Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs

Eric Mohr
Yale University
mohr@cs.yale.edu

David A. Kranz
Laboratory for Computer Science
M.I.T.
kranz@ai.mit.edu

Robert H. Halstead, Jr.
DEC Cambridge Research Lab
halstead@crl.dec.com

Abstract

Many parallel algorithms are naturally expressed at a fine level of granularity, often finer than a MIMD parallel system can exploit efficiently. Most builders of parallel systems have looked to either the programmer or a parallelizing compiler to increase the granularity of such algorithms. In this paper we explore a third approach to the granularity problem by analyzing two strategies for combining parallel tasks dynamically at run-time. We reject the simpler *load-based inlining* method, where tasks are combined based on dynamic load level, in favor of the safer and more robust *lazy task creation* method, where tasks are created only retroactively as processing resources become available.

These strategies grew out of work on Mul-T [14], an efficient parallel implementation of Scheme, but could be used with other applicative languages as well. We describe our Mul-T implementations of lazy task creation for two contrasting machines, and present performance statistics which show the method's effectiveness. Lazy task creation allows efficient execution of naturally expressed algorithms of a substantially finer grain than possible with previous parallel Lisp systems.

1 Introduction

There have been numerous proposals for implementations of applicative languages on parallel computers. All have in some way come up against a granularity problem—when a parallel algorithm is written naturally, the resulting program often produces tasks of a finer grain than an implementation can exploit efficiently. Some researchers look to hardware specially designed to handle fine-grained tasks [2, 8], while others have looked for ways to increase task granularity by grouping a number of potentially parallel operations together into a single sequential thread. These latter efforts can be classified by the degree of programmer involvement required to specify parallelism, from parallelizing compilers at one end of the spectrum to language constructs giving the programmer a fine degree of control at the other.

In the most attractive world, the programmer leaves the job of identifying parallel tasks to a parallelizing compiler. To achieve good performance, the compiler must create tasks of sufficient size based on estimating the cost of various pieces of code [5, 12]. But when execution paths are highly data-dependent as with recursive symbolic programs, the cost of a piece of code is often unknown at compile time. If only known costs are used, the tasks produced may still be too fine-grained. And for languages that allow side effects it can be quite complex to determine where parallel execution is safe, and opportunities for parallelism may be missed.

At the other end of the spectrum a language can leave granularity decisions up to the programmer, providing tools for building tasks of acceptable granularity such as the *propositional parameters* of Qlisp [4, 6, 7]. Such fine control can be necessary in some cases to maximize performance, but there are costs in programmer effort and program clarity. Also, any parameters appearing in the program require experimentation to calibrate; this work may have to be repeated for a different target machine or data set. Or, when the code is run in parallel with other code or on a multi-user machine, a given parameterization may be ineffective because the amount of resources available for that code is unpredictable. Similar problems arise when a parallelizing compiler is parameterized with details of a certain machine.

We've taken an intermediate position in our research on Mul-T [14], a parallel version of Scheme based on the future construct of Multilisp [9, 10]. The programmer takes on the burden of identifying *what* can be computed safely in parallel, leaving the decision of exactly *how* the division will take place to the run-time system. In Mul-T that means annotating programs with future to identify parallelism without worrying about granularity; the programmer's task is to *expose* parallelism while the system's task is to *limit* parallelism.

In our experience with the mostly functional style common to Scheme programs, a program's parallelism can often be expressed quite easily by adding a small number of future forms (which however may yield a large number of concurrent tasks at run time). The effort involved is lit-

tle more than that required for systems with parallelizing compilers, where the programmer must be sure to code in such a way that parallelism is available. (We note that these dynamics of parallel programming are shared by functional languages; the philosophy and goals of the "para-functional" approach [11, 13] are similar to ours.)

In order to support this programming style we must deal with questions of efficiency. The Encore Multimax[1] implementation of Mul-T [14], based on the T system's ORBIT compiler [15, 16], is proof that the underlying parallel Lisp system can be made efficient enough; we must now figure out how to achieve sufficient task granularity. For this we look to dynamic mechanisms in the run-time system, which have the advantage of avoiding the parameterization problems mentioned earlier. The key to our dynamic strategies for controlling granularity is the fact that that the `future` construct has several correct operational interpretations. The canonical `future` expression

(*C* `(future` *X*`)`)

declares that a child computation X may proceed in parallel with its parent continuation C. In the most straightforward interpretation, a child task is created to compute X while the parent task computes C.[2] Reversing the task roles is also possible; the parent task can compute X while the child task computes C. Finally, and most importantly for fine-grained programs, it is also usually correct for the parent task to compute first X and then C, ignoring the `future`.[3] This *inlining* of X by the parent task eliminates the overhead of creating and scheduling a separate task and creating a placeholder to hold its value.

Inlining can mean that a program's *run-time granularity* (the size of tasks actually executed at run time) is significantly greater than its *source granularity* (the size of code within the `future` constructs of the source program). A program will execute efficiently if its average run-time granularity is large compared to the overhead of task creation, providing of course that enough parallelism has been preserved to achieve good load balancing.

The first dynamic strategy we consider is *load-based inlining*. In this strategy, `(future` *X*`)` means, "If the system is not loaded, make a separate task to evaluate X; otherwise inline X, evaluating it in the current task." A load threshold T indicates how many tasks must be queued before the system is considered to be loaded. Whenever a call to `future` is encountered, a simple check of task queue length determines whether or not a separate task will be created.

The simple load-based inlining strategy works well on some programs, but its several drawbacks (see Section 3) led us to consider another strategy as well: why not inline every task provisionally, but save enough information so that tasks can be selectively "un-inlined" as processing resources become available? In other words, create tasks lazily. With this *lazy task creation* strategy, (*C* `(future` *X*`)`) means "Start evaluating X in the current task, but save enough information so that its continuation C can be moved to a separate task if another processor becomes idle." We say that idle processors *steal* tasks from busy processors; task stealing becomes the primary means of spreading work in the system.

The execution tree of a fine-grained program has an overabundance of potential fork points. Our goal with lazy task creation is to convert a small subset of these to actual forks, maximizing run-time task granularity while preserving parallelism and achieving good load balancing. In the subsequent discussion, this is contrasted with *eager task creation*, where all fork points result in a separate task. When referring to the implementation of these strategies in Mul-T we will sometimes use the terms *lazy futures* and *eager futures*.

An example will help make these ideas more concrete.

2 An Example

As a simple example of the spectrum of possible solutions to the granularity problem, consider the following algorithm (written as a Scheme program) to sum the leaves of a binary tree:

```
(define (sum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (sum-tree (left tree))
         (sum-tree (right tree)))))
```

(where `leaf?`, `leaf-value`, `left`, and `right` define the tree datatype). The natural way to express parallelism in this algorithm is to indicate that the two recursive calls to `sum-tree` can proceed in parallel. In Mul-T we might indicate this by adding one `future`:[4]

```
(define (psum-tree tree)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (future (psum-tree (left tree)))
         (psum-tree (right tree)))))
```

The natural expression of parallelism in this algorithm is rather fine-grained. With eager task creation this program would create 2^d tasks to sum a tree of depth d; the average number of tree nodes handled by a task would be 2. Figure 1 shows this execution pictorially; each circled subset of tree nodes is handled by a single task. Unless task creation is very cheap, this task breakdown is likely to lead to poor performance.

[1] Multimax is a trademark of Encore Computer Corporation.

[2] `(future` *X*`)` returns an object called a *future*, a placeholder for the eventual value of X. Any task attempting to use the value of this future before X has completed is suspended until the value is available.

[3] Such inlining is not always correct; sometimes it can lead to deadlock as described in Section 3.3.

[4] This strategy for adding `future` relies on `+` evaluating its operands from left to right; if argument evaluation went from right to left, then `(psum-tree (right tree))` would evaluate to completion before `(future (psum-tree (left tree)))` began, and no parallelism would be realized.

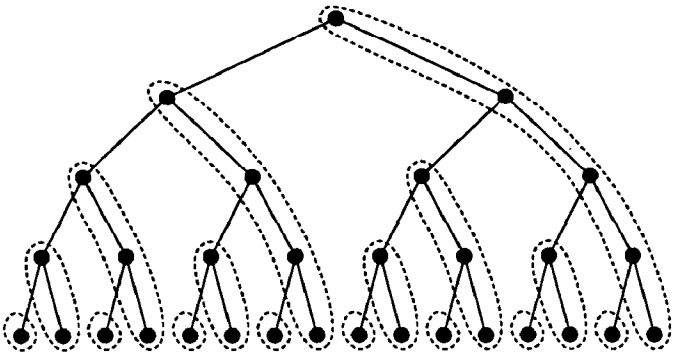

Figure 1: Direct execution of psum-tree.

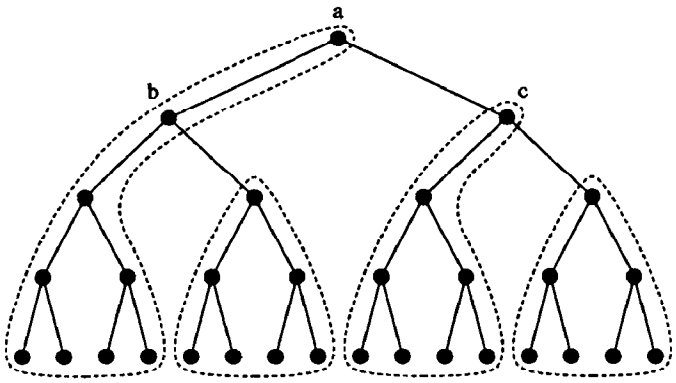

Figure 2: BUSD execution of psum-tree on 4 processors.

The ideal task breakdown is one which maximizes the run-time task granularity while maintaining a balanced load. For a divide-and-conquer program like this one, that means expanding the tree breadth-first by spawning tasks until all processors are busy, and then expanding the tree depth-first within the task on each processor. We will refer to this ideal task breakdown as BUSD (Breadth-first Until Saturation, then Depth-first). Figure 2 shows this execution pictorially for a system with 4 processors.

How can we achieve this ideal task breakdown? A parallelizing compiler might be able to increase granularity by unrolling the recursion and eliminating some futures, but in this example we *want* fine-grained tasks at the beginning so as to spread work as quickly as possible (breadth-first). The compiler might possibly produce code to do this as well if supplied with information about available processing resources, but making such a transformation general is a difficult task and would still have the parameterization drawbacks noted earlier.

What if we control task creation explicitly as in Qlisp? In many of Qlisp's parallel constructs the programmer may supply a predicate which, when evaluated at run time, will determine whether or not a separate task is created (one such predicate, (qemptyp) [7], tests the length of the work queue, achieving the same effect as our load-based inlining). If such a Qlisp-style mechanism were used to create a hypothetical qfuture construct, we might write psum-tree in this way (very similar to an example in [4]):

```
(define (psum-tree-2 tree cutoff-depth)
  (if (leaf? tree)
      (leaf-value tree)
      (+ (qfuture (> cutoff-depth 0)
           (psum-tree-2 (left tree)
                        (- cutoff-depth 1)))
         (psum-tree-2 (right tree)
                      (- cutoff-depth 1)))))
```

Here cutoff-depth specifies a depth beyond which no tasks should be created. The predicate (> cutoff-depth 0) tells qfuture whether or not to inline the recursive call. A cutoff-depth value of 2 would achieve the BUSD execution shown in Figure 2; below level 2 all futures are inlined.

This solution has two problems. First, the code has become more complex by the addition of cutoff-depth—it is no longer completely straightforward to tell what this program is doing. Second, the program is now parameterized by the cutoff-depth argument, with the associated calibration issues noted previously.

Both load-based inlining and lazy task creation can approximate the BUSD performance of psum-tree-2 without sacrificing the clarity of psum-tree. When psum-tree is run with load-based inlining, the first three occurrences of future (probably at nodes a, b, and c of Figure 2) find that processors are free, and separate tasks are created (breadth-first). Depending on the value of the threshold parameter T, a few more tasks may be created before the backlog is high enough to cause inlining. But since there is a large surplus of work, most tasks are able to defray the cost of their creation by inlining a substantial subtree (depth-first). This general behavior has been modeled analytically by Weening [23, 24].

When psum-tree is run with lazy task creation on a four processor system, the future at a (representing the subtree rooted at b) is provisionally inlined, but its continuation (representing the subtree rooted at c) is immediately stolen by an idle processor. Likewise, the futures at b and c are inlined, but their continuations are stolen by the two remaining idle processors. Now all processors are busy; subsequent futures are all provisionally inlined but no further stealing takes place and each processor winds up executing one of the circled subtrees of Figure 2.

This execution pattern depends on an *oldest-first* stealing policy: when an idle processor steals a task, the oldest available fork point is chosen. In this example the oldest fork point represents the largest available subtree and hence a task of maximal run-time granularity.

Because of the dynamic nature of both lazy task creation and load-based inlining, real-life execution of a program like psum-tree may not match Figure 2 exactly. For example, some additional tasks may be created at the tail end of program execution as processors become idle, but this is unlikely to have a noticeable effect on overall efficiency.

3 Comparison of Dynamic Methods

Needless to say, not all programs are as simple as `psum-tree`. Load-based inlining does in fact produce good results for some programs [14] but it also has drawbacks, consideration of which led to the idea of lazy task creation. After summarizing these drawbacks we discuss each in turn as a basis for comparing the two dynamic strategies. The problems with load-based inlining:

1. The programmer must decide when to apply load-based inlining, and at what load threshold T.
2. Load-based inlining is irrevocable; processors can starve even though many inlined tasks are pending.
3. Deadlock can result if inlining is used on some types of programs.
4. For programs with irregular call trees, load-based inlining can create many more tasks than an optimal BUSD division would create.
5. Load-based inlining is ineffective for fine-grained linear recursions (loops).

3.1 Programmer Involvement

The first drawback is straightforward; even though load-based inlining is an automatic mechanism, it still requires programmer input. Some programs run better without inlining, so the programmer must identify where it should be applied. The programmer must also supply a value for the load threshold T. Lazy task creation needs no such parameterization, and, because stealing tasks is the primary means of scheduling, lazy task creation can be used with any program.

3.2 Irrevocability

The irrevocability of load-based inlining can mean that processors become idle even though many inlined tasks are pending, due to either *bursty task creation* or *parent-child welding*. *Bursty task creation* refers to the fact that opportunities to create tasks may be distributed unevenly across a program. At the moment when a task is inlined, it may appear that there are plenty of other tasks available to execute, but by the time these tasks finish executing there may be too few opportunities to create more tasks. Consequently, processors may go idle because the inlined tasks are not available for execution. This problem never arises with lazy task creation because inlined tasks are always available for stealing.

Parent-child welding refers to the fact that inlining effectively "welds" together a parent and child task. If an inlined child becomes blocked waiting for a future to resolve (or for some other event), the parent is blocked as well and is not available for execution. With lazy task creation, the information kept for each inlined child allows the child to be decoupled if it becomes blocked, allowing the parent to continue.

3.3 Deadlock

Irrevocability is also behind the deadlock problem. To see how inlining can lead to deadlock, consider the program in Figure 3 for finding primes. It uses a standard prime-finding algorithm, checking each (odd) integer n for primality by looking for divisors among the primes found so far, up to $\sqrt{n}$. Futures introduce parallelism, as well as getting around the difficulty of accessing both ends of a single list (adding primes to the end while reading primes from the front).

`all-primes` is initially bound to a lazily generated list of all the odd primes.[5] The function `find-primes>=n` generates a tail of `all-primes` by first making a future to find all (odd) primes above n, and then checking n for primality by walking down `all-primes`, using the primes already generated.

This program could deadlock with inlining. Say a task T calls `(find-primes>=n 5)`, and inlines the subsequent calls for $n = 7, 9, ..., 25$. Then T makes a real future for $n = 27$ and calls `prime?` to test the primality of 25. `prime?` will attempt to use the second element of `all-primes`, but will block because it hasn't been computed yet. But T itself was computing this second element, so deadlock arises because T has blocked on itself. No such deadlock is possible with lazy task creation, because of the decoupling of blocked tasks mentioned above. Programs that are deadlock-free with eager task creation are also deadlock-free with lazy task creation.

3.4 Irregular Call Trees

The fourth drawback of load-based inlining only shows up in programs with irregular call trees. In programs with regular call trees, load-based inlining gives a good approximation to BUSD—a few initial forks saturate the machine, creating tasks which expand subtrees of essentially identical shape. Tasks are always inlined because all processors are busy. But consider a program where the call tree is not a complete tree (*e.g.* a search program like n queens). As above, a few initial forks saturate the machine. But now the subtrees being expanded by each task have widely varying shapes, so some processors will run out of work early on in the program execution. The next task received by such a processor is determined dynamically, by whatever other task happens to come to a fork point next. Because the majority of fork points lie toward the leaves of the tree, our processor is likely to get a task representing only a small subtree. This results in tasks of small run-time granularity, so many more tasks are created than with an ideal BUSD execution.

When lazy task creation is used with an oldest-first stealing policy this problem doesn't arise. When a processor

[5] **delay**, which creates a future object but does not spawn a task, is used instead of **future** to avoid a race condition in the **letrec** binding.

```
(define (find-primes limit)
  (letrec ((all-primes (cons 3 (delay (find-primes>=n 5))))
           (find-primes>=n (lambda (n)
                             (if (> n limit)
                                 '()
                                 (let ((rest (future (find-primes>=n (+ n 2)))))
                                   (if (prime? n all-primes)
                                       (cons n rest)
                                       rest))))))
    (cons 2 all-primes)))

(define (prime? n primes)
  (let ((prime (car primes)))
    (cond ((> (* prime prime) n)  #t)
          ((zero? (mod n prime))  #f)
          (else (prime? n (cdr primes))))))
```

Figure 3: Program `find-primes` could deadlock with load-based inlining.

becomes idle, it steals a task from the oldest outstanding fork point in the call tree. In a divide-and-conquer program (even one with an irregular call tree) the oldest fork point is very likely to represent a substantial subtree, leading to large run-time task granularity and a closer approximation to BUSD execution. The performance statistics in Section 5 bear this out.

3.5 Linear Recursion

Of course not all parallel programs have bushy call trees; for example, some programs contain data-level parallelism expressed by a linear recursion over a data structure. Load-based inlining is not effective in increasing the run-time granularity of such programs. To see why, consider the following versions of parallel `map`, exemplifying the two straightforward methods of parallelizing a linear recursion on a list:

```
(define (parmap-cars f l)
  (if (null? l)
      '()
      (let* ((elt (future (f (car l))))
             (rest (parmap-cars f (cdr l))))
        (cons elt rest))))

(define (parmap-cdrs f l)
  (if (null? l)
      '()
      (let* ((rest (future
                    (parmap-cdrs f (cdr l))))
             (elt (f (car l))))
        (cons elt rest))))
```

With both load-based inlining and lazy task creation, efficiency is increased for fine-grained programs when each task is able to inline many other tasks, increasing the average run-time task granularity and reducing overhead due to task creation. In both of these versions of parallel map, many tasks are unable to inline any subtasks, leading to high task-creation overhead when `f` is fine-grained.

In `parmap-cars` a parent task loops through the list, calling `future` for each application of `f` to a list element. In this program, inlining futures can only increase the granularity of the parent task; any child tasks created will be fine-grained because they have no inlining opportunities. So at best we will have one task of large granularity and many of small granularity, leading to poor performance.

In `parmap-cdrs`, `future` appears around a call to map down the rest of the list; the parent task then applies `f` to the current list element. It is conceivable in this case that inlining could create several tasks of large granularity; the parent could inline several futures before making a real future F_1, F_1 could inline several futures before making a real future F_2, *etc.* In practice however, the system load is low initially and many small tasks are created. With numerous processors, tasks are executed faster than they can be created so the backlog necessary for load-based inlining never builds up.

Unfortunately, lazy task creation suffers the same problems as load-based inlining on this type of program. The eager stealing policy necessary for timely scheduling of tasks leads in this case to many small tasks and poor performance. We have considered several strategies for addressing this problem, and will have more to say about it at the end of the paper.

But as we have seen, lazy task creation overcomes all of the other drawbacks of load-based inlining, warranting a study of its implementation costs.

4 Implementation

As described above, our dynamic methods increase efficiency by ignoring selected instances of `future`. But ignoring a `future` will never be as cheap as no `future` at all—lazy task

creation requires maintaining enough information when a future is provisionally inlined to allow another processor to steal the future's continuation cleanly. The cost of maintaining this information is *the* critical factor in determining the finest source granularity that can be handled efficiently. The cost is incurred whether a new task is created or not, so a large overhead would overwhelm a fine-grained program. By comparison the cost of actually stealing a task is somewhat less critical; if enough inlining occurs the cost of stealing a task will be small compared to the total amount of work the task ultimately performs.

Still, the cost of stealing a continuation must be kept in the ballpark of the cost of creating an eager future. Stealing a continuation requires splitting an existing stack, which in a conventional stack-based implementation requires the copying of frames from one stack to another. We discuss such copying costs later, but it is attractive to consider instead using a linked-frame implementation where splitting a stack requires only pointer manipulations. However, care must be taken with such an implementation that the normal operations of pushing and popping a stack frame have comparable cost with conventional stack operations.

We have pursued both avenues of implementation: a linked-frame implementation for the ALEWIFE multiprocessor as well as a conventional stack-based implementation for the Encore Multimax version of Mul-T.

In both implementations, instances of `future` are compiled as special procedure calls. When making a *lazy future call*, a task T first pushes a pointer to the `future`'s continuation onto a queue of continuations that are stealable from T (its *lazy future queue*). If upon return the continuation has not been stolen by another processor, T dequeues it. We refer to the processor making lazy future calls as the *producer* of lazy futures; another processor stealing them is called a *consumer*. Consumers remove frames from the head of the lazy future queue while the producer pushes and pops frames from the tail.

In order to steal the continuation to a lazy future call, the consumer must change the producer's stack to make it look as though an eager future call had been made. Consider again the expression:

(*C* (`future` *X*))

The producer makes a lazy future call to compute X, queueing its continuation C. Sometime later, a consumer decides to steal C. It creates a placeholder and modifies the producer's call stack so that the value returned by the call to X will determine (*i.e.* supply a value for) the placeholder rather than being passed directly to the continuation C. The consumer then calls C itself, passing the undetermined placeholder as an argument. It now looks as though an eager future had been created, with one processor (the producer) evaluating the child X and another (the consumer) evaluating the parent C.

Implementations must take care to guard against two kinds of race conditions to ensure correctness of the stealing operation. First, a producer trying to return to a continuation may race with a consumer trying to steal it; second, two consumers may race to steal the same continuation.

4.1 ALEWIFE implementation

The ALEWIFE machine is a cache-coherent machine being developed at MIT with distributed, globally shared memory. The processing elements are SPARC[6] chips modified to support rapid context switching, traps for strict operations on futures, and full/empty bits in the memory [1].

For the ALEWIFE implementation of lazy futures, stacks are represented as a two-way doubly linked list of stack frames [17] in order to minimize copying in the stealing operation. An important feature of this scheme is that stack frames are not deallocated when popped. A subsequent push will re-use the frame, meaning that in the average case the cost of stack operations associated with procedure call and return is very close to the cost of such operations with conventional stacks.

In this implementation the lazy future queue is threaded through the doubly linked list of stack frames, using frame slots reserved for the queue pointers. The easiest way to understand the lazy future operations is to look at pseudocode for a lazy future call first from the point of view of the producer and then from that of the consumer. In this code we use the following register names:

`FP` Frame pointer register. Points to the current stack frame.

`LFT` Lazy future tail register. Modified only by the producer.

`LFH` Head of the lazy future queue. This must be in memory so that consumers on other processors can steal frames from the head of the queue. Its full/empty bit serves as the lock limiting access to one potential consumer at a time.

Each stack frame has the following slots:

`lf-next` Frame slot points to the next frame on the lazy future queue (toward the tail of the queue). This location's full/empty bit is the lock arbitrating between a consumer stealing a continuation and the producer trying to invoke that continuation.

`lf-prev` Frame slot points to the previous frame on the lazy future queue (toward the head of the queue).

`lf-link` The lazy future call code stores the return address for the consumer in this slot. The consumer reads out the return address and stores the placeholder object it creates in this slot as well. When the producer tries to invoke a stolen continuation it will trap and instead store the returned value in this placeholder.

`next` Frame slot points to the "next" frame, which will become current if a stack-frame push operation is performed.

[6] SPARC is a trademark of Sun Microsystems, Inc.

cont Frame slot points to the "continuation" frame, which will become current if a stack-frame pop operation is performed.

regs Some number of slots for local variable bindings and temporary results.

Pseudo-code for lazy future call:

```
store address of continue: in lf-link[FP]
store LFT,lf-prev[FP]
store FP,lf-next[LFT] # allow stealing
move  FP,LFT
call procedure
load  lf-prev[FP],LFT # pop tail back to lft
test  lf-next[LFT] and trap if stolen
continue:
```

Pseudo-code for consumer:

```
acquire LFH lock
dequeue continuation frame from head of
    lazy future queue if present
copy the continuation frame
read return address from lf-link slot
    of old frame
create placeholder and store in lf-link
    slot of old frame, setting empty bit
adjust links in old and new frame
return to continuation by jumping to the
    return address that was in lf-link
```

We have omitted some details of the synchronization between the producer and consumer. As mentioned, the race condition between the producer trying to return to a continuation frame and the consumer trying to steal it is controlled by the lock in the `lf-next` slot of the frame. The producer, after detecting that its continuation has been stolen, must also wait for the consumer to fill in the `lf-link` slot with the placeholder. This synchronization is controlled by the lock (empty bit) in the `lf-link` slot. The lazy future call sequence shown ensures that the producer writes the return address into the `lf-link` slot before the consumer can access it.

Figures 4–6 show the lazy future call and stealing operations graphically. Figure 4 shows how the stack frames and relevant registers might look before a lazy future call. Note that each frame's `next` pointer points to the next frame toward the top of the stack and each `cont` pointer points to the next frame toward the bottom of the stack. An "X" in the left-hand part of a frame slot indicates that the full/empty bit of the corresponding memory word is set to "empty"; note that the `lf-next` slot of a frame is always empty unless that frame is part of the lazy future queue. If a frame slot's contents are left blank in the figure, its contents are either unimportant (they will never be used) or indeterminate: for example, the `next` slot of the leftmost frame in Figure 4 could either be empty or point to another, currently unused frame.

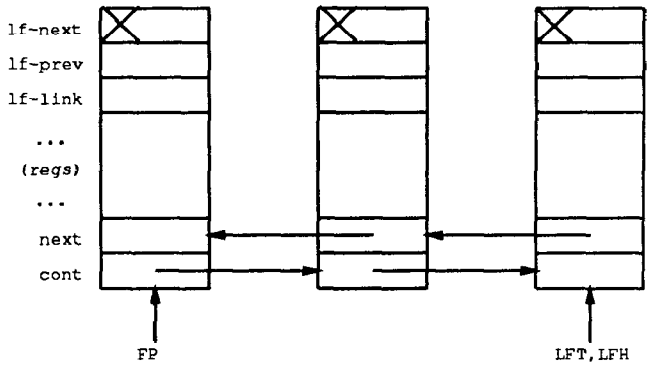

Figure 4: Before lazy future call.

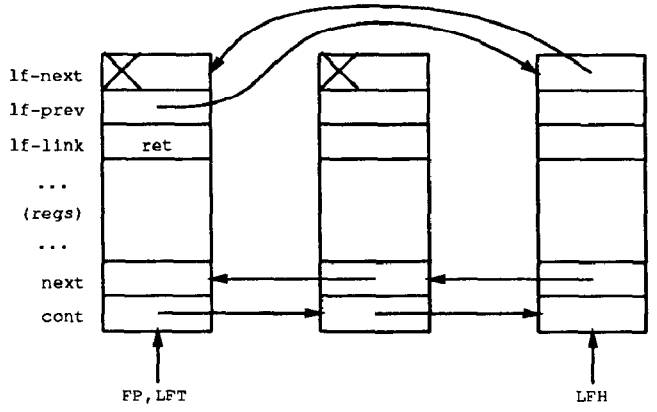

Figure 5: After lazy future call.

Figure 5 shows the situation after a lazy future call. The current frame (pointed to by FP) has joined the lazy future queue. Accordingly, LFT has changed to point to that frame, and `lf-next` and `lf-prev` links have been updated as needed to maintain the doubly linked lazy future queue. The address for the lazy future call's continuation has been stored in the current frame's `lf-link` field. The middle frame is not part of the lazy future queue, but simply part of the stack.

Finally, Figure 6 shows the state of the producer and consumer tasks if a consumer steals the continuation from the task shown in Figure 5 (the consumer task's state variables have a "c" appended, *e.g.* LFHc). The stealer has made the producer's `lf-prev` link point to a dummy frame that will cause a trap when the producer tries to return from the lazy future call. Note that the consumer's stack now looks just like the producer's did in Figure 4, just before the original lazy future call. The consumer has also made the producer's `lf-link` field point to a newly-created placeholder; the placeholder will be determined by the producer and is also passed to the consumer. (The synchronization here is slightly unusual, with `lf-link` marked "empty" even though it contains useful data.)

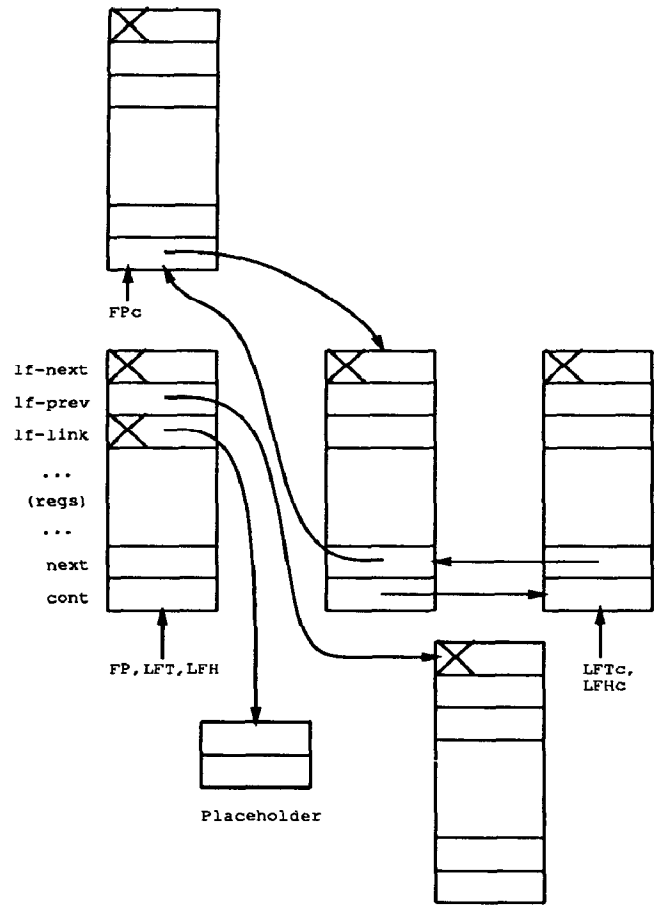

Figure 6: After steal.

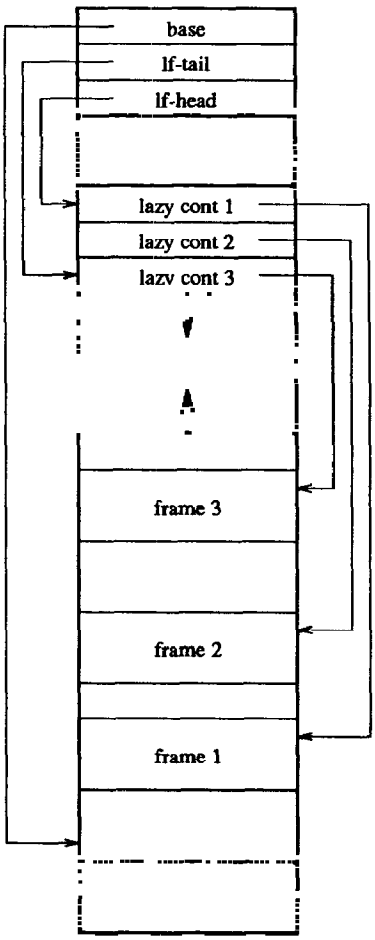

Figure 7: Lazy futures implemented using a conventional stack.

4.2 Encore Implementation

We have also implemented lazy futures in the version of Mul-T running on the Encore Multimax, a bus-based shared-memory multiprocessor. In this implementation stacks are represented conventionally, in contiguous sections of the heap. (When a stack overflows, its contents are copied to a new stack of twice the original size.)

As seen in Figure 7, the lazy future queue is kept in contiguous memory in the "top" part of a stack. As the producer pushes lazy continuations the queue grows downward while the stack frames grow upward. Stealing continuations effectively shrinks the stack by removing information from both ends (the head of the lazy future queue and the bottom frames of the stack).

To steal from the stack pictured, a consumer first locates the oldest continuation by following the **lf-head** pointer, through the **lazy cont 1** pointer, to **frame 1**. The consumer then replaces **frame 1** in the stack with a continuation directing the producer to determine a placeholder. Next the consumer copies frames from **frame 1** down to the bottom of the live area of the stack (indicated by **base**) to a new stack, updating **base** and **lf-head** appropriately.

To guard against the race conditions mentioned earlier there is a lock on the entire stack plus a lock on each lazy continuation. As with the ALEWIFE implementation, only the producer modifies **lf-tail**, and only consumers modify **lf-head** and **base**.

4.3 Discussion

What are the advantages and disadvantages of these implementations? The main disadvantage of the conventional stack implementation is in copying. It would appear that the amount of copying required for a stealing operation is potentially unlimited, so that the cost of stealing a lazy task is also unlimited. While this is technically true it is somewhat misleading; the overhead of copying when stealing a lazy continuation should be viewed against the cost of creating the continuation in the first place. A program with fine source granularity does little work between lazy future calls, and so is not able to push enough items onto the stack to require significant copying. A program which creates large continuations (requiring stealers to do lots of copying) must do a fair amount of work to push all that information on the

stack, and the cost of copying is unlikely to be a significant overhead.

One exception to this argument is a program which builds up a lot of stack and then enters a loop which generates futures:

```
(define (example)
  (build-up-stack-and-then-call loop))
(define (loop)
  (future ... )
  (loop))
```

Stealing the continuation to the first lazy future requires copying the built-up stack. As argued, that cost is unlikely to be significant compared with the cost of building up the stack in the first place. But in this example the stolen continuation immediately creates another lazy future, so the next steal must copy the same information again. In fact, spreading work to n processors in this example via lazy futures requires the built-up stack information to be copied n times.

There are two easy solutions to this problem. First, `loop` can be rewritten to resemble `parmap-cdrs` rather than `parmap-cars` (see section 3.5), resulting in a program where the built-up stack is never copied. Or, a `future` could be inserted around the call to `loop`, resulting in a program where the built-up stack is copied only once.

It appears then that the effects of copying in a conventional stack implementation can be minimized. But it is still attractive to eliminate copying altogether using the linked-frame implementation described for the ALEWIFE. Such an implementation is certainly more efficient on lazy future operations. It is somewhat more difficult to gauge exactly the overhead introduced in sequential sections of code. One ramification of re-using stack frames is that all frames have a fixed size; choosing the correct frame size involves a trade-off. If a small frame size is chosen, frames needing more space will need to create an overflow vector, increasing costs for accessing frame elements and for memory allocation. If a large frame size is chosen, most frames will contain a lot of unused slots. This could lead to more frequent garbage collection and might use up valuable space in cache and/or virtual memory, although these latter factors could well be minimal in today's memory-rich systems. The current ALEWIFE implementation uses a frame size of 22 slots. We must accumulate more experience with this promising implementation technique before making a final evaluation.

5 Performance

In this section we present performance figures for both Mul-T implementations. Experiments for the conventional stack version used Yale's Encore Multimax, configured with 18 NS-32332 processors and 64 megabytes of memory.

Figures for the linked-frame version were obtained using a detailed simulator of the ALEWIFE machine. We stress that this simulation is very realistic: the Mul-T implementation for the simulator would run essentially as-is on an ALEWIFE machine if one were available. The Mul-T run-time system as well as code for the benchmarks are compiled to SPARC instructions, which are interpreted by the simulator. Overheads due to future creation, blocking, scheduling, *etc.*, are accurately reflected in the statistics. Memory-referencing delays were not simulated in these experiments.

As mentioned earlier, it is crucial to minimize the overhead of lazy future calls. Below are statistics for both implementations on the additional cost of a lazy future call over that of a conventional call, namely pushing a continuation onto the lazy future queue and popping it off.

Encore	12 instructions, 12.6 μsec
ALEWIFE	9 instructions, 14 SPARC cycles

For the Encore, 4 instructions could be eliminated by using a compiler optimization for continuations to lazy future calls, saving roughly 3 μsec. Still, the ALEWIFE sequence is probably the cheaper of the two, since the RISC instructions of the SPARC are simpler than NS-32332 instuctions. Another important factor in the Encore time is that synchronization must be done by the rather expensive mechanism of a test-and-set instruction which acquires exclusive access to the bus. ALEWIFE's full/empty bits provide much cheaper synchronization.

Table 1 shows performance statistics for several Mul-T programs, comparing eager and lazy task creation for both Encore and ALEWIFE implementations. Statistics for load-based inlining ("LBI") are included as well for the Encore implementation, with the load threshold T chosen in each case to give the fastest time on 16 processors.

The columns marked t show elapsed time, in seconds for the Encore and thousands of simulated SPARC cycles for ALEWIFE. The columns marked nT show the number of tasks actually created. Statistics are shown for running the parallel code on 1, 2, 4, 8, and 16 processors, as well as for a sequential version of the benchmark with `future` compiled as a no-op macro. The eager and lazy sequential times differ slightly for ALEWIFE because of a one-instruction optimization in the standard procedure calling sequence under lazy futures. Encore times are somewhat variable despite our best efforts; a given column of times may change by 5% after a re-initialization, preventing exact comparisons between "lazy" and the other columns.

`fib` is the standard brute-force doubly recursive program for computing the nth Fibonacci number ($n = 16$ in this case). This program is very fine-grained, and to complicate matters was written with two `future`s in the procedure body instead of one. With eager task creation the overhead of creating futures completely overwhelms the calculation. Load-based inlining eliminates much of the overhead, but still creates many more tasks than an ideal BUSD execution would create. This is because of the program's non-uniform call tree, as discussed in section 3.4. Lazy task creation pro-

fib										
	Encore						ALEWIFE			
	Eager		LBI ($T=2$)		Lazy		Eager		Lazy	
n	t	nT	t	nT	t	nT	t	nT	t	nT
seq	.048	—	.048	—	.049	—	62	—	59	—
1	.714	3192	.055	244	.093	0	881	3192	88	0
2	.366	3192	.044	327	.050	8	442	3192	46	6
4	.198	3192	.038	381	.030	26	223	3192	26	35
8	.113	3192	.028	567	.019	112	114	3192	17	104
16	.086	3192	.027	832	.014	159	60	3192	11	190

prime-factor										
	Encore						ALEWIFE			
	Eager		LBI ($T=2$)		Lazy		Eager		Lazy	
n	t	nT	t	nT	t	nT	t	nT	t	nT
seq	.81	—	.81	—	.81	—	746	—	713	—
1	1.04	999	.82	55	.82	0	1129	999	721	0
2	.53	999	.43	153	.42	7	565	999	362	5
4	.27	999	.23	284	.21	14	284	999	183	20
8	.14	999	.12	344	.11	38	143	999	93	52
16	.07	999	.06	419	.06	96	74	999	53	322

queens										
	Encore						ALEWIFE			
	Eager		LBI ($T=1$)		Lazy		Eager		Lazy	
n	t	nT	t	nT	t	nT	t	nT	t	nT
seq	1.02	—	1.02	—	1.02	—	1707	—	1703	—
1	1.55	2056	1.01	5	1.06	0	2450	2056	1730	0
2	.70	2056	.53	170	.54	9	1146	2056	868	10
4	.35	2056	.28	328	.28	27	569	2056	438	33
8	.19	2056	.15	558	.15	91	301	2056	224	81
16	.12	2056	.10	745	.10	231	171	2056	122	352

speech										
	Encore						ALEWIFE			
	Eager		LBI ($T=1$)		Lazy		Eager		Lazy	
n	t	nT	t	nT	t	nT	t	nT	t	nT
seq	95.9	—	95.9	—	96.4	—	85.4 K	—	85.3 K	—
1	104.7	39856	96.2	1378	96.9	0	100.1 K	39856	85.6 K	0
2	53.7	39856	50.0	6173	49.3	583	51.3 K	39856	44.0 K	613
4	27.8	39856	26.5	12577	25.8	1907	26.8 K	39856	23.3 K	1946
8	14.9	39856	15.3	18011	14.4	4440	14.8 K	39856	13.0 K	4807
16	8.8	39856	10.6	21667	9.3	7875	8.8 K	39856	7.8 K	9930

Table 1: Performance of Mul-T benchmarks (times in seconds for Encore, in 1000's of SPARC cycles for ALEWIFE).

duces a much better approximation to BUSD, as shown by the smaller number of tasks.

prime-factor uses a generalized divide-and-conquer method to find the prime factors of all integers in a given interval (1 to 1000 in this case). The recursive division of the interval has very fine source granularity; the bulk of the computation occurs with medium source granularity at the leaves. Lazy task creation successfully eliminates most of the tasks, although nT rises faster than it would with pure BUSD execution. Our current scheduler is fairly naive in choosing *which* processor to steal a task from; more attention to this area may produce a scheduler which more closely approximates an oldest-first stealing policy.

queens gives results for the 8 queens benchmark.

speech is a "real" program, part of a multi-stage speech understanding system being developed at MIT. This stage is essentially a graph-matching problem, finding the closest dictionary entry to a spoken utterance. The program has medium source granularity, so eager futures don't perform too badly and the improvement with lazy futures is modest.

The overhead of lazy future calls in all these benchmarks can be measured by comparing the sequential time to the lazy time with one processor. For these benchmarks, the overhead seems acceptably low.

6 Related Work

Load-based inlining has been studied previously in the Mul-T parallel Lisp system [14], and is also available in Qlisp by using `(local-deque-size)` or `(qemptyp)` to sense the current load [7, 18]. An analytical model of load-based inlining for programs like `psum-tree` has been developed by Weening [23, 24]. His analytical results generally agree with empirical observations of load-based inlining in both Mul-T and Qlisp; however, neither the prior Mul-T work nor the prior Qlisp work have explored the alternative of lazy task creation.

The potential for deadlock when using load-based inlining was described in [14], but the example of Section 3.3 is more plausible than the scenario painted in [14]. It is interesting to note that selective load-based inlining, as is possible in Qlisp, could be used by a sophisticated programmer to ensure that inlining is never performed where it might cause deadlock. However, this solution requires the programmer to accurately recognize all situations where the potential for deadlock exists, and still does not offer the other advantages of lazy task creation.

WorkCrews [22] is a package that does perform lazy task creation, intended for use with a fork-join or `cobegin` style of programming. It is implemented on top of Modula-2+ (an extension of Modula-2). For every task that is to be created lazily, a WorkCrews program calls `RequestHelp(`*proc*`,`*data*`)` and then proceeds with other work. A free processor looks for unanswered help requests, "steals" one, and applies its *proc* to its *data*. When the requestor finishes its other work, it calls `GotHelp` to see whether the `RequestHelp` task was stolen. If not, it proceeds to do the work itself; if so, it looks for other work to do. The performance of WorkCrews was evaluated on several parallel Quicksort programs and on `MultiGrep`, a program that searches for occurrences of a given string in a group of files [22].

The principal difference between WorkCrews-style lazy task creation and Mul-T's lazy futures is that invoking lazy task creation in WorkCrews requires a significantly larger amount of source code to be written—the work performed by *proc* must be broken out into a separate procedure, the argument block to be passed as *data* must be explicitly allocated and filled in, and finally the `RequestHelp` and `GotHelp` procedures must be called. Moreover, synchronization with and value retrieval from the lazily created task are explicit responsibilities of the programmer. By contrast, in Mul-T it is only necessary to insert the keyword `future` to begin enjoying the benefits of lazy task creation.

These stylistic differences lead to some implementation differences: our lazy future implementations directly manipulate implementation objects such as stack frames and are thus more "built in" to the implementation than in the case of WorkCrews. We think some efficiency improvements result from our approach, but the systems are different enough that it is hard to make a conclusive comparison. In any case, although the mechanics of the two systems are rather different, there is a very close relationship between their underlying philosophies.

Our philosophy of encouraging programmers to expose parallelism while relying on the implementation to curb excess parallelism resembles that of data-flow researchers who have been concerned with *throttling* [3, 19]. However, the main purpose of throttling is to reduce the memory requirements of parallel computations, not to increase granularity (which is generally fixed at a very fine level by data-flow architectures [2, 8]). Throttling thus serves the same purpose as our preference for depth-first scheduling and is not directly related to lazy task creation.

7 Conclusions and Future Work

We are encouraged that our performance statistics support the theoretical benefits of lazy task creation. A remaining problem, described in section 3.5, is that programs with data-level parallelism (for example where a fine-grained operation is applied to all members of a set) can have poor parallel performance when lists are used to represent sets. Other set representations such as arrays can increase program complexity because a coarse-grained re-coding is necessary to achieve good performance.

Our current plan is to provide a data abstraction for sets which supports global operations efficiently in parallel. Using such an abstraction would require a bit more work from the programmer, but far less than specifying the details of a decomposition directly in the source code. Two attempts in this direction with something of a SIMD flavor are the *zappings* of Connection Machine Lisp [21] and the Paralation model [20].

As with sequential algorithms, where the best set representation varies depending on the mix of operations, it is likely that no one data structure will be "right" for all parallel algorithms. But by analyzing the requirements of numerous algorithms we can provide a spectrum of representations. One such representation will certainly be tree-like; we know that lazy task creation will give us good parallel performance on a divide-and-conquer implementation of global set operations.

There is also the important issue of scalability. Both the Encore machine and the ALEWIFE simulation described assume that all memory references are of equal cost, an unreasonable assumption for a large-scale multiprocessor. We are investigating how our lazy task creation strategy can be augmented to take advantage of *locality* in shared-memory systems where the physical memory is distributed. In particular, the copying of the continuation frame when a task is stolen is very expensive in a distributed-memory system, so we are implementing a version of lazy task creation that avoids the copy. We are also implementing a scheduler that takes locality into account.

Because of their extra record-keeping burden, lazy future calls are unlikely ever to be as cheap as the cheapest implementation of normal calls, but the incremental cost of a lazy future call can be strongly influenced by a multiprocessor's hardware architecture. For example, the linked-frame implementation shown in Section 4.1 benefits greatly from the ALEWIFE architecture's support for full/empty bits in memory that can be accessed efficiently as a side effect of a load or store instruction.

Nevertheless, the linked-frame implementation still requires some memory operations for every call, and even a few more memory operations for every lazy future call. For architectures whose processors have register windows we have contemplated another approach with the potential of eliminating most memory operations: each register window could have an associated bit in a processor register indicating whether it is logically part of the lazy future queue, but only when a register window was unloaded due to a window overflow trap would the frame actually be linked into the in-memory data structure representing the queue. This would further reduce the cost of lazy future calls, since one might expect a large fraction of lazy future calls to return without their associated register window ever having been unloaded. However, some mechanism would have to be provided for querying a processor to see if it contains any stealable continuations, in the event that none are found in memory, and for interrupting a processor to request it to unload stealable continuations needed by other processors. The costs and benefits of this idea are not currently known.

The larger quest in which we have been engaged is to provide the expressive power and elegance of `future` at the lowest possible cost. Complete success in this endeavor would make it unnecessary for programmers ever to shun `future` in favor of lower-level, but more efficient, constructs. Success would also encourage programmers to express the parallelism in programs at all levels of granularity, rather than forcing them to hand-tune the granularity (at the source-code level) for the best performance. Lazy task creation moves us closer to this ideal, producing very acceptable performance and greatly reducing the number of tasks created for all of the benchmark programs of Section 5. And while the ideal may never be achieved completely, every step in the direction of making `future` cheaper increases the number of situations in which the cost of `future` is no bar to its use.

8 Acknowledgments

Thanks to Randy Osborne, Richard Kelsey and Paul Hudak for helpful comments on drafts of the paper, to Kirk Johnson for the speech application, and to the Sloan foundation, IBM, the Department of Energy (FG02-86ER25012) and DARPA (N00014-87-K-0825) for their support.

References

[1] Agarwal, A., Lim, B.H., Kranz, D. and Kubiatowicz, J. "APRIL: A Processor Architecture for Multiprocessing," *17th Annual Int'l. Symp. on Computer Architecture*, Seattle, May 1990.

[2] Arvind and D. Culler, "Dataflow Architectures," *Annual Reviews in Computer Science*, Annual Reviews, Inc., Palo Alto, Ca., 1986, pp. 225–253.

[3] Culler, D.E., "Managing Parallelism and Resources in Scientific Dataflow Programs," Ph.D. thesis, M.I.T. Dept. of Electrical Engineering and Computer Science, Cambridge, Mass., June 1989.

[4] Gabriel, R.P., and J. McCarthy, "Queue-based Multiprocessing Lisp," *1984 ACM Symp. on Lisp and Functional Programming*, Austin, Tex., Aug. 1984, pp. 25–44.

[5] Goldberg, B., "Multiprocessor Execution of Functional Programs," *Int'l. J. of Parallel Programming 17:5*, Oct. 1988, pp. 425–473.

[6] Goldman, R., and R.P. Gabriel, "Preliminary Results with the Initial Implementation of Qlisp," *1988 ACM Symp. on Lisp and Functional Programming*, Snowbird, Utah, July 1988, pp. 143–152.

[7] Goldman, R., R. Gabriel, and C. Sexton, "Qlisp: Parallel Processing in Lisp," paper presented at the U.S./Japan Workshop on Parallel Lisp, June 5–7, 1989, Tohoku University, Sendai, Japan.

[8] Gurd, J., C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer," *Comm. ACM 28:1*, January 1985, pp. 34–52.

[9] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. on Prog. Languages and Systems 7:4*, October 1985, pp. 501–538.

[10] Halstead, R., "An Assessment of Multilisp: Lessons from Experience," *Int'l. J. of Parallel Programming 15:6*, Dec. 1986, pp. 459–501.

[11] Hudak, P., "Para-Functional Programming," *Computer*, 19(8):60-71, August 1986.

[12] Hudak, P., and B. Goldberg, "Serial Combinators: 'Optimal' Grains of Parallelism," *Functional Programming Languages and Computer Architecture*, Springer-Verlag LNCS 201, September 1985, pp. 382–388.

[13] Hudak, P., and L. Smith, "Para-Functional Programming: a Paradigm for Programming Multiprocessor Systems," *12th ACM Symposium on Principles of Programming Languages*, January 1986, pp. 243–254.

[14] Kranz, D., R. Halstead, and E. Mohr, "Mul-T, A High-Performance Parallel Lisp", *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 81–90.

[15] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "Orbit: An Optimizing Compiler for Scheme," *Proc. SIGPLAN '86 Symp. on Compiler Construction*, June 1986, pp. 219–233.

[16] Kranz, D., "ORBIT: An Optimizing Compiler for Scheme," Yale University Technical Report YALEU/DCS/RR-632, February 1988.

[17] Moss, J.E.B., "Managing Stack Frames in Smalltalk," *Proc. SIGPLAN '87 Symp. on Interpreters and Interpretive Techniques*, June 1987, pp. 229–240.

[18] Pehoushek, D., "Low Cost Forks and Dynamic Control," paper presented at the U.S./Japan Workshop on Parallel Lisp, June 5–7, 1989, Tohoku University, Sendai, Japan.

[19] Ruggiero, C.A., and J. Sargeant, "Control of Parallelism in the Manchester Dataflow Machine," Springer-Verlag LNCS 274, *Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987, pp. 1–15

[20] Sabot, G., *The Paralation Model*, M.I.T. Press, 1988.

[21] Steele, G.L. Jr., and W.D. Hillis, "Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing," *1986 ACM Symp. on Lisp and Functional Programming*, Cambridge, MA, August 1986, pp. 279–297.

[22] Vandevoorde, M., and E. Roberts, "WorkCrews: An Abstraction for Controlling Parallelism," *Int'l. J. of Parallel Programming 17:4*, August 1988, pp. 347–366.

[23] Weening, J., "An Analysis of Dynamic Partitioning," paper presented at the U.S./Japan Workshop on Parallel Lisp, June 5–7, 1989, Tohoku University, Sendai, Japan.

[24] Weening, J., "Parallel Lisp Programs," Stanford Computer Science Report STAN-CS-89-1265, June 1989.