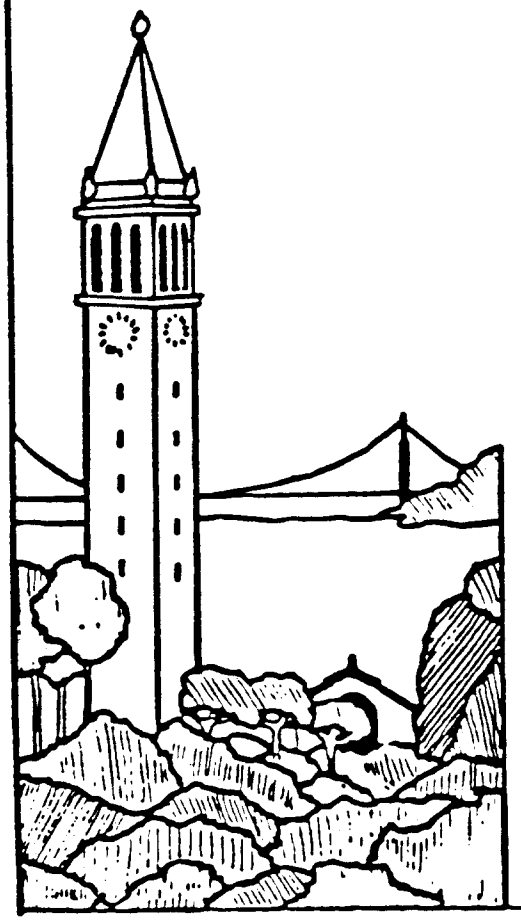
Curare: Restructuring Lisp Programs For Concurrent Execution

James R. Larus



Report No UCB/CSD 87/344

February 25, 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Curare:
Restructuring Lisp Programs
for
Concurrent Execution

James R. Larus[1]
Computer Science Division
Electrical Engineering and Computer Science Department
University of California
Berkeley, CA 94720

February 25, 1987

[1]Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4817, monitored by Space & Naval Warfare System Command under Contract No. N00039-84-C-0089. The author was also supported by a California Microelectronics Fellowship.

Abstract

This paper describes the techniques used by CURARE, a program transformer, to restructure Lisp programs for concurrent execution in multiprocessor Lisp systems. CURARE tries to eliminate control and data-dependencies that prevent concurrent execution of the invocations of recursive functions. CURARE also inserts a variety of synchronization devices to ensure that unremovable dependencies do not impair execution of a program. The product of this process is semantically equivalent to the original Lisp program, but executes faster on a multiprocessor than would the original program.

Contents

1 Introduction

CURARE is a program that transforms Lisp programs into equivalent programs that take advantage of the concurrency available on multiprocessors by executing some portion of their tasks in parallel. The resulting programs contain synchronization constructs sufficient to produce correct and deterministic results and to create a workload appropriate to the parallelism of a target machine. This paper describes CURARE's underlying assumptions and its algorithms.

CURARE addresses the problem of effectively and efficiently programming shared-memory multiprocessors with the programming language Lisp. Currently, there are a variety of shared-memory multiprocessors, such as SPUR [12], Butterfly, Sequent, and others, but few non-trivial uses of their potential parallelism exist. In addition, most research on programming multiprocessors has concentrated on solving numeric programs written in FORTRAN [2,16, 17].

Lisp, because of its flexible control and data-structures, is typically used for symbolic, not numeric, computation such as in artificial intelligence or compiler writing. The flexibility and generality of Lisp, which are its strengths in these applications, impose serious limits on the program analysis necessary to restructure programs automatically. Lisp's three major impediments are its ubiquitous pointers, its side-effects, and its flexible control-structures, all of which make the techniques applied to FORTRAN programs unsuitable for Lisp programs.

CURARE cannot avoid these problems, but it attempts to solve them by concentrating its efforts on restructuring a single program feature—recursive function calls—and by accepting programmer-supplied declarations in place of information that is impossible to collect mechanically. CURARE's goal is to restructure Lisp programs that use only the common features of the language to improve their execution speed and ensure their correct concurrent execution. Although its final product may not match the speed of a hand-crafted multiprocessor program, the average quality of its product should be better than an average programmer's attempts at programming a multiprocessor.

1.1 Why Lisp?

Why use Lisp instead of an existing programming language or a language invented for programming multiprocessors?

Lisp has several advantages over existing languages. First, it is widely used so that many programmers are familiar with it and many programs have been and are being written in it. For these reasons among others, various groups are investigating and implementing Lisp on diverse multiprocessors [8,10,11]. These systems provide a variety of environments for testing CURARE. Second, Lisp is a flexible language whose syntax and semantics is easily extended for experiments. Few other languages can be transformed and manipulated as easily as Lisp. Finally, Lisp has the serious impediments mentioned above. Pointer-manipulating program are difficult to restructure, but are common in non-numeric applications so that some method of mechanically preparing them for concurrent execution is desirable. Lisp pointers are well-suited to this experiment since they are better behaved than pointers in other languages that

allow arbitrary computations on pointer values.

Lisp is also preferable to a new multiprocessor language for several reasons. First, it allows existing programs to be used without rewriting them in a new and uncommon language. Programs can also be developed and debugged in the simpler and more hospitable uniprocessor environments for which Lisp is famous. Second, developing a new language is difficult without programming experience in its application domain. Multiprocessors are too new to have such a body of experience. CURARE's approach of extending an existing language with declarations will illuminate the areas in which Lisp and similar languages need to adapt to multiprocessors. Finally, programming languages must balance the degree of control given a programmer against the portability of the resulting programs. It is always tempting to provide imperative features that correspond to the architecture of a particular machine or group of machines, but fit poorly into the design of other computers. CURARE's declarative approach limits this temptation by having the programmer supply advice rather than describe the details of a computation.

1.2 Typical Multiprocessor Lisp System

The techniques described in this paper are intended to transform programs for execution on small-to-medium sized, shared-memory multiprocessors. A small-to-medium multiprocessor has roughly 5 to 100 autonomous processing units, each of which is capable of executing Lisp functions. We limit the size of the target computer because CURARE is unlikely to produce programs that can effectively keep more than 100 or so processors busy. Furthermore, very large numbers of processors may require different programming paradigms, such as the data-parallelism advocated for the Connection Machine [13].

Each processor must be an autonomous unit capable of executing Lisp functions within a single shared Lisp address space (Figure 1). The Lisp system has multiple, concurrent threads of control. A Lisp process can access any data within the system, except for a small amount that may be kept private to a process. The system does not mediate access to shared data; programs must, therefore, synchronize explicitly.

This model is a simple generalization of existing uniprocessor, multiprocess Lisp systems [1] and the multiprocessor Lisp systems mentioned above. The model allows us to separate the details of CURARE from the language extensions and hardware features embodied in these systems.

Finally, we also assume that Lisp process creation, deletion, and context-switching are noticeably more expensive than function invocation. In addition, this imbalance will persist for several reasons. First, processes are larger program units than functions and as such have more state, such as scheduling information. Maintaining this information has a cost in execution time. Second, as long as function calls are more frequent than context switches, it would be inappropriate to make the latter less costly at the expense of the former. This is not an argument against reducing the cost of processes. However, programmers and program transformation systems cannot treat processes as a free and infinite resource (cf. Halstead's Multilisp, [10]).

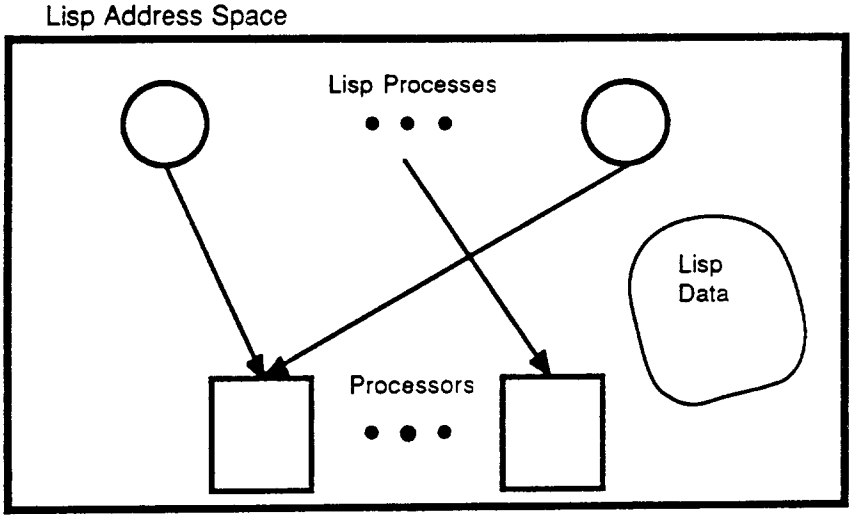


Figure 1: A prototypical multiprocessor Lisp system executes processes within a shared address space containing programs and data. The processes are autonomous and concurrent, although the system may contain more processes than processors, thereby requiring multiprogramming.

1.3 Overview of Technique

This section contains a descriptive overview of the techniques used by CURARE. The details, which are elaborated in later sections, will be more understandable if seen in the context presented below.

CURARE takes advantage of a multiprocessor's parallelism by transforming Lisp programs so that the multiple invocations of a recursive function execute concurrently. Two invocations are concurrent if part of invocation i executes at the same time as part of invocation j. This model is called Concurrent Recursive Invocations (CRI). We concentrate on recursive functions for several reasons. First, recursive functions are a more general class of control-structure than the explicit loops used in other investigations of parallelism. Any loop can be easily transformed into a recursive function, but the converse is not true because recursive functions have state that requires additional, explicit data-structures. Second, a recursive function delimits a discrete, repeatedly-executed task. Each invocation applies the body of the function to a portion of the data. Finally, a function's body is a partially-closed scope with a well-defined set of imported entities. There are fewer and simpler relationships between invocations of a function than between arbitrary pieces of code.

Two relationships between objects in a program prevent its correct, concurrent execution. The first relationship is *control dependency*, which occurs when the execution of a piece of code is contingent on the result of another piece of code. For example, both arms of a conditional statement are control dependent on the predicate of the statement. Control dependencies are easily detected and represented [9] and will not be discussed in detail.

The other relationship is *data-dependency*, which occurs when statement A writes a location subsequently read or written by statement B. Reversing the execution order of A and B may change their results. For example, consider the statements X ← 1 and Y ← X. If X's initial value is not 1, then Y's final value depends on the order of execution of the statements. Statements A and B also are said to *conflict* over their use of X's memory location.

The three possible types of data-dependency have been given names:

- *Flow Dependency:* A writes a location that B reads.
- *Antidependency:* A reads a location that B writes.
- *Output Dependency:* A writes a location that B also writes.

Non-atomic operations complicate detecting data-dependencies. Intermingling pieces of non-atomic operations can totally change their results.

Some data-dependencies, such as the modification of a variable, are easy to detect. Others, such as the modification of a cell in a list structure, are much more difficult to find. In either case, to execute a program concurrently, we must transform it so that conflicts cannot affect the final result of its execution.

This paper discusses three such transformations. The first requires the use of *lock* and *unlock* primitives to synchronize access to shared-locations. Locks are expensive, but are always able to order accesses. The next technique rewrites a function's body to delay subsequent, conflicting invocations until after the execution of operations that conflict with them. This approach may be less expensive than locking, but will not not work for all recursive functions. The final approach is still less general and costly. In some circumstances, the order of operations does not affect their final result, in which case constraints arising from data-dependencies do not exist.

User-supplied declarations are an important aspect of conflict-detection and program-transformation. Without declarations, a system such as CURARE must use the most conservative assumptions about any relationship it cannot deduce. In particular, CURARE would have to assume the worst possible aliasing between input parameters, which would prevent any possibility of concurrent execution. Also, deducing that the semantics of operations permits their reordering is impossible in general. However, such information is frequently obvious to a programmer, who could thus supply the information necessary to produce efficient multiprocessor programs.

1.4 Overview of Paper

The rest of this paper describes the algorithms used by CURARE. Section 2 describes the techniques used to detect conflicts in memory-resident structures. Section 3 discusses the semantics of concurrent execution and three ways of ensuring its correctness. Section 4 describes the execution model. Section 5 discusses ways to expand the class of functions that CURARE can transform.

2 Conflicts in Memory-Resident Structures

The locus of a memory conflict, which itself is the cause of a data-dependency, is either a program variable or a location in a memory-resident object. Conflicts among uses of variables are easily detected in most cases. Only the most general features of Lisp, such as the `set` and `eval` functions, frustrate this analysis. However, these features are infrequently used in most programs, therefore, a program analyzer can reasonably assume the worst about their side-effects.

The FORTRAN-restructuring literature contains an extensive discussion of the techniques for detecting conflicts among accesses to arrays (for example, the work done at Rice, IBM, and the University of Illinois, [2,5,24]). The techniques developed for FORTRAN can be applied to Lisp arrays also. The major difference between the two types of arrays are that Lisp arrays can contain pointers but FORTRAN arrays contain only numbers. In addition, FORTRAN programs rarely use double indirect array references, such as `A[A[I]]`, that simulate pointers.[1] Pointers can cause two apparently distinct array references to lead to a conflicting memory location. This problem with pointers is discussed in detail below. The results from that discussion can be applied to arrays as well, thereby enabling the use of FORTRAN conflict detection techniques for Lisp arrays.

The rest of this section discusses detecting conflicts among references to a third type of object—memory-resident data-structures. These objects are a contiguous block of memory with named fields, for example list-cells or structures produced by `defstruct`. These objects frequently contain pointers linking them into arbitrary graphs.

Detecting conflicts in programs that manipulate arbitrary graphs is impossible in general. Any two references into the graph can lead to the same location. Instead, we hope to identify a class of structures for which a simple program analysis can indicate the absence of conflicts and to provide a way for a programmer to label objects belonging to this group.

2.1 Conflicts Among Structures

A *data-structure* is a set of named fields $f_1, f_2, \ldots f_q$ for $1 \leq q$. We use Pascal notation to designate a reference to field f_i of an instance I: $I.f_i$. Although programs typically use more than one data-structure, we will make the simplifying assumption that a program only manipulates one type of structure. Generalizing to more complex programs requires distinguishing the type of object manipulated by a reference, which is possible because of the requirement in Lisp that structure accessors have unique names.[2] We further assume that fields in a structure can be partitioned into two sets, those pointing to other instances of the structure $(f_1, f_2, \ldots f_r)$ and those pointing to other types of objects $(f_{r+1}, f_{r+2}, \ldots f_q)$. Not every program makes a clear distinction between the sets (particularly list-cells). However,

[1] Because of the difficulties caused by pointers, most FORTRAN transformation systems will not work on programs containing this construct.

[2] In an inheritance system, such as Flavors or PCL [1,4], more than one class of objects can be manipulated by a given accessor. However, the behavior of a related group of objects should be similar enough that an analysis can apply to objects from all such classes.

```
(setf (cadr x) 5)
(print (caadr x))
```

Figure 2: Pair of conflicting statements.

user-defined structures typically have fields pointing to their predecessors and successors in a graph and other fields containing information specific to the node. A programmer can easily declare the class in which a structure field belongs. This information is simply the type declarations required in strongly-typed languages.

A *link*, l, is a triple $\langle I_1, f_s, I_2 \rangle$ such that I_1 and I_2 are instances of a structure and $I_1.f_s = I_2$. The first element of a triple is its *source* (S), the second is its *field* (F), and the third is its *target* (T). For example, $S(l) = f_s$. A link is an assertion that instance I_1 contains a pointer in field f_s to instance I_2 of the *same* type of structure.

A *path* P is an ordered sequence of links, $l_1, l_2, \ldots l_k$, such that $T(l_i) = S(l_{i+1})$ for $1 \leq i \leq k$. Define the length $|P| = k$. The *destination* of a path is its last target: $T(l_k)$. In other words, a path is an ordered traversal of some nodes in the graph formed by instances of a structure.

The *accessor*, $A(P)$, of a path, P, is the ordered sequence of fields along the elements of the path: $F(l_1), F(l_2), \ldots F(l_k)$ where the l_i's are links in the path. Given the initial source of a path and its accessor, the path can be recreated by applying the accessor to nodes in the graph. Let $I = S(l_1)$ in a path P. Then, $A(P) \circ I$ is the application of the accessor to the instance, which yields the destination of the path.

A structure *access* is a pair consisting of an accessor, $A(P)$, and a structure instance, I: $\langle A(P), I \rangle$. It corresponds to reading the value of the location $A(P) \circ I$.

A structure *modification* is a triple consisting of an accessor, $A(P)$, a structure instance, I, and a value, V: $\langle A(P), I, V \rangle$. It corresponds to replacing the value in the location $A(P) \circ I$ by V.

To detect conflicts in a Lisp program, an analyzer must identify a set of structure accessors and detect when the destination of a path used in a write operation is equal to a source or target in the path of another operation. For example, the statements in Figure 2 conflict because the destination of the path of the first statement, `x.cdr.car`, is used in the path of the second statement, `x.cdr.car.car`.

This process is complicated by two factors: aliasing and program variables. When two distinct structure instances contain a pointer to the same location—there exist links l_i and l_k such that $S(l_i) \neq S(l_k)$ but $T(l_i) = T(l_k)$—the paths containing the instances are *aliased*. The structure instances shared by these paths no longer have a unique identity since they can be reached by distinct accessors. Arbitrary aliasing among structures makes analysis fruitless since any two accessors can point to the same location.

However, some forms of aliasing are so well-defined that they are benign. For example, a doubly-linked structure has an infinite number of paths to any instance in it. However, this

set of paths can be reduced to a finite set of unique paths by combining adjacent successor-predecessor pairs in a path. The *canonicalization*, C, of a path is an injective mapping from paths to paths. The effect of a canonicalization function is specific to a particular structure and depends on the linkage between instances of the structure. In the doubly-linked example:

$$C(l_1, l_2, \ldots l_m, \langle I_x, \mathit{succ}, I_y\rangle, \langle I_y, \mathit{pred}, I_x\rangle, l_n \ldots l_k) \Rightarrow C(l_1, l_2, \ldots l_m, l_n \ldots l_k)$$

$$C(l_1, l_2, \ldots l_m, \langle I_x, \mathit{pred}, I_y\rangle, \langle I_y, \mathit{succ}, I_x\rangle, l_n \ldots l_k) \Rightarrow C(l_1, l_2, \ldots l_m, l_n \ldots l_k)$$

$$C(l_1, l_2, \ldots l_k) \Rightarrow (l_1, l_2, \ldots l_k)$$

To provide this information to CURARE, a programmer would declare that the *succ* and *pred* operations are inverses of each other and that a canonical path does not contain adjacent pairs of these operations.

The set of structure instances *accessible* from a given instances, I, denoted $\mathit{accessible}(I)$, is the smallest set of instances satisfying:

$$\mathit{accessible}(\mathtt{nil}) = \emptyset$$

$$\mathit{accessible}(I) = \{I\} \cup \mathit{accessible}(I.f_1) \cup \mathit{accessible}(I.f_2) \ldots \cup \mathit{accessible}(I.f_r)$$

An instance of a structure, I, has the *single access path property* (SAPP) if there exists only one *canonical* path to any instances in $\mathit{accessible}(I)$. In effect, this property requires that instances form a tree rather than a general graph. We are measuring how often this occurs in Lisp programs.

The second factor complicating this analysis is that instances of structures are stored in program variables. Our analysis would be more useful if it could report conflicts among the variables in a program. However, the values of variables change during the execution of a program, which complicates the bookkeeping.

For any two references to the same variable, v, let the *transfer function*, τ_v, be the accessor of the difference in the value of v between the two references. If v's value did not change between references, then $\tau_v = \emptyset$. If the relationship between the values of v cannot be determined or no accessor can map the first value to the second, then $\tau_v = A^*$, where A is the alphabet of accessors and $*$ is the Kleene star. If v is a parameter of a recursive function and a is the accessor of the function used to compute the actual value of the parameter, then $\tau_v = a^+$, where $a^+ = aa^*$. For example, the function in Figure 3 has a transfer function $\tau_l = \mathtt{cdr}^+$. If a variable, v, can be assigned one of many distinct values with accessors $a_1, a_2, \ldots a_m$, then $\tau_v = a_1|a_2|\ldots|a_m$, where "|" is the regular expression disjunction operator. This combination is flow-insensitive since the information from various paths through the program is combined into a form that does not permit us to distinguish the portion that is valid at a particular point in the program. The n-fold composition of τ_v, written ${\tau_v}^n$, is $\underbrace{\tau_v \ldots \tau_v}_{n}$.

Transfer-functions can be combined with access paths to detect conflicts in programs. First, we need to define a conflict precisely. Let $M = \langle A(P_1), I_1, V\rangle$ be a structure modification and $A = \langle A(P_2), I_2\rangle$ be a structure access. M conflicts with A, written $M \otimes A$,

```
(defun f (l)
  (when l
    (print (car l))
    (f (cdr l))))
```

Figure 3: A simple recursive function.

if

$$T(l^{P_1}_{|P_1|}) = \begin{cases} S(l^{P_2}_i) & \text{for } 1 \le i \le |P_2|, \text{ or} \\ T(l^{P_2}_{|P_2|}) \end{cases}$$

where l^x_j is the j^{th} link in path x. This formula says that a structure modification conflicts with a structure access if the location modification by the former is read by the latter.

Similarly, two modifications, $M_1 = \langle A(P_1), I_1, V_1 \rangle$ and $M_2 = \langle A(P_2), I_2, V_2 \rangle$, conflict if the first one modifies a location read or written by the second:

$$T(l^{P_1}_{|P_1|}) = \begin{cases} S(l^{P_2}_i) & \text{for } 1 \le i \le |P_2|, \text{ or} \\ T(l^{P_2}_{|P_2|}) \end{cases}$$

Consider two references to a variable v. The first reference, $v.A_1$, is the application of accessor $A_1 = a^1_1.a^1_2 \ldots a^1_m$ to v. Similarly, the other reference, $v.A_2$, is the application of $A_2 = a^2_1.a^2_2 \ldots a^2_m$ to v. Let $\tau_v = t_1|t_2| \ldots |t_p$ be the transfer function between the values of v at the references. If the first reference is a modification operation, then

$$\begin{aligned} v.A_1 \otimes v.A_2 &\Leftrightarrow v.A_1 \otimes v.\tau_v.A_2 \\ &\Leftrightarrow a^1_1 \ldots a^1_m \preceq t_1 \ldots t_p a^2_1 \ldots a^2_n \end{aligned}$$

where $\preceq$ is the prefix operator so that $a \preceq b$ if a is a prefix of b. In other words, the references conflict if the location modified by the first one is used by the second one, taking into account the change in v's value between them. Similarly, if the second reference is a modification operation, then

$$\begin{aligned} v.A_2 \otimes v.A_1 &\Leftrightarrow v.A_2 \otimes v.\tau_v.A_1 \\ &\Leftrightarrow a^2_1 \ldots a^2_m \preceq t_1 \ldots t_p a^1_1 \ldots a^1_n \end{aligned}$$

This technique relies heavily on the SAPP to ensure that every location has only a single name so that conflicts can be detected by matching strings.

For example, consider the function in Figure 4. Let A_1 be `car.cdr`, A_2 be `car`, and τ_l be `cdr`. Then, $A_1 \otimes A_2$ under τ_l because $\tau_l \circ A_2 = \texttt{cdr.car} = A_1$.

The same technique works for more complicated transfer functions, as long as the prefix operation matches a string against a regular expression. In addition, the technique can be

```
(defun f (l)
  (when l
    (setf (cadr l)        ; A1 = cdr.car
          (car l))        ; A2 = car
    (f (cdr l))))
```

Figure 4: A recursive function with a conflict between invocations.

used to find the distance of a conflict. The *distance* of a conflict in a recursive function is the number of invocations separating the conflicting references. For example, in `f`, the distance of the conflict is 1 since the location written in an invocation is read in the subsequent one. If $A_1 \preceq \tau_v{}^d \circ A_2$, then A_1 and A_2 conflict at distance d, written $A_1 \overset{d}{\otimes} A_2$.

2.2 Example: Lists

This section describes the application of the technique described above to a particular data-structure—the Lisp list. A list cell has two fields, `car` and `cdr`. Assume that both fields point only to other list cells. Lists do not require a canonicalization function, so the SAPP can be restated that lists do not contain cycles or shared substructures.

List accesses are strings in the language $\{\texttt{car}, \texttt{cdr}\}^+$. Transfer functions are regular expressions over the alphabet $\{\texttt{car}, \texttt{cdr}\}$.

For example, the function in Figure 5 contains three accessors

A_1 `cdr`
A_2 `cdr.car` (modify)
A_3 `car`

and a transfer function τ_l = `cdr`. Since A_2 is the only accessor that modifies a location, the only conflicts are between A_2 and A_1 or A_3. A_2 does not conflict with A_1 since $\texttt{cdr}^+\texttt{.car}$ can never be a prefix of `cdr`. However, $A_2 \otimes A_3$ since $\texttt{cdr.car} \preceq \texttt{cdr}^+\texttt{.car}$.

3 Concurrent Execution of Recursive Functions

This section describes the Concurrent Recursive Invocation (CRI) execution model for functions restructured by CURARE. In this model, invocations of a recursive function execute concurrently, subject to the constraints necessary to preserve the result of the function. This exposition has three parts. The first one elaborates the execution model and calculates its potential parallelism. The second discusses the semantics that must be preserved in spite of parallel execution. And, the third shows how to ensure these semantics with locks and other devices.

```
(defun f (l)
  (cond ((null l) nil)
        ((null (cdr l))          ; A1
         (f (cdr l)))
        (t
          (setf (cadr l)         ; A2
                (+ (car l)       ; A3
                   (cadr l)))
          (f (cdr l)))))
```

Figure 5: A complex recursive function with a conflict between invocations.

3.1 Mechanics of Concurrent Execution

Consider a function, f, that contains self-recursive calls $C_1, C_2, \ldots C_n$. If f returns the result returned by these calls, it is a *tail-recursive* function. If f does not use the result returned by one of these calls, say C_i, then C_i is a *free* call.

Every statement, $S_1, S_2, \ldots S_m$, in the body of f can be partitioned into one of two sets: the *head*, H_f, or the *tail*, T_f, of f. A statement, S_i, belongs in the tail of f if S_i is not a recursive call and is dominated by a recursive call. A statement that is not in f's tail is in its head. The head contains all recursive calls and all statements that might execute before a recursive call.

The i^{th} invocation of f, denoted I_i^f, is the $i-1^{th}$ recursive call on f. I_0^f is the external call that began the recursion.

Consider for a moment a simple recursive function, f, that contains no loops or function calls other than a single self-recursive call C_1. In the normal course of recursion, invocations $I_0^f, I_1^f, \ldots I_d^f$ execute statements from the head of f followed by a phase of executing statements from the tail of f as the recursion unwinds (Figure 6).

How must f change so that its invocations can execute concurrently? Ignore, for the moment, side-effects in f that cause conflicts and that may produce incorrect results (this problem is discussed in Section 3.1.1). The first part of the question is when does an invocation begin executing? One approach would create processes for all invocations and simultaneously begin their execution. However, the agent spawning the processes must know how many to create (the depth of the recursion) and the actual arguments for every call. It is impossible to know this information *a priori* for any but the simplest functions.

A more practical approach treats a recursive call as the creation of a new process to execute the subsequent invocation asynchronously. Obviously, this approach will not yield parallelism if the spawning process needs to wait for an answer from the recursive call. However, if the spawning process is not strict in its use of the result (e.g., it stores the result in a data structure rather than looking at its value), then a Multilisp *future* provides process creation

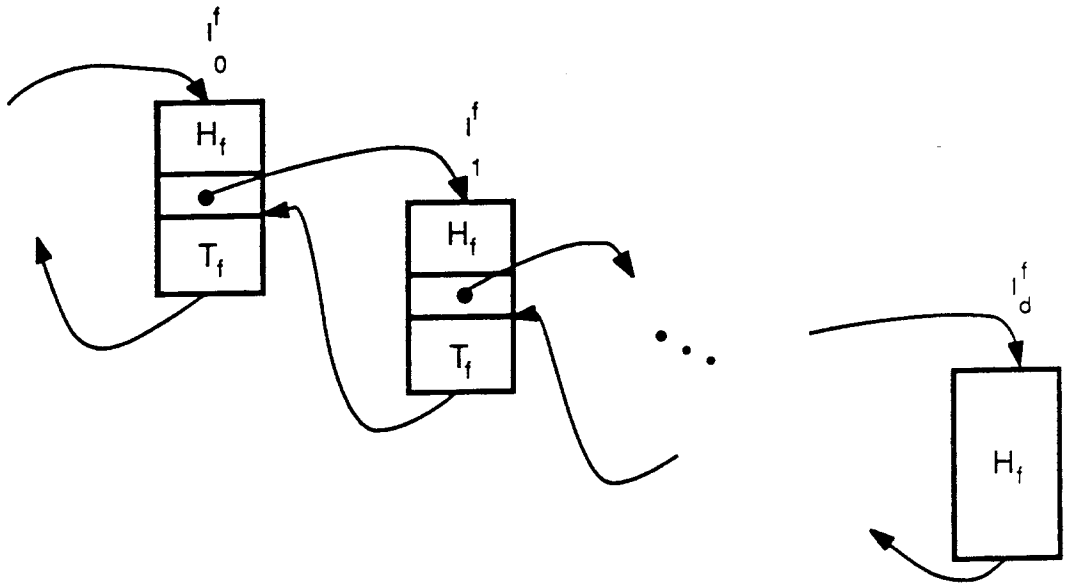


Figure 6: The normal course of a recursive call on a simple function f that contains only a single self-recursive call.

and synchronization features that permit concurrent execution [10]. Furthermore, if the spawning invocation never uses the result from the recursive call, as, for example, when it calls for effect, then a future's synchronization mechanism is unnecessary. Either way, the recursive call takes the form shown in Figure 7. CURARE can transform some functions that use results from self-recursive calls into equivalent functions that do not use these values (Section 5).

The concurrency of this approach, and therefore its potential speedup on a multiprocessor, depends on the relative size of f's head and tail. Let the size of the two sets, $|H_f|$ and $|T_f|$ respectively, be some measure of the execution time spent in each set of statements. A variety of measures are discussed and evaluated by Sarkar and Hennessy [20,21].

The number of processes that execute simultaneously—the concurrency of the system—is given by

$$\frac{|H_f| + |T_f|}{|H_f|}$$

Proof. In the time required to execute a single invocation, $|H_f| + |T_f|$, we can spawn off $\frac{|H_f|+|T_f|}{|H_f|}$ new processes since $|H_f|$ is the time from the beginning of an invocation through a recursive call. ∎

Note that $|H_f|+|T_f|$ is fixed, so that the only way to increase the concurrency is to decrease the amount of code executed before a self-recursive call. In contrast to sequential function execution, in which tail-recursive functions have major benefits in speed and space, concurrent function execution favors functions in which recursive calls occur as early as possible (*head-recursive* functions).

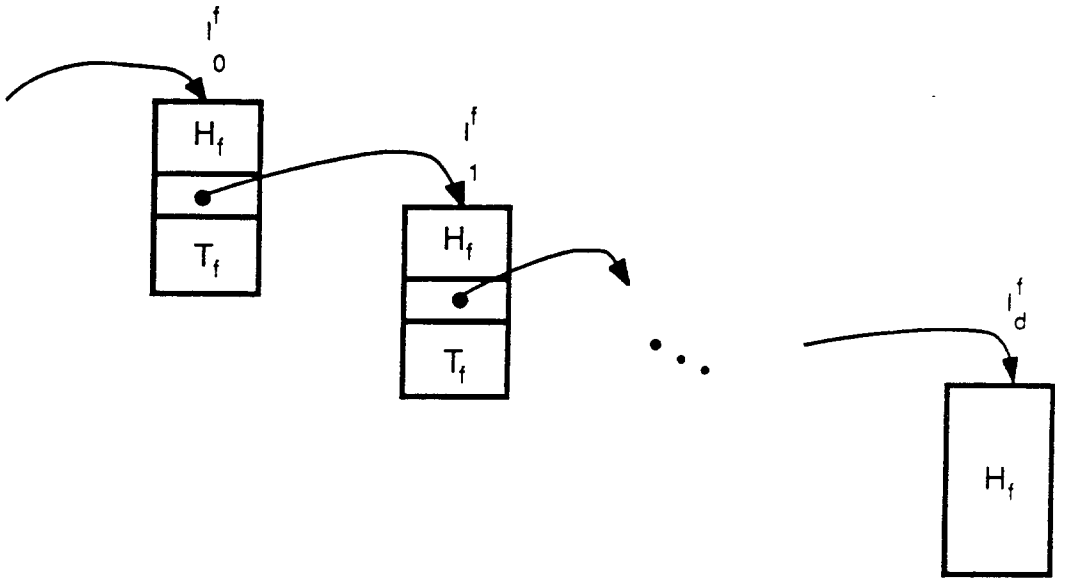


Figure 7: Control flow between recursive calls when a recursive call spawns off a process to execute its subsequent invocation.

A similar principle applies to functions containing more than one recursive call. The code executed before each such call reduces the potential concurrency. However, calculating the concurrency for these functions is much more complicated. In addition, semantic constraints discussed below impose further limits on the interleaving of recursive calls from different calls sites.

3.1.1 Semantics of Concurrent Execution

In the previous section, our discussion ignored the consequences of concurrent execution on the results produced by a function with side-effects. This section rectifies this omission and shows how to preserve the semantics of sequential execution, albeit at a cost in execution time.

Throughout the rest of this paper, we will view a function invocation as a transaction, that is, an atomic action that takes a program from consistent state to consistent state. Invocations are the appropriate granularity for primitive operations since they are the smallest independently scheduled tasks.

Conventional database systems usually guarantee that execution of a set of transactions will be serializable. An execution is *final-state serializable* if its result is equal to the final result of a serial execution of the same set of transactions in some unspecified order [18]. This guarantee is not appropriate for concurrent functions for two reasons. First, a function executed under this guarantee can produce non-deterministic results. Most programming languages guarantee reproducible results unless a programmer uses inherently non-deterministic language features. Although some argue that multiprocessing is such a feature, changing

this principle would constitute a major revision of the semantics of Lisp and would necessitate rewriting most programs submitted to CURARE. Second, CURARE can make stronger promises than a database because it pre-analyzes programs before execution and knows the sequential execution order of its transactions.[3]

CURARE's guarantee is that the execution of a set of transactions (invocations) is sequentializable. An execution is *final-state sequentializable* if its final result is equal to the final result of the serial execution of the same set of transactions in their sequential order. This result is the one that a programmer expects: concurrent execution improves the speed of a program but does not change its result.

CURARE ensures that a program is sequentializable by conflict serializing its concurrent execution with its sequential execution. Execution of a set of transactions is *conflict serializable* if it is conflict equivalent to a serial execution of the same transactions. Executions of two sets of transactions are *conflict equivalent* if they contain the same transactions and all conflicts are resolved the same way in both executions. We use conflict serializability as our correctness criterion because it is intuitive, easily verified, and corresponds to the conflicts discussed in Section 2.

3.2 Ensuring Correct Execution

The next few sections discuss various techniques for ensuring the correct execution of concurrent function invocations. The methods are discussed in order of decreasing cost and generality.

3.2.1 Locking

Locking conflicting items is a simple and effective method of ensuring the conflict serializability of a set of transactions. Formally, assume that invocations I_i^f and I_j^f, for $i < j$, access location M and that at least one contains a statement that modifies M. To ensure that I_i^f has exclusive use of M before I_j^f (the sequential execution order) CURARE inserting a lock statement, `Lock(M)`, in the head of f and an unlock statement, `Unlock(M)`, in the body of f. The head of I_i^f executes before any portion of I_j^f, so location M will be locked when the latter invocation begins and will remain locked until the first invocation finishes with the location.

If a third invocation, I_k^f for $j < k$, accesses location M and conflicts with the other two invocations, the locking still works correctly. Since I_j^f contains a `Lock(M)` statement in its head, it cannot spawn subsequent invocations, including I_k^f, until it has locked M. Therefore, the order of access to conflicting locations is preserved even if more than two invocations conflict over a location.

The naive locking scheme described above can be improved in several respects. First, if invocations I_i^f and I_j^f conflict over a set of locations, $M_1, M_2, \ldots M_m$, and all such sets are

[3] Other attempts at using database semantics in concurrent Lisp systems have taken the database approach of foregoing analysis and detecting conflicts at runtime [15].

disjoint, then replace the m locks by a single lock. For example, suppose that a function conflicts over the locations `1.car`, `1.car.cdr`, and `1.car.cdr.car` and that τ_l = `cdr`, then locking the location `1.car` is enough. Second, move unlock statements so that they execute as soon after their lock statements as possible. `Unlock(M)` should be put in the body of f after all uses of M and after all lock statements. The latter condition ensures that the locking follows a two-phase protocol for avoiding deadlocks. Finally, replace exclusive locks by read-write locks in cases in which more than one invocation reads M.

Locking has two costs: the costs of the locks themselves and the resulting loss of concurrency. The locks described above are expensive since they are fine-grained locks for single memory locations. Some computer architectures associate a tag with each memory location that can be used as a lock.[4] Other architectures require a more-costly, dynamically-allocated collection of locks (the number of locks depends on the data and the depth of the recursion).

The other cost of locking is a loss of concurrency. Assume that invocations of f conflicts with invocations offset by distances $d_1, d_2, \ldots d_u$. The maximum concurrency of f is no more than $min(d_1, d_2, \ldots d_u)$ if an invocation release its locks just before it terminates. This estimate is slightly pessimistic if invocations release their locks as soon as they finish with a location.

3.2.2 Delays

Another way of achieving the same result as locking, without some of its overhead, is to delay operations so that the second operation in a conflicting pair always occurs after the first [22]. In the CRI model, the only inherent ordering on statement execution is that heads of functions execute sequentially.

Moving conflicting statements into the head of function ensures their correct execution order. Let statements S_i and S_j in the body of f conflict over location M. Assume that S_i always executes before S_j in an invocation of f. By moving S_i to the head of f—also moving any statements upon which it depends—the conflict between S_i and S_j is always resolved in accordance with sequential execution since S_i executes before S_j executes in either the current or subsequent invocations. If S_i sometimes executes before S_j and sometimes after, then both statements must move to f's head.

Which statements must be moved with S_i? All statements that produce values used by S_i. Furthermore, in moving, we must preserve the control dependencies between statements. Considerable work has been done on algorithms and data-structures for code-motion [6,9,19, 23].

The cost of this approach is the loss of concurrency caused by increasing the size of f's head. Such a loss may be acceptable if the resulting concurrency is still greater than the parallelism of a computer or is greater than that produced by locking.

[4] SPUR is not one of these architectures. Although LISP pointers have a tag byte, SPUR's test-and-set instruction destroys the information stored in the bytes of a word other than the low-order byte.

```
(setq a (+ a 1))
(setq a (+ a 2))
```

Figure 8: A pair of statements that do not conflict.

3.2.3 Reordering

Some conflicts between statements impose constraints that are stronger than necessary for correct execution. For example, if addition is an atomic operation, then the apparent conflict between the statements in Figure 8 is illusionary and ignoring the ordering constraints will not affect the final result. Below are three categories of operations for which unnecessary constraints arising from conflicts can be easily identified and removed.

The first type is atomic, commutative, and associative operations, such as addition. Commutativity and associativity ensure the same result regardless of the order of application of an operator. Atomicity guarantees a consistent result, even if two operations overlap. Non-atomic commutative and associative operations can be made atomic with the aid of locks.

The next class of operations consists of those that put a value into an unordered data-structure. Lisp contains several such structures, such as hashtables and unordered lists, beside the ones that users define. Since the order in which items are added to these objects does not matter, constraints related to this order are unnecessary.

The third class of operations is searching unordered sets or searching for one of many acceptable results. If a program is willing to accept any result meeting a criterion, then a search can proceed in parallel without the additional constraint of having to find the same result as a sequential search.

Operations in these three categories are difficult to identify mechanically in all but the most trivial cases. Generally, these properties are closely connected to the semantics of a program, as for example, whether a set is ordered or unordered, and cannot be deduced from an analysis of the program. User-supplied declarations are the only way to provide such information to a program like CURARE.

4 An Implementation

This section rounds out the previous discussion by investigating an approach to implementing CRI in a typical multiprocessor Lisp system as described in Section 1.2. This discussion is concerned with balancing the work from a recursive call among the available processors, not with nitty-gritty details, such as the appropriate synchronization primitives. CURARE is a program transformer that can accommodate a wide variety of target language features simply by changing its final, code-generator stage, which produces Lisp code from CURARE's internal representation.

The larger question of producing programs with parallelism appropriate for a particular

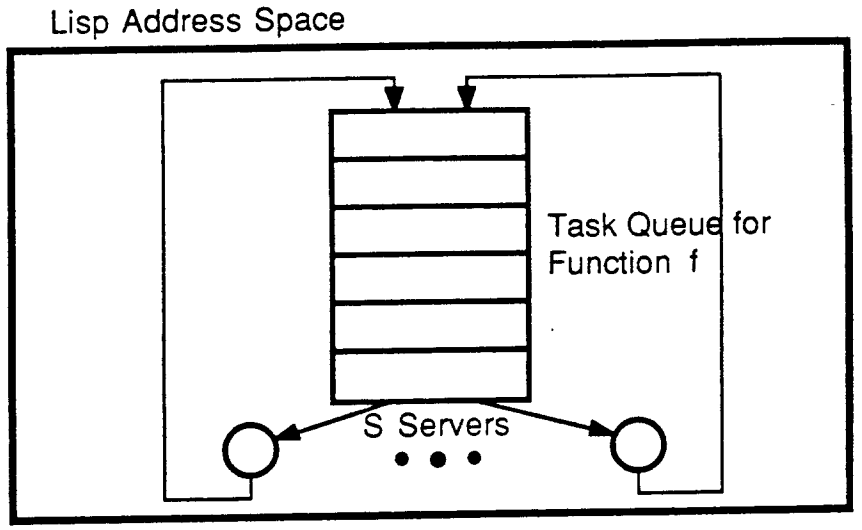


Figure 9: The arrangement used to execute concurrent invocations of recursive functions. Servers repeatedly execute f's body, using parameters obtained from a task queue and enqueuing parameters for subsequent invocations.

machine has been neglected in most of the multiprocessor literature (except Sarkar and Hennessy's work, [20,21]). The correct balance between the concurrency of a program and the parallelism of a processor permits the former to execute efficiently, without wasteful context switches or idle processors. The ideal would be to have a queue of non-preemptable tasks sufficient to keep all processors busy.

This ideal is approachable in our execution model. Because every transaction executes an identical function body, we can have a collection of servers that repeatedly execute this piece of code. Each server only needs to obtain the arguments to an invocation to begin executing a new task. It does not need to execute a process context switch. Since the invocations are well-ordered, tasks execute without preemption.

4.1 Details

Assume a multiprocessor with P processors. A subset of these processors, numbering S, will execute servers for invocations of function f, which they obtain from a central task queue (Figure 9). One question that arises is the appropriate number of servers for a given function invocation.

An abstract model of a server is:

```
while ¬ *recursion-done* do
        dequeue parameters;
        {body of f}
end
```

A server first checks whether the recursion terminated. If not, the server dequeues parameters for the next invocation and executes f's body. Servers repeat this process until the recursion ends. CURARE modifies f's body to enqueue arguments to recursive calls, instead of making the calls directly. These arguments include not only the actual parameters to a call, but also values for non-local variables referenced in f. A practical server would need a more elaborate arrangement to avoid memory bottlenecks. For example, the invocation that terminates the recursion can enqueue tokens that kill the other severs, thereby eliminating the termination flag. However, the central queue will remain a potential bottleneck and limit on the maximum concurrency. This bottleneck will not adversely affect performance, if the time spent executing an invocation is much longer than the time spent waiting for the queue.

If f contains only a single self-recursive call, then queue entries for its invocations are enqueued in their sequential order. In this case the task queue never grows to more than N items, where N is the number of entries in the queue when the servers began executing. Execution of a task removes an item from the queue and that task adds at most one item to the queue, so its length never increases.

If, on the other hand, f contains multiple self-recursive calls, then the order of invocations can be scrambled by the queue. Such an outcome would destroy the temporal ordering between invocations that CURARE depends on. This problem can be resolved by maintaining an ordered set of queues, one for each call site, and by having a server use the next queue only after it finishes executing all calls in the current queue.

The first step in analyzing the performance of this arrangement is to find the time required to execute a recursive call. Consider a simple recursive function f with $h = |H_f|$ and $t = |T_f|$. Let c_f be the maximum concurrency allowed by f, that is the minimum of $\frac{h+t}{h}$ and the distances of f's conflicts. Finally, let d be the depth of f's recursion.

To a first approximation, the invocations are divided into $\left\lceil \frac{d}{S} \right\rceil$ sets of S tasks, each set of which takes $Sh+t$ time steps to execute, so that an entire recursive call requires $\left\lceil \frac{d}{S} \right\rceil (Sh+t)$ time steps (Figure 10). However, this analysis assumes that all processes in a group begin executing simultaneously and ignores the possible overlap between groups. The second group can begin execution as soon as a server from the first group finishes $h+t$ steps. The possible overlap is shown in Figure 11. The total execution time is therefore,

$$\left(\left\lceil \frac{d}{S} \right\rceil - 1\right)(h+t) + (Sh+t)$$

time steps when $S \leq d$.

Setting the first derivative of the equation with respect to S equal to zero, we obtain a minimum at $S = \sqrt{\frac{d(h+t)}{h}}$. This value minimizes the execution time of a simple recursive function when there are more invocations than servers (if there are more servers than invocations, then we should use d servers). The value of S calculated above has to be balanced against c_f, the maximum number of servers that a function can keep occupied. We should use the minimum of these two values.

Allocating processors to a function with multiple self-recursive calls is not more difficult with the multiple queue approach mentioned above. The required number of servers can be

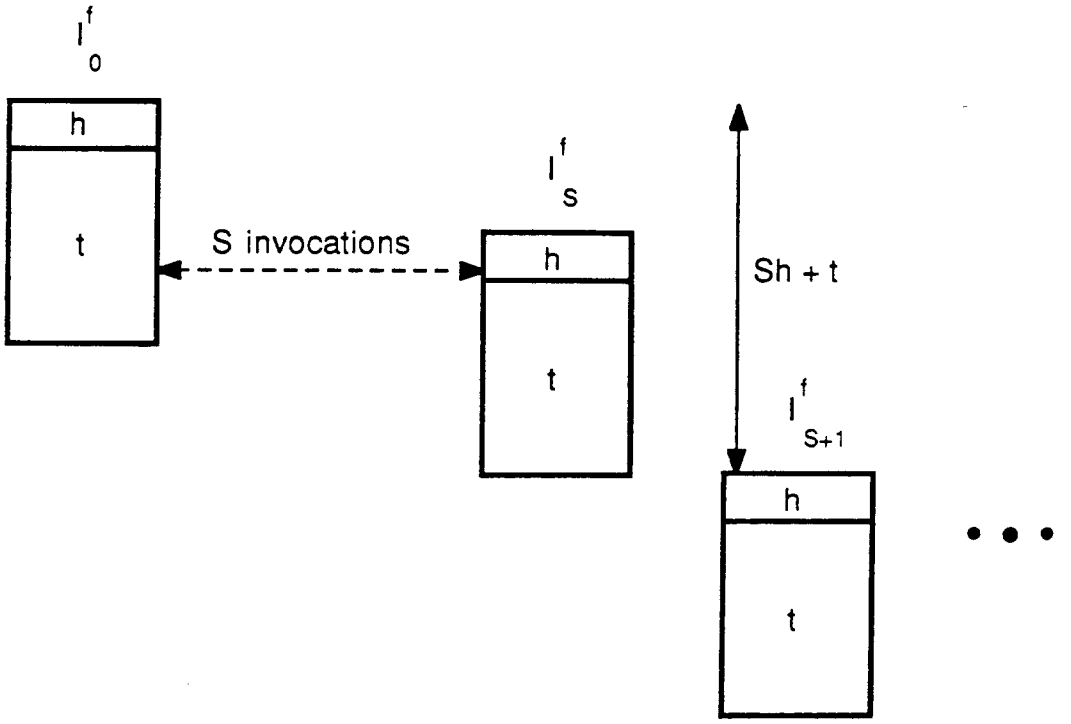


Figure 10: First approximation to the total execution time required by S servers to execute d tasks.

calculated separately for each call site and we can either allocate a fixed number of servers for the entire function execution or dynamically allocate servers as the execution proceeds.

However, the problem of allocating processors to an entire Lisp program is more difficult. A program generally contains many recursive functions, some of which invoke each other. Consider two recursive functions f_1 and f_2 such that f_1 invokes f_2. Furthermore, assume that f_1 requires S_1 servers and f_2 requires S_2 servers. Looking at the call graph of a program, an analyzer might allocate $S_1 \times S_2$ servers to handle the S_1 possible invocations of f_2. However, extravagant allocation of this sort is not practical for larger programs that contain many recursive calls as it will soon exhaust the pool of processors. In addition, processors allocated this way will rarely all execute simultaneously, so the program's concurrency will be small.

Another option is to dedicate only S_2 processes to f_2 and require the S_1 invocations of f_1 to wait their turn for the processors necessary to execute f_2. This may reduce the concurrency of f_1's execution to the point at which it should execute sequentially.

As the preceding discussion points out, load balancing problems are extremely difficult when enough tasks are involved. These problems have not been solved for simpler tasks running on small collections of machines. To compound the problem, the times used in the calculations are simple estimates. Both factors suggest that a simple allocation scheme, with a dynamic component, is the best approach.

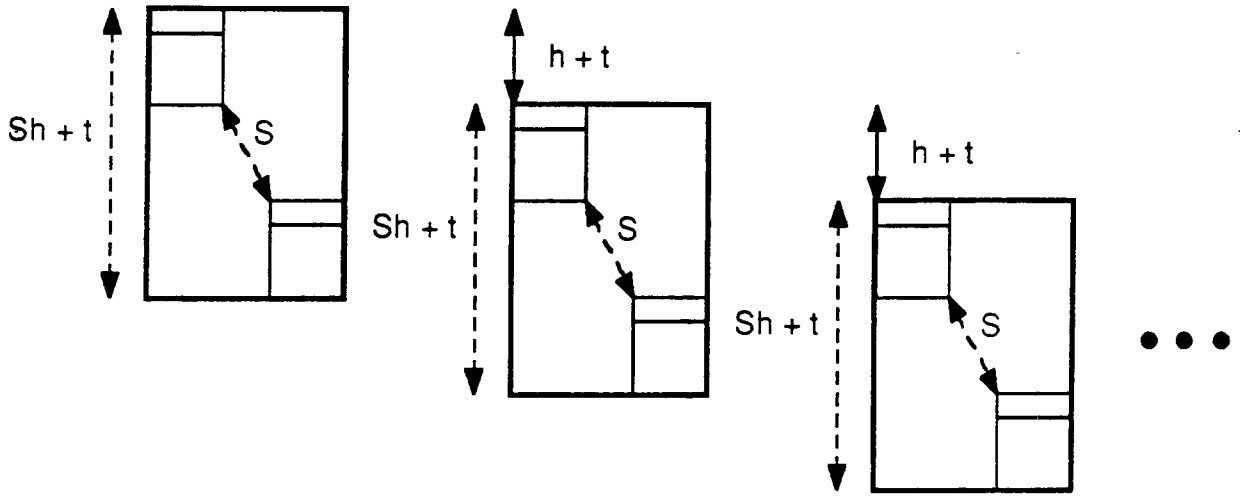


Figure 11: Execution time of S servers executing d tasks.

5 Expanding the Class of Transformable Functions

CURARE only transforms functions that have the specific forms described above. However, some functions that are initially unacceptable to CURARE can be changed into equivalent functions that CURARE can manipulate. We distinguish this second group of transformation from those described earlier to avoid obscuring the issues involved in concurrent execution.

Described below are two transformations that removed impediments to concurrent execution. The problem that they solve is that CURARE only transforms recursive functions that do not return a value that is used within the body of a function. Functions that use the result from a self-recursive call must wait for the recursive invocations to finish executing, which precludes concurrent execution. However, we can change some functions that use their results into equivalent functions that do not use these values and hence make them candidates for concurrent execution. Two such transformations are: changing recursion to iteration and modifying functions to use a destination-passing style.

The first one produces iterative functions from certain types of recursive functions. Iterative functions, in turn, are equivalent to tail-recursive functions. CURARE can transform these functions. Although each invocation of a tail-recursive function returns a result, the function never uses these values. Changing the single return that produces a value into an assignment eliminates the return. The result of a call on the function will appear in a variable. Restricted classes of recursive functions can be transformed into iterative functions by a set of well-known transformations [3,7,14]. Some of these transformations, particularly those described by Huet and Lang, depend on subtle properties of a function's operations, such as commutativity and associativity, and so require information like that provided by CURARE's declarative model.

```
(defun remq (obj lst)
  (cond ((null lst) nil)
        ((eq obj (car lst))
         (remq obj (cdr lst)))
        (t
          (cons (car lst) (remq obj (cdr lst))))))
```

Figure 12: The function **remq** returns a list with all items in **lst** that are **eq** to its first argument removed.

```
(defun remq-d (dest obj lst)
  (cond ((null lst)
         (setf (cdr dest) nil))
        ((eq obj (car lst))
         (remq-d dest obj (cdr lst)))
        (t
          (let ((cell (cons (car lst) nil)))
            (remq-d cell obj (cdr lst))
            (setf (cdr dest) cell)))))
```

Figure 13: The function **remq-d** is the **remq** function rewritten in the destination-passing style. Its first argument is a list-cell whose **cdr** is the destination for results.

The second transformation eliminates return statements in a function whose recursive calls return values that are stored in a data-structure. Although these functions can execute concurrently with the aid of futures, their transformed versions need not incur the overhead of these devices and may be more amenable to CURARE's optimizations.

This transformation produces a function that uses a *destination-passing style* (DPS). In this style, instead of returning a result that is immediately stored in a structure, a function is passed the structure as an argument and stores the value directly. For example, the function in Figure 12 returns a copy of a list with all elements eq to its first argument removed. The function in Figure 13 shows this function rewritten in the destination-passing style. Its first argument (dest) contains a list-cell whose cdr is the destination for returned values. Although this function is less attractive than the original one, it does not execute any more operations than the conventional function; it just performs them in a different order.

At first glance, the DPS function appears to contain more side-effects and is more difficult to analyze than the original. This comparison is correct if both functions were input to CURARE. It would not be able to conclude that the second function contains no conflicts because CURARE's conflict-detection algorithm is flow-insensitive and hence the function would need

synchronization code. However, the destructive operations in this function are the product of the DPS transformation and so CURARE does not start with a blank slate. It can conclude that since the result from a recursive call in the original function is stored in a single, unique location, the store operations in the DPS function also operate on unique locations and do not conflict. The additional information about the origin of the destructive operations enables CURARE to transform the DPS function.

6 On the Nature of Declarations

CURARE relies upon a programmer for a wide variety of information that it cannot collect by analyzing a program, for example:

- whether a structure field points to other structures,
- the type of actual arguments to a function,
- the canonicalization function for a structure,
- whether to restructure a function,
- whether an operation has characteristics necessary for reordering, and
- constraints on data-structures and on functions' results.

The first two items are commonly declared in strongly-typed programming languages. The other items are increasingly abstract program properties of a type that is rarely provided by programmers. CURARE must make the form of declarations as simple as possible, so that a programmer can add them with confidence.

Fortunately, a programmer does not have to write these declarations in a vacuum or supply them for all functions in a program. These declarations can be added as part of an iterative process of tuning a program's performance on a multiprocessor, by examining CURARE's output and program timings. CURARE can help this process by indicating why it failed to transform a function. One crude form of feedback is the locks that CURARE inserted. More helpful are the unresolved conflicts that necessitate these locks. Even more useful is knowledge of the information that would permit CURARE to transform a function.

However, as long as CURARE makes pessimistic assumptions about unknown information, the absence of declarations will not cause it to produce incorrect programs—only slow ones. This lack of speed is a valuable form of feedback about when and where to insert declarations.

7 Conclusion

CURARE is a first attempt at automatically restructuring Lisp programs for concurrent execution. It uses several new ideas, including a simple conflict-detection algorithm for memory-resident data structures that contain pointers, an emphasis on recursive functions instead

of program loops, an expanded vocabulary of transformations for improving a program's structure and concurrent preformance, and a simple execution environment.

Implementation and testing of these ideas is under way. A working prototype of CURARE can already apply some of these transformations to list-manipulating Scheme programs. It is written in Common Lisp and Portable Common Loops. A more complete version, which should be capable of preparing small programs for concurrent execution, will be finished this spring.

References

[1] *Reference Guide to Symbolics-Lisp.* Symbolics, Inc., Cambridge, Mass., 1985.

[2] Randy Allen and Ken Kennedy. *Programming Environments for Supercomputers.* Technical Report Rice COMP TR85-18, Dept. of Computer Science, Rice Univ., March 1985.

[3] R. S. Bird. Notes on recursion elimination. *Communications of the ACM*, 20(6):434–439, June 1977.

[4] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: merging Lisp and object-oriented programming. In *OOPSLA '86: Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 17–29, September 1986.

[5] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[6] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, pages 70–85, January 1986.

[7] J. Darlington and R. M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[8] Issac Dimitrovsky. *ZLISP 0.1 Reference Manual.* Technical Report, Courant Institute, NYU, 1986.

[9] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. *The Program Dependence Graph and its Use in Optimization.* Technical Report CS-TR 8 6-8, Michigan Technological University, Houghton, Michigan, August 1986. To appear in TOPLAS.

[10] Robert H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] W. Ludwell Harrison III. *Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor.* Technical Report CSRD Rpt. No. 565, Center for Supercomputing Research & Development, Univ. of Illinois at Urbana-Champaign, March 1986.

[12] Mark D. Hill, Susan J. Eggers, James R. Larus, George S. Taylor, et al. SPUR: a VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–24, November 1986.

[13] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[14] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, 1978.

[15] Morris J. Katz. *ParaTran: A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme.* Master's thesis, MIT, June 1986.

[16] David J. Kuck et al. The effect of program restructuring, algorithm change, and architecture choice on program performance. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 129–138, August 1984.

[17] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[18] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[19] John H. Reif and Harry R. Lewis. Efficient symbolic analysis of programs. *Journal of Computer and System Sciences*, 32(3):280–314, 1986.

[20] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, June 1986.

[21] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 202–211, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986.

[22] Dennis Shasha and Marc Snir. *Efficent and Correct Execution of Parallel Programs that Share Memory.* Ultracomputer Note 96. Dept. of Computer Science, Courant Institute, NYU, March 1986.

[23] Joe Warren. A hierarchical basis for reordering transformations. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*, pages 272–282, January 1984.

[24] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Technical Report UIUCDCS-R-82-1105, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, October 1982.

This report has been superseded by latter work. For more details, please see: James Larus and Paul Hilfinger, "Restructuring Lisp Programs for Concurrent Execution," in the *Proceedings of the ACM/SIGPLAN PPEALS 1988,* SIGPLAN Notices, Vol. 23, No. 9, Sept. 1988, pp. 100-110.