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

Eric Mohr
Yale University
mohr@cs.yale.edu

David A. Kranz
M.I.T.
kranz@lcs.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 [15], an efficient parallel implementation of Scheme, but could be used with other 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.

Key words and phrases: parallel programming languages, load balancing, program partitioning, process migration, parallel Lisp, task management.

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, 9], 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 [6, 13]. But when execution paths are highly data-dependent (as for example 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 mutation of shared variables 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 [5, 7, 8]. 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.

0

We've taken an intermediate position in our research on Mul-T [15], a parallel version of Scheme based on the `future` construct of Multilisp [10, 11]. 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 little 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 [12, 14] 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 [15], based on the T system's ORBIT compiler [16, 17], 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

(K `(future` X`))`

declares that a child computation X may proceed in parallel with its parent continuation K. In the most straightforward interpretation, a child task is created to compute X while the parent task computes K.[2] Reversing the task roles is also possible; the parent task can compute X while the child task computes K. Finally, and most importantly for fine-grained programs, it is also usually correct for the parent task to compute first X and then K, ignoring the `future`. 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.[3]

[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. The placeholder is said to be *unresolved* until X's value becomes available. Any task attempting to use the value of an unresolved future is suspended until the value is available. A *touch* is a use of a value V that will cause a task to be suspended if V is an unresolved future.

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

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, (K `(future` X`))` means "Start evaluating X in the current task, but save enough information so that its continuation K 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.

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)))))
```

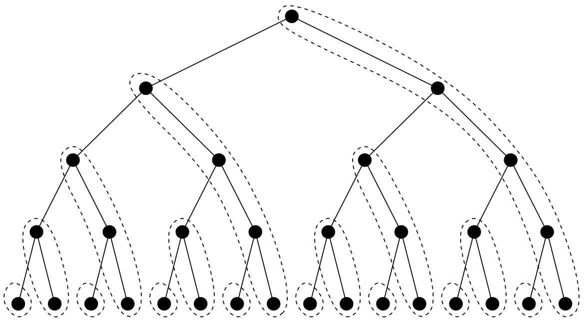


Figure 1: Direct execution of `psum-tree`.

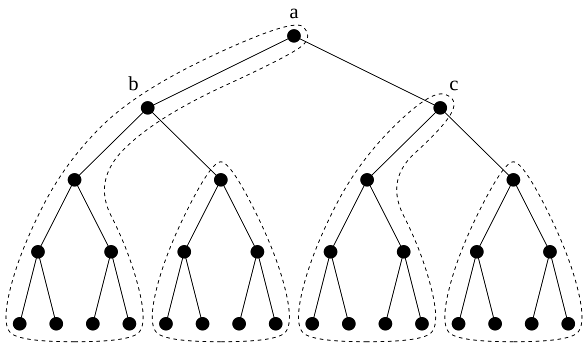


Figure 2: BUSD execution of `psum-tree` on 4 processors.

(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.

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.

```
(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)))))
```

Figure 3: Code for `psum-tree-2`

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)` [8], 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` as in Figure 3 (very similar to an example in [5]).

In this example, `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.

Load-based inlining and lazy task creation are both attempts to approximate the BUSD perfor-

[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.

mance of `psum-tree-2` without sacrificing the clarity of `psum-tree`. In an ideal run of `psum-tree` on a four-processor system with load-based inlining, the first three occurrences of `future` (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).

In an ideal run of `psum-tree` with lazy task creation, 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.

We now consider how these idealized execution patterns match up with real-life execution patterns for these methods.

3 Comparison of Dynamic Methods

Load-based inlining has an appealing simplicity and does in fact produce good results for some programs [15], but we have noted several factors which decrease its effectiveness. A major factor is that inlining decisions are irrevocable—once the decision to inline a task has been made there is no way to revoke the decision at a later time, even if it becomes clear at that time that doing so would be beneficial.

The following list summarizes the drawbacks of load-based inlining; the following sections discuss each in turn as a basis for comparing the two dynamic strategies.

1. The programmer must decide when to apply load-based inlining, and at what load threshold T.
2. Inlined tasks are not accessible; processors can starve even though many inlined tasks are pending.
3. Deadlock can result if inlining is used on some types of programs.
4. In an implementation with one task queue per processor, load-based inlining creates many more tasks than would be created with an optimal BUSD division.
5. Load-based inlining is ineffective in programs where fine-grained parallelism is expressed through iteration.

3.1 Programmer Involvement

Even though load-based inlining is an automatic mechanism it still requires programmer input. Some programs run significantly faster with eager task creation than they do with load-based inlining, so the programmer must identify where load-based inlining should be applied. For example, load balancing is crucial in a coarse-grained program creating relatively few tasks—inlining even a few large tasks can hurt load balancing by lengthening the "tail-off" period when processors are finishing their last tasks. With lazy task creation however, load balancing can't suffer because all inlining decisions are revocable. At worst, all lazily-inlined tasks will have their continuations stolen. But because the cost of stealing a task is comparable to that of creating an eager task, performance will not be significantly worse than with eager task creation. Thus lazy task creation can be used safely on such programs without the danger of degrading performance.

With load-based inlining, the programmer must also get involved by supplying a value for the load threshold T. Experience has shown that choosing the right value for T is crucial for good performance, but is difficult to do except by experimentation [26]. Since lazy task creation requires no parameterization the programmer is freed of this burden as well.

3.2 Irrevocability

The irrevocability of load-based inlining can mean that processors become idle even though the continuations of many inlined tasks have not yet begun to execute. Such problems can be caused by *bursty task creation* and *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 continuations of the inlined tasks are not available for execution. This problem never arises with lazy task creation because these continuations 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

Perhaps the most serious problem with load-based inlining is that, for some programs, *irrevocable inlining is not a correct optimization.* To see how inlining can lead to deadlock, consider the program in Figure 4 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 as they are found while reading primes from the front during divisor testing).

Initially, `all-primes` is 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.

Figure 5 shows different possible scenarios in the execution of `find-primes`. 5a shows that with eager task creation a separate future is created to test each value of n for primality. These futures could be scheduled in any order, but a future testing a large value of n might block when walking down the list of known primes if the futures testing small values of n were unresolved. 5b shows a possible scenario during eager execution; the futures for $n = 3, 5, 9, 13, 15, 17$ have been executed, while the futures for $n = 7, 11$ are still unresolved. No blocking is shown; for example, the future for $n = 17$ was able to run to completion because only the first two elements of the list of primes $(3, 5)$ were needed to determine that 17 is prime.

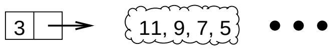
Figure 5(c) shows a possible execution snapshot with load-based inlining. Inlining a future in `find-primes` causes the parent task to test an additional value of n; if several successive futures are inlined, a task will test several values of n. 5c illustrates an important manifestation of load-based inlining in `find-primes`: because an inlined task (containing the recursive call to

[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.

(a) With eager task creation a future is created for each value of n.

(b) With eager task creation execution order is arbitrary, subject to data dependencies.

(c) With load-based inlining a single future may handle several values of n.



(d) Deadlock could occur with load-based inlining.

Figure 5: Possible execution scenarios for `find-primes`.

`find-primes>=n`) is executed before its continuation, a task testing several values of n *tests the largest value first.* This is the interaction that can cause deadlock, as would happen in the scenario of 5d. Here, inlining three futures has created a task T to test $n = 11, 9, 7, 5$, in that order. We expect T's ultimate value to be a list containing the elements $5, 7, 11$ (and another future representing the rest of the list). Let F be the future representing this value. The call to `prime?` for $n = 11$ will attempt to access the second element of `all-primes`, but will block because the second element is "inside" the unresolved future F! The second element won't be available until T itself gets around to testing $n = 5$, so deadlock has occurred.

This type of deadlock is not possible with lazy task creation because of the decoupling of blocked tasks mentioned above. Any inlined task can be separated from its parent, so programs that are deadlock-free with eager task creation are also deadlock-free with lazy task creation.

3.4 Too Many Tasks

The behavior of load-based inlining for programs like `psum-tree` has been analyzed by Weening [26, 27]. He assumes, as we do, that each processor maintains its own local task queue and that inlining decisions are based only on the local queue's length. He shows two ways in which the need to maintain at least one task on the local queue leads to non-BUSD execution. First, a lone processor P executing a subtree of height h creates h tasks instead of just one; second, removing a task

```
(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 4: Program `find-primes` could deadlock with load-based inlining.

from P's queue at an inopportune moment (a "transfer") can lead to the creation of $O(h^2)$ tasks. He derives an upper bound of $O(p^2h^4)$ tasks using p processors, and points out that this bound guarantees asymptotically minimal task creation overhead as the problem size grows exponentially in h. But this doesn't tell the whole story—asymptotically acceptable overhead may only be achieved when the problem size grows to a running time measured in days. In our experience, the overhead of task creation with load-based inlining is significant for problems of substantial size.

The bottom line is that load-based inlining with distributed task queues is unable to achieve oldest-first scheduling; many of the tasks created represent small subtrees. For example, consider what happens when a transfer removes a task from the queue of a processor P. The next time P encounters a future call, P will find that its queue is empty and so will create a new task to evaluate the call. But the position of T in the program's call tree is really a matter of chance, determined only by the timing of the transfer operation. Since the majority of potential fork points lie toward the leaves of the tree, T is likely to represent only a small subtree.

It is possible that using one central queue instead of several distributed queues would decrease the number of tasks, but the contention introduced by this alternative would probably be unacceptable and would certainly not be scalable. A much better alternative is the oldest-first scheduling policy of lazy task creation; as can be seen by the task counts in Section 5, lazy task creation results in many fewer tasks than load-based inlining. Tasks created by oldest-first scheduling are able to inline larger subtrees, giving a much better approximation to BUSD execution.

3.5 Fine-Grained Iteration

Not all parallel programs have bushy call trees; for example, some programs contain data-level parallelism expressed by iteration over a linear data structure. Load-based inlining is not effective in increasing the run-time granularity of such programs. To see why, consider the two versions of parallel `map` shown in Figure 6. Both versions apply a function `f` to every element of a list `l`, exemplifying the two methods of parallelizing an iterative loop.[6]

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

[6] This example involves recursion on lists; similar dynamics appear with iteration on arrays.

```
(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))))
```

Figure 6: Code for two versions of parallel `map`

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. Other approaches are possible; we have considered more complex dynamic methods as well as making use of compiler granularity information in cases like this.

But it may be too harsh to fault our dynamic task combination strategies for failing to improve the execution of programs where fine-grained data-level parallelism is expressed using iteration. The sequentiality of iteration inherently limits parallelism; even if task creation overhead were nonexistent, the parallelism in a loop like `parmap-cars` will never exceed t_f/t_l (the cost of computing `f` *vs.* the cost of one loop iteration) [11]. For a fine-grained loop this ratio represents a real limitation on the number of processors that can be kept busy.

This is a case where expressing an algorithm in the most convenient way inherently limits parallel performance. We mentioned our view that the programmer should identify *what* can be computed in parallel without worrying about *how* the division will take place at run-time. But iteration says a lot about the "how"—we can't increase performance if the programmer doesn't provide an algorithm with enough inherent parallelism.

One conclusion of this line of reasoning is that lists are a bad data structure to use for a program with fine-grained data-level parallelism because their very structure requires sequential, iterative-style access. Even if we knew in advance the optimal number of list elements handled by a single task, all elements still need to be traversed to create the tasks. The random-access nature of arrays make them a better choice for a division of this nature. But we'd still like to free the programmer from specifying explicitly how array elements should be chunked together in tasks.

One way of solving the "apply-to-all" problem within our philosophical framework is to use a divide-and-conquer division of an array's index set, as in `parmap-interval`, shown in Figure 7. (No array is visible here; the function `f` would use its index argument in an array calculation.) This method exposes abundant parallelism without requiring the programmer to specify an exact partition of array elements to tasks. And lazy task creation interacts well with this program's bushy call tree, approximating a BUSD partition at runtime. This is not a perfect solution though, as `parmap-interval` is somewhat more complex than the corresponding iterative loop. But the increase in complexity is small, and the program is free of parameterization. We discuss some ideas for improving on this solution at the end of the paper.

4 Implementation

We have seen that lazy task creation has several strong advantages over load-based inlining. We now explore the implementation issues to determine whether the overhead of lazy task creation can be acceptably minimized.

Both of our dynamic methods increase efficiency by ignoring selected instances of `future`. But 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

```
(define (parmap-interval f lo hi)
  (if (= lo hi)
      (f lo)
      (let* ((mid-lo (quotient (+ lo hi) 2))
             (mid-hi (+ mid-lo 1))
             (lo-half (future (parmap-interval f lo mid-lo))))
        (parmap-interval f mid-hi hi)
        (touch lo-half))))
```

Figure 7: Code for `parmap-interval`

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. Alternatively, we could use a linked-frame implementation where splitting a stack requires only pointer manipulations. However, care must be taken with such an implementation to ensure 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 conventional stack-based implementation for the Encore Multimax version of Mul-T as well as a linked-frame implementation for the ALEWIFE multiprocessor. The basic data structures and operations for lazy task creation are common to both implementations however, and are discussed next.

4.1 The Lazy Task Queue

Each task maintains a queue of stealable continuations called the *lazy task queue*, shown abstractly in Figure 8. When making a *lazy future call* corresponding to an instance of `future` in the source code, a task T first pushes a pointer to the `future`'s continuation onto the lazy task queue. If upon return the continuation has not been stolen by another processor, T dequeues it. We refer to T as the *producer* of lazy tasks; another processor stealing them is called a *consumer*. Consumers remove frames from the head of the lazy task queue while the producer pushes and pops frames from the tail.

Figure 8 tells a lazy task creation story for a producer task P. 8a shows P's stack (growing upward) with eight frames. Three of these frames are continuations to lazy future calls; pointers to these frames have been placed on the lazy task queue. Note that the oldest continuation is at the head (bottom) of the queue while the newest continuation is at the tail (top) of the queue. At this point a lazy future call occurs, corresponding to the following code:

(K `(future` X`))`

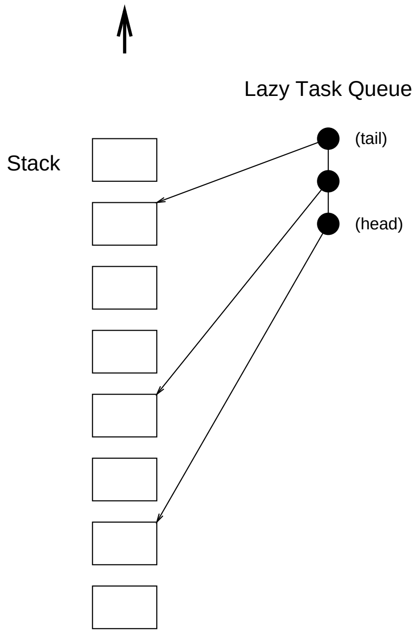
As shown in 8b, a new continuation K is pushed onto the stack, and a pointer to K is pushed on the lazy task queue. The inlined future completes execution of X before any stealing occurs, so P simply returns to K after first popping the lazy task queue (removing the pointer to K from the tail of the queue); this is shown in 8c.

Now an idle consumer C decides to steal a continuation from P's lazy task queue. To do this correctly, C must change P's stack to make it look as though an eager future had actually been created to compute X. C does this by creating a placeholder and modifying P's stack so that the value returned by the call to X will resolve (*i.e.*, supply a value for) the placeholder rather than being passed directly to the continuation K. The consumer then calls K itself, passing the unresolved placeholder as an argument. 8d shows the completed steal operation; it now looks as though an eager future had been created, with one processor (the producer P) evaluating the child X and another (the consumer C) evaluating the parent K. Note an important feature of the stealing operation: *the consumer never interrupts the producer.*

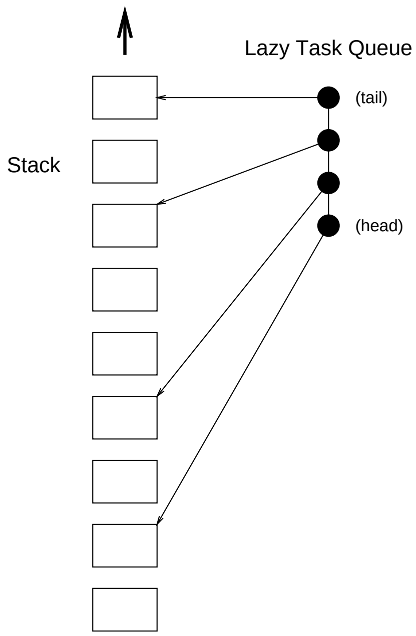
Implementations must take care to guard against two kinds of race conditions to ensure correctness of the stealing operation. First, two consumers may race to steal the same continuation; second, a producer trying to return to a continuation may race with a consumer trying to steal it.

4.2 Encore Implementation

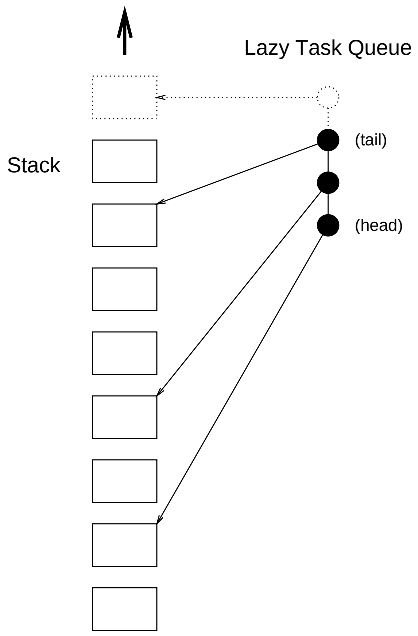
We have implemented lazy task creation in the version of Mul-T running on the Encore Multimax, a bus-based shared-memory multiprocessor. Our Multimax has 18



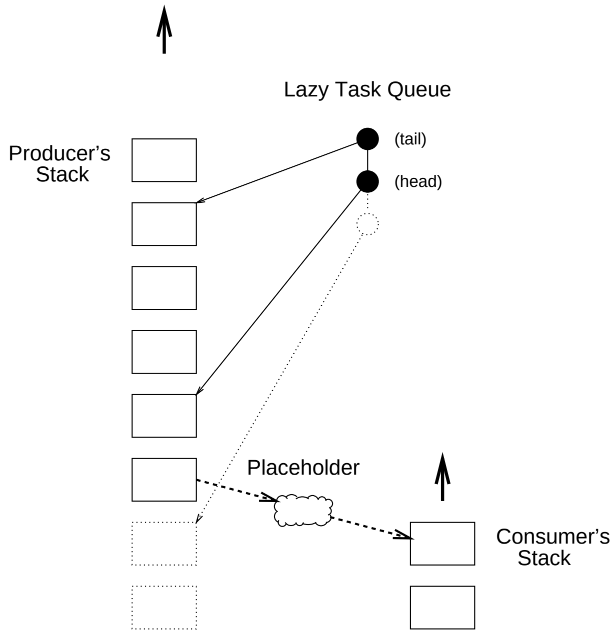
(a) Data structures for lazy task creation.



(b) A lazy future call causes a continuation to be queued.



(c) Returning from a lazy future call causes a continuation to be dequeued.



(d) A continuation is stolen.

Figure 8: Lazy task queue data structures and operations.

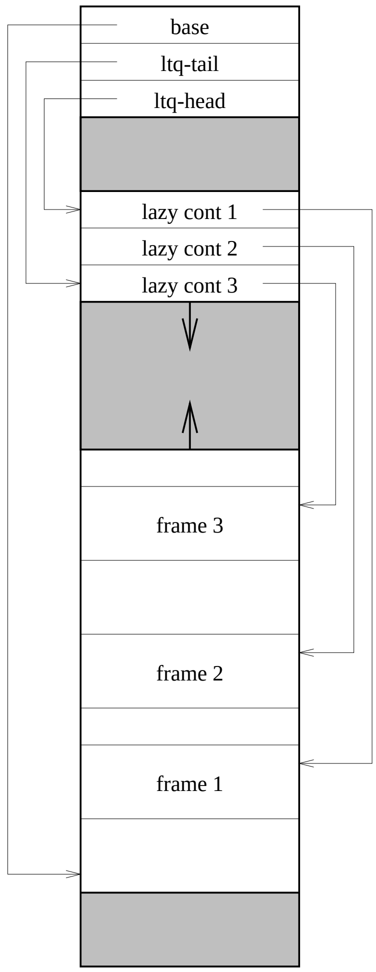


Figure 9: Lazy task queue implemented in conjunction with a conventional stack.

processors; the National Semiconductor 32332 processors used have relatively few general-purpose registers (8) but fairly powerful memory addressing modes. Synchronization between processors is possible only by using a test-and-set instruction which acquires exclusive access to the bus.

In this implementation stacks are represented conventionally, in contiguous sections of the heap. As seen in Figure 9, the lazy task 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 task queue and the bottom frames of the stack). When a stack overflows (*i.e.*, when the gap between stack frames and lazy task queue gets too small), it may either be repacked to reclaim space created by steal operations or its contents may be copied to a new stack of twice the original size.

To steal from the stack pictured, a consumer first locates the oldest continuation by following the `ltq-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 resolve 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 `ltq-head` appropriately.

To guard against the race conditions mentioned earlier there is a lock for the entire stack plus a lock for each continuation on the lazy task queue. Only the producer modifies `ltq-tail`, and only consumers modify `ltq-head` and `base`.

4.2.1 Lazy Future Call and Return

We now present the lazy task queue operations in somewhat more detail. Figure 10 gives assembler pseudocode showing how the expression

```
(g (future (f x)))
```

would be compiled in Encore Mul-T with lazy task creation. The lazy future call and return in this example show the crucial lazy task queue operations of enqueuing and dequeuing a lazy continuation.

The first block (`entry` and `call-g`) shows the compiled code for the lazy future call to `f` and its continuation, containing the standard call to `g`. `stack` is a pointer to the current stack; lazy task queue pointers such as `ltq-tail` are referenced via an offset to this pointer.[7]

The code shows that 2 longwords (4 bytes each) are allocated in the lazy task queue area of the stack for each lazy continuation—one for the continuation itself and one for a lock. After storing the continuation pointer `call-g` and initializing the lock to 0 we increment the `ltq-tail` pointer, which makes the lazy continuation available for stealing. There is no need to test explicitly for overflow of the lazy task queue; the stack overflow check on entry simply tests the size of the empty region between the actual stack (growing upwards) and the lazy task queue (growing downwards).

Before calling `f` we push `return-from-lf-call` on the stack as the return address. This is a shared, out-of-line routine that serves as the continuation to all lazy future calls. It is shown in the second block of code. Here we see synchronization to guard against interference by a consumer trying to steal the same lazy continuation the producer is trying to return to. The returning producer first acquires the lazy task queue item lock (using the Encore's interlocked test and set instruction), busy-waiting if the lock is currently held

[7] This is a slight simplification; in actuality, the current stack is stored in a block of data kept locally by each processor; `ltq-tail` is referenced using the double indirection capability of the NS 32332.

```
    (lambda (x)
      (g (future (f x))))

entry:
 Standard stack overflow test (3 instructions).
 push-addr   call-g                 # push return address (a.k.a. current continuation) on stack
 move        ltq-tail(stack),r1     # get pointer to tail of lazy task queue
 move        sp,8(r1)               # store pointer to stack continuation in lazy task queue
 move        $0,12(r1)              # initialize lazy task queue entry lock
 add         $8,ltq-tail(stack)     # lazy continuation officially enqueued
 push-addr   return-from-lf-call    # call to f will return to return-from-lf-call
 Standard call to unknown procedure f (5 instructions).
call-g:
 Standard continuation code, including call to unknown procedure g (6 instructions).

return-from-lf-call:
 move        ltq-tail(stack),r1     # get pointer to lazy task queue tail
 test&set    4(r1)                  # try to lock tail item of lazy task queue
 br-if-clr   pop-ltq                # if successful, go pop it
 Busy-wait loop to lock tail item of lazy task queue.
pop-ltq:
 sub         $8,ltq-tail(stack)     # lazy continuation officially dequeued
 adjust-sp   $-4                    # remove return-from-lf-call address from stack
 Standard return (2 instructions).
```

Figure 10: Assembler pseudo-code showing lazy future call and return in the Encore implementation.

by a consumer. Once the lock is acquired the return address on top of the stack is guaranteed to be valid; in this case it will be either the original value `call-g` or else `resolve-placeholder` if the continuation has been stolen. After dequeuing the tail entry of the lazy task queue we return normally.

If, as is usually the case, the continuation to a lazy future call is known (*i.e.*, unless `future` appears in tail-call position), the code shown in Figure 10 can be streamlined by generating the `return-from-lf-call` code in line. This optimization, which saves 4 instructions (and increases the code size slightly), has not yet been implemented in the current system.

4.2.2 Steal Operation

Figure 11 gives the algorithm for stealing a lazy continuation from another processor's lazy task queue. The task to be stolen is chosen by a round-robin search of other processors' lazy task queues. Two locks must be acquired before a continuation is stolen—the producer's stack is locked to avoid races with other consumers and the continuation itself is locked to avoid a race with the producer trying to return to it.

Once a stealable continuation has been chosen and the necessary locks obtained, we replace it in the producer's stack with a continuation to resolve the newly created placeholder, and we update the producer's `base` and `ltq-head` pointers. At this point the producer's stack is in a consistent state, so we unlock the head item of the lazy task queue.[8] Then the bottom of the producer's stack is copied to the consumer's stack (taking care to use the *old* continuation rather than the newly swapped-in one!) and the consumer can begin executing the stolen continuation, passing the placeholder as an argument. The producer (or another processor if further stealing occurs!) will eventually return to our swapped-in continuation, providing a value for the placeholder.

4.2.3 Blocking

There is one remaining loose end in this discussion: what happens to the lazy task queue when a task T blocks by touching an unresolved future? It is not sufficient to save the lazy task queue as part of T's state because the queued lazy tasks would become inaccessible. We would then have the same potential deadlock problem that arises with load-based inlining.

The simple solution adopted here is for T to "bite its tail." T's stack is split above the most recent lazy con-

[8] The producer's stack is not unlocked at this point because of the possibility of stack overflow—the repacking operation discussed earlier would conflict mightily with a stealer's copying operation.

- Allocate and initialize data structures: a placeholder P, a new task object T_2, and a new stack S_2.
- Look for a continuation to steal.
  - Poll other processors to find one whose current stack S_1 has a non-empty lazy task queue (*i.e.* `ltq-tail` $\geq$ `ltq-head`).
  - Try to lock stack S_1; if it's already locked, skip to next processor.
  - Try to lock head item of S_1's lazy task queue Q; if it's already locked, skip to next processor.
- Steal the continuation. In the head item (now locked) of Q is a pointer CP into the stack S_2. CP points to a stack frame C representing a stealable continuation. The bottom of the stack (the portion between CP and S_1's `base` pointer) must be copied to the new stack S_2.
  - Replace C in S_1 with the continuation (`resolve-placeholder` P).
  - Update `base` and `ltq-head` pointers in S_1.
  - S_1 is now in a consistent state; unlock head item of Q.
  - Copy bottom portion of S_1 into S_2.
  - Unlock stack S_1.
  - "Return" to top continuation in new stack S_2, passing placeholder P as the argument.

Figure 11: Algorithm for steal operation in Encore implementation.

tinuation (at the tail of the lazy task queue), and only the top piece is blocked along with T. As with a steal operation, a placeholder is created to communicate a value between the two pieces of the split stack. The executing processor P can continue using the other piece of the stack, which contains all of the continuations on the lazy task queue. No lazy tasks are inaccessible. P dequeues the tail lazy continuation and returns to it, passing the placeholder as an argument.

In essence, P has stolen a task from the *tail* of T's lazy task queue. One problem with this solution is that it goes against our preference for oldest-first scheduling, since we have effectively created a task at the newest potential fork point. Performance can suffer because this task is more likely to have small granularity. And further blocking may result, possibly leading to the dismantling of the entire lazy task queue. An improved solution which avoids these problems has been implemented for ALEWIFE, and is discussed in the next section.

4.3 ALEWIFE implementation

The Encore implementation of lazy task creation just described performs reasonably well by lowering the overhead of using the `future` construct, but it still has several other sources of overhead:

1. Strict operations doing `future?` checks on their operands.
2. Checking for stack overflow.
3. Use of a global resource (the bus) for locking operations.

The ALEWIFE machine—a cache-coherent machine being developed at MIT with distributed, globally shared memory—is designed to address these problems. Its processing elements are modified SPARC[9] chips [1]. The modifications of interest here are fast traps for strict operations on futures and support for full/empty bits in each memory word. If a strict arithmetic operation or memory reference operates on a future a trap occurs; thus, explicit checks are not needed. The full/empty bits allow fine-grained locking: ALEWIFE includes memory-referencing instructions that trap when the full/empty state of the referenced location is not as expected.

For the ALEWIFE implementation of lazy task creation, a stack is represented as a doubly linked list of stack frames in order to minimize copying in the stealing operation [19]. In this scheme, each frame has a link to the previously allocated frame and another link to the next frame to be allocated. Thus push-frame and pop-frame operations are simply load instructions. 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. The "next frame" link is set to empty when no next frame has been allocated. This strategy avoids the need to check explicitly for stack overflow when doing a push-frame operation: if no next frame is available the push-frame operation will trap and the trap handler will allocate a new frame.

An earlier version of this paper [18] described an initial ALEWIFE implementation. In that version, stealing a lazy task involved copying the topmost stack frame. The version described here avoids this copying and also fixes a subtle bug in the original version.

Each frame is divided into two separate data structures, referred to as the *stack frame* and the *frame stub*. The stack frames form a doubly linked list as described

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

at the beginning of this section. Each stack frame contains local and temporary variables as in an ordinary stack frame. In addition, it contains a pointer to its associated frame stub. Each frame stub also has a pointer back to its associated stack frame. The reason for separating these two structures will become clear later.

In this implementation the lazy task queue is threaded through the frame stubs. Figures 12–15 show the lazy future call and stealing operations graphically. In these figures we use the following register names:

`FP` Frame pointer register. Points to the current stack frame (not frame stub).

`LTQT` Lazy task queue tail register. Modified only by the producer. Points to the current frame stub.

`LTQH` Head of the lazy task 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:

`next` This slot points to the "next" frame, which will become current if a stack-frame push operation is performed. The push-frame operation is thus performed simply by loading `next[FP]` into `FP`. If the next frame has not yet been allocated, `next[FP]` is marked as empty.

`cont` This slot points to the "continuation" frame, which will become current if a stack-frame pop operation is performed. The pop-frame operation is thus a load of `cont[FP]` into `FP`.

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

`lf-frame` This slot points to the associated frame stub.

Each frame stub has the following slots:

`ltq-next` This slot points to the next frame stub on the lazy task 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.

`ltq-prev` This slot points to the previous frame stub on the lazy task queue (toward the head of the queue).

`ltq-link` The lazy future call code stores in this slot the return address that the consumer should use if it steals this frame's continuation. If the continuation is stolen, the consumer reads out this return address and replaces it with the placeholder object it creates.

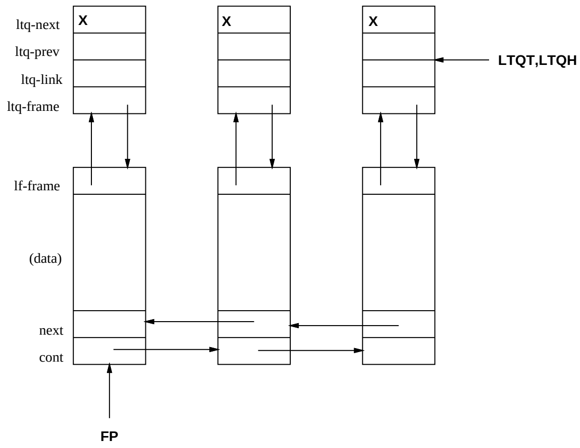


Figure 12: Just before lazy future call.

`frame` This slot points to the associated stack frame.

In this implementation, every call—whether a lazy future call or an ordinary procedure call—is preceded by a push-frame operation and followed by a pop-frame operation. This contrasts with the more common approach of pushing a frame upon procedure entry and deallocating it at procedure exit; further comments on this subject appear at the end of this section.

Figure 12 shows how the stack frames and relevant registers might look just before a lazy future call (but after that call's push-frame operation has already occurred). Note that each stack frame's `next` pointer points to the next frame toward the top of the stack and each `cont` pointer points to the next stack frame toward the bottom of the stack. If a memory location'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 12 could either be empty or point to another, currently unused frame. An "X" in the left-hand part of a frame slot (see, for example, the `ltq-next` slots in Figure 12) indicates that the full/empty bit of the corresponding memory word is set to "empty."

The lazy task queue in Figure 12 has no frames in it. A consumer would discover this by seeing that the `ltq-next` slot of the frame stub pointed to by `LTQH` is empty—if this task had stealable frames, this slot would point to the first such frame.

Figure 13 shows the situation just after the lazy future call. The frame stub associated with the current stack frame (pointed to by `FP`) has joined the lazy task queue. Accordingly, `LTQT` has changed to point to that frame stub, and the `ltq-next` and `ltq-prev` links have been updated as needed to maintain the doubly linked lazy task queue. Note that the rightmost frame stub in Figure 13 is not logically part of the lazy task queue—it is serving as a convenient header object for the doubly

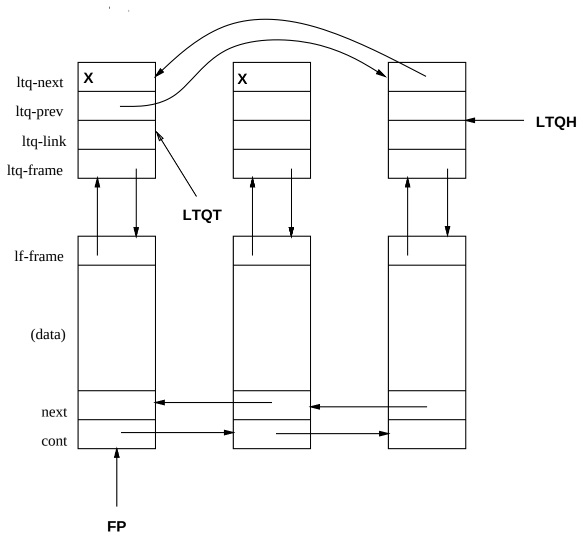


Figure 13: Just after lazy future call.

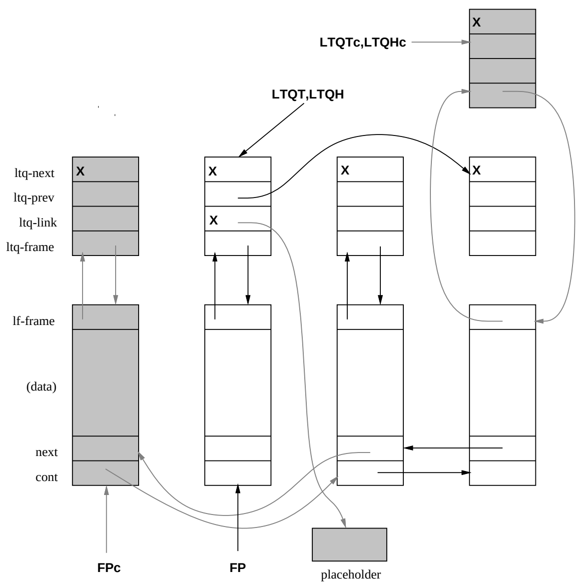


Figure 14: After steal; new structures shaded.

linked queue. The middle frame is also not part of the lazy task queue; it is simply part of the stack. The current frame stub's `ltq-link` field contains the address for the lazy future call's continuation, as required.

If no consumer steals this continuation, then this lazy future call will eventually return. The code for the return will restore the state of affairs depicted in Figure 12, after which the pop-frame operation associated with the lazy future call can be performed.

Figure 14 shows the state of the producer and consumer tasks if instead a consumer steals the continuation from the task shown in Figure 13. The consumer task's state variables are shown with a `c` appended, as in `LTQHc`. The shaded areas and shaded arrows show structures that have been created by the consumer. An

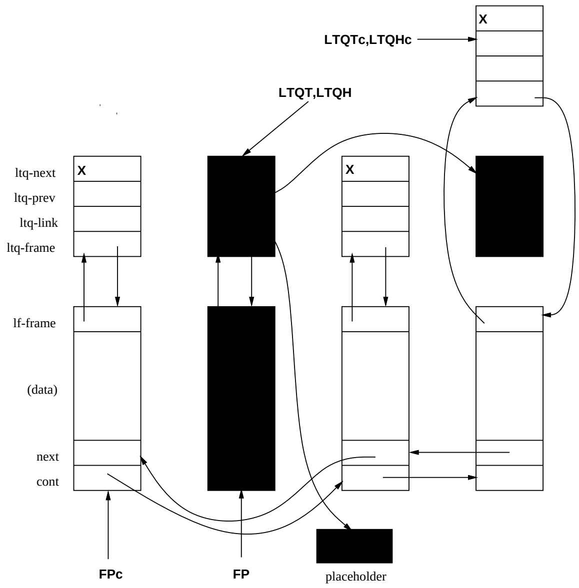


Figure 15: After steal; frames belonging to producer shown in black.

alternate view of this situation is shown in Figure 15. Note that the consumer's stack (the part that is not blacked out in Figure 15) now looks just like the producer's stack did in Figure 12 just before the original lazy future call (and just like the producer's stack would have looked in the case of a normal return from the lazy future call). Effectively, the consumer has "taken over" the continuation, created a placeholder to stand for the value of the called computation (which is still being performed by the producer), and forced an early return from the lazy future call, supplying the placeholder as the call's returned value. (No arrow is shown from any of the consumer's data structures to the placeholder because that value is returned in one of the consumer's registers.)

The consumer has also made the producer's `ltq-link` field point to the newly created placeholder. When the producer completes its computation and finds that its continuation has been stolen, it looks here to find the placeholder that should resolve to this computation's value. The synchronization here is unusual in that `ltq-link` is marked "empty" even though it contains useful data. This technique handles close races between a returning producer and a stealing consumer. By inspecting `ltq-next` and `ltq-prev` pointers, a returning producer can discover that its continuation has been stolen before the consumer has actually stored the placeholder in the `ltq-link` field. Correct operation is ensured by having the consumer set the `ltq-link` field's "empty" flag when the placeholder is installed, and having the producer wait for this "empty" flag before attempting to read out the placeholder.

A producer returning from a lazy future call distinguishes between the situations shown in Figures 13 and 14 by locating the frame stub F pointed to by the `ltq-prev` field of the frame stub pointed to by `LTQT` and looking at the `ltq-next` field of F. In Figure 14, where the continuation has been stolen, this field in F is empty; in Figure 13, where the continuation has not been stolen, it is not.

The algorithm for lazy future calls is spelled out in more detail in the pseudo-code shown in Figure 16. The in-line code for a lazy future call starts at the label `lf-call`; the code at `stolen` is out-of-line code shared by all lazy future calls. The algorithm for a consumer to find and steal a continuation is given in Figure 17.

Since the producer is not explicitly notified when a steal operation is performed on its stack, any resources the producer may continue to use after a continuation is stolen may not be used by the consumer. Some complexity in the algorithm for stealing results from this fact. In particular, the consumer must copy the rightmost frame stub in Figure 14 so it can use the `ltq-next` slot in that frame stub when it performs lazy future calls. If this frame stub were shared with the producer, such calls by the consumer could confuse the producer.

This approach has the drawback that a push-frame operation occurs at every procedure call (lazy or not) instead of at the entry point of a procedure, but there are several mitigating factors:

1. Push-frame and pop-frame operations are inexpensive (one instruction).
2. Compiler optimizations can eliminate some of them (*e.g.*, pop-push sequences that cancel out can be detected and eliminated).
3. Some push-frame operations at procedure entry turn out to be unnecessary due to conditional branches; this approach delays them until they are sure to be necessary.

We do not know the net effect of using this approach but we believe that the difference is not significant.

Finally, we return to the issue of what to do with the lazy task queue when a task blocks on an unresolved future. To preserve both oldest-first scheduling and laziness in task creation we would like to make the lazy task queue accessible for normal stealing by consumers. This is accomplished by placing the entire blocked task, lazy task queue and all, on the task queue of an appropriate processor.[10] Consumers may steal either from a task that is actually running or from a queued blocked task; a processor may steal from the lazy task queue of one of its own blocked tasks if it runs out of other useful work. This solution addresses the problems raised in Section 4.2.3.

[10] Of course, the task is marked as blocked, so the processor will not attempt to run it.

1. Select a processor for inspection and load its `LTQH` pointer into a register H, leaving `LTQH` empty. If `LTQH` is found already empty, move on to another processor. (This enforces mutual exclusion among consumers.)
2. Load `ltq-next[`H`]` into a register F, leaving `ltq-next[`H`]` empty. If `ltq-next[`H`]` is found already empty, then this processor's lazy task queue is empty—write H back into `LTQH` and move on to another processor.
3. Store F into `LTQH`. This step commits the steal operation and ends the exclusion of other consumers. Other consumers can now steal other continuations from this processor, even as this consumer continues its steal operation.
4. Load `ltq-link[`F`]` into a register *retpc*. This is the consumer's return address from the lazy future call.
5. Create a placeholder object and save its address in a register *retval*.
6. Store *retval* into `ltq-link[`F`]`, leaving `ltq-link[`F`]` empty. If the producer is already trying to return to the continuation being stolen, this step frees the producer to proceed and deposit its result into the placeholder.
7. Create the stack frame and the two frame stubs shown shaded in Figure 14 and link them together as shown by the shaded arrows in Figure 14. These data structures are accessed only by this consumer and hence neither the producer nor other consumers need to synchronize with these operations.
8. Set the `FPc`, `LTQTc`, and `LTQHc` pointers properly and execute a return to the address *retpc*, giving *retval* as the returned value.

Figure 17: Algorithm for steal operation in ALEWIFE implementation.

4.4 Discussion

What are the advantages and disadvantages of these implementations? The main disadvantage of the conventional stack implementation is the copying it performs. 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.

```
lf-call:
 load    next[FP],FP               # Push stack frame.
 load    lf-frame[FP],temp         # Address of new frame stub.
 store   $continue,ltq-link[temp]  # PC for consumer's return.
 store   LTQT,ltq-prev[temp]       # Make lazy task queue backward ...
 store   temp,ltq-next[LTQT]       # ... and forward links.
 move    temp,LTQT                 # Advance lazy task queue tail pointer.
 Call the procedure.
 load    ltq-prev[LTQT],temp       # Dequeue from lazy task queue tail,
 empty   ltq-next[temp]            # trap to stolen if continuation stolen.
 move    temp,LTQT                 # Reset lazy task queue tail pointer.
continue:
 load    cont[FP],FP               # Pop stack frame.

stolen:
 Wait for ltq-link[LTQT] to be empty.
 load-e  ltq-link[LTQT],temp       # Get placeholder to resolve.
 Resolve the placeholder in temp to the value returned by the procedure.
 Terminate the current task and find new work to do.
```

Figure 16: Assembler pseudo-code showing lazy future call and return in the ALEWIFE implementation.

While this is technically true it is somewhat misleading; the overhead of copying when stealing a 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 significant in comparison.

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

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

Stealing the first lazy task's continuation 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 task, so the next steal must copy the same information again. In fact, spreading work to n processors in this example via lazy tasks 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 task 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 17 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. The Mul-T run-time system, as well as code for the benchmarks, are compiled to SPARC instructions that 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.[11]

5.1 Comparison to Sequential T

When assessing the performance of a multiprocessor system it is important to make comparisons with the "best" sequential implementation. This assessment can be done in two steps:

1. How does the parallel version running on one processor compare to the sequential version?
2. How much does performance improve as processors are added?

For the first step we compare Mul-T with T3.1, which is not the best possible sequential implementation but is close enough for our purposes [17]. The second step is considered in Section 5.3.

For the Encore implementation, the sequential comparison shows the overhead due to compiler-inserted `future?` checks on strict operations. Although the Encore implementation is engineered to minimize future-checking overhead [15], the cost can be significant for some programs. Table 1 compares running times of several sequential programs[12] in T3.1 with the same program run in Mul-T on one processor. The Mul-T programs run between 1.4 and 2.2 times as long as their T3.1 counterparts.

ALEWIFE's hardware traps eliminate future-checking overhead, so "sequential" and "parallel" times on one processor are identical.[13] Thus speedup figures can measure how close we get to the ideal of linear speedup compared to the sequential version.

[11] Minimizing memory-referencing delays is crucial to good performance in a distributed-memory machine. ALEWIFE's distributed caching scheme [3] reduces the need for remote references; preliminary results of current research at MIT on a new scheduler for ALEWIFE Mul-T show good performance even with simulation of network delays.

[12] These programs are described either in Section 5.3 or in [15].

[13] It is interesting to note that the presence of hardware tag checking may be more significant in machines supporting parallel Lisp than in machines supporting sequential Lisp.

	Time (seconds)		
Program	Mul-T	T	ratio
`fib`	0.24	0.12	2.00
`queens`	1.07	0.74	1.45
`mergesort`	1.82	0.99	1.84
`speech`	95.9	43.4	2.21
`compiler`	159	98	1.62
`permute`	11,600	8,500	1.36

Table 1: Comparison of Running Times for Encore Mul-T and T3.1.

5.2 Cost of Lazy Task Queue Operations

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

For the Encore, 4 instructions could be eliminated by using the compiler optimization mentioned in section 10, 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 instructions. 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.

The cost of stealing a continuation from another processor's task queue is not as critical, since steals are relatively rare. As seen below, stealing a task in the Encore implementation has comparable cost to creating an eager future. Stealing a task in the ALEWIFE implementation is noticeably cheaper; the linked-frame stack implementation allows a much cleaner steal.

Machine / Operation	Number of Instructions
Encore / Eager Future	118
Encore / Steal	150 + 4 per word copied
ALEWIFE / Steal	100

These instruction counts include all aspects of creating and executing a task; *e.g.*, allocating and initializing placeholder, task, and stack objects, queueing and dequeuing the task, and resolving the placeholder.

5.3 Benchmarks

We begin our discussion of actual Mul-T programs with the synthetic benchmark `grain`, designed to measure the effectiveness of the various task-creation strategies over a range of task granularities. `grain` adds up a perfect binary tree of 1's using a parallel divide and conquer structure very similar to `psum-tree`, but before returning 1 at any leaf it executes a delay loop of a specified length, allowing granularity control. By timing trials using a range of granularities we can get an "efficiency profile" for each task-creation strategy. The efficiency E for a given trial is calculated using the formula

$$E = \frac{T_s}{nT_p}$$

where in this case the sequential time T_s is for a Mul-T program without futures and the parallel time T_p was measured using $n = 16$ processors. Efficiency of 1.0 means perfect speedup. The tree depth of 16 (65,536 1's) used in these trials ensures that processor idle time at start-up and tail-off is minimal, so close-to-perfect speedup should be achievable.

The granularity figures across the top of Table 2 tell how many NS-32332 instructions were used at the leaves to execute the delay loop and return 1; they do not include the instructions which implement the basic divide and conquer loop. The average source granularity is actually half of the given figure because internal nodes of the tree (where no delay loop is executed) account for half of the futures in this program. The instruction counts would be different for ALEWIFE due to its RISC instruction set, but because the source code is the same the efficiency figures are roughly comparable.

As expected, the high cost of eager task creation leads to poor efficiency at fine granularities. Load-based inlining performs somewhat better, although perhaps not as well as one might expect. The main problem is that with 16 processors far too many tasks are created; roughly 20-30% of the possible total of 65,536. We see here the impact of the mechanism discussed in Section 3.4. With lazy task creation less than 1% of future calls are converted to actual tasks, resulting in much better performance. Still, the overhead of lazy future calls is significant, hurting efficiency at the finest granularities. The lower overhead of lazy future calls in ALEWIFE leads to somewhat greater efficiency.

Table 3 shows performance statistics for several Mul-T programs, allowing comparison of speedup under each task creation strategy. The column labelled "seq time" gives the elapsed time for running the benchmark in Mul-T on one processor with all futures removed. Times are given in seconds for Encore and in 1000's of simulated SPARC cycles for ALEWIFE. The remaining columns show the relative speedup when the parallel benchmark is run using 1, 2, 4, 8, and 16 processors. For load-based inlining ("LBI"), the load threshold T was chosen in each case to give the fastest time on 16 processors.

Table 4 shows the number of tasks created under each strategy. The header for each benchmark shows the maximum number of tasks possible for that benchmark, which equals the number produced with eager task creation.

`fib-20` is the standard brute-force doubly recursive program for computing the nth Fibonacci number ($n = 20$ in this case). This program is very fine-grained; extremely little computation is performed between `future` calls. With eager task creation the overhead of creating futures completely overwhelms the calculation—even with 16 processors the sequential time is not improved on. Load-based inlining eliminates much of the overhead, but still creates many more tasks than an ideal BUSD execution would create. Lazy task creation produces a much better approximation to BUSD, as shown by the smaller number of tasks. At such a fine granularity the overhead of lazy future calls is significant, as can be seen by the 1-processor "speedup" column.

`queens` finds all possible solutions to the well-known 8 queens problem. A queen is placed on one row of the board at a time; each time a queen is legally placed, `future` appears around a recursive call to find all possible solutions stemming from the current configuration. The source granularity of `queens` is not particularly fine, but lazy task creation still makes a noticeable improvement. The 1-processor "speedup" column in Table 3 gives an indication of the overhead of task creation for each strategy. The eager number (0.69) shows that the cost of creating eager futures is significant for this program. The number for load-based inlining (0.99) is much better because the overhead of load-based inlining is very small when no task is created. With lazy task creation, the higher cost of provisionally inlining a task leads to a slightly smaller number (0.95). But lazy task creation substantially reduces the number of tasks created, allowing the performance increase when many processors are used.

`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. It uses a method resembling `parmap-interval`—a divide-and-conquer division of the dictionary where each leaf compares the utterance against one dictionary entry. The granularity of this comparison is rather coarse, so eager task creation doesn't perform too badly and the improvement with lazy task creation is modest.

`grain`—Efficiency										
	Leaf Task Granularity (number of NS-32332 instructions)									
Machine / Strategy	6	12	24	48	96	192	384	768	1536	3072
Encore / Eager	.07	.07	.09	.12	.18	.29	.45	.64	.78	.88
Encore / LBI ($T = 1$)	.19	.21	.25	.29	.39	.56	.66	.83	.88	.94
Encore / Lazy	.56	.59	.65	.73	.78	.87	.92	.95	.97	.99
ALEWIFE / Lazy	.74	.78	.82	.86	.91	.95	.97	.98	.99	1.00

Table 2: Efficiency of implementations for synthetic grain benchmark.

`fib-20`							
	seq	Relative Speedup					
Machine / Strategy	time	seq	1	2	4	8	16
Encore / Eager	.24	1.00	0.09	0.18	0.34	0.60	0.83
Encore / LBI ($T = 2$)	.24	1.00	0.69	1.09	1.60	2.40	3.43
Encore / Lazy	.25	1.00	0.63	1.25	2.27	4.17	6.25
ALEWIFE / Lazy	318	1.00	0.71	1.38	2.65	4.84	8.06

`queens`							
	seq	Relative Speedup					
Machine / Strategy	time	seq	1	2	4	8	16
Encore / Eager	1.07	1.00	0.69	1.35	2.68	5.10	8.23
Encore / LBI ($T = 1$)	1.07	1.00	0.99	1.95	3.45	5.63	8.92
Encore / Lazy	1.01	1.00	0.95	1.91	3.61	6.73	11.22
ALEWIFE / Lazy	1282	1.00	0.97	1.93	3.72	6.93	12.53

`speech`							
	seq	Relative Speedup					
Machine / Strategy	time	seq	1	2	4	8	16
Encore / Eager	95.9	1.00	0.92	1.79	3.45	6.44	10.90
Encore / LBI ($T = 1$)	95.9	1.00	1.00	1.92	3.62	6.27	9.05
Encore / Lazy	96.4	1.00	0.99	1.96	3.74	6.69	10.37
ALEWIFE / Lazy	85K	1.00	1.00	1.94	3.66	6.56	10.94

Table 3: Performance of Mul-T benchmarks (absolute times are in seconds for Encore and in 1000's of SPARC cycles for ALEWIFE).

	`fib` (10945 eager)			`queens` (2056 eager)			`speech` (39856 eager)		
	Encore		ALEWIFE	Encore		ALEWIFE	Encore		ALEWIFE
n	LBI(2)	LTC	LTC	LBI(1)	LTC	LTC	LBI(1)	LTC	LTC
1	172	0	1	5	0	1	1378	0	0
2	587	5	4	62	9	9	6173	583	613
4	1140	23	15	371	27	37	12577	1907	1946
8	1693	58	54	702	71	106	18011	4440	4807
16	2083	106	118	817	238	183	21667	7875	9930

Table 4: Number of Tasks Created in Mul-T benchmarks

For all 3 benchmarks, Table 4 shows the effects of the mechanism discussed in Section 3.4; lazy task creation gives a consistently better approximation to BUSD execution than does load-based inlining. The task counts for lazy task creation are rather variable from run to run due to the variable timing of steal operations; we do not believe that the differences between Encore and ALEWIFE task counts reflect significant differences in the implementations.

6 Related Work

Load-based inlining has been studied previously in the Mul-T parallel Lisp system [15], and is also available in Qlisp by using `(deque-size)` or `(qemptyp)` to sense the current load [8, 26]. An analytical model of load-based inlining for programs like `psum-tree` has been developed by Weening [26, 27]. 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.

Pehoushek and Weening [26] also present a strategy which reduces task creation overhead when a queued task is executed by the processor that created it. This strategy takes advantage of the same phenomenon that lazy task creation leverages: that when parallelism is abundant most tasks are executed locally. Executing such tasks with lazy task creation appears to be cheaper than with their scheme; furthermore, their scheme only works in programs with a fork/join style of parallelism. Lazy task creation has no such restriction, interacting well with the unlimited lifetime of futures in Mul-T.

The potential for deadlock when using load-based inlining was described in [15], but the example of Section 3.3 is more plausible than the scenario painted in [15]. 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 [23] 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 requester 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 [23].

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* [4, 20]. 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, 9]). 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. For programs with bushy call trees the programmer can use `future` to identify parallelism, effectively ignoring granularity considerations. One group of programs that deserve further study are those with fine-grained parallelism expressed iteratively (see Section 3.5). Our suggested solution finesses the problem by rewriting such programs to have bushy call trees, but a higher-level solution involving less work from the programmer would be preferable. We envision developing a data abstraction for aggregates and a set of high-level operations with efficient underlying parallel implementations, perhaps following some of the ideas in Connection Machine Lisp [22] or the Paralation model [21].

Such a high-level language for manipulating aggregates would need to exhibit the following properties:

- Powerful enough to express common patterns of parallel computation over aggregates.
- Translatable to efficient parallel code.

There are really two parts to the efficiency question, two rather different sources of overhead to worry about. Certainly we must minimize parallel processing overheads due to task creation, idle processors, and communication (*i.e.*, weighing the conflicting needs for good partitioning, load balancing, and locality). Lazy task creation addresses some of these issues, although more work is definitely needed. But before worrying about these overheads we have to first deal with the considerable overhead introduced by using high-level constructs (functional expressions on data aggregates) in the first place. If in abstraction we lose too much efficiency we won't be able to regain the cost with parallelism.

For this second efficiency issue, the work of Waters [24, 25] seems relevant. He observes that programming with high-level functional expressions on data aggregates is very powerful, but that such approaches are typically much less efficient than writing the code "by hand". His solution essentially provides a high-level language for manipulating aggregates that is:

- Powerful enough to express most commonly-used looping idioms.
- Translatable to efficient sequential code (loops).

There is definitely some commonality between our problem and that addressed by his approach. He has provided a solid framework which captures common patterns of sequential computation on aggregates and hides low-level details. We are attempting to capture common patterns of parallel computation on aggregates and hide low-level details. Beyond noting this promising similarity our ideas for extending Waters' approach to handle parallelism are still in the preliminary stages.

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.

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 4.3 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 in 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

Special thanks to Dan Nussbaum for the final ALEWIFE version. 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] Chaiken, D., Kubiatowicz, J., and Agarwal, A., "LimitLESS Directories: A Scalable Cache Coherence Scheme," MIT VLSI Memo. Submitted for publication.

[4] 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.

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

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

[7] 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.

[8] Goldman, R., R. Gabriel, and C. Sexton, "Qlisp: Parallel Processing in Lisp," Springer-Verlag LNCS 441, *Proceedings of U.S./Japan Workshop on Parallel Lisp*, June 5–7, 1989, Tohoku University, Sendai, Japan.

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

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] Mohr, E., Kranz, D., and Halstead, R., "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs," *Proceedings of Symposium on Lisp and Functional Programming*, June 1990.

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

[20] 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

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

[22] 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.

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

[24] Waters, R.C., "Series", in *Common Lisp: the Language*, Second Edition, G. Steele, Jr., Digital Press, Maynard MA.

[25] Waters, R.C., *Optimization of Series Expressions: Part I: A User's Manual for the Series Macro Package*, Massachusetts Institute of Technology technical report AIM-1082, January 1989.

[26] Weening, J., "An Analysis of Dynamic Partitioning," Springer-Verlag LNCS 441, *Proceedings of U.S./Japan Workshop on Parallel Lisp*, June 5–7, 1989, Tohoku University, Sendai, Japan.

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