)))
                 (update-placeholder!)       ; (4)
                 (lambda ()                  ; (5)
                   (for-each
                    (lambda (task)
                      (activate task
                                (lambda (status)
                                  (or (eq? status 'WAITING)
                                      (eq? status 'DISJOIN)))
                                placeholder))
                    waiters))))
              (else (update-placeholder!) ; (6)
                    (lambda () 'OK))))))
  (what-to-do)                               ; (7)
  value)                                     ; (8)
```

See the text of Section 7 for footnotes.

Fig. 10. Code for determine!

vation test permits only tasks that are waiting for a placeholder (i.e., those with status **waiting** or **disjoin**) to be awakened.

6. If the placeholder previously has a mutable value, then it can't have a set of tasks waiting for its value to appear. Thus, in step 7 we don't need to take any special action.
7. Now that the placeholder is unlocked, perform whatever work is necessary.
8. The value returned by determine! is (arbitrarily) the value that has been given to the placeholder.

8. PROCESSOR COORDINATION

MultiScheme provides two methods, global interrupts and synchronizers, for coordinating the activities of processors. Unlike placeholders, which coordinate the activity of *tasks*—logical processes generated by running progams—these two operations deal with the physical processing units of the hardware. As such, they are more often used by the MultiScheme system itself than by application programs. The two operations, while not necessarily novel, have the virtues of simplicitly and compatibility. They were motivated by the difficulty of initiating a garbage collection, but they serve as a base for higher-level constructs (see the examples of Section 8.2 and 8.3).

8.1. Starting Garbage Collection

One of the early problems encountered in moving from a sequential simulation of MultiScheme to a truly parallel implementation was modifying the mechanism used to initiate a garbage collection. MultiScheme, like sequential MIT Scheme, uses a stop-and-copy garbage collector, although MultiScheme's garbage collector uses a parallel algorithm. The subsequent discussion, with only slight modification, applies equally well to initiating the space flip in a real-time copying garbage collection algorithm.

In MIT Scheme, garbage collection is initiated in three phases, using a system modeled after a hardware priority interrupt mechanism:

Interrupt Request

During some operation the processor notices that it is low on memory and sets a bit requesting a garbage collection interrupt.

Interrupt Detect

The interpreter and compiled code periodically poll the interrupt bits. A pending interrupt is serviced if no higher level interrupt is pending or in progress.

Interrupt Service

Before executing the next instruction the machine calls the interrupt handler for the current interrupt level, a procedure supplied by the Scheme runtime system. The level of the interrupt determines which handler to call, and there is a level devoted to garbage collection interrupts.

In moving to a parallel-processor hardware base there was no need to modify the basic interrupt mechanism but some of the details were modified. Because all of the processors share a common address space for

the heap it is essential that they cease computing before the garbage collector begins relocating objects. The system must, therefore, support some mechanism for forcing all of the processors to synchronize. The ability to initiate such global synchronization from software is essential to several system services (see the examples of Sections 8.2 and 8.3).

Only three modifications to MIT Scheme are used in MultiScheme to provide coordination among the processors. The interrupt levels intersperse global interrupts that are pertinent to all processors with local interrupts that are pertinent to the current processor only. The procedure global-interrupt allows any processor to interrupt the others. And synchronization objects permit all processors to proceed in unison (known as **barrier synchronization**).

The new operation global-interrupt is the software interface that initiates a global interrupt:

```
(global-interrupt priority-level interrupt-handler all-clear?)
```

The interrupt-handler is a procedure that is to be executed by all the *other* processors. Because a global interrupt requires the cooperation of all processors, initiating such an interrupt must be serialized. A processor receives permission to initiate a global interrupt only when no interrupt (local or global) of a higher priority is pending. At that time, it calls the all-clear? procedure to determine whether or not the interrupt should actually be initiated. This test is used, for example, to guarantee that a garbage collection global interrupt is issued exactly once even though the need for it may be detected independently by multiple processors. The value returned by a call to global-interrupt is the value returned by all-clear? so the processor issuing the interrupt can determine whether or not the interrupt was actually generated.

Global-interrupt returns control to the caller only after the interrupt is initiated or the all-clear? procedure indicates that no interrupt should take place. It guarantees that all of the processors will stop their ordinary work as soon as they poll their own interrupt bits, an event that the interpreter and compiler force to occur fairly often. This alone, however, is not sufficient to solve the problem of starting a garbage collection. Before *any* processor can begin the actual garbage collection operation, *all* processors must be entering the garbage collection operation. The global interrupt mechanism provides a way of initiating an action, but does not provide synchronization.

Instead, MultiScheme provides a pair of procedures for this purpose: make-synchronizer and await-synchrony. These jointly provide a mechanism for creating a cooperative barrier synchronization. To synchronize all of the processors, one processor makes a synchronizer object

and then forces all of the other processors to call await-synchrony with the synchronizer as argument. When all processors are waiting for synchrony on the same synchronizer object they all return from the call to await-synchrony. Typically, one processor makes one or more synchronizers and then uses global-interrupt to force the other processors to begin waiting on them.

While separating these two operations can cause deadlocks if they are used improperly, the operations do serve two distinct purposes. The examples of the next two sections show how the separate operations can be used to provide higher-level operations that are not as easily provided if the low-level operations are bundled together.

8.2. Pause-Everything

Clamen[13] describes an early investigation into debugging tools for dealing with the parallelism of MultiScheme programs. He identified a variety of situations requiring a program to temporarily stop all other work on the system, perform some action, and then allow the work to proceed. In order to provide this ability, he implemented the original pause-everything procedure. A new implementation extending Clamen's original version is depicted in Fig. 11, simplified for purposes of explana-

```
(define (pause-everything)
  (let ((drain-synch (make-synchronizer))
        (proceed-synch (make-synchronizer)))
    (define (interrupt-handler)
      (saving-state                          ; (1)
       (lambda ()
         (await-synchrony drain-synch)
         (await-synchrony proceed-synch))))
    (global-interrupt high-priority
      interrupt-handler (lambda () #T))
    (await-synchrony drain-synch)
    (let ((pool (drain-work-pool)))          ; (2)
      (await-synchrony proceed-synch)
      (make-returned-object pool))))         ; (3)
```

Notes:

1. Saving-State stores away the state of the task currently executing on this processor and places it in the work pool. It then calls the procedure which is its only argument. Saving-State never returns to its caller: it looks for work from the pool when the argument procedure is finished. See the discussion in Section 4.3 and the code in Figure 7.
2. Drain-work-pool empties the pool of tasks awaiting processors and returns a weak list of the tasks removed. See the description of drain-work-pool in Section 3.3.
3. Make-returned-object creates the message-accepting object that is the result of a call to pause-everything. See the discussion of make-returned-object in Appendix A and the code in Figure 13.

Fig. 11. Simplified Code for Pause-Everything.

tion. It provides a message-passing interface to an object representing the tasks that were available for execution at the time of the call to pause-everything. This interface is described in Appendix A. A procedure with structure very similar to pause-everything but without this elaborate interface is also used to initiate garbage collection.

Pause-everything uses both the global interrupt mechanism and the synchronizers. A global interrupt is necessary to force the other processors to save their state and become idle. The synchronizers are used to divide the work into two phases.

The first phase is initiated by the call to global-interrupt and ends when all processors have arrived at the first synchronization point. The interrupt guarantees that all processors except the one that called pause-everything will begin executing the code in interrupt-handler. Thus the other processors save away the state of the task they are executing and place it in the pool of work to be performed. They then wait for all processors to execute (await-synchrony drain-synch). When all the processors arrive at this point, the tasks available for execution (including the ones that were formerly executing) have been saved in the work pool.

The processors proceed past the synchronization point, beginning the second phase. All but the initiating processor will arrive immediately at the second rendezvous point, the call to (await-synchrony proceed-sync). The initiating task, however, first saves away the contents of the work pool in the variable pool and empties the pool. All the processors again rendezvous, ending the second phase.

The initiating task makes the message accepting object based on the value of pool by calling the procedure make-returned-object shown in Fig. 13. This becomes the value of the original call to pause-everything. The other processors, however, have now finished the procedure that is the argument to saving-state. But saving-state does not return to the procedure that called it. Instead it tries to get work from the (now empty)

```
(define (within-task task thunk)
  ...
  (with-task-locked task
    (lambda (task-still-runnable?)
      (if (eq? (task.status task) 'RUNNING)
          (begin
            (set-task.status! task 'WITHIN-TASK)
            (set-task.code! task thunk)
            (global-interrupt high-priority
             (lambda () (if (eq? (current-task) task) (reschedule)))
             (lambda () #T)))
          ...))))
```

Fig. 12. Simplified Code for Within-Task.

```
(define (make-returned-object the-queue)
  (lambda (message)
    (cond ((eq? message 'ANY-TASKS?)     ; (1)
           (and (not (eq? the-queue #T))
                (not (eq? the-queue '()))))
          ((eq? message 'RESTART-TASKS) ; (2)
           (if (eq? the-queue #T)
               (error "Attempt to re-use a pause object!")
               (begin
                 (for-each               ; (3)
                  (lambda (task)
                    (activate task
                              (lambda (status) (eq? status 'PAUSED))
                              'NO-RELEVANT-WAKE-UP-VALUE)
                    the-queue))
                  (set! the-queue #T))))
          ((eq? message 'THE-TASKS)      ; (4)
           (if (eq? the-queue #T) '() the-queue))
          (else (error "Pause: unknown message" message)))))
```

Notes:

1. Code to handle the Any-Tasks? message.
2. Code to handle the Restart-Tasks message.
3. If this is the first time the restart-tasks message is received, any tasks that are still paused are activated. Notice that the code for `await-placeholder` shown in Figure 9 will have activated any of these tasks that were touched after the call to `pause-everything`. Hence, the test here.
4. Code to handle the The-Tasks message.

Fig. 13. Make-Returned-Object, support for pause-everything.

work pool. So at this point, the initiating task is still running on the initiating processor. All the other processors have relinquished the task they were executing and are waiting for more work. The system has "paused."

An interesting detail has been deliberately omitted in this description. What should happen if, while other work is suspended, the running task touches the placeholder associated with one of the suspended tasks? While there is no obviously correct answer, the actual MultiScheme system marks these suspended tasks **paused**. Touching a placeholder that is marked **paused** is handled by the scheduler in exactly the same way the scheduler handles touching a placeholder marked **delayed**: the associated task will be reactivated (see the code for await-placeholder in Fig. 9). One of the advantages of writing the scheduler in MultiScheme is that such decisions can be easily changed. For example, it is quite easy to add a "scheduler hook" to allow users to specify their own way of handling this situation.

8.3. Within-Task

The final example of processor coordination is a critical part of the user interface, allowing the user to interact with previously started tasks. The primary use of this facility occurs in the top level interaction between a user and the MultiScheme system. In MIT Scheme, a user can at any time interrupt execution of a program and create an interaction environment within the state of the program at the time the interrupt was serviced by issuing a "breakpoint interrupt." A user can also interrupt the program and force it to return back to an earlier interaction environment, effectively aborting the current computation.

The direct extension of this into MultiScheme would allow a user to interrupt the system and interact with any one of the tasks in the system at the time of the interrupt. Because a task has state information visible to the programmer it is important that the correct task actually interact with the user. As a result, there must be some way to force a selected task to call the procedure that implements the interaction environment or invoke a continuation that aborts the current computation.

Within-task is designed to facilitate this and other similar operations. Some of Clamen's debugging tools, for example, rely on within-task in order to report the progress of a program back to the user. A simplified version of this procedure is shown in Fig. 12. It expects two arguments, a task and a thunk, and forces the task to execute the thunk before continuing with whatever processing it is currently doing. The operation of this procedure is intertwined with the scheduler (recall the code for store-my-state in Fig. 2) but the overview demonstrates a different use of the global interrupt mechanism.

Within-task works by testing whether the task is currently executing on one of the processors. If so, it marks the task indicating that it must be forced to execute the specified code and then initiates an interrupt on all processors, forcing the one running the chosen task to reschedule the task for later execution. Recall that the procedure reschedule uses store-my-state to store the state of the task in such a way that the specified code is executed before the task resumes its current computation.

Unlike the earlier uses of global interrupts, this one does not need any synchronization of the processors. The purpose of the interrupt is merely to get the attention of a particular processor without knowing in advance which one. All of the processors are briefly interrupted from their duties to process the interrupt, but they can resume immediately and independently.

9. CONCLUSION

The scheduler is the "heart" of the MultiScheme system, and is designed as a flexible and extensible mechanism for supporting a variety of experiments. The primary data structures of the scheduler are the placeholder and the task. Using these data structures, a variety of ways of creating, suspending, and managing tasks have been implemented and used. The ease of experimentation has facilitated the construction of a number of different approaches to parallelism, including speculative computation and data flow simulation.

Most of the material presented is "nuts and bolts engineering" but serves to demonstrate the ease with which parallel processing support can be described within an existing language framework. The small number of changes required to the sequential Scheme language is particularly pleasing.

APPENDIX: SINGLE TASK INTERLUDES

The scheduler supports the ability to switch from a parallel processing mode where many tasks are simultaneously active to one in which only a single task is active. This operation, embodied in the procedure pause-everything, has already been discussed in Section 8.2 and the code is shown in Fig. 11. This section presents the underlying support routine that was omitted in that earlier version.

As mentioned in the earlier discussion of pause-everything its job is to suspend all other tasks on the system. It returns as its value an object that encapsulates these other tasks through a "message passing" interface. The returned object is implemented as a procedure of one argument, the message. It accepts three messages.

Any-Tasks?

Returns a boolean answer of #T is there were other tasks running at the time of the call to pause-everything and they have not yet been restarted.

The-Tasks

Returns a list of the tasks that were suspended by the call to pause-everything provided they have not yet been restarted.

Restart-Tasks

Activates the tasks that were suspended by the call to pause-everything by releasing them to the underlying task distribution mechanism. It can be called only once.

The procedure Make-Returned-Object, shown in Fig. 13, creates the message accepting object that will be returned by pause-everything.

REFERENCES

1. James Miller, *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Massachusetts Institute of Technology (August 1987). Available as MIT LCS/TR/402.
2. R. Halstead, Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, p. 501–538 (October 1985).
3. Sharon L. Gray, *Using Futures to Exploit Parallelism in Lisp*, Master's thesis, Massachusetts Institute of Technology (1986).
4. A. Wang, *Exploiting Parallelism in Lisp Programs with Side Effects*, Bachelor's thesis, Massachusetts Institute of Technology (May 1986).
5. John McCarthy, A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, (ed.), *Computer Programming and Formal Systems*, North-Holand (1963).
6. R. P. Gabriel and J. McCarthy, Queue-based multi-processing lisp. In *ACM Symp. on Lisp and Functional Programming*, p. 25–44, Austin, Texas (August 1984).
7. M. Katz, *ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme*, Master's thesis, Massachusetts Institute of Technology (May 1986).
8. Tom Knight, An architecture for mostly functional languages. In *Symposium on LISP and Functional Programming, ACM*, p. 105–112 (1986).
9. H. Baker and C. Hewitt, *The Incremental Garbage Collection of Processes*, AI Memo 454, Massuchetts Institute of Technology, Artificial Intelligence Laboratory (December 1977).
10. *MIT Scheme Reference, Scheme Release 7*, Massachusetts Institute of Technology, Cambridge, Massachusetts (1988).
11. Jonathan Rees and William Clinger (eds.). Revised[3] report on the algorithmic language scheme. *ACM Sigplan Notices*, **21**(12) 37–79. Also available as MIT AI Memo 848a (December 1989).
12. James Miller, Garbage collection in multischeme. (*in preparation*)
13. Stewart Michael Clamen, *Debugging in a Parallel Lisp Environment*, Bachelor's thesis, Massachusetts Institute of Technology (1986).