Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors

by
James Richard Larus

Abstract

CURARE, the program restructurer described in this dissertation, automatically transforms a sequential Lisp program into an equivalent concurrent program that executes on a multiprocessor.

CURARE first analyzes a program to find its control and data dependences. This analysis is most difficult for references to structures connected by pointers. CURARE uses a new data-dependence algorithm, which finds and classifies these dependences. The analysis is conservative and may detect conflicts that do not arise in practice. A programmer can temper and refine its results with declarations.

Dependences constrain the program's concurrent execution because, in general, two conflicting statements cannot execute in a different order without affecting the program's result. A restructurer must know all dependences in order to preserve them. However, not all dependences are essential to produce the program's result. CURARE attempts to transform the program so it computes its result with fewer conflicts. An optimized program will execute with less synchronization and more concurrency.

CURARE then examines loops in a program to find those that are unconstrained or lightly constrained by dependences. By necessity, CURARE treats recursive functions as loops and does not limit itself to explicit program loops. Recursive functions offer several advantages over explicit loops since they provide a convenient framework for inserting locks and handling the dynamic behavior of symbolic programs. Loops that are suitable for concurrent execution are changed to execute on a set of concurrent server processes. These servers execute single loop iterations and therefore need to be extremely inexpensive to invoke.

Restructured programs execute significantly faster than the original sequential programs. This improvement is large enough to attract programmers to a multiprocessor, particularly since it requires little effort on their part. Although restructured programs may not make optimal use of a multiprocessor's parallelism, they make good use of a programmer's time.

Committee Chair

This research was funded by DARPA contract numbers N00039-85-C-0269 (SPUR) and N00039-84-C-0089 (XCS) and by an NSF Presidential Young Investigator award to Paul N. Hilfinger. Additional funding came from the California MICRO program (in conjunction with Texas Instruments, Xerox, Honeywell, and Phillips/Signetics).

Acknowledgements

The germ of this work began five years ago at Bolt Beranek and Newman in Cambridge, Massachusetts. I spent a year working with their Butterfly multiprocessor. That experience convinced me that explicitly controlling parallelism is a difficult and error-prone way of programming. My supervisor, Donald Allen, posed the problem of automatically producing parallel Lisp programs. I claimed that the approach was impractical, if not impossible. This thesis demonstrates that I was wrong.

Many people provided invaluable assistance at various stages of this project. Mike Harrison answered my many questions about automata and text processing tools. Larry Carter provided the reference to Tarjan's pointer machines. Ken Rimey suggested the network flow algorithm used in Chapter 5. David Wood discussed the difficulties in implementing various synchronization primitives in a memory system. Tom Reps and his colleagues improved my understanding of dependence analysis, both through their papers and in private conversations. Bert Halstead and Jim Miller answered my questions about parallel Lisp systems and encouraged this work.

My officemates—Kinson Ho, Ken Rimey, Luigi Semenzato, Ed Wang, and Ben Zorn—made an over-crowded and under-ventilated office a pleasant place to spend my time.

The SPUR research project supported me for many years while I explored various facets of Lisp and multiprocessing. The many members of SPUR were not only colleagues, but also good friends. Professors Dave Patterson, Randy Katz, John Ousterhout, and Paul Hilfinger demonstrated that graduate students can produce interesting research and build systems as well. For several years, I was also supported by a California Microelectronics Fellowship.

The members of my committee, Richard Fateman and Charles Stone, provided helpful comments on my dissertation. Morris Katz and Ken Rimey greatly improved this thesis with their comments. My advisor, Paul Hilfinger, strongly supported this work, even when I was not quite sure that it was possible. His Delphic and perceptive comments were inevitably right on target when I learned enough to understand them.

Jeremy Larus-Stone thoughtfully held off his arrival until SPUR was over and my thesis was nearly done. My parents instilled in me the belief that a PhD was worth obtaining; however they never dreamed that it would be a PhD in Computer Science. My wife, Diana Stone, deserves my endless gratitude for happily accepting years of promises that I would finish in six months. She greatly improved this thesis and only occasionally quibbled at the jargon. Without Diana, these years would have been a lot less fun.

Contents

List of Figures

The style in which it was written was the curious jewelled style, vivid and obscure at once, full of argot *and of archaisms, of technical expressions and elaborate paraphrases, that characterizes the work of some of the finest artists of the French school of* Symbolistes.
–Oscar Wilde, The Picture of Dorian Gray

Chapter 1

Introduction

Symbolic programs are very different from numeric programs. Nevertheless, both share a common shortcoming: they often execute too slowly. Users of both types of programming languages have sought to remedy this problem with more effective compilers and faster computers. However, specialized, fast computers are expensive and their use is limited. In addition, fundamental physical constants and difficult engineering problems limit the eventual speed of conventional, von Neumann-style computers. Inexpensive, shared-memory multiprocessor computers offer a promising remedy because of their greatly increased performance for programs able to take advantage of parallelism.

The Lisp community has not embraced these computers. Its reluctance stems in part from Lisp's complex data structures and its data-dependent execution behavior. These factors complicate programming multiprocessors and automatically translating sequential programs into parallel programs. Because the latter task is more difficult, most research has concentrated on developing extensions to Lisp for controlling parallel execution. Paradoxically, these extensions make multiprocessors less attractive because they introduce a different program semantics for parallel execution and force programmers to rewrite their programs to take advantage of these machines.

By contrast, CURARE, the program restructurer described in this thesis, automatically transforms a sequential Lisp program into an equivalent concurrent program.[1] CURARE can be seen as a compiler for a new type of computer that contains multiple, asynchronous processing elements. The source language is Scheme, augmented with a few declarations. CURARE detects a program's data dependences and identifies its recursive functions that could benefit from concurrent execution. Restructured functions execute their iterations on concurrent server processes. The restructured program produces the same result as the original program. Figure 1.1 depicts the translation process.

[1] *Restructuring* is the process of automatically preparing a sequential program for concurrent execution. Parallelizing is a more descriptive but ugly word that sounds too much like paralyzing.

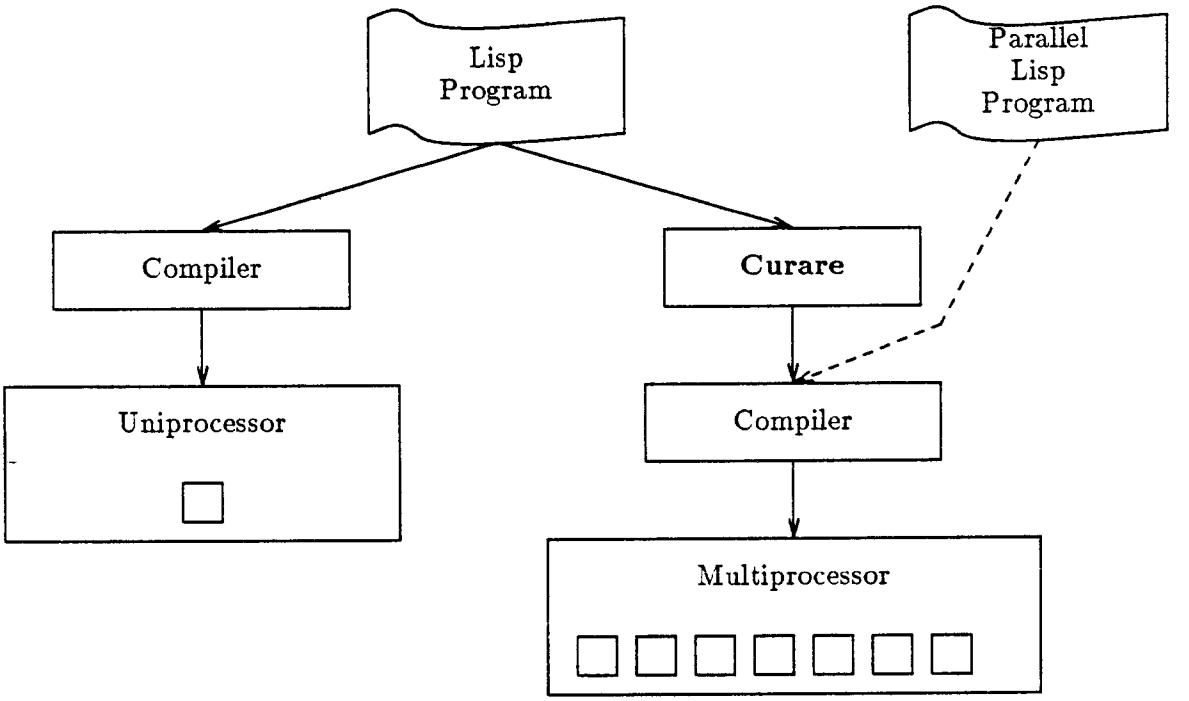


Figure 1.1: CURARE restructures a sequential Lisp programs so that it executes concurrently on a multiprocessor.

This transformation process is complicated by two of Lisp's features. Its ubiquitous pointers require extensive analysis to detect data dependences. And, Lisp's dynamic execution behavior requires optimizations to improve a program's concurrency and flexible scheduling of the parallel tasks to achieve good performance. This thesis describes the methods CURARE uses to handle these problems. Measurements show that CURARE's approach achieves good performance for many different styles of loops as well as for real programs.

1.1 Programming Multiprocessors

Parallel computers are programmed in two ways. Either a programmer writes in a language containing explicit features for controlling a program's parallel execution or a restructurer transforms a program written without these features into a concurrent program. Both approaches have advantages. By carefully controlling a program's parallelism and communication, a programmer can optimize the program to make maximum use of a particular parallel computer. Also, some applications fit nicely on certain parallel computers so that an explicitly parallel program is a natural expression of an algorithm. Finally, many programmers have more confidence in low-level languages that give a programmer substantial control over the executed code than in more abstract translation processes in which an imperfect, mechanical translator determines most of the details.

On the other hand, languages without explicit parallel constructs—conventional languages and languages designed for parallel processing—offer advantages as well. A program written in these languages is not tied to a particular computer architecture, and, given translators, will execute efficiently on a wide variety of machines. These translators perform low-level and machine-specific optimizations that a programmer would not attempt because they are too error-prone or too destructive of a program's structure. Finally, developing programs in these languages is easier since debugging can be done on a sequential computer where the program can be deterministically reexecuted to understand the sequence of events leading to an error. In general, this approach makes multiprocessors more attractive to the large group of programmers who are unwilling to rewrite their programs for these machines.

The parallel execution of a restructured program is predicated on good translators that produce efficient, parallel programs from sequential ones. The best examples today are FORTRAN translators—for example, Parafrase [49], PFC [6], and PTRAN [4]—that transform a sequential FORTRAN program into a concurrent program for a vector-processor, multiprocessor, or other parallel computer [20]. Newer, and not yet as effective, are translators for side-effect-free, applicative languages [75] and single-assignment languages [24].

This thesis explores the process of restructuring another type of program: non-numeric or symbolic programs with side effects. These programs are common and best characterized by the programming language features that they use: data structures linked by pointers into arbitrary graphs, conditional statements that give the programs an unpredictable control flow, and many small functions that are frequently invoked. Programs with these characteristics commonly are labeled as artificial intelligence (AI) programs, but the category is much broader and includes compilers, CAD tools, simulators, etc. These features complicate the translation process and are the subject of this dissertation.

The next section of the introduction briefly discusses the correct, concurrent execution of programs with side effects. Section 1.3 describes the translation process used by CURARE and outlines this thesis. Section 1.4 surveys previous work on multiprocessor Lisp.

1.2 Concurrent Execution

Intuitively, a transformed concurrent program should compute the same (correct) result as its sequential version. In general, some sequential programs may be non-deterministic and may produce many equally correct results. A concurrent program can produce any of these answers. Each of these results is the product of a sequential execution of the program. Therefore, we will focus on the more constrained task—producing the same result as a deterministic program.

Researchers in the field of databases have extensively studied the problem of correctly executing concurrent tasks with overlapping data. The theory from this area describes the correct execution of a concurrent program. Assume that we have a set of statements to execute concurrently. These statements read and write shared data, which introduces the well-known problems that result when side effects and concurrency meet. For example, if

our statements are $x \leftarrow x + 4$ and $x \leftarrow x * 7$ and the initial value of x is 3, then, ignoring the possibility of interleaved execution, the final value of x may be 49 or 25. Which answer is correct and why?

Database theory compares a concurrent execution of the statements to their sequential execution and declares the concurrent execution correct if the two are equivalent. Three common definitions of equivalence are: final-state equivalence, view equivalence, and conflict equivalence. A *schedule* is a list of statements from a program in the order in which they actually execute. A statement may appear more than once. These equivalence relations compare schedules from two executions of the same program.

Two schedules are *final-state equivalent* if the values of all variables are identical at the end of the schedules. Schedules are *view equivalent* if the value of variables read by the same occurrences of statements in each schedule are identical. Both definitions of equivalence have some abnormalities, but their major flaw is that the problems of deciding if a schedule is final-state or view equivalent to another schedule are NP-complete (see, for example, Papadimitriou [61]).

The third relation, conflict equivalence is the criterion of choice because checking it is not NP-complete. Two statement occurrences *conflict* if they both access the same location and at least one of them modifies the value in the location. Two schedules are *conflict equivalent* if they resolve all conflicts in the same way. In other words, the same statement from each conflicting pair first reads or writes the location in both schedules. This definition permits substantial freedom to rearrange statements in a schedule so long as conflicting statements do not exchange order.

At this point, we diverge from database practice. In that area, the tasks are transactions that arrive unannounced and execute concurrently with other transactions. A set of transactions executes correctly if they are conflict equivalent to some sequential execution. This standard is called *conflict serializability*. Deterministic programs, on the other hand, have only a single execution order so their standard of correctness is conflict equivalence with this order, which is known as *conflict sequentializability*.

It is easy to imagine situations in which this standard is too restrictive. For example, consider a function that adds the integers in a list. The parallel version of this function need not duplicate the sequential additions to produce the same result. However, if the sequential function produces side-effects—if the partial sums are accumulated in a variable—then conflict sequentializability would prevent reordering and concurrency. Most programming languages do not provide a way of specifying that the order of operations is irrelevant, so a restructurer must detect special cases such as these additions.

Database systems serialize their transactions by locking the data that a transaction reads or writes to prevent other transactions from imposing their conflicting reads or writes among those of the first transaction. Locking is a natural strategy when all accesses to shared data go through a central data manager that is unaware in advance of each transaction's data requirements. More sophisticated schemes—such as one in the SDD-1 distributed database [11] that preanalyze a transaction's requirements and schedule tasks to reduce conflicts—have been proposed but are not widely used.

On the other hand, a program transformation system has the source code of all tasks and can, in theory, analyze the potential conflicts; find a correct, concurrent execution order; and, ensure this order with less overhead than general locking. Locking overhead is particularly important for programs in which the cost of a non-conflicting access, such as a variable, array, or structure reference, is very small. However, the usual undecidability questions about programs, the need to approximate answers during program analysis, the unpredictability of execution times, and the large number of possible program behaviors, force a transformation system into heuristic methods that guarantee a correct, but not perfect solution. That is the case for CURARE, which does not attempt to find an optimal schedule, just a correct one.

To restructure programs properly, we must classify conflicts precisely. A *data dependence* is the relation introduced between two conflicting statements. There are three types of data dependences: a *flow dependence*, in which one statement writes a value read by the other; an *anti-dependence*, in which one statement reads a location subsequently written by the other; and a *def-order (or output) dependence*, in which both statements write the same location. Dependences can be *loop-independent* or *loop-carried*, depending whether the conflicting statements execute in the same or different loop iterations. The *distance* of a loop-carried conflict is the number of loop iterations separating the conflicting statements. Another type of dependence occurs when two statements are *control dependent* because the execution of one is contingent on the value of the other. Dependences are discussed in greater detail in Section 4.1.

Program analysis can easily find control dependences and some data dependences, in particular, those involving variables. Other dependences—for example, those between statements referencing structures—are much more difficult to define precisely and are a major focus of this thesis. The algorithms developed in this work detect a superset of the data dependences among structure references and thereby ensure a program's correct concurrent execution. Sometimes the analysis is too conservative and a programmer must refine it with declarations that increase the program's potential concurrency by eliminating false conflicts.

Once CURARE determines a program's dependences, it can restructure the program into a concurrent program. Restructuring has two phases. The first, called *optimization*, changes the program's structure dramatically to increase its potential concurrency by removing dependences. These transformations do not preserve conflict equivalence and are verified in other ways. The second phase, *code-generation*, finds areas of the program that could benefit from concurrent execution and introduces low-level parallel constructs. These transformations preserve conflict equivalence. These phases are called transformations because their results are expressed in the same language as their inputs (Lisp in this case).

This thesis concentrates on finding parallelism in the most likely area: program loops. Other areas of potential parallelism, such as the parallel evaluation of expressions, can be handled similarly.

1.3 Overview of Thesis

This thesis describes the pieces of CURARE in the opposite order to the one in which they are applied. This presentation results in a bottom-up description of CURARE that illustrates how a component is used before explaining how it is constructed.

Chapter 2 describes how recursive functions can be transformed to execute concurrently in spite of their control and data dependences—which must be detected by program analysis. The transformed functions spawn portions of their bodies as concurrent tasks, which are evaluated by servers running on multiple processors. Data dependences between these tasks must be serialized with locks. This chapter also shows how to insert the fewest locks necessary for sequentialization.

Chapter 3 describes transformations that eliminate data dependences and further reduce the need for locks. These optimizations improve a program's concurrent behavior by permitting CURARE more freedom in scheduling and executing parallel tasks and by reducing the cost and delays due to synchronization.

Both of these chapters assume that a program's data dependences are known precisely. Chapter 4 describes a data-flow framework for detecting and classifying data dependences. This framework is independent of the way in which a program references objects. Chapter 5 applies this framework to the difficult problem of detecting dependences between references to data structures connected by pointers.

Chapter 6 describes some declarations by which a programmer can refine the dependence analysis and enable additional dependence-removing optimizations.

Finally, Chapter 7 presents some measurements that validate CURARE's execution model and show the performance of several transformed programs.

The index at the end of the thesis points to definitions of terms.

Figure 1.2 shows CURARE's overall arrangement. The double boxes enclose components of CURARE described in detail in this thesis. The components in the other boxes are either well-known or beyond the scope of this work and can be explored by following the appropriate references in the thesis. CURARE contains four major pieces.

1. Data-dependence analysis finds potential conflicts among statements in a program and describes them in a program dependence graph (PDG) similar to the one described by Ferrante *et al.* [21]. The data-flow analysis framework established in Chapter 4 carries over to the difficult problem of detecting conflicts among structure accesses, which is discussed in Chapter 5.

2. This analysis, however, is conservative and can detect conflicts that do not occur. Chapter 6 describes a set of programmer-supplied declarations that refine the analysis by providing information that is not available in a program's text and by correcting conservative analytic results.

3. Program optimizations remove some dependences and increase potential concurrency by eliminating synchronization. These optimizations are described in Chapter 3.

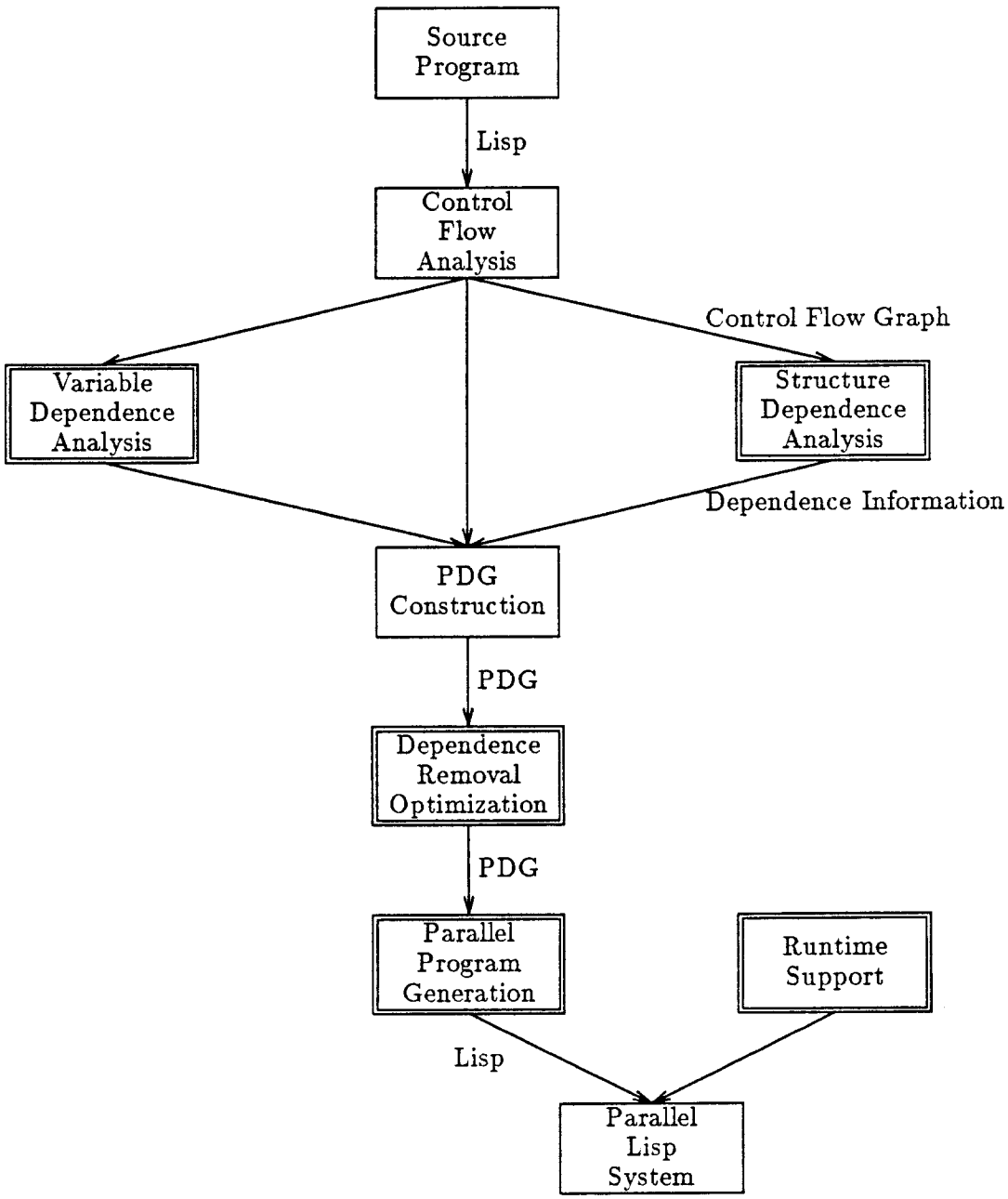


Figure 1.2: Overview of CURARE's transformation process. CURARE first analyzes a program to uncover its control and data dependences. It then produces a parallel program that executes, with a small support library, in almost any concurrent Lisp system. The double boxes enclose pieces of CURARE described in detail in this thesis.

4. The concurrent execution of a restructured program results from a combination of program transformations that introduce parallelism and synchronization into a sequential program and run-time mechanisms that support this concurrency. Chapter 2 describes both.

The details of the target parallel Lisp system are not particularly important to CURARE, since it only requires concurrent processes and inexpensive locks. However, the processes must execute on a shared-memory multiprocessor so each task can access any object in the heap. Lisp systems based on message-passing (for example, CCLISP running on hypercubes [12]) require major changes to Lisp and CURARE.

1.3.1 A Note on Notation

The sample programs in this thesis are written in a slightly extended version of Scheme. Because standard Scheme does not provide structures or a generalized location updating mechanism, CURARE accepts the `defstruct` and `setf` constructs from Common Lisp [67]. This thesis does not discuss the analysis of Scheme-specific features, such as the use of functions as general values or `call-with-current-continuation`. These constructs are used and CURARE could analyze them with the techniques used in Scheme compilers [47] or those proposed by Shivers [66], but their complications are orthogonal to this work.

The examples shift back and forth between Lisp structure accessors, e.g., `(car x)`, and a Pascal-like notation, `x.car`. In the latter notation, `x` contains a pointer that is automatically dereferenced, so that it would be written `x↑.car` in proper Pascal.

Almost all of the examples use cons-cells as their data structure. This bias is not because our data-dependence analysis cannot handle more general structures—it can—but rather because programs written with `cons`, `car`, and `cdr` are shorter and require less background context.

The algorithms in this thesis are written either in a language tenuously related to SETL [65] or in English, depending on the level of detail and the complexity of the data structures. CURARE is written in Common Lisp and is, of course, less concise because of error-checking and efficiency concerns.

1.4 Related Work

Although most research in multiprocessor Lisp systems has explored explicit parallelism, a few researchers have investigated restructuring Lisp. We will discuss their approaches first and then briefly survey language extensions for parallelism.

1.4.1 Restructuring Lisp

Gray investigated inserting futures into side-effect-free Lisp programs [29]. A *future* is a construct in Multilisp [31] that creates a process to evaluate an expression and returns a

token that synchronizes any process that reads the expression's value. Gray sought concurrency in two places. The first was concurrent execution of syntactically parallel tasks, such as the evaluation of actual arguments in a function call. The second was overlapping execution of a subexpression and its containing expression, for example, executing a function concurrently with the computation of its arguments. Since side-effect-free programs contain only simple flow dependences, data-dependence analysis was unnecessary and futures provided all synchronization. Her major difficulty was to avoid spawning too many processes. A process is useless if the cost of its task is less than the cost of creating the process or if another process soon waits for the first process' result. The performance of Gray's approach is unclear since she only presented a speedup curve for one small benchmark.

Boyle *et al.* also transformed pure (side-effect-free) Lisp programs into concurrent programs [15]. They derived parallel FORTRAN programs from sequential Lisp specifications by applying many small rewriting rules. Syntactic transformations of this type are easier for applicative languages, which have the Church-Rosser property, than for languages with side effects in which conflicts are not syntactically visible. Like Gray, Boyle also faced the problem of producing too many processes for the available processors and made the crucial distinction between a process and a server for a repeatedly executed task (see Chapter 2). They presented impressive speedup curves for a non-trivial, transformed version of their transformation system running on a multiprocessor.

Harrison was one of the first to propose a technique for concurrently executing non-pure Lisp [32,33]. His transformation system, PARCEL, requires a new representation for lists that allows the use of parallel algorithms normally associated with numeric programs. This list representation, which is similar to cdr-coding [14], stores the *car* pointers of a list segment contiguously, without the *cdr* pointers. The representation also contains the number of elements remaining in the list. Its main advantage is that the i^{th} element of a list can be obtained in nearly constant time by treating the block of pointers as a vector. Given this property, parallel prefix algorithms can apply an associative function to n elements of a list in $\log n$ steps.

Harrison's technique, however, is not general. The new list representation forbids direct side effects on list cells because of the unpredictability that results from the extensive sharing of sublists necessary to make these structures as flexible as conventional lists. Harrison described replacements for many destructive list operations, such as nconc, but not for the fundamental operation of replacing a structure field. His representation is also specific to list structure and cannot extend to user-defined data structures. PARCEL's data-dependence analysis relies on the Scheme programming paradigm of implementing mutable objects as collections of variables in function closures. By combining variable flow analysis with closure lifetime analysis, Harrison can eliminate some dependences as infeasible.

Katz proposed the first general approach for concurrently executing Lisp programs with side effects [42]. His system, ParaTran, relies on a combination of program transformations and run-time error-checking to implement optimistic concurrency. In this scheme, which is borrowed from databases, a transaction is assumed to have few conflicts with other transactions. Transactions execute without regard to possible conflicts. A monitor examines

the read and write traces for each transaction to detect conflicts and restart transactions that violate serializability. A transaction commits its results when it can no longer conflict with any other transaction.

To achieve good performance with this approach, few transactions should conflict and need to be restarted, and testing for a conflict must be inexpensive. Katz depends on undescribed program analysis to find the potential processes in a sequential program and to encapsulate shared reads and writes with the appropriate tests. The results presented in a recent paper are simulated speedup curves that appear to level off rapidly for a small number of processors [73].

1.4.2 Explicitly Parallel Lisp Systems

The other approach to executing Lisp programs concurrently is to write them in a parallel Lisp dialect and run them on a multiprocessor Lisp system. Several such systems operate on a variety of computers.

Halstead's Multilisp was the first operational parallel Lisp system [31]. This dialect encourages functional programming with its two primary constructs of futures and parallel argument evaluation. Multilisp, however, is a general Scheme implementation that provides the usual side-effect-producing operations. Multilisp's byte-code interpreter slowed program execution substantially and the system ran on a one-of-a-kind multiprocessor. Nevertheless, it demonstrated the potential of multiprocessor Lisp and the power of the future construct.

BBN's Butterfly multiprocessor has three parallel Lisp implementations. The first is Miller's MultiScheme [58], which is a parallel Scheme dialect also based on futures. Miller reimplemented Multilisp's features in a higher-quality Scheme system and investigated the appropriate primitive constructs for building parallel Lisp systems and supporting futures and user-definable process scheduling. BBN extended MultiScheme with a compiler and Common Lisp features to produce the first multiprocessor Common Lisp [3].

The third Lisp for the Butterfly is Portable Standard Lisp (PSL), which Swanson, Kessler, and Lindstrom ported to the Butterfly [69]. Their initial system did not correctly implement futures (programs had to touch futures explicitly to recover their values), but it did demonstrate the feasibility of the operations necessary to build an efficient system.

Gabriel and McCarthy developed another major parallel Lisp dialect, Qlambda [23]. This language introduced concurrency into several Lisp commands. The extended variable binding command can evaluate variables' initial-value expressions in parallel and possibly overlap their execution with that of the statements that use the bindings. The function constructor can produce concurrent functions that execute asynchronously and read their arguments from a queue. Lucid, Inc. is extending its commercial Common Lisp with these features to produce a Lisp (Qlisp) for the Alliant multiprocessor [26]. Experience has shown the inadequacy of Qlambda's language extensions and has led to increasingly complex language constructs [27].

SPUR Common Lisp contains a set of multiprocessor features that take the opposite approach [78]. These features are simple primitives that either can be used directly or can form the basis of more complex language features. We will use these constructs, which are

briefly described in Section 2.1, in the parallel versions of the transformed programs. Franz Inc. is building a Common Lisp for the Sequent Symmetry with these features.

Regarding the actual making of curare,
there are almost as many conflicting accounts
as there have been witnesses....
– *Richard G. Gill,* White Water and Black Magic

Chapter 2

Parallel Execution of Loops

Data-dependence detection algorithms and declarations expose a program's dependences. These dependences constrain the program's concurrent execution but do not prescribe its concurrent behavior. The program's execution depends on its expected behavior, the frequency of dependences, the target multiprocessor, and the program restructurer. In this chapter, we describe Curare's technique for executing loops (including loops formed by recursion) concurrently on asynchronous, shared-memory multiprocessors.

Loops are a natural place to seek concurrency. They repeatedly invoke a piece of code on a set of data, which raises the possibility of simultaneously executing the operations on each value. If the task can execute concurrently, the loop's execution time is reduced by the number of parallel processes. Many programs spend most of their time in a few loops, so concurrent execution can greatly improve the speed of the entire program.

Data dependences divide loops into four categories:

1. Naturally parallel loops with no loop-carried dependences.

2. Recurrences in which results from previous iterations are used in the current iteration. In other words, the loop body contains loop-carried flow dependences.

3. Loops containing an anti-dependence in which a statement reuses a location read in an earlier iteration.

4. Loops containing a def-order dependence in which statements in two iterations write the same location.

Only the first type of loop can execute correctly without synchronization. Fortunately, optimizations that change the locations used by statements can eliminate dependences in some of the latter three types of loops. The remaining dependences must be serialized by synchronization devices such as locks.

The execution cost of a loop body will vary if it preforms different operations on each value. This variance requires a flexible, run-time scheduling mechanism to allocate tasks

efficiently to the available processors. This mechanism should also schedule the iterations to reduce delays due to synchronization.

This chapter describes a scheme for concurrently executing loops that do not contain many data dependences. The approach is simple and can be efficiently implemented in most concurrent Lisp systems. It only requires parallel processes, queues, locks, and function closures.

All loops that CURARE restructures are written as recursive functions. As Steele argued, recursion is a more fundamental control structure than iteration since all explicit loops can be rewritten trivially as tail-recursive functions [68]. Scheme programmers use recursion heavily, so a program restructurer must handle this feature. Recursion also provides a convenient and flexible framework for concurrently executing Lisp loops, particularly those with dependences.

Because of the close relation between recursion and iteration, we use the terms interchangeably. An *iteration* or *invocation* of a recursive function is a single execution of the function's body.

The next section briefly describes several of SPUR Lisp's multiprocessing features. Section 2.2 describes the server model and shows how it executes simple recursive functions that do not contain dependences. Section 2.3 expands the model to include functions with data dependences. Section 2.4 discusses how locks synchronize the conflicts. The next two sections expand the model to accommodate complex recursive functions and control dependences. Section 2.7 describes how servers are allocated to loops. The final section surveys related work.

2.1 Multiprocessor Language Features

The server model described in this chapter is not closely tied to a particular multiprocessor or concurrent Lisp system. Nevertheless, we need a concrete language in which to demonstrate the model. We primarily use SPUR Lisp's multiprocessing extensions [78], which are well-suited for this role. The features described below are slight extensions of the actual constructs. The changes simplify the presentation but are not essential.

SPUR Lisp provides three sets of primitives: processes, mailboxes, and signals. We only need the first two. A *process* is a concurrent thread of control created by the `make-process` function to evaluate an expression. SPUR Lisp provides a variety of operations to control processes, none of which we use.

Mailboxes are FIFO queues. The function `make-mailbox()` returns an unbounded mailbox. The function `send(item, mailbox)` returns when the item is enqueued in the mailbox. The function `receive(mailbox)` returns the next item from the mailbox. Each mailbox also contains two user-definable fields, `mb-barrier` and `mb-count`.

We also use *locks*, which are not part of SPUR Lisp. `make-lock(state)` returns a lock that is initially `locked` or `unlocked`, depending on the argument. The function `acquire-lock(l)` returns when the process locks `l`. If multiple processes simultaneously try to lock a lock, only one process will succeed. The other processes wait. The function

`release-lock(l)` atomically changes the lock's status to `unlocked`. Locks are heavily used and should be directly implemented with the low-level features provided by most multiprocessors.

Another non-SPUR Lisp construct is *counters*. The function `make-counter(n)` returns a counting semaphore initialized to `n`. The function `decr-counter(s)` reduces the value of the counter `s` by 1. The function `wait-counter(s)` returns when the value of the counter reaches 0. Counters are easily implemented with locks although some computers support them directly.

2.2 Parallel Execution Without Data Dependences

Before describing the details of CURARE, we will present a simplified example that illustrates the main points. The simplest loop to execute concurrently is one in which no data dependences extend between iterations. For example, consider the function:

```
(defun mapc (f lst)
  (cond ((null? lst))
        (t
          (f (car lst))
          (mapc f (cdr lst)))))
```

which applies a function, `f`, to each element of a list, `lst`. If invocations of `f` do not conflict, then `mapc` contains no loop-carried dependences. Each iteration, however, is control dependent on the termination test in previous iterations.

CURARE's first step in restructuring `mapc` is to move the recursive call so that it occurs as early as possible in each iteration. The recursive call will spawn a new process, so the sooner it happens, the more concurrency is possible. Since there are no conflicts, the invocation of `f` and the recursive call have no dependences and can be exchanged without affecting `mapc`'s result:[1]

```
(defun mapc1 (f lst)
  (cond ((null? lst))
        (t
          (mapc1 f (cdr lst))
          (f (car lst)))))
```

CURARE then changes the recursive call into a parallel call, which, conceptually, spawns a new process to evaluate the next iteration:

[1] The examples follow the convention that refinements of a function are numbered in increasing order: e.g., **mapc**, **mapc1**, **mapc2**, ...

```
(defun mapc2 (f lst)
  (cond ((null? lst))
        (t
          (make-task mapc2 f (cdr lst))
          (f (car lst)))))
```

The call may execute asynchronously because the code executed by the call does not conflict with the statements following the call. `mapc` is now a parallel function. Each invocation examines its arguments to see if all tasks are complete, and if not, spawns a task to attend to the rest of the list and executes `f` on the head of the list.

These tasks do not require full-fledged processes, which have potentially large overhead in some systems. Instead, tasks are closures of functions over their actual arguments. They are enqueued and executed by a server running on one of several processors. A server executes a simple loop:

```
(defun server (queue)
  (let ((task (receive queue)))
    (apply (task-function task) (task-arguments task))
    (server queue)))
```

`make-task` simply saves its arguments on the servers' queue:

```
(defun make-task (function . arguments)
  (send (make-task-object function arguments) task-queue))
```

where a `task-object` is a record of the function and its actual arguments and the variable `task-queue` contains the servers' queue.

2.2.1 Restructuring Simple Recursive Functions

Simple recursive functions have no loop-carried data dependences and are *linearly recursive* (i.e., they have at most one recursive call on any path from the function's entry to exit). This section examines the restructuring process for simple recursive functions. The result of this process is a concurrent function that distributes its work among a set of concurrent servers. The servers can also execute more complex recursive functions, such as those with data dependences (Section 2.3) and non-linearly recursive calls (Section 2.5).

Throughout this section, we assume that a function contains only a single recursive call to simplify the discussion. The technique applies equally well to functions containing several recursive calls along different paths.

The first step in restructuring these functions is to move the recursive call so it executes as early as possible during an invocation of the function. The earlier the call occurs, the sooner another concurrent task is created and the more parallelism results.

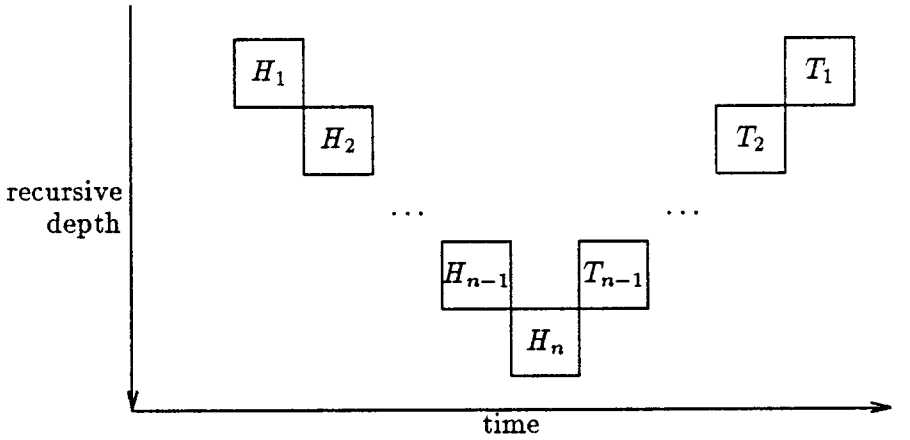


Figure 2.1: In a sequential, simple recursive function, the function heads execute in order until the recursion terminates. Then the function tails execute in reverse order.

A function's *head* contains the statements that execute before a recursive call. These are the statements along acyclic paths to the call statement. The *tail* contains all statements that execute after a recursive call. These are the statements dominated by the call.

To reduce the size of a function's head, we move statements to its tail. However, a statement S cannot move after a recursive call C if:

1. There exists a control or data dependence (transitively) from S to C.
2. S has a loop-carried dependence with a statement T. Moving S from the head to the tail reverses the execution order of S and T.

After moving the statements, the function's head contains statements that must execute before the recursive call either because the call is flow dependent on them or because they have a loop-carried dependence. Since simple recursive functions do not contain loop-carried dependences, the head only contains the former type of statements.

The next step checks that the recursive call may execute concurrently. Sequential recursion imposes a single execution order on the statements in a function. Executing the recursive call concurrently permits many orders, all of which must preserve dependences. Although we claimed that simple recursive functions contain no loop-carried dependences, we can relax this restriction to accommodate functions whose dependences are preserved by concurrent execution.

Figure 2.1 shows the normal order of execution of a recursive function's heads and tails. Figure 2.2 shows the concurrent execution order. Concurrent execution preserves loop-independent dependences and loop-carried dependences from a statement in the head to a statement in a later iteration since the parallel call executes after the head. Figure 2.3 shows the preserved dependences. However, loop-carried dependences leading to a statement in an earlier tail are not preserved

Spawning the recursive call will make parallel those simple recursive functions with no

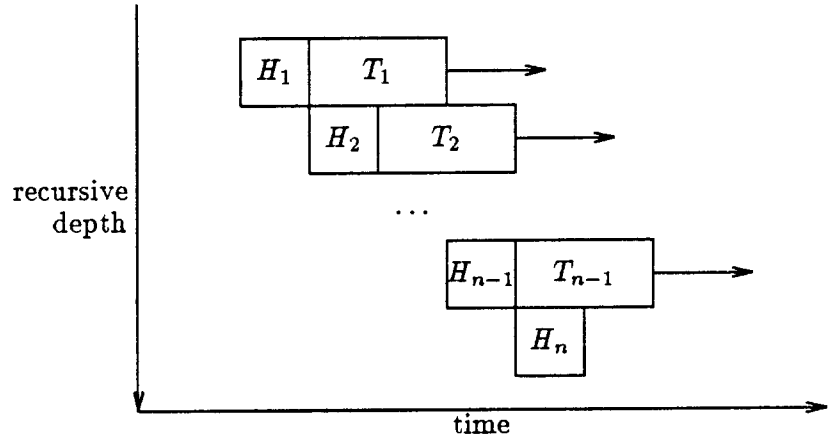


Figure 2.2: In a concurrent, simple recursive function, the recursive call spawns a new process, so the function's tails overlap subsequent iterations. In this figure, a block indicates the earliest time at which a tail can execute. The arrows show the range of time over which the code may execute.

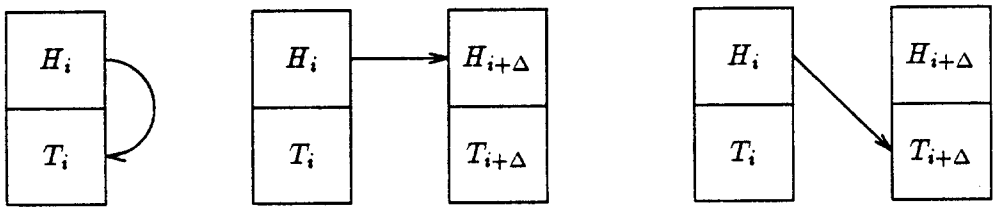


Figure 2.3: The dependences between statements in a function's head and other statements that are preserved when a recursive call executes concurrently.

loop-carried dependences or with dependences that are preserved. A function modified in this manner executes part of each invocation concurrently. However, functions that invoke the restructured function may not want to see this concurrency because the results of the computation may be incomplete. To avoid this, we add barrier synchronization to wait for the servers to complete their tasks. A restructured function has two entry points: a *synchronous* entry, which waits for all tasks to finish and preserves the sequential semantics; and an *asynchronous* entry, which returns when all tasks are scheduled. (Figure 2.4).

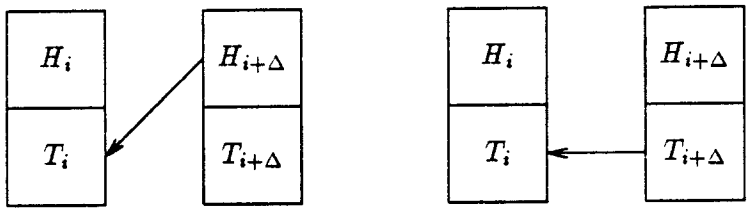


Figure 2.4: The dependences between statements in a function's tail and other statements that are not preserved when a recursive call executes concurrently.

To continue the mapc example:

```
(defun mapc-sync (f lst)
  (let ((task-queue (mapc-async f lst)))
    (server task-queue)
    (wait-counter (mb-barrier task-queue)))
  nil)

(defun mapc-async (f lst)
  (let ((task-queue (obtain-servers)))
    (mapc3 task-queue f lst)
    task-queue))

(defun mapc3 (task-queue f lst)
  (cond ((null? lst) (terminate-servers task-queue))
        (t
          (make-task task-queue mapc3 task-queue f (cdr lst))
          (f (car lst)))))
```

This example illustrates several important conventions. The function **obtain-servers** gets the servers for a function and directs their attention to a new task queue, which it returns. An optional argument to **obtain-servers**, omitted in the example, is the desired number of servers (see Section 2.7). The asynchronous version of a function returns the task queue, so the synchronous version can also service the queue. This function then waits on the counter (in the queue's **mb-barrier** field) for all servers to finish their tasks. The function **terminate-servers** redirects the servers when they finish. It cannot execute before scheduling the last iteration of the recursive function since it enqueues tasks that redirect the servers.

Recursive functions that return a result to previous iterations contain a loop-carried dependence and are discussed later.

2.2.2 Servers and Tasks

Servers are concurrent processes that execute function invocations. In a system in which creating processes is extremely inexpensive, **make-task** could create a new process for each task. However, in most systems, processes are likely to be much more expensive to create and schedule than tasks, and **make-task** will not create actual processes. In particular, the cost of creating and scheduling a process in machines with large register files, such as SPUR [34], will be much longer than a function call time because of the cost of saving and restoring registers. In general, the cost of creating a process will be higher than the cost of enqueuing a task, which contains the absolute minimum information necessary to execute the work concurrently. A fully general process mechanism requires more context

information. Tasks are also not preemptable, which simplifies their implementation. The server and queue approach is portable and can be used in any concurrent Lisp system.

Below, we describe a simple implementation of tasks and servers. A task is a pair consisting of a function and its actual arguments:[2]

```
(defstruct (task-object
             (:conc-name task-)
             (:constructor make-task-object (function arguments)))
  function
  arguments)
```

A server is a concurrent process that executes the loop:

```
(defun server (queue)
   (let ((task (receive queue)))
     (cond ((task-object? task)
            (apply (task-function task) (task-arguments task))
            (server queue))
           (t))))
```

A server reads an object from its queue. If the object is a task, the server executes the task's function and then looks for more work. Any other value causes the server to terminate.

The function `obtain-servers` finds idle servers for a loop, directs these servers to a new queue, and returns the queue. The number of servers assigned to a loop is determined partially by CURARE and partially by availablity of processors at execution time. For now, we postulate an oracle, `number-of-servers`, which takes the requested number of servers and returns the actual number of servers for a function (see Section 2.7). Idle servers are found waiting on the `*idle-server-queue*`.

[2]In systems in which consing the argument list and applying a function to a list are expensive, a task could be a closure that applies the function to the actual arguments. The choice depends on the cost of the two operations in a particular Lisp system.

```
(define *server-queue-lock* (make-lock 'unlocked))

(define *idle-server-queue* (make-mailbox))

(define *number-of-idle-servers* 0)

(defun obtain-servers (&optional desired-number)
  (acquire-lock *server-queue-lock*)
  (let ((n (number-of-servers desired-number)))
    (set! *number-of-idle-servers* (- *number-of-idle-servers* n))
    (release-lock *server-queue-lock*)

    (let ((queue (make-mailbox)))
      (set! (mb-barrier queue) (make-counter n))
      (set! (mb-count queue) (+ n 1))  ; Count this process also
      (obtain-n-servers n queue)
      queue)))

(defun obtain-n-servers (n queue)
  (cond ((= n 0))
        (t
          (make-task *idle-server-queue* idle-server queue)
          (obtain-n-servers (- n 1) queue))))

(defun idle-server (queue)
  (server queue)
  (decf-counter (mb-barrier queue))
  (acquire-lock *server-queue-lock*)
  (incf *number-of-idle-servers*)
  (release-lock *server-queue-lock*))
```

The function **obtain-servers** executes only one request at a time to eliminate a race between computing the number of servers and obtaining them. The function that **obtain-servers** enqueues invokes **idle-server**. When the call on **server** in this function returns, it decrements the barrier for its task queue and increments the number of idle servers. The process will then be back in the server loop for the idle server queue.

The server pool is initialized by assigning free processors to be servers:

```
(defun initialize-servers (number-of-processes)
  (set! *idle-server-queue* (make-mailbox))
  (set! *number-of-idle-servers* number-of-processes)
  (set! *server-processes* (initialize-n-servers number-of-processes))
  (set! (mb-barrier *idle-server-queue*)
        (make-counter number-of-processes))
  (set! (mb-count *idle-server-queue*) number-of-processes))

  (defun initialize-n-servers (number-of-processes)
    (cond ((= number-of-processes 0))
          (t
            (make-process (loop (server *idle-server-queue*)))
            (initialize-n-servers (- number-of-processes 1)))))
```

where `(loop e)` $\equiv$ `(letrec ((loop () (e) (loop))))`. These servers execute an infinite loop in which they wait to be assigned to a function, execute iterations for that function, and then return to the `*idle-server-queue*` to look for more work.

The function `terminate-servers` returns servers to the idle queue when a function finishes:

```
(defun terminate-servers (queue)
  (terminate-n-servers (mb-count queue) queue))

(defun terminate-n-servers (n queue)
  (cond ((= n 0))
        (t
          (send nil queue)
          (terminate-n-servers (- n 1) queue))))
```

It enqueues several `nil`'s, each of which terminate a server loop. The n servers from the idle pool return and look for other tasks. When the call on `server` in the synchronous function terminates, the function waits for the counter associated with the task queue to indicate that all other tasks are finished.

2.3 Restructuring with Data Dependences

Not all functions are as amenable to concurrent execution as `mapc`. Some functions contain loop-carried dependences that are not preserved by concurrent execution. These dependences must either be synchronized or removed. The next chapter discusses eliminating them. This section addresses the problem of serializing the conflicting statements.

Consider a dependence from statement S_1 to statement S_2. The simplest synchronization is a lock that is initially locked. The earlier statement (S_1) accesses the location and then

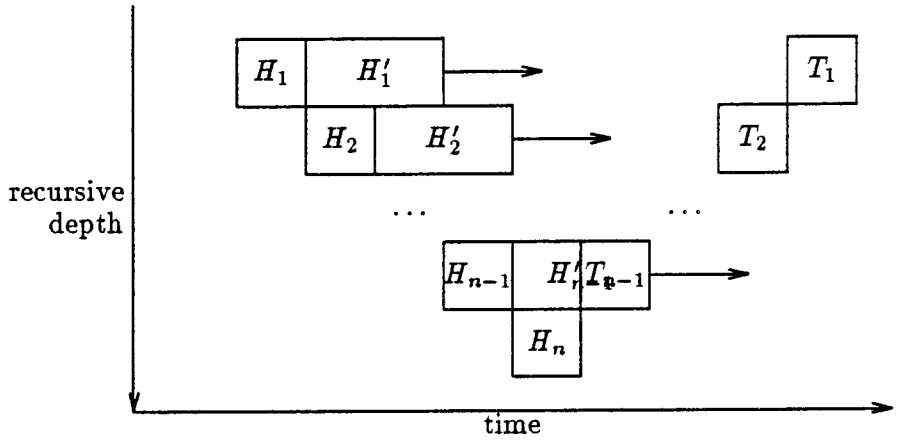


Figure 2.5: Tasks spawned in the head of a recursive function execute concurrently with subsequent iterations. However, the tails execute sequentially.

unlocks the lock. The other statement (S_2) tries to lock the lock and succeeds after S_1 finishes. Besides the cost of these additional operations, locking introduces the possibility of deadlock and complicates the server model. Nevertheless, it is necessary to ensure the correct execution of functions with data dependences.

The next subsection describes the concurrent execution of functions with loop-carried dependences. It assumes the simple locking outlined above. Section 2.4 shows how to improve the synchronization by inserting the minimal locks and by locking conflicting statements instead of memory locations.

2.3.1 Concurrent Execution with Dependences

In addition to necessitating synchronizations, dependences reduce concurrency by preventing the movement of statements, thereby decreasing the size of a function's tail. If the tail is too small, it is not worth spawning the call concurrently. Therefore, we will examine other approaches to scheduling functions with loop-carried dependences. These approaches spawn portions of a function's head or tail—not the recursive call.

The first technique spawns a task containing the statements from a function's head that have loop-carried dependences but are not the source of loop-independent dependences to the call or statements in the tail. These tasks are scheduled in order and overlap subsequent iterations (Figure 2.5). This technique is particularly appropriate for tail-recursive functions with loop-carried dependences.

For example, consider the function:

```
(defun f (lst)
  (cond ((null? (cddr lst)))
        (t
          (set! (car (cddr lst)) (+ (car lst) 1))
          (f (cdr lst)))))
```

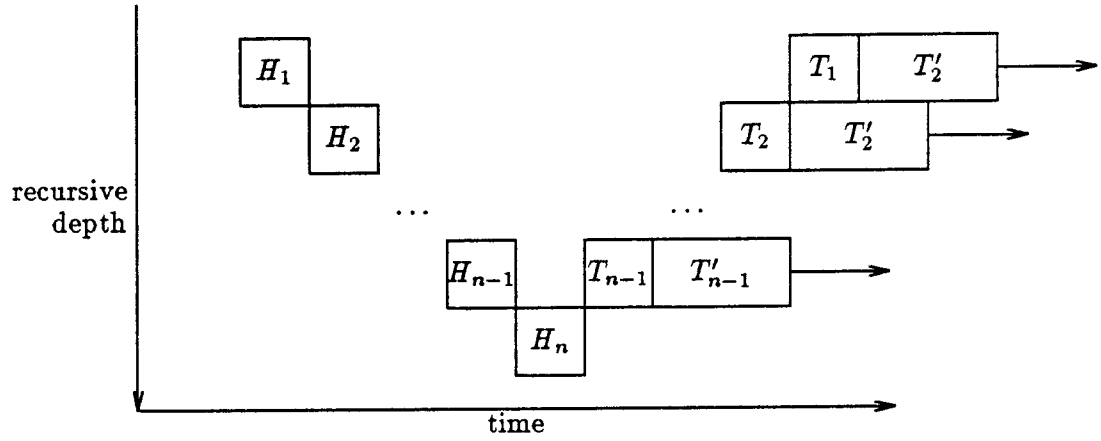


Figure 2.6: Tasks spawned in the function's tail overlap subsequent iterations.

The assignment statement has a loop-carried flow dependence of distance 2 with itself. The restructured function spawns this statement as a separate task:

```
(defun f1 (task-queue lst)
  (cond ((null? (cddr lst))
         (terminate-servers task-queue))
        (t
          (make-task task-queue
                     (lambda ()
                       lock (l.car)
                       (set! (car (cddr lst)) (+ (car lst) 1))
                       unlock (l.cddr.car))
                     ())
          (f1 task-queue (cdr lst)))))
```

The *lock* and *unlock* statements protect the locations described by their arguments and are discussed below. The first two locations in the list are initially unlocked. All other elements are initially locked. Since `f1` schedules tasks in order, they cannot deadlock.

All data dependences involving statements in the spawned region must be synchronized. Dependences between statements in the rest of the function's body are preserved without synchronization, since these statements execute in the usual order.

A similar approach works for statements in the function's tail (Figure 2.6). For example, consider the function:

```
(defun g (lst)
  (cond ((null? (cddr lst)))
        (t
          (g (cdr lst))
          (set! (car (cddr lst)) (+ (car lst) 1)))))
```

which has a loop-carried anti-dependence between instances of the last statement. The restructured function spawns and synchronizes this statement:

```
(defun g1 (task-queue lst)
  (cond ((null? (cddr lst)))
        (t
          (g1 task-queue (cdr lst))
          (make-task task-queue
                     (lambda ()
                       lock (lst.car)
                       (set! (car (cddr lst)) (+ (car lst) 1))
                       unlock (lst.cddr.car))
                     ()))))
```

The function traverses the list and schedules the tasks as the recursion unwinds. Because the task queue is FIFO and dependences in the tail run from later to earlier iterations, tasks do not deadlock. The elements of the list, with the exception of the last two, are initially locked. Also, the servers cannot terminate until the recursion is fully unwound, so the function **terminate-servers** reassigns servers when the restructured function returns:

```
(defun g-async (lst)
  (let ((task-queue (obtain-servers)))
    (g1 task-queue lst)
    (terminate-servers task-queue)
    task-queue))
```

The number of servers that can be profitably used by a function with dependences depends on the distance of the smallest loop-carried dependence and the amount of code not involved in the dependence. In f or g, no more than 2 servers are necessary or useful.

To estimate the potential speed improvement from spawning pieces of a function, we need to know the execution cost of various statements. In general, assume that the concurrently executed head or tail has the following form:

where S are the statements executed between the two statements that conflict at distance d, B are the statements executed before S, and A are the statements executed after S. Let T_S, T_B, and T_A be estimates of the execution time of each group of statements. Then, the upper bound on the number of concurrently executed server processes is

$$d\left\lfloor\frac{T_A}{T_S}\right\rfloor + d\left\lfloor\frac{T_S}{T_S}\right\rfloor + d\left\lfloor\frac{T_B}{T_S}\right\rfloor \;=\; d\left(1 + \left\lfloor\frac{T_A}{T_S}\right\rfloor + \left\lfloor\frac{T_B}{T_S}\right\rfloor\right),$$

where the first term comes from executing the epilogue (A) of the previous invocations concurrently with the current d invocations of S; the second term accounts for those invocations; and the third term comes from executing the prologue (B) of the next invocations. If $T_A \gg T_S$ or $T_B \gg T_S$, the amount of concurrency is not seriously limited by the locking. However, if d is small or $T_A \sim T_S$ and $T_B \sim T_S$, the tasks do not have much concurrency. In the limit (as in `f`), $d = 1$, $T_A = 0$, $T_B = 0$, and the function should execute sequentially. In the function `g`, B and A are empty, so $T_B = T_A = 0$. However, since $d = 2$, two tasks can execute simultaneously.

If we ignore data dependences, which necessitate locking and introduce unpredictable delays, we can compare the potential gains of spawning the head, the recursive call, or the tail. Let T_H be the cost of the function's head, T_T be the cost of its tail, n be the number of iterations, and s be the number of servers. If we spawn the recursive call, then the earliest completion time is (see Figure 2.2)

$$\max\left(nT_H, \left\lceil\frac{nT_T}{s}\right\rceil\right).$$

If we spawn the function's entire head (the best case), then the earliest completion time is (see Figure 2.5)

$$\left\lceil\frac{nT_H}{s}\right\rceil + nT_T.$$

Finally, if we spawn the function's entire tail, then the earliest completion time is (see Figure 2.6)

$$nT_H + \left\lceil\frac{nT_T}{s}\right\rceil.$$

Spawning the recursive call is always better than spawning the tail since the recursive calls partially overlap the concurrently executing code. Spawning the head may be better than spawning the call when $T_H > T_T$ so the expensive portion executes concurrently.

2.4 Inserting Locks

Synchronizing data dependences with locks requires a lock for each location and pair of statements that conflict over the location. In addition to inserting the *lock* and *unlock* statements described above, we must ensure that the lock is unlocked even if the protected statement never executes, which requires additional `unlock` statements on alternative paths

through the function. This approach has two flaws. First, it inserts too many locks in a program since many of them are redundant. Second, the locks are too fine-grained since each protects a single memory location. Let us first consider how to find the minimal set of locks. Section 2.4.3 shows how to lock statements instead of locations.

A *lock dependence graph* (LDG) for a recursive function is a graph formed as follows:

1. Create $d + 1$ copies of the control-flow graph for recursive function `f`, where d is the longest distance of a conflict between the statements of `f`. The arcs in the graphs are *flow arcs*. Label the graphs $G_1, \ldots, G_{d+1}$ and label statement S_i in graph G_j as $S_{i,j}$.

2. Add a flow arc from the recursive call in G_j to the entry point of G_{j+1}. Add another flow arc from the exit of G_{j+1} to the statement following the recursive call in G_j.

3. For each conflict for which there is a loop-carried dependence from S_i to S_m, add every possible *lock arc* from $S_{i,j}$ to $S_{m,j+d_c}$, where d_c is the distance of the conflict.

The entry of an LDG is the entry of G_1 and its exit is the exit of G_{d+1}. A path from $S_{i,j}$ to $S_{m,n}$ asserts that statement $S_{m,n}$ must execute after $S_{i,j}$, either because of serial execution constraints or synchronization.

A lock (represented by a lock arc from $S_{i,j}$ to $S_{m,n}$) is *redundant* if, excluding the arc itself, there is a path from $S_{i,j}$ to $S_{m,n}$. This path means that some other combination of sequential statements and locks ensures the synchronization. We can compute the minimal set of locks by finding the transitive reduction of the LDG.

The *transitive reduction* of a directed, acyclic graph G is the unique, smallest graph G' whose transitive closure is equal to that of the original graph, $G^T = G'^T$. In other words, G' is the graph with the fewest edges that has a path from node u to node v if and only if G has a path between the nodes. Aho, Garey, and Ullman show the transitive reduction of an acyclic graph is unique and is found by removing the graph's redundant arcs in any order [2].

An LDG must be acyclic since two statements in a dependence cycle have no legal execution order. In computing the transitive reduction of an LDG, we cannot remove control arcs, but this does not matter since removing only the lock arcs produces the graph with the fewest lock arcs.

Theorem 1 *The graph resulting from computing the transitive reduction of an LDG by only removing redundant lock arcs has the fewest lock arcs of any graph whose transitive closure is equal to that of the original graph.*

Proof Let G be the graph found by a transitive reduction algorithm that removes only lock arcs and H be another graph with the same transitive closure and fewer lock arcs. Let $e = \langle u, v \rangle$ be a lock arc in G but not in H. Since e is not in H, there must be a path in H from $u \rightarrow x \rightarrow v$. The paths $u \rightarrow x$ and $x \rightarrow v$ must be in G since $G^T = H^T$ so e is redundant in G. But G has no redundant arcs and e does not exist. ∎

After finding the minimal set of locks, we must insert *unlock* statements so that every lock is unlocked after the first conflicting statement either executes or is bypassed. Below

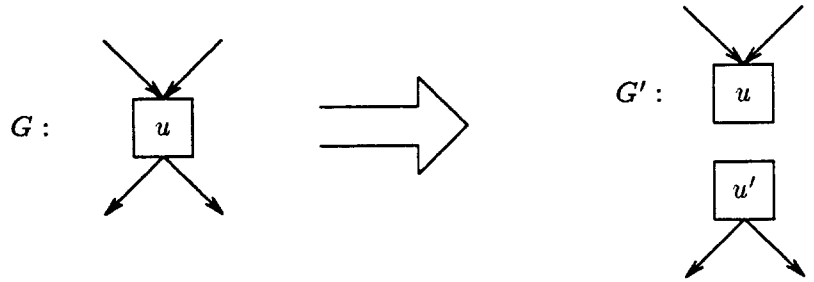


Figure 2.7: The first step in finding the minimum set of arcs to unlock is to split the graph G at the block u that contains the *unlock* statement.

are two algorithms for inserting the additional *unlock* statements, which are called *bypass unlocks*.

The first algorithm inserts the minimum *unlock* statements necessary to ensure that exactly one of these constructs executes on any path through the function. The second and more practical algorithm may insert more *unlock* statements but ensures that a lock is unlocked as early as possible, thereby increasing the possible concurrency.

2.4.1 Min-Cut Bypass Unlock Algorithm

Consider a function f that has a control-flow graph G with entry block s and exit block e. Block u contains the *unlock* statement following the first conflicting statement. To find the minimum bypass *unlock* statements, this algorithm finds the smallest set of arcs that must be removed to partition G into two pieces, one of which contains s and the other e. If we insert *unlock* statements along these arcs, any path from s to e will execute exactly one of these constructs.

The first step splits G at block u to account for the *unlock* statement in that block. It forms a new graph G' (see Figure 2.7). The next step finds the minimum cut-set necessary to partition block s from block e. This set is found by standard network-flow techniques under the assumption that every arc in G' has weight 1. Let the minimum cut-set be C.

The simplest way to insert an *unlock* statement along an arc in C is to split each arc into two arcs and put a new block containing the *unlock* between these arcs (see Figure 2.8). However, if the destination of the arc has only a single predecessor, the *unlock* statement can be the first statement in that block.

2.4.2 Earliest Bypass Unlock Algorithm

Another algorithm also inserts *unlock* statements along each path through a function and has the advantage of ensuring that locks are unlocked as early as possible. This means that a lock is unlocked as soon as control reaches a block that cannot lead to the original *unlock* statement.

Again, we start with an acyclic flow graph $G = (V, E)$ with start block s, final block e, and an *unlock* statement in block u. Partition V into two disjoint, mutually-exclusive sets

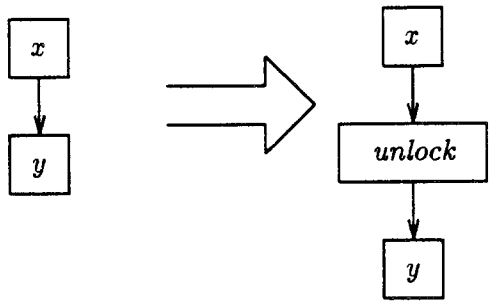


Figure 2.8: Each arc in the min-cut set C must have an *unlock* statement inserted along it. The easiest way to accomplish this is to split the arc and add a new block containing the statement.

B and A:

$$\begin{aligned} B &= \{v \in V \mid G \text{ has an acyclic path } s \rightarrow v \rightarrow u\} \\ A &= V - B. \end{aligned}$$

The notation $x \rightarrow y$ indicates there exists a (possibly empty) path from node x to node y. B is the set of blocks executed no later than block u. A is the set of blocks executed after block u.

In this algorithm, the cut set contains the edges that have an endpoint in both sets, $C = \{\langle x, y\rangle \mid x \in B, y \in A, \text{ and } x \neq u\}$. The *unlock* statements are inserted along these edges in the same manner as above.

Theorem 2 *Inserting locks along arcs in C ensures that there is exactly one* unlock *statement along each acyclic path from a node $n \in B$ (in particular s) to e.*

Proof When $|V| = 1$, the graph contains a single block. This block must contain an *unlock* statement, so any path through the flowgraph (i.e., the block) encounters this statement.

Assume the theorem for all graphs with $|V| < k$. Consider a graph G with k blocks. Consider each successor z of node $n \in B$. If $z \in A$, then either $n = u$ or the algorithm would insert an *unlock* statement along the arc $\langle n, z\rangle$. Therefore, there is exactly one *unlock* along all paths from s to z. Since $z \in A$, it cannot have a successor in B. The algorithm would not insert any more *unlock* statements along paths beginning with $\langle n, z\rangle$.

If $z = u$, no lock statement is inserted along the arc $\langle n, z\rangle = \langle n, u\rangle$, since both nodes are in B, or along any arc leaving u, by the definition of C. However, u contains the original *unlock* statement so paths from z to e will release the lock.

Otherwise, $z \neq u$ and $z \in B$, so there is no *unlock* statement along the path $s \rightarrow z$. Consider the subgraph headed by z. Since u is unchanged, $B_z = B \cap V_z$, $A_z = A \cap V_z$, and $C_z = C \cap V_z$, where V_z contains the nodes in the graph headed by z and B_z, A_z, and C_z are the respective sets for this graph. Since G is acyclic, the subgraph contains fewer than

k blocks and the induction assumption applies. The paths from z to e contain exactly one unlock statement. ∎

This algorithm is more desirable than the minimum-cut algorithm because the additional *unlock* statements that it may introduce only increase the static size of a program by a small amount. Both algorithms cause the same number of *unlock* statements to execute in each invocation, but this algorithm may increase the potential concurrency by releasing locks earlier.

2.4.3 What Are We Locking?

In the description above, locks serialize accesses to a location in memory. On most computers, it is difficult to lock a single memory location. An alternative is to add locks to the data. However, sets of locks are difficult to preallocate for Lisp programs since the size of data structures, such as trees and linked lists, cannot be precomputed.

By slightly shifting our perspective, we can find a more manageable approach to this problem. A lock serializes the execution of two statements d iterations apart. The location plays no part except to determine the constant d. Therefore, we divorce a lock from a location and pass the lock between the statements.

Let the statements $S_{i,j}$ and $S_{m,n}$ have a non-redundant conflict over location l and let $d = n - j$. To serialize these statements, add the following code to the function containing them:

1. Add formal parameters $l_1, \ldots, l_d$ and new local variable l_0. Non-recursive calls on the restructured function must pass correctly initialized locks as the initial values of the new parameters.

2. Before the recursive call, add the assignment

   $$l_0 \leftarrow \textit{make-lock('locked)}$$

3. In the recursive call, pass l_0 as the new value of l_1, l_1 as the value of l_2, and so on.

4. Before $S_{m,n}$ add the statement

   $$\textit{acquire-lock}(l_0)$$

5. After $S_{i,j}$ add the statement

   $$\textit{release-lock}(l_d)$$

6. Add the statements

   $$\textit{release-lock}(l_1);$$
   $$\ldots$$
   $$\textit{release-lock}(l_d);$$

   to the termination test of the recursive function.

For example, the correctly locked version of the function from Section 2.3.1 is:

```
(defun f2 (task-queue l1 l2 lst)
  (cond ((null? (cddr lst))
         (release-lock l1)
         (release-lock l2)
         (terminate-servers task-queue))
        (t
          (let ((l0 (make-lock 'locked)))
            (make-task task-queue
                       (lambda ()
                         (acquire-lock l0)
                         (set! (car (cddr lst)) (+ (car lst) 1))
                         (release-lock l2))
                       ())
            (f2 task-queue l0 l1 (cdr lst))))))
```

2.5 More Complex Functions

The linear recursive functions discussed above execute at most one recursive call in each invocation and correspond to a simple loop. However, functions that execute more than one recursive call per invocation can also execute concurrently. Their primary problem is that concurrent execution does not preserve any order among the statements so all conflicts must be serialized.

Non-linearly recursive functions may execute multiple recursive calls in a single invocation and consequently do not have a well-defined ordering among their statements. Although the series of invocations from a call site are ordered, each invocation spawns other, unsynchronized series of recursive calls. For example, consider the function:

```
(defun traverse (tree)
  (cond ((leaf? tree)
         (set! (value tree) (+ 1 (value tree))))
        (t
          (traverse (left tree))
          (traverse (right tree)))))
```

The recursive calls from traversing the left children can be treated like conventional recursion. However, each invocation also begins a series of calls down the chain of right children. These calls, in turn, traverse the left and right children of subtrees. The heads and tails of `traverse` execute without any apparent order.

`traverse` also illustrates the difference between a pair of nested loops and a pair of consecutive recursive calls. Nested loops alternate: the inner loop runs to completion

before each iteration of the outer loop. Consecutive recursive calls are interleaved in a data-dependent manner.

Ensembles of mutually-recursive functions share this problem as well. For example, if function f invokes function g, which calls f, then the call on g is, in effect, a recursive call on f. Recursion introduced through another function presents no new problems and may improve the program's concurrent execution by increasing the amount of work done by each task.

In functions without data dependences between iterations, the absence of ordering is not a serious problem since no conflicts must be serialized. Each recursive call can execute concurrently. For example, if `traverse` is applied to a tree (not a DAG), the function may be rewritten:

```
(defun traverse1 (task-queue tree)
  (cond ((leaf? tree)
         (set! (value tree) (+ 1 (value tree))))
        (t
          (make-task task-queue traverse1 task-queue (left tree))
          (make-task task-queue traverse1 task-queue (right tree)))))
```

Another approach to restructuring these functions is to transform only one recursive call. This increases the amount of work each task performs (by a data-dependent amount), but does not restore order among statements.

Non-linearly recursive functions with data dependences are much more difficult to execute concurrently within the server model. Synchronization and the lack of order between statements require that the servers be preemptable so that a server can turn its attention to other tasks when a task waits for a lock. Preemptable servers are general processes. In concurrent Lisp systems that provide inexpensive processes, these servers are easily implemented by spawning a process to execute each task. In systems with expensive processes, this approach is impractical and these functions must execute sequentially.

2.6 Control Dependences

Control dependences impede parallelism more than data dependences because they offer less flexibility in executing statements concurrently. If statement B's execution is contingent on statement A's result, the two statements can execute concurrently only if: B has no side effects, its side effects do not matter, or the side effects delay until A terminates. We will examine the third option to speed evaluation of the most common control dependences in recursive functions.

These functions frequently have the form:

```
(defun f (...)
  (cond (test1 body1)
```

```
        .
        .
        .
      (testn bodyn)))
```

where the tests may be elaborate expressions involving most of the work in the function. Even if the tests do not have data dependences, $test_i$ is control dependent on $test_1$ through $test_{i-1}$. In addition, $body_i$ is control dependent on $test_1$ through $test_i$. For the rest of the section, assume that there are no conflicts between the tests, which is reasonable since the predicates are frequently side-effect-free.

Below is a simple scheme for evaluating the tests concurrently. The goal is to find the first test in the sequence that succeeds. As soon as a predicate succeeds and all earlier predicates have failed, the corresponding body can execute.

Consider a cond statement:

```
(cond (test0 body0)
        .
        .
        .
      (testn-1 bodyn-1)
      (t bodyn))
```

in which the last test always succeeds. If the last test is not constant, add a new final clause: `(t nil)`. Clause i is *earlier* than clause j if $i < j$.

When a predicate terminates with a true result, we want to know if all earlier predicates have failed. Let the result of predicate p_i be i if the predicate is true and n if false. Therefore, when p_i succeeds, we want to know if its result is minimum among $\{p_0, \ldots, p_i\}$.

The restructured conditional statement becomes:

```
(let ((heap (make-vector (* 2 n))))
  (defun body (i)
    (case i
      ((nil))
      (0 body0)
        .
        .
        .
      (n bodyn)))

  (make-task cond-queue
             (lambda ()
               (body (find-first heap
```

```
                                    0
                                    (if (test0) 0 n)
                                    n))))
      .
      .
      .
(make-task cond-queue
           (lambda ()
              (body (find-first heap
                                    n-1
                                    (if (testn-1) n-1 n)
                                    n)))))
```

Concurrent tasks evaluate the n non-trivial tests by executing their predicates and passing the result and predicate's index to `find-first`, which returns either the index of the earliest true clause or `nil`. If this value is non-nil, the server executes the appropriate clause body. At this point, the server should terminate the other processes.

This scheme is practical for conditional statements in which several predicates execute non-trivial, but side-effect-free tasks. Program analysis of type described later can identify these predicates.

2.7 Allotting Servers

Earlier in this chapter, we deferred decisions about allotting servers to loops by postulating a function **number-of-servers** that accepts a loop's desired number of servers and returns an allotment less than or equal to this quantity. This section discusses the behavior of this function.

The simplest case is a concurrent recursive function that is not nested in another concurrent function. At execution time, this function should be given all available servers, up to the number that it requests. These servers would otherwise sit idle until the loop terminates. On the other hand, **number-of-servers** should not allocate more servers than requested since servers assigned to a loop, but not used, incur a slight but unnecessary overhead.

The requested number of servers should not be larger than the number of tasks for the same reason. The most general approach is to count the number of tasks (e.g., the number of items in a list) before calling **obtain-servers**. Other ways of determining this quantity are to rely on programmer-supplied declarations or to empirically measure the value by profiling the program.

The situation is more complex when recursive functions nest within recursive functions, either because of simple nested loops or non-linearly recursive functions. These two cases are different and will be discussed separately. In properly-nested recursive loops, the outer loop should be favored in allotting servers since its body takes longer to execute and better

amortizes the overhead of parallelism. This bias occurs naturally with the scheme describe previously. The outer loop first obtains its allotment of servers. Any remaining ones may be used by inner loops that start subsequently.

On the other hand, non-linearly recursive functions do not have clear inner and outer loops. For these loops, the best policy is to be sparing in requesting servers so every loop fully utilizes all processors that it obtains. The other general rule—to assign servers to large tasks—is difficult to apply since it depends on the algorithm and data. For example, a depth-first search may quickly reach the small tasks at the leaves of a tree and assign them to servers. In the same problem, breadth-first search would have assigned servers to the larger subtrees higher in the original tree.

If a loop does not receive any servers because they are busy with other tasks, the function could fall back on the original, non-parallel code and avoid the overhead of enqueuing tasks for a single processor. This scheme requires only a few small changes to the framework described above.

2.8 Related Work

The server model is similar to the Uniform System for BBN's Butterfly Multiprocessor [72], although both are used in different manners. The Uniform System is a library of routines that hide the structure of a multiprocessor by concurrently executing a set of tasks produced by a user-supplied generator. A programmer calls system routines to initialize the generator and writes the task bodies. None of the code is automatically generated.

An early version of the techniques in this chapter was presented in a paper on Curare [53]. This paper discussed concurrent execution in which dependences were preserved but did not describe the servers.

Midkiff developed an algorithm for removing redundant locks similar to the one in Section 2.4 [56,57]. His algorithm, however, was limited to straight-line code without conditional statements. Although his scheme was similar to transitive-reduction, he incorrectly claimed that the problem was NP-hard (instead of being equivalent to computing the transitive closure of a graph). In attempting to extend his algorithm for conditional statements, he did not realize that the locking relation is preserved by the bypass `unlocks`. Thus, he had to build an LDG for every possible path through the function body.

As far as we can judge, the voluntary muscular system of the entire body is completely paralyzed—the utterly limp, flaccid, dishraggy kind of paralyzed—the opposite of the rigid, convulsive spastic kind.
– Richard G. Gill, White Water and Black Magic

Chapter 3

Optimization of Concurrent Programs

This chapter briefly describes some optimizations that are in CURARE or that could easily be added to it to improve the performance of concurrently executed recursive functions. The goal of these optimizations is to remove data dependences, for, as we saw previously, dependences require additional synchronization and prevent concurrent execution. The optimizations remove dependences by changing a program to perform its actions in a different manner or order. Obviously, dramatic changes, such as rewriting a program in an applicative style, are beyond the ability of a general transformation system.

The optimizations are transformations that take a Lisp function and produce an equivalent function. The two functions are equivalent because they produce the same result, not because they compute the result in the same manner. In fact, a goal of the transformations is that the two functions are not conflict equivalent. The transformations preserve final-state or view equivalence so the two statements compute the same result.

There are many possible transformations that could eliminate dependences. The transformations that improve a particular program depend on its structure and dependences. This chapter describes five generally useful transformations: anti-dependence elimination, destination-passing, associative composition, recursion-removal, and loop-splitting. The final section surveys related work.

3.1 Anti-Dependence Elimination

Some anti-dependences can be eliminated by copying values. Assume statement S_W modifies a location, l, read by a preceding statement S_R. These statements do not have a dependence if the value from l is saved in another place (by statement $S_{R'}$) before either statement executes (see Figure 3.1). S_R is modified to read from this new location. The head of the function is the natural place to save this value since it executes before either statement. For example, the function:

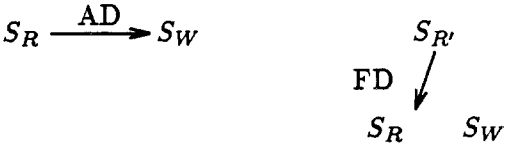


Figure 3.1: The anti-dependence between S_R and S_W can be eliminated by saving the conflicting location's value in $S_{R'}$.

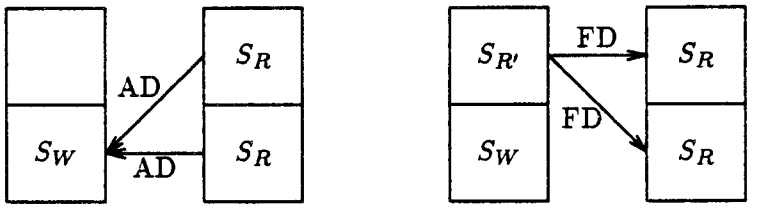


Figure 3.2: Anti-dependence removal transforms a loop-carried anti-dependences (on the left) into a flow dependences (on the right). The new dependences are preserved when the recursive call is spawned.

```
(defun f (lst)
   (cond ((null? (cdr lst)))
         (t
           (f (cdr lst))
           (set! (cadr lst) (+ (car lst) 1)))))
```

contains an anti-dependence because the assignment modifies the location `(car lst)` reads in the next invocation. This dependence is easily removed:

```
(defun f1 (lst pv)
   (cond ((null? (cdr lst)))
         (t
           (f1 (cdr lst) (cadr lst))
           (set! (cadr lst) (+ pv 1)))))
```

This transformation eliminates anti-dependences with the following characteristics (see Figure 3.2):

1. S_R has a loop-carried anti-dependence of distance d with statement S_W, which is in the tail of the function.
2. The conflicting location is referenced by the access path $v_1.\alpha_1$ in S_R and $v_2.\alpha_2$ in S_W, where v_i is a variable and α_i is a sequence of structure field names.
3. The reference $v_2.\alpha_2$ cannot cause an error if executed at the recursive call.
4. No other statements conflict with S_R or S_W over the location (this condition can be relaxed to no flow or output dependence over the location).

The transformation is:

1. Add d additional arguments to the function: $pv_1, \ldots, pv_d$.
2. Pass the appropriate values for these new arguments at all non-recursive calls on the function.
3. At the recursive call, pass $v_2.\alpha_2$ as pv_1, pv_1 as pv_2, etc.
4. Change all references to $v_1.\alpha_1$ in S_R to use pv_d.

This transformation (which is not part of CURARE) is clearly beneficial for anti-dependences over a small number of iterations since they normally greatly reduce the concurrency and because the transformed function gains only a few new arguments. However, when d is large, the cost of passing many arguments, particularly through a concurrent call, may outweigh the benefits of increased parallelism.

3.2 Destination-Passing

A common and crippling data dependence occurs when a statement in the tail of a recursive function uses the result of the recursive call. This loop-carried flow dependence constrains the statement to execute after the subsequent iterations and prevents server-based execution since the call returns a value. For example, consider the function:

```
(defun mapcar (f lst)
  (cond ((null? lst) ())
        (t
          (cons (f (car lst))
                (mapcar f (cdr lst))))))
```

which returns a list of results from applying `f` to the elements of a list. The recursive call returns a result to the call on `cons`, so the function cannot be restructured, even if it contains no other data dependences.

This section describes CURARE's transformation that removes this type of flow dependence. The transformation, destination-passing, is limited to functions in which a recursive call's result is passed to a structure-allocation function (such as `cons`).

A *destination-passing* function receives, as one of its arguments, the location in which its result is to be stored. The function can produce and store its result and does not need to return a value. It is easy to change recursive function calls embedded in calls to allocation functions into this form. The destination-passing version of `mapcar` is:

```
(defun mapcar-dp (dest f lst)
  (cond ((null? lst)
         (set! (cdr dest) ()))
        (t
          (let ((tmp (cons nil nil)))
            (set! (car tmp) (f (car lst)))
            (mapcar-dp tmp f (cdr lst))
            (set! (cdr dest) tmp)))))
```

The new argument, **dest**, contains a **cons** cell whose **cdr** will receive the result from the function's invocation. All destructive operations in **mapcar** and **mapcar-dp** occur in the same order, so the functions are conflict equivalent.[1]

The recursive call no longer returns a result so that **mapcar-dp** can execute concurrently:

```
(defun mapcar-dp1 (task-queue dest f lst)
  (cond ((null? lst)
         (set! (cdr dest) ()))
        (t
          (let ((tmp (cons nil nil)))
            (make-task task-queue
                       mapcar-dp1 task-queue tmp f (cdr lst))
            (set! (car tmp) (f (car lst)))
            (set! (cdr dest) tmp)))))
```

The recursive call and the preceding assignment in **mapcar-dp** can be exchanged because subsequent iterations of the function cannot modify the **car** of the **cons** cell (which is not even allocated when those iterations execute in **mapcar**). The absence of data dependences over the cons cell is clear in the original function and is preserved by the conflict-equivalent transformation.

More generally, let $alloc(v_1, \ldots, v_n)$ be an allocation function that produces an object with n fields, $f_1, \ldots, f_n$, that are initialized to the values $v_1, \ldots, v_n$, respectively. f is a recursive function with formal parameters $a_1, \ldots, a_m$ in which each recursive call is either tail recursive or is the i^{th} argument to a call on *alloc*. CURARE forms the destination-passing version of f, *f-dp*, as follows:

1. Add a new, first formal parameter, *dest*, to f.
2. Replace calls on *alloc* in which the recursive call is the i^{th} argument

$$tmp \leftarrow alloc(v_1, \ldots, f(e_1, \ldots, e_m), \ldots, v_n)$$

[1] The destructive operations in **mapcar** are hidden by the definition of **cons**, which must destructively assign the fields of the new **cons** call.

by

$tmp \leftarrow alloc(nil, \ldots, nil)$;
$tmp.f_1 \leftarrow v_1$;
...
f-dp$(tmp, e_1, \ldots, e_m)$;
...
$tmp.f_n \leftarrow v_n$;

3. Replace each tail-recursive call, $f(e_1, \ldots, e_m)$ by: *f-dp*$(dest, e_1, \ldots, e_m)$.

4. Replace each return statement, **return** v, by an assignment: $dest.f_i \leftarrow v$.

5. Create a wrapper function for f with the following body

function $f(a_1, \ldots, a_m)$
  $tmp \leftarrow alloc(nil, \ldots, nil)$;
  f-dp$(tmp, a_1, \ldots, a_m)$;
  return $tmp.f_i$;

The explicit assignments introduced into *f-dp* do not conflict with each other. However, they may conflict with other uses of fields of the newly created object.

Generalizing this technique to functions other than allocation functions is difficult because it relies on the non-strictness of allocation functions, which provide a place for a value without examining the value itself. Strict functions, which examine their arguments, impose more constraints since the function cannot execute until its arguments are available.

3.3 Associative and Commutative Composition Functions

The reduction of a set of values by an associative, commutative, and side-effect-free function does not depend on the order in which the operations are applied. The composition $x_0 \circ x_1 \circ \ldots \circ x_n$ forms a linear recurrence in which the partial sum is given by $s_i = s_{i-1} \circ x_i$. Asymptotically efficient techniques for evaluating these recurrences in parallel are well-known. These techniques, however, use random-access, fixed-size data structures to store intermediate results without synchronization and so are ill-suited to Lisp programs and data. Fortunately, CURARE adapts the approach to Lisp programs.

Consider a function that returns the sum of the elements of a list:

```
(defun add-all (lst)
  (cond ((null? lst) 0)
        (t
          (+ (car lst) (add-all (cdr lst))))))
```

Because of a loop-carried flow-dependence, no addition occurs until subsequent terms are summed. However, the order of addition does not affect the final result.[2] Therefore, if we had locations to store intermediate results, the additions could execute concurrently. The area for temporary results must accommodate a varying number of values and must be concurrently accessible, so we use a queue (mailbox). The rewritten function is:

```
(defun add-all (lst)
  (let ((num-queue (make-mailbox)))
    (send 0 num-queue)
    (add-all1 num-queue lst)
    (receive num-queue)))

(defun add-all1 (num-queue lst)
  (cond ((null? lst))
        (t
          (send (car lst) num-queue)
          (send (+ (receive num-queue) (receive num-queue))
                num-queue)
          (add-all1 num-queue (cdr lst)))))
```

If `add-all1` executes sequentially, it adds the list's elements in reverse order, but the result is unchanged. More importantly, to execute this function concurrently, we enlarge its tail and spawn the recursive call:

```
(defun add-all (lst)
  (let ((num-queue (make-mailbox))
        (task-queue (obtain-servers)))
    (send 0 num-queue)
    (add-all2 task-queue num-queue lst)
    (server task-queue)
    (wait-counter (mb-barrier task-queue))
    (receive num-queue)))
```

[2]Except for non-associative floating-point addition.

```
(defun add-all2 (task-queue num-queue lst)
  (cond ((null? lst)
         (terminate-servers task-queue))
        (t
          (make-task task-queue
                     add-all2
                     task-queue num-queue (cdr lst))
          (send (car lst) num-queue)
          (send (+ (receive num-queue) (receive num-queue))
                num-queue))))
```

This function traverses the list, creating processes that enqueue an item from the list, add two items from the queue, and enqueue their sum. To achieve concurrency, we need the two statements in the tail. Combining them into a single statement

```
(send (+ (car lst) (receive num-queue)) num-queue)
```

prevents more than a single process from executing simultaneously. If the list contains n items, $n/2$ processes can execute the first round of addition concurrently. These processes do not begin executing simultaneously but are offset by the time spent in the function's head. The entire reduction requires $\log n$ rounds of addition.

The composition operator, in this case +, must be commutative because values are enqueued in an arbitrary order. Standard techniques for solving recurrences avoid this requirement by providing a fixed set of locations for intermediate results so operands are not commuted. To allocate this set for data structures linked by pointers, a program must traverse the structure to find the number of elements and initialize the temporary area. This approach may be useful for expensive operations that are associative but not commutative (such as append), but will not be described here.

Contention for the queue is limited if the number of concurrent processes (servers) is low, the cost of the composition operation is high, and queue operations are inexpensive. If queue contention is a problem, the queue can be split into multiple queues at the expense of some complexity.

The general form of this transformation modifies a function f in which all recursive calls are either tail-recursive or are an argument to an associative, commutative function, g. It produces a new function $f1$ and wrapper function f.

1. Rename f to $f1$ and add a new, first argument, q.

2. Replace each call on f that is an argument to the function g

   $$g(f(a_1, \ldots, a_m), v)$$

   by

$f1(q, a_1, \ldots, a_m)$;
enqueue(v, q);
enqueue$(g($*dequeue*$(q),$ *dequeue*$(q)), q)$;

3. Replace each tail-recursive call

$$f(a_1, \ldots, a_m)$$

by

$$f1(q, a_1, \ldots, a_m).$$

4. Replace each return statement, **return** v, by

enqueue(v, q).

5. Create a new wrapper function

function $f(a_1, \ldots, a_m)$
  $q \leftarrow$ *make-queue*();
  enqueue(0, q);
  $f1(q, a_1, \ldots, a_m)$;
  return *dequeue*(q);

where 0 is the identity element for g.

The transformed function contains no data dependences among the *enqueue* and *dequeue* operations and may be rewritten as a concurrent function.

Determining that function g is associative and commutative is impossible in general. A programmer must supply this information through declarations such as those in Chapter 6. Without this knowledge, CURARE cannot apply the transformation since the result would be undefined if g does not have those properties.

3.4 Recursion Removal

Transformations (other than destination-passing) that produce tail-recursive programs from properly recursive ones are well-known [13,19,40]. These transformations can aid in restructuring programs. For example, the iterative version of the function:

```
(defun reverse (lst)
  (cond ((null? lst) ())
        (t
          (append (reverse (cdr lst) (list (car lst)))))))
```

is:

```
(defun reverse1 (lst r)
  (cond ((null? lst) r)
        (t
          (reverse1 (cdr lst) (cons (car lst) r)))))
```

and can be developed by applying these transformations. The iterative version of a program may be better suited to concurrent execution because the flow dependence between iterations is removed (`reverse1` is still not suitable).

The difficulty with these transformations is that they require specific information about the properties, such as associativity, of the operators that they manipulate (see the appendix of Huet and Long's paper [40] for one of the few systematic presentations of the preconditions on transformations). This information must be provided by the programmer who wrote a function; it cannot be inferred in general. Hence, a set of declarations—similar to CURARE's—is necessary.

3.5 Loop-Splitting

Of the transformations developed for restructuring FORTRAN programs, loop-splitting is the most useful for Lisp programs. This transformation breaks a single loop containing a data dependence into two loops. The two loops execute concurrently, one after the other, thereby assuring that the dependence is respected. Usually, partial results from the first loop are collected in a new data structure that the second loop uses. CURARE does not yet implement this optimization.

For example, consider the function:

```
(defun f (lst)
  (cond ((null? (cdr lst)))
        (t
          (set! (car lst) (+ (car lst) (cadr lst)))
          (f (cdr lst)))))
```

that replaces each element in a list by the sum of it and the next element. It contains a loop-carried anti-dependence between the reference `(cadr lst)` and the subsequent assignment to `(car lst)`. The loop cannot execute concurrently since exchanging the call and assignment reverses the dependence. However, this function produces the same result and can execute concurrently:

```
(defun f (lst)
  (defun f1 (lst)
    (cond ((null? (cdr lst)))
          (t
            (cons (+ (car lst) (cadr lst))
```

```
                (f1 (cdr lst))))))

(defun f2 (sums lst)
  (cond ((null? (cdr lst)))
        (t
          (set! (car lst) (car sums))
          (f2 (cdr sums) (cdr lst)))))

(f2 (f1 lst) lst))
```

Function `f1` produces the new values. It can execute concurrently after being modified to use destination-passing. Function `f2` replaces the list elements by their new values.

This transformation breaks anti- and def-order dependences in which values do not flow between iterations. Flow dependences still require synchronization between the production and use of a value. In general, let f be a recursive function with a single anti- or def-order dependence from statement S_1 to statement S_2 (S_2 assigns the location and S_1 may use or assign the location, depending on the type of dependence). The transformation produces two new functions f_1 and f_2 and a new definition for f.

1. f_1 creates a new list containing results from the invocations of S_1.

   function f1 (a_1, ..., a_n)
     if done **then return** *nil*;
     else return *cons*(SL_1, f1(...));

   The code in SL_1 is the slice of f with respect to statement S_1 [38]. This slice is the code in f that may produce a value necessary for S_1. The results from S_1 are collected in a list, which is returned.

2. f_2 takes this list and the original arguments and computes the function, using the previously computed values of S_1 in place of that statement. Hence f_2 is the original function f, with a new argument *values*, and occurrences of S_1 replaced by *values.car*.

3. f becomes the expression $f2(f1(a_1, \ldots, a_m), a_1, \ldots, a_m)$. The resulting function no longer has a dependence between S_1 and S_2 and f_1 can be transformed to destination-passing. f_2 may execute concurrently, depending on the remaining dependences. The new argument *values* is distinct from the existing arguments so no statement conflicts with the references *values.car*.

3.6 Related Work

Transforming programs to facilitate parallel execution has long been used to prepare programs for parallel machines. The specific transformations described in this chapter are

not original, although few have been previously applied to restructuring Lisp programs for concurrent execution.

The anti-dependence and loop-splitting transformations are particular examples of Cytron's and Ferrante's more general technique [18]. Their algorithm can eliminate all anti- and def-order dependences in a program by "renaming" the locations referenced by statements, but it may greatly increase the amount of storage consumed by the program.

Wadler previously and independently described destination-passing in his dissertation on eliminating intermediate lists from applicative programs [76]. He called the transformation *tail recursion modulo cons* and used it as a technique for producing tail-recursive functions. He, however, did not consider its potential for producing concurrent functions.

Fast algorithms for solving recurrence problems on parallel computers have long been known [45,46]. The essential idea of these algorithms is to use associativity to rearrange the expression $x_0 \circ x_1 \circ \ldots \circ x_n$ from a long, stringy tree into a bushy one and then to operate on parallel branches. These algorithms heavily rely on arranging the data in an array so that the operands can be efficiently partitioned among the processors. Kruskal, Rudolph, and Smir described an equally efficient solution to the data-dependent version of the problem in which the data is in an array, but the next item in the sequence is designated by a pointer and is not the next array element [48]. Their solution will not work for Lisp programs without a pretraversal of the data to collect it into an array, in which case the slightly faster, conventional algorithm can be used.

Harrison's PARCEL parallel Lisp system depends on the parallel solution of recurrences [32]. To use the technique, he proposed a new data structure that contiguously allocates elements of a list. However, this structure only works for linear lists, not general structure objects, and does not permit destructive operations.

The technique of loop-splitting was first described by Kuck *et al.* [50] and has been used in translators since at least PARALYZER for the ILLIAC IV [62]. Its use in Section 3.5 is similar to its use in FORTRAN programs, except for the dynamic construction of the intermediate result.

If two contrary actions be excited in the same subject,
a change must necessarily take place in both,
or in one alone, until they cease to be contrary.
– Benedict de Spinoza, Ethics, *Part Five (Axiom 1)*

Chapter 4

Detecting and Classifying Data Dependences

A data dependence constrains the execution of two statements because both expect to use a common storage location that can accommodate only one value at a time. Statements S_1 and S_2 either read from or write to a location l, which leads to the three data dependences named by Kuck [49]:

1. *Flow Dependence.* S_1 writes a value into location l that is subsequently read by S_2.
2. *Anti-Dependence.* S_1 reads a value from location l and S_2 subsequently modifies l.
3. *Def-Order Dependence* (*Output Dependence*). S_1 and S_2 write values to location l.

Two statements *conflict* if they share a data-dependence. Changing the execution order of conflicting statements may affect a program's result.

Most programming languages specify a single execution order for most of their constructs, thereby permitting programmers to control the order of side effects within their programs. In some circumstances, however, it is necessary to know a program's dependences. For instance, optimizing compilers commonly take the liberty of rearranging non-conflicting statements. Many traditional compiler tools, such as reaching definition data flow and use-definition analysis, detect or represent data dependences.

Restructuring programs requires more precise data-dependence analysis than does simply compiling programs. To take advantage of concurrency, large portions of a program must be carefully analyzed because their execution order will be indeterminate. Compilers traditionally regard dependences in aggregate objects, such as arrays and data structures, to be too difficult to analyze. They treat a collection of locations as a single location. This shortcut is not feasible for restructuring since the simple analysis finds many false dependences that prevent concurrent execution.

A wide range of techniques has been proposed to overcome the two fundamental problems that limit dependence analysis: aliases and imprecise references. *Aliases* occur when

there are different ways of referencing an object. Explicit pointers in data structures and language features, such as reference parameters or FORTRAN COMMON blocks, cause aliases. An aggregate reference is *imprecise* if the exact location accessed is unknown. Non-constant array subscripts and pointer variable references cause imprecise references since they permit a statement to access many different locations.

This chapter discusses the data-dependence problem in general and presents a framework for determining the dependences between two statements. The next chapter uses this framework to solve the structure-access dependence problem. The first section of this chapter defines the terminology for this problem. Section 4.2 describes an abstract formulation of the data-dependence problem that encompasses variable, array, and structure references. Section 4.3 describes complications that arise in real programs. The final section surveys related work.

4.1 Definitions

A *location* is an unspecified place in which a program can store and retrieve values. Different languages provide different ways of aggregating and accessing locations, but the most common are variables, arrays, and data structures. Assume that the locations accessed through these mechanisms are disjoint. So, for example, we cannot treat a structure as a vector or access variables through pointers.

A statement accesses a subset of L, the set of all locations. Each location may be *read*, in which case its value is examined, but not modified; or it may be *written*, in which case the value in the location is replaced. A statement may access locations from more than one partition of L. For example, `a[1, 2]` reads the value of variable `a` and a value from the array contained there.

The variables accessed by a statement are usually easy to determine, though some language constructs, such as dynamic binding or the use of functions as first-class values, complicate the analysis. Determining the locations referenced by an array access is more difficult, but the problem has been the subject of considerable work since Banerjee's original paper [7]. Determining the locations referenced by structure accesses is an even more difficult problem, as we will see in the next chapter. For now, assume that we know the locations accessed by a statement.

The three data dependences were informally defined above. More precise definitions (after Horwitz, Prins, and Reps [37]) are below. To simplify the definitions, assume that a statement does not read and write the same location. Split statements that violate this prohibition (e.g., $x \leftarrow x + 1$) into two statements ($t \leftarrow x + 1$; $x \leftarrow t$). Statement S_2 is *flow-dependent* on statement S_1 over a location l if:

1. S_1 writes to l and S_2 reads from the same location.
2. There exists a path in the control-flow graph from S_1 to S_2 along which no statement writes to location l. In the traditional compiler terminology, there is an *l-definition-free* path from S_1 to S_2.

Statement S_2 is *anti-dependent* on statement S_1 over location l if:

1. S_1 reads from l and S_2 writes to the same location.
2. There exists a path in the control-flow graph from S_1 to S_2 along which no statement writes to l.

Finally, statement S_2 is *def-order dependent* on S_1 over location l if:

1. Statements S_1 and S_2 both write to location l.
2. There is a third statement S_3 that is flow dependent on S_1 and S_2 over location l.
3. S_1 occurs first in a canonical sequential execution of the program.

Dependences can be further classified as loop-independent or loop-carried, depending on whether the conflicting statements execute in the same or different iterations of a loop. A flow or anti-dependence is *loop-carried* if, in addition to the conditions above:

3. S_1 and S_2 are both contained in a loop.
4. There is an l-definition-free path from S_1 to S_2 that includes a back edge of the loop.

A loop-carried conflict's *distance* is the number of loop iterations between the execution of S_1 and S_2.

A flow or anti-dependence is *loop-independent* if, in addition to the conditions above:

3. There is a definition-free path from S_1 to S_2 that includes no loop back edges in the control-flow graph.

A def-order dependence is loop-independent if both flow dependences are loop-independent. It is loop-carried if either flow dependence is loop-carried. If a def-order dependence is loop-independent, the definition above can be strengthened by the additional constraint:

4. The execution of S_1 and S_2 is not mutually exclusive. Precisely determining the property is impossible in general. However, a simple approximation is that both statements are in the same branch of every conditional statement that encloses both of them.

4.2 Computing Dependences

A data-flow calculation, similar to reaching-definition flow analysis, will detect flow and anti-dependences. Classifying a dependence as loop-independent or loop-carried requires a slightly more elaborate computation but does not fundamentally change the problem. The flow dependences determine the def-order dependences. These calculations are known (e.g., [4,39]), but are worth restating in a problem-independent manner.

4.2.1 Computing Flow Dependences

In computing dependences, we group together locations that are accessed in the same manner, for instance those containing different instances of a dynamic variable. Let L be a (finite) set of names of locations accessed by the statements in a program. For example, L may be the set of variable names or, as we shall see in the next chapter, labels for data structure fields. In this chapter, I will use the term "location" interchangeably with "name of location" when describing the computation. Let S be the set of statements in the program and $G = \langle S, E \rangle$ be the control-flow graph of the program. The set of *definitions*, $D = S \times L$, contains pairs of statements and the locations they modify.

For each statement $s \in S$, let R_s be the set of locations read by s and W_s be the locations written by s. Two other sets describe a statement's effects on the definitions. $\mathrm{GEN}_s = \{\langle s, l \rangle \in D \mid l \in \mathrm{W}_s\}$ are the definitions created by executing s. $\mathrm{KILL}_s = \{\langle t, l \rangle \in D \mid l \in \mathrm{W}_s \text{ and } l \in \mathrm{W}_t \text{ for some } t \in S\}$ are the definitions killed when s executes.

The definitions that reach statements in a program are calculated by solving the following equations with any of the standard techniques:

$$DD_s^{\mathrm{in}} = \bigcup_{p \in pred(s)} DD_p^{\mathrm{out}} \tag{4.1}$$

$$DD_s^{\mathrm{out}} = (DD_s^{\mathrm{in}} - \mathrm{KILL}_s) \cup \mathrm{GEN}_s. \tag{4.2}$$

The reaching definitions for statement s, DD_s^{in}, determine its flow dependences. If s reads location l and $\langle t, l \rangle \in DD_s^{\mathrm{in}}$, then there is an l-definition-free path from the definition at t to the use in s, and hence s is flow dependent on t.

Computing whether a flow dependence is loop-independent or loop-carried requires a more complicated calculation. The first step is to identify the loop heads and back edges in a program's control-flow graph. If this graph is reducible, the back edges are found by computing the dominators in the flow graph and finding the edges whose target dominates their source. Let B be the set of back edges.

Loop-independent dependences are found by solving the reaching-definition problem over $E - B$, in other words, replacing Equations 4.1 and 4.2 by

$$LI_s^{\mathrm{in}} = \bigcup_{p \in pred(s) \text{ and } \langle p, s \rangle \notin B} LI_p^{\mathrm{out}}$$

$$LI_s^{\mathrm{out}} = (LI_s^{\mathrm{in}} - \mathrm{KILL}_s) \cup \mathrm{GEN}_s.$$

The resulting set contains only definitions that did not traverse a back edge and identifies loop-independent flow dependences.

Loop-carried dependences are found from the kill-free paths in a program. A *kill-free path* at statement t is a pair, $\langle s, l \rangle$, that indicates there exists a path from statement s to statement t along which no statement modifies location l. Kill-free paths are found by

solving the following equations with standard data-flow techniques:

$$K_s^{\text{in}} = \bigcup_{p \in pred(s)} \left[K_p^{\text{out}} \cup \{\langle p, l\rangle \mid l \in L\} \right]$$
$$K_s^{\text{out}} = K_s^{\text{in}} - \{\langle t, l\rangle \mid t \in S \text{ and } l \in \mathrm{W}_s\}.$$

There is a loop-carried flow dependence over location l between S_1 and S_2 if:

1. S_1 and S_2 are contained in a loop $\mathcal{L}$.
2. S_1 writes to location l and S_2 reads from location l.
3. There is a kill-free path for l from S_1 to H, i.e., $\langle S_1, l\rangle \in K_H^{\text{in}}$.
4. There is a kill-free path for l from H to S_2, i.e., $\langle H, l\rangle \in K_{S_2}^{\text{in}}$.
5. H does not kill l.

Since S_1 and S_2 are in $\mathcal{L}$, a path $S_1 \rightarrow H \rightarrow S_2$ must include a loop back edge, and because the path is l-definition-free, there is a loop-carried flow dependence from S_1 to S_2.

4.2.2 Computing Anti-Dependences

Anti-dependences are flow dependences in the reverse control-flow graph. The *reverse* of graph $G = \langle V, E\rangle$ is a graph $G^R = \langle V, E^R\rangle$ where $E^R = \{\langle v, u\rangle \mid \langle u, v\rangle \in E\}$, in other words, the graph with its edges reversed.

Theorem 3 *A program has an anti-dependence in its control-flow graph G if and only if it has a flow dependence in G^R.*

Proof Assume there exists an anti-dependence over l between S_1 and S_2. Then S_1 reads l and S_2 writes l and there is a definition-free path from S_1 to S_2. In G^R, there is a definition-free path from S_2 to S_1 and hence a flow dependence. By a similar argument, a flow dependence in G^R means there is an anti-dependence in G. ∎

The reverse of a reducible flow graph is not necessarily reducible. The faster algorithms for solving data-flow equations may not work for these graphs. Nevertheless, the equations can still be solved by node-splitting or with standard iterative techniques.

4.2.3 Computing Def-Order Dependences

Def-order dependences can be computed from the results of the flow dependence calculation. The def-order dependences at statement s, which uses location l, are computed:

```
/* u and v are statements in S */
if ⟨u, l⟩ ∈ DD_s^in and ⟨v, l⟩ ∈ DD_s^in and u ≠ v then
  if ⟨u, l⟩ or ⟨v, l⟩ are loop-carried
       or same-common-conditional-arms?(u, v) then
    if occurs-first?(u, v) then
      v is def-order dependent on u;
    else u is def-order dependent on v;
```

The predicate *same-common-conditional-arms?* returns *true* if both statements are in the same arm of every conditional statement that encompasses both statements. For example, in:

```
(set! x 1)                ; S1
(if p (set! x 2))         ; S2
(print x)

(if p
  (set! x 1)              ; S3
  (set! x 2))             ; S4
(print x)
```

S_1 and S_2 are def-order dependent, but S_3 and S_4 have no loop-independent data dependence.

4.3 Aliases and Imprecise References

The discussion above presumes that finite representations of the locations accessed by each statement (R_s and W_s) are known. However, many statements access unknown locations because of aliases and imprecise references. Aliases cause problems because a reference to one location may affect locations known under other names. By calculating the aliases and approximating the accessed locations with a superset of the true locations, we can find a conservative solution. Imprecise references cause different problems because accesses to locations in an aggregate cannot be distinguished, so several non-overlapping references can appear to conflict. These problems are solved by more precisely characterizing the subset of locations referenced by a statement.

Variable references are precise because they access a non-aggregate, but they are subject to aliases. The earliest published interprocedural data-dependence problem is side-effect determination, which is a flow-insensitive conflict-detection problem. This analysis finds the variables potentially modified by a function call. Banning found the modified variables by first computing the aliasing due to call-by-reference parameter-passing and then propagating a corrected set of modified locations through the call graph [8].

Array references are imprecise but usually not aliased. The dependence problem is to determine if two accesses to an array can touch the same element. Since this analysis

is usually performed intraprocedurally on nested loops, dependences are often classified by ad hoc techniques—such as Allen and Kennedy's and Wolfe's pairwise comparisons of statements containing array references [6,77]—instead of by a data-flow framework such as the one described above. The precision of these comparisons is increased by Banerjee's test, which determines if two array index expressions overlap by comparing their ranges.

Array dependence analysis is complicated by procedure calls, which introduce composite statements that access many array locations. This reduces the precision of the analysis since it is difficult to compare these sets. Triolet *et al.* and Li and Yew propose describing locations as sets of linear inequalities or descriptions of the index expressions [74,54].

Aliasing between array references can arise because of globally accessible variables (e.g., COMMON blocks) and parameter passing. Burke and Cytron suggested transforming the affected array references into the underlying memory reference and comparing the resulting expressions [16]. The transformed references are not aliased and can be directly compared.

The other aggregate dependence problem is references to data structures, which are connected by pointers into structure graphs. These references are both imprecise and aliased. Larus and Hilfinger proposed a technique (see Chapter 5) for determining potential conflicts by creating a finite model of the structure graph and comparing the nodes that statements access within this *alias graph* [52]. Horwitz *et al.* improved this approach in two ways: by combining the construction and analysis phase and by classifying the type of dependences, in effect solving the flow-sensitive version of this data-dependence problem [35].

These techniques for improving the analysis are necessary because the mapping from program quantities (variables, array accesses, etc.) to memory locations are necessarily imprecise. Therefore, the data-flow framework alone may not produce accurate enough results.

4.4 Related Work

Bernstein [10] identified the relations between the locations read and written by two data-dependent statements. Appropriately, he developed the equations to express the constraints on two statements that execute concurrently on a parallel computer. Bernstein's paper did not describe how to compute the locations accessed by a statement or account for aliasing or aggregate references, nevertheless it clearly presented the essentials of the data-dependence problem.

Kuck published the modern formulation of this problem and identified the three types of dependences [49]. He did not show how to calculate these dependences in general and did not distinguish loop-carried and loop-independent dependences. This distinction was first drawn by Allen [5].

The first published, systematic solution to the reaching-definition problem was in Kildall's paper on data-flow analysis [44]. Faster techniques for solving data-flow equations have since been published.

The first interprocedural data-flow calculation was side-effect analysis, which finds the variables potentially modified by a function call, taking into account the aliasing introduced

by call-by-reference parameters. Barth and Banning published solutions to this problem, which is simpler than data-dependence analysis since it is flow-insensitive [8,9].

Horwitz, Prins, and Reps presented an attribute grammar that computed the reaching definitions, flow dependences, and def-order dependences in a program [36]. Their solution is fundamentally equivalent to the formulation in this chapter, although the outward appearances are different.

Total grandeur of a total edifice,
Chosen by an inquisitor of structures
For himself. He steps upon this threshold
As if the design of all his words takes form
And frame from thinking and is realized.
– *Wallace Stevens,* To an Old Philosopher in Rome

Chapter 5

Dependences Among Structure Accesses

This chapter applies the data-dependence framework from the preceding chapter to the problem of detecting dependences among structure-accessing statements. Informally, a structure is a collection of named fields that contain either values or pointers to other structures. A structure graph is a group of structures linked by pointers. Variables contain the root pointers into a structure graph. A program may: read or write a field in a structure instance, dereference a pointer leading to a structure, or create new structures; however, it may not do anything else to pointers. These operations correspond to Tarjan's *pointer machine* model [70] and to Lisp's behavior. Other languages, most notably C, allow arbitrary operations on pointers. Programs using this feature cannot be analyzed in the framework of this thesis.

A statement specifies its path through the structure graph with a structure reference or access, which is a variable and a sequence of structure field names. The statement reads the contents of the fields specified by the reference, up to the last field, which the statement may modify or read.

The primary problem in detecting dependences among structure accesses is identifying the locations accessed by a structure reference. This set is potentially unbounded, as for example:

```
(defun f (x)
  (set! (car x) 2)
  (f (cons 1 2)))
```

The locations also depend on aliases in the structure graph. For example, in:

```
(if p
    (set! x y)
    (set! x z))
(set! (car x) 3)
```

the structure assignment modifies either location $y.car$ or $z.car$ as well as the location called $x.car$.

To determine which locations a statement accesses, we must know the aliases that reach the statement and have a way of labeling locations. CURARE uses *alias graphs*, which are finite representations of the structures visible at a point in a program. Locations in alias graphs are uniformly labeled, so the alias graph location accessed by a statement can represent the referenced structure locations in the data-dependence computation.

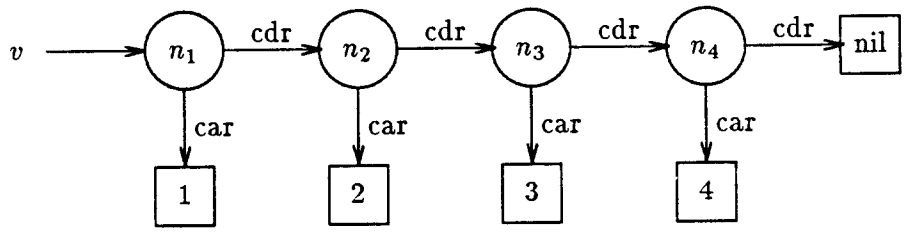
This chapter has eight sections. The first defines terminology. The next section describes alias graphs. Section 5.3 returns to the data-dependence problem and shows how alias graphs provide a basis for solving this problem. Section 5.4 and Section 5.5 describe alias graphs and present some examples. Section 5.6 show how to compute alias graphs. Section 5.7 describes a faster technique for computing alias graphs. Section 5.8 shows how to improve the precision of conflict detection with alias graphs. The final section surveys related work.

5.1 Structures and Structure Graphs

This section describes a model of structure instances and defines some basic notation. It also explains why the model is not a suitable basis for data-dependence analysis. A *structure* is an object composed of a collection of named fields. Each field may contain either a pointer to a structure or a non-pointer value. A collection of structures is modeled by a labeled, directed graph $G = \langle N, E \rangle$ called a *structure graph*. Each node in the graph, $n \in N$, corresponds to an instance of a structure. ACCESSORS is the set of names of the fields in structures in G. Program variables contain pointers to nodes in this graph.

An edge in G is a triple $\langle n, f, s \rangle \in E$, where n and s are nodes in N, and f is a field name. This edge indicates that the structure represented by node n contains a pointer in field f to the structure represented by s. A node has at most one arc with a given field name.

For example, the following structure graph describes a list of the integers from 1 to 4:



Variable v contains a pointer to the list.

An *access path* in G, $n.\alpha$, is a pair consisting of a node n and a string of fields $\alpha = \alpha_1 \ldots \alpha_l$, for $l \geq 0$, such that if $n = n_1$, $\{\langle n_1, \alpha_1, n_2\rangle, \ldots, \langle n_l, \alpha_l, n_{l+1}\rangle\} \subseteq E$.[1] Node n_1 is the *source* and n_{l+1} is the *destination* of the path. Let $dest(n.\alpha)$ denote the destination of path $n.\alpha$. In the graph above, $dest(n_1.cdr.cdr) = n_3$.

Define $\alpha_{i\ldots l}$ to be the sequence of field names: $\alpha_i \ldots \alpha_l$. Let $n.\tilde{\alpha}$ be $n.\alpha_{1\ldots(l-1)}$, in other words, an access path up to its last field. For example, if $n.\alpha$ is $v.cdr.cdr.car$, $n.\tilde{\alpha}$ is $v.cdr.cdr$. By definition, $n.\tilde{\epsilon} = n$ for the empty string ϵ.

A *location* in the graph is a pair, $\langle x.\alpha, \beta\rangle$, consisting of an access path and a field in the destination of the path: $loc(x.\alpha_{1\ldots l}) = \langle x.\tilde{\alpha}, \alpha_l\rangle$. To continue the previous example, $loc(n.\alpha) = \langle v.cdr.cdr, car\rangle$. Two locations are equal if their destinations and final fields are identical

$$\langle x.\alpha, \beta\rangle = \langle y.\gamma, \delta\rangle \iff dest(x.\alpha) = dest(y.\gamma) \text{ and } \beta = \delta.$$

Paths $x.\alpha$ and $y.\beta$ are *aliases*, written $x.\alpha \sim y.\beta$, if $dest(x.\alpha) = dest(y.\beta)$. In the example, $n_1.cdr.cdr \sim n_2.cdr$. Since aliases paths lead to the same node, $dest(x.\alpha.\delta) = dest(y.\beta.\delta)$ for all δ in ACCESSORS*, the closure under concatenation of the field names. An alias $x.\alpha \sim y.\beta$ is *minimal* if no proper prefixes of the paths are aliased.

Structure graphs are an abstraction of the data manipulated by a program, but they are too concrete to determine the locations that the program accesses. They represent a particular datum and do not abstract the commonality of a collection of values. For example, the function:

```
(defun copy-list (lst)
  (cond ((null? lst) nil)
        (t
          (cons (car lst) (copy-list (cdr lst))))))
```

terminates for arguments in the unbounded set of proper, noncircular lists. A structure graph can describe one list, but a description of all such lists requires an unbounded set of structure graphs. The other shortcoming of structure graphs is that they do not provide a uniform way of labeling locations for the dependence analysis described in Chapter 4.

5.2 Alias Graphs

A solution to these problems is to abstract structure graphs into *alias graphs*. These finite, labeled graphs represent the potential aliases in an unbounded set of structure graphs and

[1] In a strongly-typed language, legal access paths are constrained by the type system. Type information can be used to reduce the number of valid paths and potential conflicts (see, for example, Ruggieri and Murtagh [64]) but does not change the algorithms.

The term "path" is also used by Cartwright *et al.* in a similar context [17]. However, they propose paths as a replacement for pointers in high-level programming languages and do not discuss data dependences.

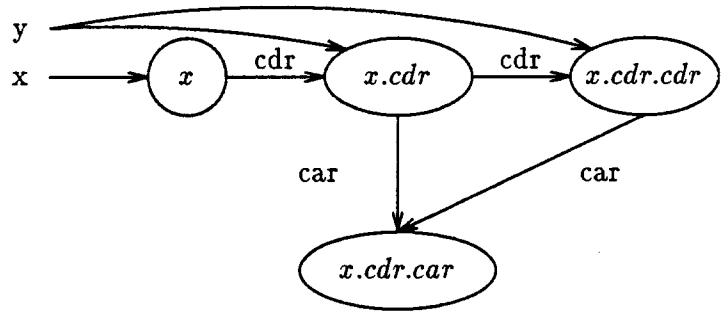


Figure 5.1: Sample alias graph.

uniquely label nodes within the graphs. However, the abstraction process removes some details, so an alias graph may not distinguish all distinct locations. Nevertheless, alias graphs are a good basis for dependence analysis.

The alias graph A_p at point p in a program contains all aliases that occur in the structure graphs that reach p during the program's execution. Therefore, if $x.\alpha \sim y.\beta$ in some structure graph at p, the paths $x.\alpha$ and $y.\beta$ form an alias in A_p. The converse is not necessarily true since alias graphs may conservatively overstate the aliases.

Alias graphs also uniformly label locations in structure graphs. Consider two points p and q in a program. If path $x.\alpha$ at point p leads to a structure instance I and path $y.\beta$ at point q leads to the same structure, the path $x.\alpha$ in A_p should lead to an alias graph node with the same label as a node pointed to by path $y.\beta$ in A_q. Therefore, by observing the labels of nodes accessed in A_p and A_q, we can find data dependences with the framework from the previous chapter.

For example, the alias graph in Figure 5.1 describes the minimal aliases:

$$\begin{aligned} x.cdr &\sim y \\ x.cdr.cdr &\sim y \\ x.cdr.car &\sim x.cdr.cdr.car \end{aligned}$$

Nodes in these graphs represent structures. Each node is labeled with the path along which it was first encountered. Arcs are labeled with the name of the field in the structure in which they are contained. If this graph reaches the statement `(set! (car (car y)) 5)`, then, the assignment modifies location $\langle x.cdr.car, car\rangle$—the *car* field in the node labeled $x.cdr.car$. We can deduce that this statement does not conflict with another statement `(set! z (car x))`, which reads location $\langle x, car\rangle$.

Formally, an *alias graph* is a labeled, directed graph, $A = \langle N_A, E_A, V_A\rangle$. The nodes, N_A, represent one or more structure instances and are labeled with path expressions (see Section 5.5.1). *label*(n) denotes the label of a node $n \in N_A$. The edges, E_A, are triples, $\langle n, f, s\rangle$, where $n, s \in N_A$ and f is a field name from ACCESSORS. A node may have more than one outgoing edge labeled with a field name, so long as their destinations are distinct

(i.e., the graph is not a multigraph). V_A is a map from program variables to nodes and identifies the entry points into the graph. These arcs are not labeled.

A *path* in an alias graph is similar to a path in a structure graph, except that nodes in an alias graph may have more than one outgoing arc with the same label, so that an alias graph path can lead to more than one node. If x is a variable and $\alpha = \alpha_1 \ldots \alpha_l$ is a string of field names, define $dest(A, x.\alpha)$ to be the set of nodes reachable along a path labeled $x.\alpha$ from the nodes pointed to by variable x in alias graph A. Two paths are aliases if $dest(A, x.\alpha) \cap dest(A, y.\beta) \neq \{\}$.

An alias graph A *correctly represents* a structure graph S at a point p in a program if:

1. Every minimal alias, $x.\alpha \sim y.\beta$, in S has an equivalent transitive alias in A so that $dest(A, x.\alpha) \cap dest(A, y.\beta) \neq \{\}$. Aliases in A are transitive if they are transitive in S, so $x.\alpha \sim y.\beta \sim z.\gamma \Rightarrow dest(A, x.\alpha) \cap dest(A, y.\beta) \cap dest(A, z.\gamma) \neq \{\}$.

2. The labeling of nodes in A is consistent with all other alias graphs for the program. Let A' be another alias graph that correctly represents structure graph S' and assume that both structure graphs contain a node n. Further assume that $dest(x.\alpha) = n$ in S and $dest(y.\beta) = n$ in S'. Then, A contains a node r in $dest(A, x.\alpha)$ with label l, A' contains a node r' in $dest(A', y.\beta)$ with label l', and the labels intersect.

An alias graph at point p is *correct* if it correctly represents all structure graphs that reach p during the program's execution.

5.3 Detecting Dependences Using Alias Graphs

Alias graphs provide a basis for comparing the locations accessed by each structure-referencing statement in a program. A location in an alias graph is a pair $\langle x.\alpha, f \rangle$, where $x.\alpha$ is the label of a node and f is the label of an arc leaving the node. This pair represents the field f in the structures denoted by the node with label $x.\alpha$. Define

$$loc_A(x.\alpha_{1\ldots l}) \;=\; \{l \mid l = label(n) \text{ for some } n \in dest(A, x.\tilde{\alpha}) \times \{\alpha_l\}\}.$$

Since nodes with intersecting labels represent overlapping sets of structure instances, alias graph locations provide the basis for the data-dependence tests described in the previous chapter.

Consider a statement t. Let A_t be the alias graph immediately before t executes. If t has the form $x.\alpha_{1\ldots n} \leftarrow y.\beta_{1\ldots m}$, then

$$\begin{aligned} R_t &= \bigcup_{i=1}^{n-1} loc_{A_t}(x.\alpha_{1\ldots i}) \cup \bigcup_{j=1}^{m} loc_{A_t}(y.\beta_{1\ldots j}) \\ W_t &= loc_{A_t}(x.\alpha) \end{aligned}$$

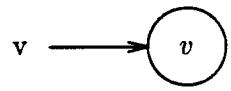
are the locations read and written by t, respectively. The locations enable the equations from Chapter 4 to find the dependences among statements.

5.4 Examples of Alias Graphs

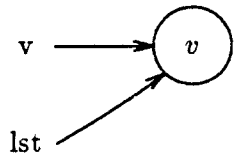
The following examples show how alias graphs are built and summarized. Section 5.6 presents the rules for constructing alias graphs from a program. In these examples, the limit $l = 2$. Consider a function `f`, which walks down a list:

```
(defun f (lst)
  (cond ((null? lst))
        (t
          (f (cdr lst)))))
```

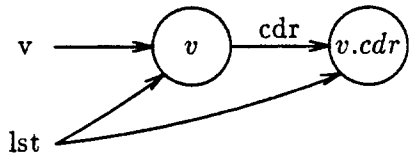
and assume that it is applied to the alias graph:



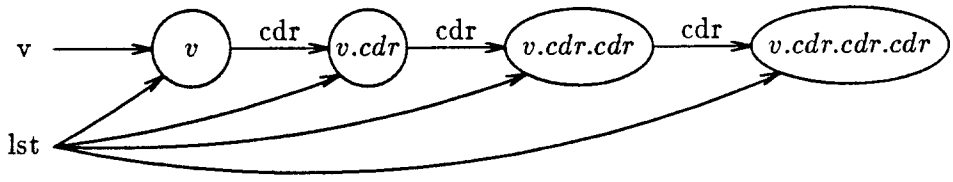
In the first iteration of the recursive function, the variable `lst` points to the same node as the actual parameter:



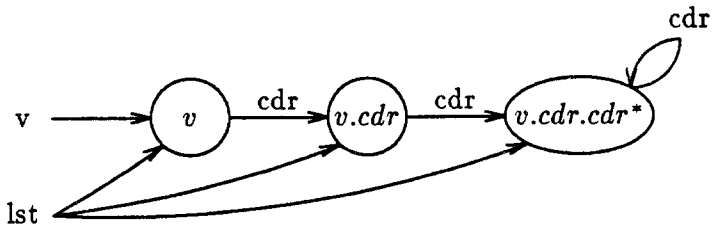
The next iteration extends the alias graph:



The following iteration also extends the graph. However, in the fourth iteration, the alias graph contains a node with a label that is too large:



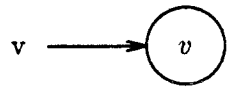
The summary of $v.cdr.cdr.cdr$ is $v.cdr.cdr^*$ so the graph contracts to:



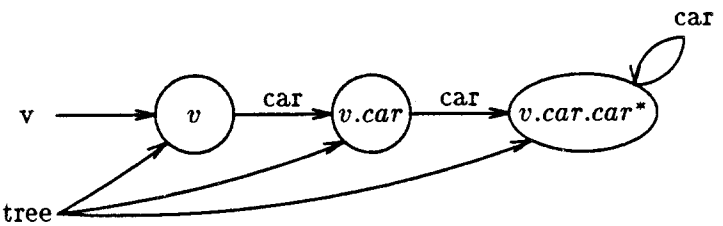
A slightly more complex example is the function g, which traverses a tree of cons cells:

```
(defun g (tree)
  (cond ((pair? tree)
         (g (car tree))
         (g (cdr tree)))
        (t)))
```

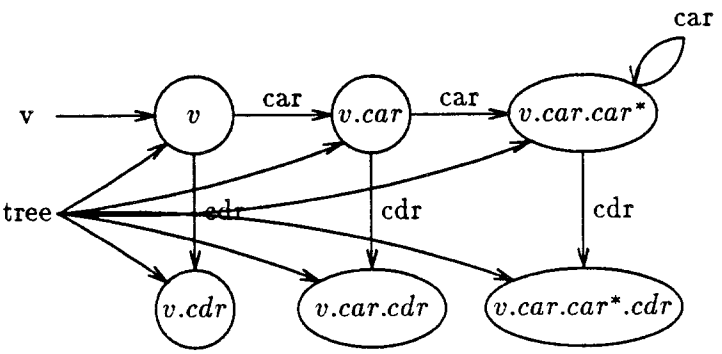
If g is also applied to the graph:



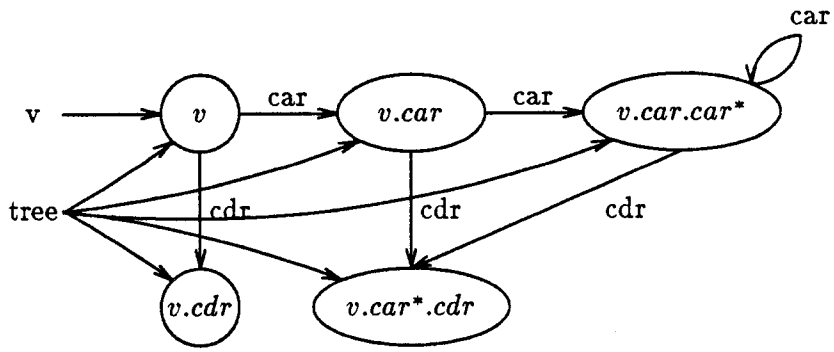
then the alias is built as follows. The alias graph is propagated through the first recursive call until the graph stabilizes:



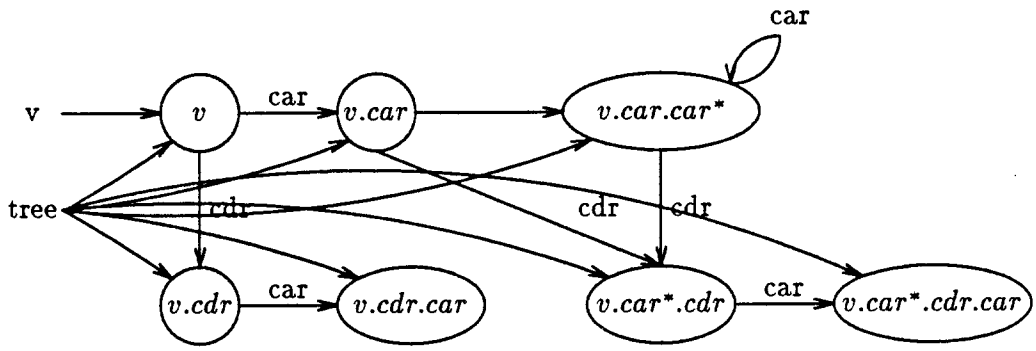
Then, propagating this graph through the second recursive call produces:



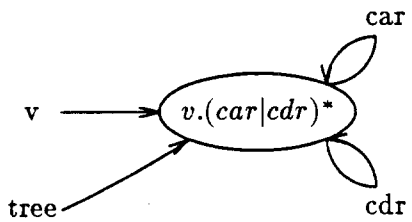
The label $v.car.car^*.cdr$ is too long and is summarized by $v.car^*.cdr$, so the graph contracts to:



If we next propagate this graph through the first recursive call, it becomes:



The path $v.car^*.cdr.car$ is summarized to $v.(car|cdr)^*$, so the graph reduces to:



which is the only finite representation of all paths in a tree. This graph, although full of spurious aliases, consistently labels all nodes in the tree so that data-dependence analysis reaches a conservative conclusion.

5.5 Details of Alias Graphs

The next few subsections describe the expressions that label alias graph nodes and define the operations on these graphs. These operations are used by the semantic equations in Section 5.6 to construct alias graphs for a program.

5.5.1 Path Expressions

The *path expressions* that label nodes are right-dominant regular expressions over the set of field names, ACCESSORS. In these path expressions, '.' denotes concatenation, '|' denotes alternation, and '$*$' denotes the reflexive, transitive closure under concatenation. A *right-dominant regular expression* (RDRE) is a regular expression in which the Kleene star operator subsumes terms to its right, so, for example, $x.x^* = \{x, x.x, \ldots\}$ but $x^*.x = x^* = \{\epsilon, x, x.x, \ldots\}$. More precisely, the interpretation of a RDRE r is the set of strings that it produces, $\mathcal{L}(r)$:

1. If $a \in$ ACCESSORS, then $\mathcal{L}(a) = \{a\}$.
2. If r_1 and r_2 are RDREs, then

$$\mathcal{L}(r_1.r_2) = \begin{cases} \mathcal{L}(\alpha.\beta^*.\delta) & \text{if } r_1 = \alpha.\beta^*,\ r_2 = \gamma.\delta, \\ & \text{and } \mathcal{L}(\gamma) \subseteq \mathcal{L}(\beta) \\ \{\gamma.\delta \mid \gamma \in \mathcal{L}(r_1) \text{ and } \delta \in \mathcal{L}(r_2)\} & \text{otherwise} \end{cases}$$

   In other words, the prefix of term r_2 that matches a closure at the end of term r_1 is dropped.

3. $\mathcal{L}(r_1^*) = \bigcup_{j=0}^{\infty} \mathcal{L}(r_1^j)$ where $\mathcal{L}(r_1^0) = \{\}$ and $\mathcal{L}(r_1^i) = \mathcal{L}(r_1^{i-1}.r_1)$.
4. $\mathcal{L}(r_1|r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$.

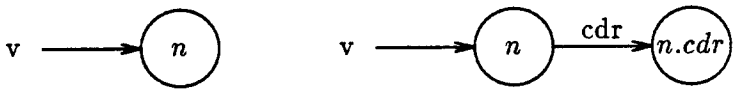
RDRE, like ordinary regular expressions, have a unique minimal representation as a deterministic finite-state automata (FSA). This representation is found by constructing a deterministic but not minimal automata. Terms dominated by closures can be found and removed by first finding all cycles in the FSA, and then repeatedly matching the body of a loop against all paths leading from an exit of the loop. The matched paths represent dominated terms and can be removed. The reduced FSA is then canonicalized with the usual techniques.

A *simple path expression* is a string of field names that does not contain closures or alternatives. A path expression is *compound* if it is not simple.

5.5.2 Labeling Nodes

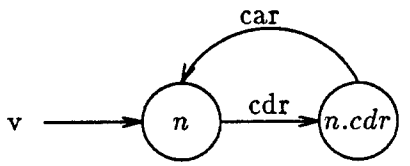
The labels on most nodes in an alias graph are the path through the graph to the node. If the node is reached by two or more paths, then the first path followed to the node determines

its label. This rule produces a consistent labeling. Ignore for the moment newly allocated nodes. A node with two or more predecessors must be contained in a correct alias graph. Therefore, any node not in the graph has a single path from its immediate predecessor. The new node's label is the concatenation of its predecessor's label and the field followed to the new node. For example, following the path *v.cdr* in the left alias graph extends it into the right alias graph:



As long as the new node is reachable (and hence part of the alias graph), it retains its label.

An arc is *continuous* if the destination node's label is the concatenation of the source node's label and the arc's label, as in the previous example. If an arc is not continuous, it is *discontinuous*. For example, the operation $v.cdr.car \leftarrow v$ produces:



and contains a discontinuous arc.

The function *next-nodes* either returns the successors of a node (n) along arcs with a given name (f) in alias graph (A) or augments the graph and returns a new node:

```
function next-node (A, n, f)
  if ∃s such that ⟨n, f, s⟩ ∈ E_A then
    return {s | ⟨n, f, s⟩ ∈ E_A};
  else let s = a (new) node in A with label label(n).f in
      add arc ⟨n, f, s⟩ to A;
      return {s};
        tel;
```

The function *dest*(A, $x.\alpha$) invokes *next-nodes* to follow or create a path through an alias graph.

```
function dest (A, x.α_{1...l})
  return destination (A, n, α_{1...l}) where ⟨x, n⟩ ∈ V_A

function destination (A, n, α_{1...l})
  if l = 0 then return {n};
  else return ⋃_{q ∈ next-node(A, n, α_1)} destination (A, q, α_{2...l});
```

A node labeled with a simple path expression is a *simple node* and denotes a single structure instance. A node labeled with a compound expression is a *compound node* and denotes more than one structure instance.

5.5.3 Limiting the Size of Alias Graphs

An alias graph is *l-limited* if no node has a label longer than l. The size of a path expression p, $size(p)$, is the number of accessors that it contains. Limiting an alias graph requires *summary nodes* that replace nodes with labels that are too large and possibly other nodes whose labels are matched by the summary node.

The function *summarize-pe* tries to find a short, compound path expression that matches a path expression that is too large. It returns $\perp$ if it fails.

function *summarize-pe* (pe, l)
  if $size(pe) \leq l$ **then return** pe;
  else let *new-pe* = *summarize-cat*(pe) **in**
    if $pe \neq$ *new-pe* **then return** *summarize-pe*(*new-pe*, l);
    else let *new-pe* = *summarize-alt*(pe) **in**
      if $size$(*new-pe*) $> l$ **then return** $\perp$;
      else return *new-pe*;

The function *summarize-cat* tries to find an expression of the form $\alpha.\beta^*.\gamma$ that matches a path expression. Expressions of this form are well-suited to describing linear lists or elements of a list.

function *summarize-cat* (pe)
  if pe $= x.\alpha.\underbrace{\beta \ldots \beta}_{i}.\gamma$ **return** $x.\alpha.\underbrace{\beta \ldots \beta}_{i-2}.\beta^*.\gamma$
  else return pe;

The function *summarize-alt* attempts a more drastic summary.[2] It returns a path expression that matches all strings over the alphabet of field names in the original path expression. This is the shortest expression that matches the original expression.

function *summarize-alt* (pe)
  if pe contains field names $\{\alpha_1, \ldots, \alpha_n\}$ **then return** $(\alpha_1 | \ldots | \alpha_n)^*$;

A summary node replaces (*subsumes*) all nodes in an alias graph that it matches. Path expression α *matches* path expression β, written $match(\alpha, \beta)$, if $\mathcal{L}(\beta) \subseteq \mathcal{L}(\alpha)$. Since path

[2] In some applications, it might be worthwhile to change *summarize-alt* so that it does not produce the shortest path expression, but rather to treat it like *summarize-cat*, which tries to match only nodes beyond the summarized node. In the example in Section 5.4, the change would not make a difference since $size((car|cdr)) = 2$, but if l was larger, then the tree structure up to the summary node would be explicit.

expressions are regular expressions, this property can be efficiently computed by the identity

$$\mathcal{L}(\beta) \subseteq \mathcal{L}(\alpha) \iff \overline{\mathcal{L}(\beta)} \cap \mathcal{L}(\alpha) = \{\}.$$

The three operations on the right side—complementation, intersection, and testing for an empty accepting set—are basic, efficient operations on automata.

When replacing node n by summary node s, arcs entering and leaving node n move to node s. The function *replace-node*(A, n, s) replaces node n by node s in alias graph A, moving all edges to the new node.

5.5.4 Union of Alias Graphs

The union of two alias graphs is an alias graph that contains all aliases in either graph. Conceptually, this operation is the graph union of the two graphs. However, the operation must preserve nodes' labels by ensuring that a summary node subsumes nodes from the other alias graph:

$$\begin{aligned}
A_1 \cup A_2 &= \langle N, E, V \rangle \text{ where} \\
N &= \{n = \mathit{subsume}(q, N_1 \cup N_2) \mid q \in N_1 \cup N_2\} \\
E &= \{\langle \mathit{subsume}(p, N_1 \cup N_2), f, \mathit{subsume}(s, N_1 \cup N_2)\rangle \mid \langle p, f, s\rangle \in E_1 \cup E_2\} \\
V &= \{\langle v, \mathit{subsume}(n, N_1 \cup N_2)\rangle \mid \langle v, n\rangle \in V_1 \cup V_2\}
\end{aligned}$$

$$\mathit{subsume}(n, N) = \begin{cases} q \in N & \text{if } q \neq n \text{ and } \mathit{match}(\mathit{label}(n), \mathit{label}(q)) \\ n & \text{otherwise} \end{cases}$$

5.6 Computing Alias Graphs

CURARE computes the alias graph at a point in a program by solving a set of data-flow equations. These equations use three auxiliary functions, *AGen*, *AKill*, and *ANew*, which are described first. Intraprocedural alias computation is infeasible since the effects of a call on an unknown function must be the extremely pessimistic assumption that the function modifies every reachable portion of the alias graph. The first three subsections define auxiliary functions. Section 5.6.4 shows how to extend control-flow graphs to entire programs. Finally, Section 5.6.5 presents the flow equations.

5.6.1 AGen

The function $AGen(x.\alpha,\ y.\beta,\ A)$ returns the new arcs added to alias graph A for the assignment $x.\alpha \leftarrow y.\beta$:

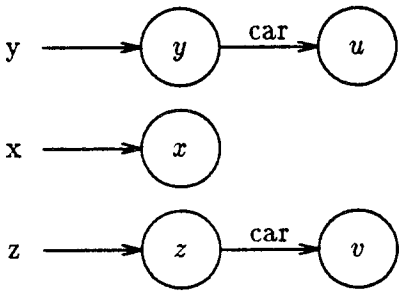
$$AGen(x.\alpha_{1\ldots l},\ y.\beta,\ A) = \{\langle n,\ \alpha_l,\ s\rangle \mid n \in dest(A,\ x.\tilde{\alpha}) \text{ and } s \in dest(A,\ y.\beta)\}.$$

This set contains an arc labeled α_l from each node in $dest(A,\ x.\tilde{\alpha})$ to each node in $dest(A,\ y.\beta)$.

5.6.2 AKill

The function $AKill(z.\gamma,\ A)$ returns the arcs in alias graph A removed by the assignment $z.\gamma_{1\ldots l} \leftarrow \ldots$. This set is not simply the arcs labeled γ_l leaving the nodes in $dest(A,\ z.\tilde{\gamma})$. The problem is that an arc in an alias graph asserts that a link *may* exist between the structures. Removing an arc from the alias graph asserts that a link *does not* exist. Using "may" information to compute "must" information is difficult.

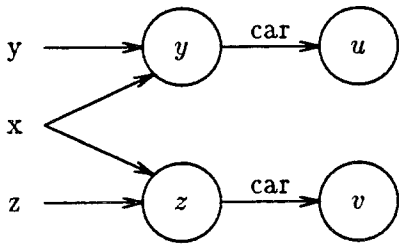
For example, assume that the alias graph:



reaches the statements:

```
(if p
    (set! x z)
    (set! x y))
(set! (car x) 5)
```

The alias graph reaching the structure assignment is:



The assignment cannot remove the arcs labeled *car* from both nodes y and z because, depending on which arm of the conditional is executed, one of these arcs is removed and the other is left alone. Removing both arcs would produce an incorrect alias graph. Removing neither arc produces a conservative but correct graph.

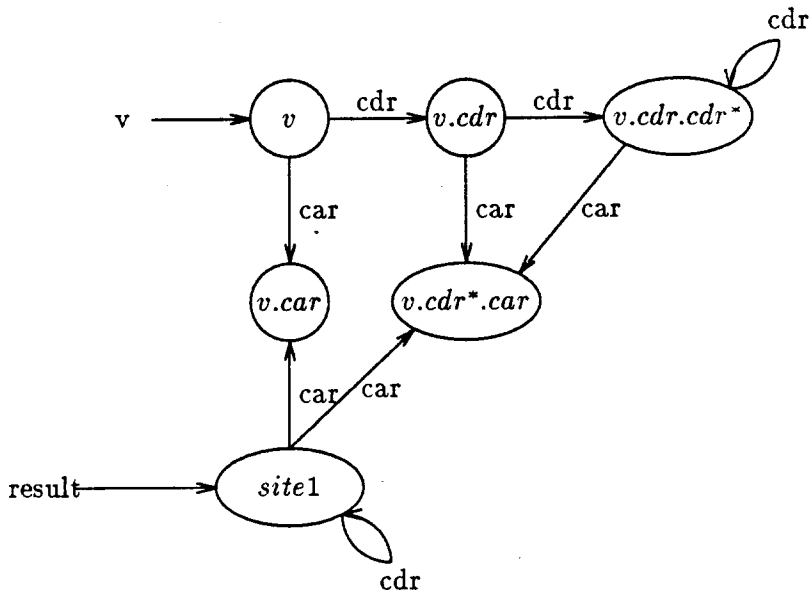
When $dest(A, z.\gamma)$ contains a single, simple nodes, only one structure field can be modified and the alias graph arc may be safely removed.

$$AKill(z.\gamma_{1\ldots l}, A) = \left\{ \langle n, \gamma_l, s\rangle \in E_A \;\middle|\; \begin{array}{l} |dest(A, z.\gamma)| = 1 \text{ and } dest(A, z.\tilde{\gamma}) = \{n\} \\ \text{and } label(s) \text{ is simple} \end{array} \right\}$$

5.6.3 ANew

The function $ANew(A, S, x.\alpha, a_1, \ldots, a_n)$ returns a subgraph containing a node for a new structure instance of type T created by the statement $x.\alpha \leftarrow alloc_T(a_1, \ldots, a_n)$ at call site S. The difficulty is to label the new node so it is distinct from the nodes for other structures.

One approach labels all nodes produced at a call site with the same label. This labeling is correct since different call sites produce distinct structures, but it is extremely conservative. For example, with this rule, `copy-list` (Section 5.6.3) produces the following alias graph:



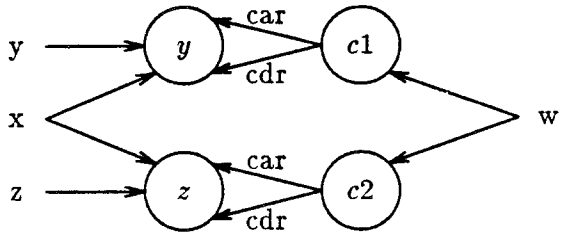
where the node labeled *site1* represents the results of the call on `cons`.

Another simple approach is incorrect. If *ANew* returns a new, distinctly labeled node for each call, the resulting flow equations are not monotone and are difficult to solve. A flow equation f is *monotone* if for all x and y, $f(x \wedge y) \leq f(x) \wedge f(y)$. To see why this version of *Anew* is not monotone, consider the statements:

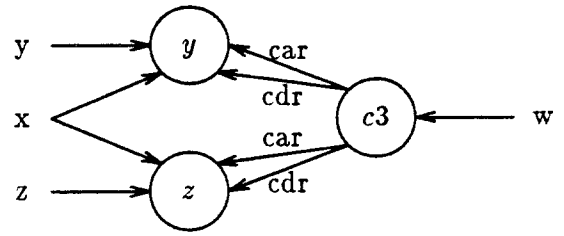
```
(if p
  (set! x y)
  (set! x z))
(set! w (cons x x))
```

The alias graph after the consequent and alternative clauses are A_1 and A_2, respectively:

Let f be an *ANew* function that creates a new node for each invocation. Then, $f(A_1) \wedge f(A_2)$ is:



where c_1 and c_2 are the labels of the new nodes. The problem arises because $f(A_1 \wedge A_2)$ is:

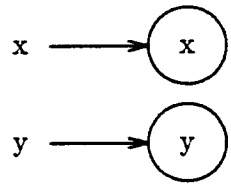


and therefore, $f(A_1) \wedge f(A_2) \not\geq f(A_1 \wedge A_2)$ so f is not monotone.

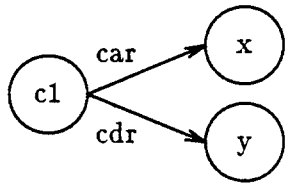
CURARE labels a freshly allocated node with a label derived from the labels of the nodes to which it points. This is the opposite of the convention for an existing node, whose label is derived from its predecessor's label. To distinguish the two, we put bars over the accessors in allocated node's labels, e.g., $\overline{car}$.

The rules for forming these labels are simple. Each call site that invokes an allocation function has a *default label*. Consider call site i with label c_i. *ANew* examines its arguments to find the sites at which they were allocated. If no nodes were allocated at site i, then *ANew* produces a node labeled c_i. If several nodes were allocated at this call site, *ANew* chooses the one with the longest label. Let this node have the label $c_1.\overline{\alpha}$ and assume that field f of the new node will point to it. *ANew* labels the new node $c_1.\overline{\alpha.f}$.

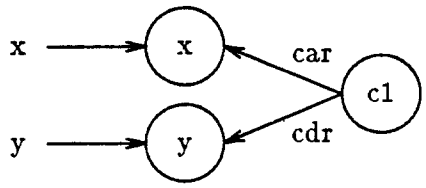
For example, consider call site 1: `(cons x y)` with label `c1`. If the graph reaching this statement is:



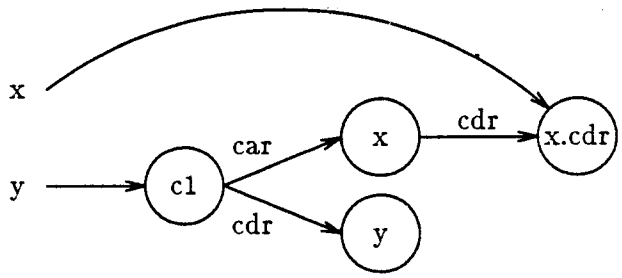
ANew produces the subgraph:



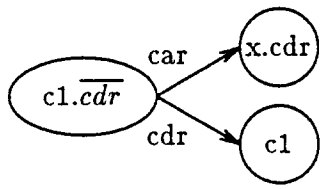
which, when added to the previous graph produces:



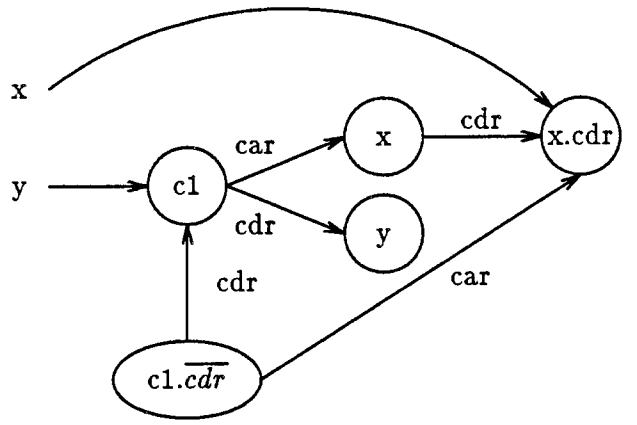
If the graph reaching the statements is:



then *ANew* produces the subgraph:



which, when added to the previous graph produces:



This labeling scheme produces a correct alias graph. Nodes from the same call site have the same default label unless one of *ANew*'s arguments was an older node from the same site. In this case, nodes with different labels must be distinct since a call on an allocation function cannot produce a circular structure. By considering the node with the shorter label to be older, the argument applies transitively to distinguish nodes from the same site with different labels.

The labeling is also consistent between alias graphs. Two nodes, in different alias graphs, represent the same object only if they have matching label. These two nodes must have originated at the same call site—in the same graph—and would have the same label there. Operations on alias graphs do not change a node's label, except to summarize it, which preserves the *match* relation.

One complication is that this labeling scheme is not monotone when the arguments to the allocation function point to more than one node. A solution is to examine each permutation of the arguments and produce a node with the appropriate label for it. This approach produces a monotone function, since $f(x) \subseteq f(x \wedge y)$, but introduces some spurious aliases.

ANew is defined:

function *ANew*(A, S, $x.\alpha$, a_1, ..., a_n)
  return $\bigcup_{d \in dest(A,\, a_1) \times \ldots \times dest(A,\, a_n)}$ *anew-sub*(A, S, $x.\alpha$, d)

function *anew-sub*(A, S, $x.\alpha$, d)
  let q = *other-node*(s, d)
    r = new node with label *label*(q).$\overline{\beta}_i$ where q is the i^{th} component of d and $\overline{\beta}_i$ is the accessor for the i^{th} field of the structure allocated at S
  in
    return $\langle \{r\}, \{\langle r, \beta_i, d_i \rangle \mid d_i \in d\} \cup \{\langle p, \alpha_l, r \rangle \mid p \in dest(A, x.\tilde{\alpha})\}\rangle$;

function *other-node*(s, d)

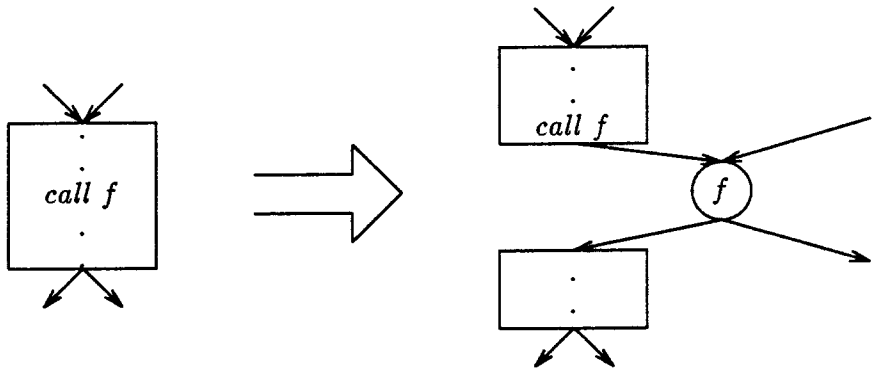


Figure 5.2: A basic block is split immediately after a call statement and new arcs are added to reflect the call and return.

$$\textbf{return} \left\{ d_i \in d \;\middle|\; \begin{array}{l} d_i \text{ is allocated at site } s \text{ and} \\ \neg\exists d_j \in d \text{ such that } size(label(d_j)) > size(label(d_i)); \end{array} \right\}$$

5.6.4 Extended Flow Graphs

Alias graphs are computed over the interprocedural *extended flow graph* of a program. This graph connects each function's control-flow graph to other functions through arcs for calls and returns. CURARE builds this graph from the functions' flow graphs. Let $f_1, \ldots, f_n$ be functions in the program and $G_1 = \langle B_1, E_1 \rangle, \ldots, G_n = \langle B_n, E_n \rangle$ be their control-flow graphs. Assume that each graph has a unique entry and exit node and that the functions invoked at each call site $s_1, \ldots, s_r$ are known.[3] The extended control graph, G, is the union of the individual graphs with new arcs to represent function calls and returns: $G = \textit{add-call-edges}(\bigcup_{i=1}^{n} G_i)$. Let $\textit{block}(s_i)$ be the basic block containing call site s_i, $\textit{entry}(s_i)$ be the entry nodes of functions possibly invoked from site s_i, and $\textit{exit}(e)$ be the exit node of the function with entry point e.

```
function add-call-edges(G)
  foreach call site s ∈ G do
    split block(s) into blocks B_c and B_r;
    foreach e ∈ entry(s) do
      add call arc from B_c to e;
      add return arc from exit(e) to B_r;
    od
```

Splitting a block after a call site requires a new block for the statements following the call in the block. If B is the old block, B_c is a new block containing statements up to and including

[3] If the functions are unknown, we can assume that any function can be invoked. However, if a program creates and invokes new functions during its execution, interprocedural flow analysis is impossible without declarations.

the call site s; B_r is a new block containing statements following s; the control-flow arcs incident on B lead to B_c; and, arcs leaving B emanate from B_r (see Figure 5.2).

If the result from a call is used, as in $x.\alpha \leftarrow f(a_1, \ldots, a_n)$, then separate the call and assignment into two statements: $f(a_1, \ldots, a_n)$ and $x.\alpha \leftarrow \langle result \rangle$, where $\langle result \rangle$ is a placeholder and the statements are in blocks B_c and B_r, respectively. If a call's result is not used, then insert the pseudo-operation $discard(\langle result \rangle)$ as the first statement of block B_r.

5.6.5 Alias Graph Equations

This section presents the data-flow equations used to compute alias graphs. The operation $A_1 \ominus A_2$ is the graph resulting from removing subgraph A_2 from alias graph A_1. The operation $A_1 \oplus A_2$ is the result of adding graph A_2 to alias graph A_1. $AGen_{i=1}^{n}(a_i, x, A)$ is an abbreviation for $AGen(a_1, x, A) \oplus \ldots \oplus AGen(a_n, x, A)$. $node(l, A)$ is the node labeled l from graph A along with its outgoing and incident arcs.

In the descriptions below, x, y, a_1, $\ldots$, a_n are variables, α and β are possibly-empty sequences of accessors, and S_1, $\ldots$, S_m are statements. A_s^{in} is the alias graph immediately before the execution of statement s. A_s^{out} is the corresponding alias graph immediately following the execution of s.

- $x.\alpha \leftarrow y.\beta$:

$$A_s^{\text{out}} = A_s^{\text{in}} \ominus AKill(x.\alpha, A_s^{\text{in}}) \oplus AGen(x.\alpha, y.\beta, A_s^{\text{in}}).$$

- $x.\alpha \leftarrow \langle literal \rangle$:

$$A_s^{\text{out}} = A_s^{\text{in}} \ominus AKill(x.\alpha, A_s^{\text{in}}) \oplus AGen(x.\alpha, \langle lit \rangle, A_s^{\text{in}}).$$

- $x.\alpha \leftarrow alloc_T(a_1, \ldots, a_n)$:
  $alloc_T$ is the allocation function for structures of type T.

$$A_s^{\text{out}} = A_s^{\text{in}} \ominus AKill(x.\alpha, A_s^{\text{in}}) \oplus ANew(A_s^{\text{in}}, S, x.\alpha, a_1, \ldots, a_n).$$

- **begin** $S_1; \ldots S_m$ **end**:

$$\begin{aligned} A_{S_1}^{\text{in}} &= A_s^{\text{in}} \\ A_{S_i}^{\text{in}} &= A_{S_{i-1}}^{\text{out}} \text{ for } 1 < i \leq m \\ A_s^{\text{out}} &= A_{S_m}^{\text{out}}. \end{aligned}$$

- **call** $f(a_1, \ldots, a_n)$:

Assume that $f_1, \ldots, f_n$ are the formal parameters of f.

$$A_s^{\text{out}} = A_s^{\text{in}} \oplus AGen_{i=1}^{n}(f_i, a_i, A_s^{\text{in}}).$$

- **call** $f(a_1, \ldots, a_n)$:
  f is a function with unknown effects.

$$A_s^{\text{out}} = A_s^{\text{in}} \oplus AGen_{i=1}^{m}(a_i, N_\perp, A_s^{\text{in}}) \oplus AGen(g, N_\perp, A_s^{\text{in}})$$

  for all global variables g. $N_\perp$ is the bottom node described below.

- **return** $x.\alpha$:
  Let $\langle result \rangle$ be a node label.

$$A_s^{\text{out}} = A_s^{\text{in}} \oplus AGen(\langle result \rangle, x.\alpha, A_s^{\text{in}}).$$

- $x.\alpha \leftarrow \langle result \rangle$:

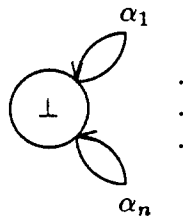
$$A_s^{\text{out}} = [A_s^{\text{in}} \ominus AKill(x.\alpha, A_s^{\text{in}}) \oplus AGen(x.\alpha, \langle result \rangle, A_s^{\text{in}})] \ominus node(\langle result \rangle, A_s^{\text{in}}).$$

- $x.\alpha \leftarrow discard(\langle result \rangle)$:

$$A_s^{\text{out}} = A_s^{\text{in}} \ominus node(\langle result \rangle, A_s^{\text{in}}).$$

5.6.6 Data-Flow Considerations

The meet operator, $\wedge$, for alias graph equations is *union-ag* (Section 5.5.4). The bottom element in the lattices of alias graphs, $\perp$, is the bottom node $N_\perp$:



By definition $match(\perp, \alpha) =$ **true** for all path expressions and ACCESSORS $= \{\alpha_1, \ldots, \alpha_r\}$. Therefore, $A \wedge \perp = \perp \wedge A = \perp$ for all alias graphs A. The bottom graph contains all possible aliases since every path leads to the same node.

5.6.7 Initial Values

The initial value of the alias graph propagated through a program must contain all aliases among the initial values of the program's variables. $\perp$ is a conservative approximation if nothing is known about these aliases. However, a more precise value will produce a better description of the possible aliases.

For example, if a global variable v contains a proper list, the declarations in Chapter 6 permit a programmer to declare this fact with:

```
(declare (distinct-loc v.cdr*))
```

5.7 Fast Computation of Alias Graphs

Fast data-flow solution techniques, such as Graham-Wegman data flow or interval analysis, can solve a slightly modified form of these equations. Since alias graphs grow large and operations on them become expensive, these techniques, which Kennedy has shown to usually require fewer operations than iterative techniques [43], can greatly reduce the cost of solving the data-dependence problem.

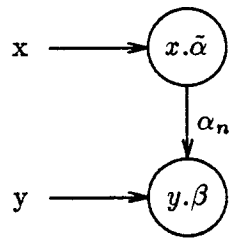
This section shows how to use the Graham-Wegman data-flow technique [28] to solve alias graph equations. We first show that alias graphs are tractable representations of the effects of a collection of statements. Next, we show that, although the equations are not Graham-Wegman fast, they can be modified to work with this technique.

5.7.1 Summary Graphs

Alias graphs can serve both as values propagated to solve the data-flow equations and as representations of statements' effects on propagated values. This dual role is analogous to the use of bit-vectors in data-flow problems to represent sets of values and data-flow GEN and KILL sets.

A *summary graph* is an alias graph that represents the effects of some statements. The statements' summary graph is computed by applying their data-flow equations to an initially empty alias graph and recording the order in which discontinuous arcs are added. The resulting graph describes the effects of the statements, except for the effect of *AKill*. However, *AKill*'s effects are limited and can be ignored. The discontinuous arcs in the summary graph describe the statements' effects.

Consider a single statement t and an empty alias graph. Let t be $x.\alpha \leftarrow y.\beta$. The resulting alias graph has a discontinuous arc $\langle n, \alpha_n, s \rangle$ where $label(n) = x.\tilde{\alpha}$ and $label(s) = y.\beta$ (unless $x.\alpha = y.\beta$, in which case the statement has no effect and the alias graph remains empty):



This discontinuous arc contains all information necessary to apply statement t to any other alias graph: the nodes at $dest(x.\tilde{\alpha})$ get an arc labeled α_n to the nodes in $dest(y.\beta)$.

A summary graph can also describe the effects of a sequence of statements. These effects need to be ordered, so discontinuous arcs must be marked with the order in which they were added to a summary graph.

Let t be a sequence of statements, S_t be its summary graph, u be another statement, and S_u be its summary graph. The effect of statements: **begin** t; u **end** is $S_{t;u} = S_u \circ S_t$, where $S_1 \circ S_2$ denotes the application of summary graph S_1 to alias (or summary) graph S_2. The function *apply-summary-graph* destructively applies summary graph S to an alias graph A:

```
function apply-summary-graph (S, A)
  foreach discontinuous arc ⟨n, f, s⟩ in S in order do
    foreach x ∈ destination+(n, f, A) do
      foreach y ∈ destination(s, A) do
        add arc ⟨x, f, y⟩ to graph A;
  return A;


function destination+(n, f, A)
  let d = dest(A, label(n)) in
    foreach x ∈ d do
      add arc labeled f from x to a node with label label(x).f;
    return d;

function destination(n, A) = dest(A, label(n))
```

The function *destination+* ensures that each node in $dest(A, x.\tilde{\alpha})$ has a continuous arc to its successor along an arc f. The destination of a compound path expression, $x.\alpha$, is the union of the destinations of the simple path expressions of size l or less matched by α. Newly allocated nodes are discussed below.

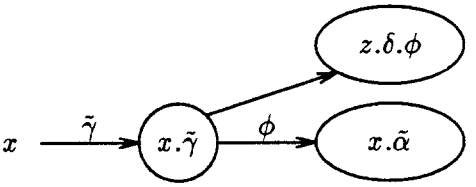
To see that this function is correct, we must prove that the composition of alias graphs is associative, so that $S_{t;u} \circ A = (S_u \circ S_t) \circ A = S_u \circ (S_t \circ A)$ for any graph A.

Theorem 4 *Let S_u be the summary graph for the statement u: $x.\alpha \leftarrow y.\beta$, S_t be the summary graph for a sequence of statements, and A be an alias graph. Then $(S_u \circ S_t) \circ A = S_u \circ (S_t \circ A)$.*

Proof S_u contains at most a single discontinuous arc $a = \langle n, f, s \rangle$, where $label(n) = x.\tilde{\alpha}$, $f = \alpha_n$ and $label(s) = y.\beta$. We will show that applying this arc to S_t produces arcs that have the same effect on A as directly applying arc a to $(S_t \circ A)$. We will only consider the interaction between $x.\tilde{\alpha}$ and S_t; a similar argument applies to $y.\beta$ and S_t.

First consider the effects of u on S_t when computing $(S_u \circ S_t)$. If no node in S_t along the path $x.\tilde{\alpha}$ is modified, $dest(S_t, x.\tilde{\alpha})$ contains a single node with label $x.\tilde{\alpha}$. Since arcs from this node are added to the summary graph after arcs from statements in t, they are applied to A in the correct order—after arcs from these statements. Therefore, the effects of u are the same if they are applied to S_t or to $S_t \circ A$.

If, on the other hand, $dest(S_t, x.\tilde{\alpha})$ contains more than one node, some statement in t, say $s: x.\gamma \leftarrow z.\delta$, must have modified the graph and $\tilde{\alpha} = \gamma.\phi$ for some $\phi \in$ ACCESSORS*. Assume the destination of this path contains only two nodes (the same argument is easily applied to more than two). The graph $S_t \circ S_u$ contains the subgraph:



When applied to A, these arcs cause *apply-summary-graph* to create arcs from $dest(A, z.\delta.\phi)$ and $dest(A, z.\tilde{\alpha})$ to nodes in $dest(A, y.\beta)$.

Now consider the other expression $S_u \circ (S_t \circ A)$. When S_t is applied to A, the arc introduced by statement s will cause *apply-summary-graph* to create arcs from nodes in $dest(A, x.\tilde{\gamma})$ to nodes in $dest(A, x.\delta)$. When S_t is subsequently applied to the resulting graph, the path $x.\alpha$ will find nodes in both $dest(A, z.\delta.\phi)$ and $dest(A, z.\tilde{\alpha})$. These sets are the same as the sets found by $(S_u \circ S_t)$, so both summary graphs have the same effect on A. ∎

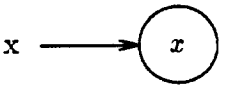
Allocated nodes slightly complicate the process of applying a summary graph. The labels of these nodes are not paths to be followed. Instead, *apply-summary-graph* must invoke *ANew* to produce a new node with the correct label derived from the nodes in the alias graph.

5.7.2 Fast Functions

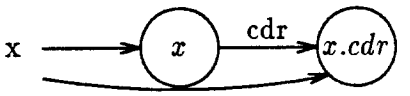
A function is *Graham-Wegman fast* if $(f(x) \wedge x) \subseteq f(f(x))$ for all x, in other words, if propagating a value once through the equations for a loop produces a minimal answer. The equations for alias graphs are not fast, as can be seen from the function:

```
(defun f (x) (f (cdr x)))
```

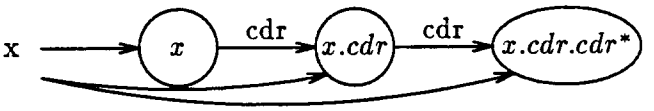
If the initial value of x is:



then, $f(x) \wedge x$ is:



but the correct solution (for $l = 2$) is:



Graham and Wegman suggest computing the *fastness closure* for non-fast functions. Define

$$\begin{aligned} f^0(x) &= x \\ f^1(x) &= f(x) \wedge x \\ f^i(x) &= f^1(f^{i-1}(x)) = f(f^{i-1}(x)) \wedge f^{i-1}(x). \end{aligned}$$

The fastness closure of f, $\overline{f}$, is $\overline{f}(x) = f^i(x)$ such that $f^1(\overline{f}(x)) = \overline{f}(x)$. The semantic functions for alias graphs (without *AKill*) have the property that $x \subseteq f(x)$ so that $f(x) \wedge x = f(x)$. Therefore, the fastness closure reduces to a simple closure of f, $\overline{f}(x) = f^{(i)}(x)$ such that $f^{(i)}(x) = f^{(i+1)}(x)$ where $f^{(i)}$ is the i-fold composition of f.

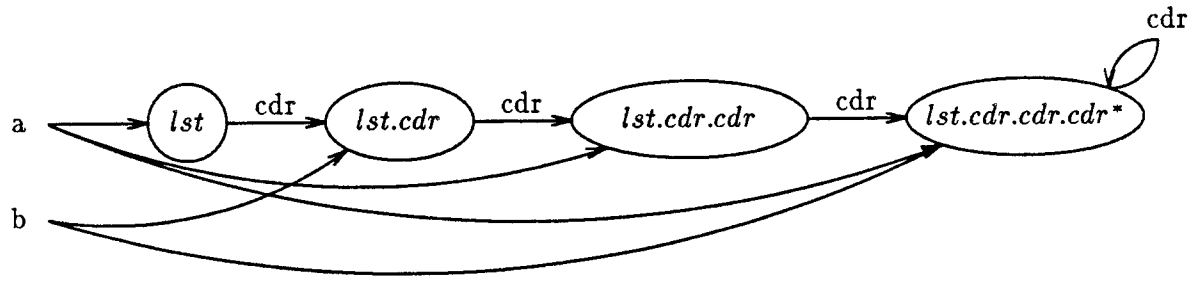
In practice, the number of iterations to compute this closure is difficult to bound because it depends on how the values are passed between iterations and the parameter l. But, for many programs, the result converges in a small number of iterations. Moreover, the cost of the operations is small since it is only applied once per loop.

5.8 More Precise Analysis of Structure Dependences

If used directly, alias graphs cause many unnecessary dependences because of the compound nodes that summarize a series of locations. For example, if the function:

```
(defun f (a b)
  (set! (car a) 1)
  (set! (car b) 2)
  (print (car a))
  (f (cdr (cdr a)) (cdr (cdr b))))
```

is invoked (`f lst (cdr lst)`), where `lst` is a proper list, then the alias graph (for $l = 3$):



reaches the assignment statements. The statements appear to conflict over the location $\langle lst.cdr.cdr.cdr^*, car \rangle$, although they do not actually modify the same location.

The precision of the analysis can be increased by examining the sequences of nodes referenced by two statements. A *sequence of nodes*, $N = \{n_1, \ldots, n_m\}$ is a collection of nodes labeled $x.\alpha.\gamma$, $x.\alpha.\beta.\gamma$, $x.\alpha.\beta.\beta.\gamma$, ... for $\alpha, \gamma \in \text{ACCESSORS}^*$ and $\beta \in \text{ACCESSORS}^+$. For example the nodes labeled *lst*, *lst.cdr*, *lst.cdr.cdr*, *lst.cdr.cdr.cdr** are a sequence. The last node in a sequence, n_m is its summary node. Assume statement S_1 accesses subset $N_1 \in N$ and statement S_2 accesses another subset $N_2 \in N$. If the sequences only overlap at the summary node, we want to know if the statements might access the same node if the sequences were longer.

To determine if such a node exists, we describe the nodes in each subsequence by a linear equation and check if there exists a common, integer solution to the equations. Let $\overline{N_1} = N_1 - \{n_m\}$ be the subsequence of non-summary nodes accessed by S_1. We try to find a linear equation, $i_j = mj + b$, that predicts the index, i_j, of the j^{th} element in $\overline{N_1}$. If no such equation exists, assume that $i_j = j$ and the statement accesses every node in the sequence. The statements in the example have the equations

$$\begin{aligned} i_j &= 2j \\ i_k &= 2k + 1. \end{aligned}$$

If the equations are $i_j = aj + b$ and $i_k = ck + d$, then if there exist integers j and k such that

$$aj + b = ck + d,$$

the statements conflict. If no integers exist (as in the example), the statements do not access the same structure instance. The bounded form of this problem occurs in detecting conflicts between array accesses and is partially solved by Banerjee's test. The unbounded equation is a linear diophantine equation that can be solved by Euclid's algorithm for the greatest common divisor.

If the statements conflict, the distance of the conflict is the difference in loop iterations between the statements that access the same location. Let i_j and i_k be the first non-negative solution to the equations. Then the conflict distance is $d = j - k$.

To perform this test, we must find an equation that describes the nodes in the sequence accessed by a statement. Either the equation is linear or we assume that the statement accesses all nodes in the sequence. An assignment statement modifies one node for each iteration of the surrounding loop. The distance between modified nodes is determined by the way in which the loop iterates over structures. The multiplier in the equation, m, is the greatest common divisor of the distances between sequence nodes passed at the recursive calls that form the loop. For example, if the function `f` contains recursive calls `(f (cddr x))` and `(f (cdddr x))`, then $m = \gcd(2, 3) = 1$. The constant b is the offset into the sequence of the first node passed to the function. These values can be found by observing the position, in the sequence, of the nodes modified by the assignment.

On the other hand, a statement may read many nodes in one invocation. If more than one of these nodes belongs to the sequence, the subsequence cannot be described with a linear equation. We then assume the statement reads every node. If only one node is in the sequence, the equation can be found like the one for an assignment statement.

5.9 Related Work

Two areas of previous work relate to alias graphs and their use in detecting data dependences among structure accesses. The first is work on describing the shape of structure graphs. The second is other attempts at solving the structure data-dependence problem.

5.9.1 Shape of Structure Graphs

Reynolds demonstrated a set of recursive equations, which he called a *data-set definition*, that describes the possible values of variables at points in side-effect-free Lisp programs [63]. He constructed the definitions for individual statements as functions of their input parameters. In a program containing a loop, these equations are recursive and can be simplified by substitution. The resulting recursive equations describe the range of possible values for a variable and the aliasing among variables.

Jones and Muchnick solved a similar problem [41], but used a more systematic approach that is the starting point for most subsequent work, including alias graphs. Their goal was a finite description of the structure graphs visible at each point in a program. They formulated the problem as data-flow equations for both side-effect-free and destructive Lisp programs and showed how to compute finite approximations as solutions. Their graphs did not directly provide enough information for data-dependence analysis, since they did not label locations. However, their equations have been extended for this analysis in this work and by Horwitz, Pfeiffer, and Reps (see below).

Ruggieri and Murtagh applied the Jones and Muchnick framework to the problem of determining the lifetimes of objects by computing a description of the sources of the objects (their allocating statements) [64]. Ruggieri's data-flow equations are similar to those for alias graphs, although both were developed independently. However, Ruggieri's equations do not label locations and cannot solve the data-dependence problem.

5.9.2 Dependences in Structures

An earlier version of this work described alias graphs and showed how to determine if two statements potentially conflict [52]. Its description of alias graphs is slightly more complicated than the one in this thesis. Its data-dependence test is markedly inferior as it did not classify dependences or improve the precision of the analysis by examining sequences of nodes. However, as this chapter showed, alias graphs can be extended to label locations and, along with the dependence framework from the previous chapter, to classify dependences.

Guarna discussed a technique for analyzing the data dependences due to pointers in C programs [30]. The problem of analyzing general C pointers is more difficult than analyzing Lisp structures because C programs can perform arbitrary computations on pointer values. Guarna does not address this aspect of C, so his work can be compared to CURARE. His technique is very different from alias graphs because he tries to maintain sets of equivalent paths (aliases) visible at each point in the program. Maintaining this information is expensive since a single assignment may create and destroy many aliases. Paths are also insufficient to compare the locations accessed by two statements since they do not label the locations. Finally, his paper does not show how paths can be computed for programs containing loops.

Horwitz, Pfeiffer, and Reps described an elegant technique for directly computing the data dependences in structure-accessing programs [35]. They use Jones and Muchnick's data-flow equations to compute a finite description of the structure graph at each point in a program. While computing this graph, they label each field with the last statement that modified its contents. By observing these labels before a statement reads or writes a field, they can directly find statements that have data dependences. This approach is simpler than CURARE's technique, but should be roughly comparable in efficiency since the major cost in both is the computation of the alias graph. Also, their technique will share the problem of spurious conflicts at summary nodes and require some external method for improving its precision at these points.

If decisions never had to be made,
life would be much easier,
and so would programming.
–Donald Knuth, The METAFONT Book

Chapter 6

Declarations of Data Dependences

Automated detection of data dependences, as discussed in the previous chapters, is one approach to detecting the constraints in parallel programs but is insufficient by itself. The results of this analysis may be conservative enough to prevent restructuring of a program. Rather than discard this analysis, we will enhance it with declarations by which programmers can assist the analyzer and correct its overly conservative conclusions. This approach relieves programmers of the burden of detecting and declaring the majority of data dependences and allows them to concentrate on refining the analysis of the most critical portions of their programs.

Before turning to the syntax and semantics of the declarations, it is worth exploring why they are necessary. The analysis algorithms previously presented have intrinsic limitations that must be understood and respected. The next section examines these limits. Section 6.2 describes a set of declarations that assist the data-dependence analysis and correct its conclusions. The final section of this chapter surveys related work.

6.1 Shortcomings of Data-Dependence Analysis

Data-dependence analysis for structure reference has some intrinsic shortcomings. These limitations reduce the precision of the analysis since two distinct locations may not be distinguished by the algorithm. However, they do not affect its correctness since merging locations creates spurious dependences but does not hide real ones. This section presents reasons why the analysis is not precise but does not propose remedies. The next section explores the solution of programmer-supplied declarations.

The first problem with this analysis is that precise (up to symbolic execution) computation of structure graph aliases is NP-complete, even within a single function. The proof is a simple variation of the argument Myers used to show that precise interprocedural flow analysis is NP-complete [60].

The structure alias problem, STALIAS, is defined as follows. Statements in a program have two effects on a structure graph (not on an alias graph). The first effect is

$AGen(x.\alpha_{1...m}, y.\beta)$, which changes field α_m in the object at $x.\tilde{\alpha}$ to point to the object at $y.\beta$. The other effect is $AKill(z.\gamma_{1...r})$, which destroys all aliases that traverse the field γ_r of the object at $z.\tilde{\gamma}$. At statement S in a program, $x.\alpha$ is an alias for $y.\beta$, written $x.\alpha \overset{S}{\sim} y.\beta$ if:

1. A statement G in the program has the effect $AGen(x.\alpha, y.\beta)$.
2. There exists a path in the control-flow graph from G to S along which there is no statement K that kills the alias.

The STALIAS problem is, given $x.\alpha$ and $y.\beta$, to find a path from the function entry to a point p such that $x.\alpha \overset{p}{\sim} y.\beta$.

Theorem 5 *STALIAS is NP-complete for programs consisting only of a single function, conditional statements, and structure and variable assignment.*

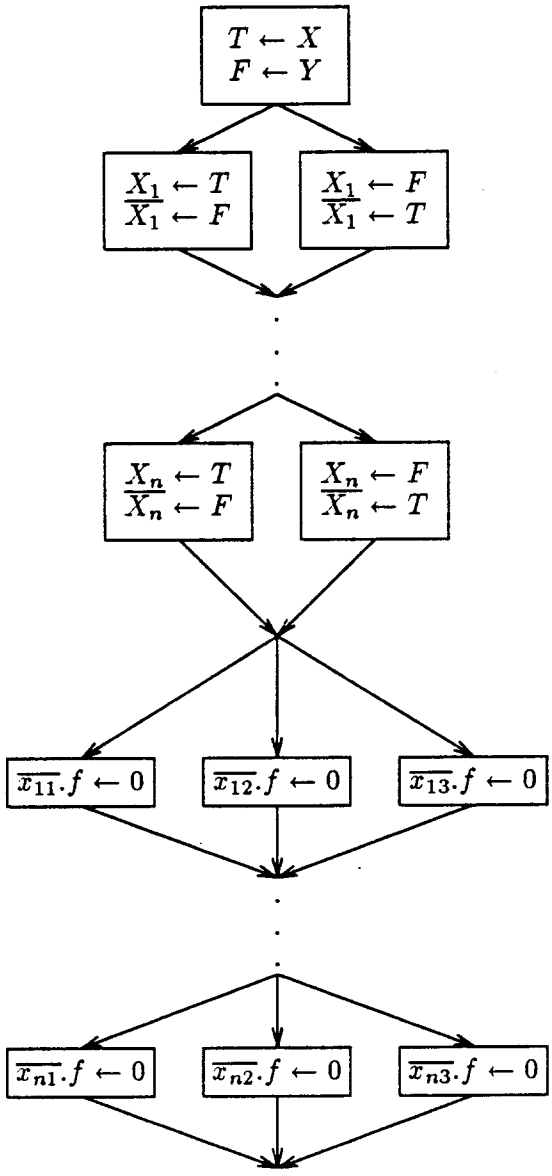
Proof STALIAS is in NP because given a solution, we can quickly check that $x.\alpha \overset{p}{\sim} y.\beta$ is generated and not subsequently killed along the path by simulating the effects of the statements along the path. We will show that STALIAS is NP-complete by reducing it to the 3SAT problem.

Let X be the variables $\{x_1, \ldots, x_n\}$ and $\overline{X}$ be their complements $\{\overline{x}_1, \ldots, \overline{x}_n\}$. Let $x_{i,j} \in X \cup \overline{X}$. The 3SAT problem is to find a consistent truth assignment to the variables in X that satisfies an equation $E = \bigwedge_{i=1}^{k}(x_{i,1} \vee x_{i,2} \vee x_{i,3})$. We will show how to construct a program flow graph in which STALIAS has a solution if and only if E can be satisfied.

The flow graph, G, consists of basic blocks $B_{i,j}$, program variables T, F, X, Y, $x_1, \ldots, x_n$, $\overline{x}_1, \ldots, \overline{x}_n$, and a structure with the field f. Figure 6.1 contains the flow graph. If E has a satisfying assignment, at least one literal in each clause in E is true, say $x_{i,j}$ for $1 \leq i \leq k$ and $1 \leq j \leq 3$. Therefore, $\overline{x}_{i,j}$ is false and there exists a path in G along which $\overline{x}_{i,j} \sim F$. Hence, the structure assignment $\overline{x}_{i,j}.f \leftarrow 0$ modifies the location $F.f$. Combining these paths produces a path through G along which $T.f$ is not modified so that $T.f \sim X.f$ at Q. Conversely, if $T.f \sim X.f$ at Q, there exists a path through G along which $T.f$ is not killed, so that a literal $\overline{x}_{i,j} \sim F$ in each clause along this path. ∎

The next problem is caused by summary nodes in alias graphs, which make these graphs finite and permit their computation by data-flow analysis. However, they also represent many structure instances with a single node, which can cause spurious dependences. Certain regular access patterns, such as traversing a list, can be characterized by examining other alias graph nodes and so can rule out spurious dependences (see Section 5.8). Other access patterns, such as traversing a tree or an arbitrary graph, are difficult to characterize or analyze more precisely. Two accesses to a summary node may appear to conflict, even if they reference different structures.

Another problem with alias graphs occurs at function entry points in the extended call graph, where the incoming alias graphs are combined into a single graph. This union may introduce spurious aliases. Aliases are also combined at basic blocks, but the two situations are different. Each call arc entering a function has a complementary return arc. Aliases



Q: Is $T.f \sim X.f$?

Figure 6.1: Program flow graph used to show that STALIAS is NP-complete. The question at the bottom is answerable if and only if the 3SAT problem is solvable.

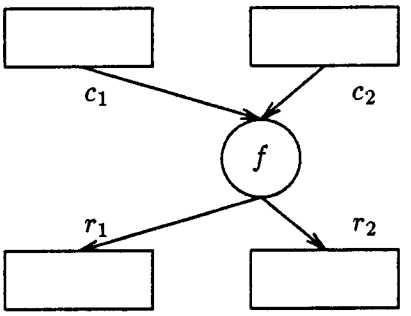


Figure 6.2: Each call arc entering a function f in an extended flow graph has a corresponding return arc leaving the function.

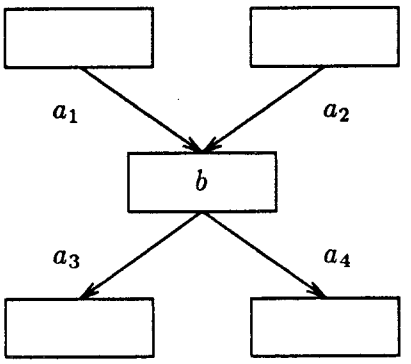


Figure 6.3: An arc entering a basic block b does not have a corresponding arc leaving the block.

entering along the call arc should only leave along the return arc. No analogous property holds for other control arcs. For example, in Figure 6.2, when control enters function f through call arc c_1, it leaves f along return arc r_1. However, in Figure 6.3, when control enters block b along arc a_1, it may leave through any exit arc.

Combining the alias graphs upon entering the function has two undesirable effects. The graph reaching statements in the function has all possible dependence-causing aliases, so that conflict-causing calls are indistinguishable from benign calls. Also, the alias graph leaving the function contains the union of the aliases, which may affect dependence analysis of statements that use the function's result.

For example, consider the function

```
(defun f (x y)
  (cond ((null?  x))
        (t
           (set! (cadr x) r)          ; S1
           (print (car y))            ; S2
           (f (cdr x) (cdr y)))))
```

If `f` is invoked with the arguments `(f 1 1)`, then statement *S2* has a loop-carried flow dependence on *S1*. If `f` is invoked `(f 1 (cdr 1))`, then *S2* has a loop-independent flow

dependence on *S1*. If `f` is invoked at both call sites, the combined alias graph makes it impossible to distinguish the cases.

There is no alternative to merging aliases, since this permits data-flow analysis to compute the effects of a function simultaneously for all call sites. An impractical alternative is to compute the aliases along each of the potentially exponential number of paths among functions. Another approach would label aliases with the call arc upon which they entered a function and then try to separate the resulting alias graph upon exit from the function. Since a value can flow through many functions, the length of these paths cannot be bounded. Tarjan's path expressions [71] use regular expressions to describe paths in a flow graph. However, they summarize all paths from the program's entry to a point and do not describe a single path.

Finally, the data-dependence calculation requires a precise call graph. For some languages and programming styles, computing such a graph is not difficult. However, for Lisp, and particularly Scheme, in which functions are first-class data objects, computing a precise call graph may be difficult or impossible. In this case, a call graph will be a superset of the actual call graph. The additional call and return arcs decrease the precision of the alias graph computation by introducing spurious aliases.

6.2 Declarations

The goal of the declarations described below is to improve CURARE's performance by refining the data-dependence analysis to eliminate spurious or unnecessary dependences and to provide additional information that is useful in transforming programs. Most declarations specify the absence of dependences, though a few declarations provide affirmative information that is difficult or impossible to infer. This bias is natural since the dependence algorithm finds a superset of the dependences and only needs to be made less conservative.

The declarations trade the precision and detail of the information that they provide against the difficulty of providing the information. At one extreme is a direct declaration that statement S_1 has a loop-carried anti-dependence of distance 2 with statement S_2. At the other extreme is a higher-level declaration that an argument to a function is always a tree. The second declaration is easier to provide since it is probably part of the programmer's unwritten knowledge about a program and may be necessary to the reasoning process that produced the first declaration. Nevertheless, detailed declarations have a role in specifying the precise dependence between two statements when an analyzer cannot refine its results even with the assistance of other declarations.

The consequences of providing these declarations are similar to those of other Lisp declarations. A correct declaration does not affect a program's result; it only increases the speed with which the result is calculated. The converse is that an incorrect declaration may change a program's result since the declaration overrides any analysis.

The syntax of the declarations is also based on Common Lisp. There are three constructs:

- `(declare` *property*) asserts that the property holds within the nearest enclosing scope.

```
⟨path expression⟩ :: = '%' |
                       ⟨accessor⟩ |
                       ⟨path expression⟩ '.' ⟨path expression⟩ |
                       ⟨path expression⟩ '*' |
                       ⟨path expression⟩ '|' ⟨path expression⟩ |
                       (⟨path expression⟩)
```

Figure 6.4: Syntax of path expressions in declarations. % is the empty string.

- (`locally` *property body*) asserts that the property holds within the statements in the body.
- (`the` *property expression*) returns the expression's value and asserts that the property holds for the expression's result.

Within the following descriptions, v, $v1$, $v2$, ... are variables and pe, $pe1$, $pe2$, ... are path expressions with the syntax in Figure 6.4. An object is *reachable* from a variable V if there is a sequence of pointers beginning at V that lead to the object.

6.2.1 Alias Declarations

These declarations refine the alias graph by specifying the connectivity among objects in the alias graph.

- `(declare (distinct-loc v.pe))`
  Asserts that the locations along the compound path $v.pe$ are distinct from each other. For example, a non-cyclic list is declared

  ```
  (declare (distinct-loc lst.cdr*))
  ```

  and a tree is declared

  ```
  (declare (distinct-loc tr.(car|cdr)*)).
  ```

- `(declare (distinct-loc v1.pe1 v2.pe2))`
  Asserts that the locations along path $v1.pe1$ do not intersect locations along path $v2.pe2$. For example, two distinct lists are declared

  ```
  (declare (distinct-loc lst1.cdr* lst2.cdr*))
  ```

  and the arguments to the example in Section 5.8 are distinguished by

```
(declare (distinct-loc a.car b.car)).
```

- `(the distinct-loc expr)`
  Asserts that the value returned by the expression is distinct from all previously reachable values. For example,

  ```
  (declare (distinct-loc (cons x y))).
  ```

- `(declare (unique-obj v.pe))`
  Asserts that the objects along the path *v.pe* are not reachable along any other path within the scope of the declaration. For example,

  ```
  (declare (unique-obj x))
  ```

  asserts that the object contained in variable `x` cannot be accessed except through this variable and

  ```
  (declare (unique-obj lst.cdr*))
  ```

  specifies that the cons cells in the list are only accessible from the previous item in the path.

- `(the unique-obj expr)`
  Asserts that the value returned by the expression is not reachable along any path. For example,

  ```
  (the unique-obj (cons x y)).
  ```

As a larger example, consider the function in Figure 6.5, which destructively concatenates each sublist in a list to its successor. A programmer can supply two useful assertions. First, each sublist must be distinct (or the algorithm will loop forever):

```
(declare (distinct-loc lists.cdr*.car)).
```

Second, cells in each of these lists must be distinct for the same reason:

```
(declare (distinct-loc lists.cdr*.car.cdr*)).
```

With these two assertions, or the equivalent knowledge of the structure of the actual parameter, the data-dependence algorithm can conclude that the assignments do not conflict.

```
(defun nconc (lists)
  (defun last-cell (x)
    (cond ((null? x) nil)
          ((null? (cdr x)) x)
          (t (last-cell (cdr x)))))

  (cond ((null? lists) nil)
        (t
          (set! (cdr (last-cell (car lists))) (cadr lists))
          (nconc (cdr lists))
          lists)))
```

Figure 6.5: Definition of the **nconc** function, which destructively appends each list in its argument.

6.2.2 Dependence Declarations

These declarations override the data-dependence algorithm and directly specify dependences between statements. They are particularly useful for programs that manipulate complex data structures, in which a simple description of the aliases is impossible but the programmer carefully controls potential conflicts. An example is a graph traversal algorithm in which each node is examined once although it may be visited many times. These declarations can also specify cases in which data dependences can be ignored since they cannot affect the eventual result. An example is adding a sequence of numbers. If the addition is atomic and associative, the order of additions does not matter.

- `(the sole-access expr)`
  Asserts that no other statement within the function containing the expression accesses the result of the expression. For example, in nconc, the assignment could be written

  ```
  (set! (the sole-access (cdr (last-cell (car lists))))
        (cadr lists)).
  ```

- `(the sole-read expr)` `(the sole-write expr)`
  Asserts that the result of the expression is not read (written) by any other statement within its function containing the expression. Other statements, however, may write (read) the location.

- `(the no-dependences expr)` `(the no-lc-dependences expr)`
  Asserts that the result of the expression does not cause any (loop-carried) dependences with another statement within the function containing the expression. The assignment from nconc could also be written

```
(set! (the no-dependences (cdr (last-cell (car lists))))
      (cadr lists)).
```

- `(declare (no-dependences f))`
  Asserts that invoking function `f` does not cause any conflict with any statement within the scope of the declaration. For example,

  ```
  (declare (no-dependences last-cell))
  ```

  specifies that this function does not conflict with the assignment in nconc.

- `(declare (no-dependences f g))`
  Asserts that invocations of functions `f` and `g` within the scope of this declaration do not conflict.

  ```
  (declare (no-dependences f f))
  ```

  asserts that multiple invocations of function `f` do not conflict with each other.

- `(the (dependence name type {func}) expr)`
  Asserts that the expression is involved in a dependence with another statement, which must be labeled with a declaration containing the same name (an arbitrary symbol). The type of the dependence is one of `FD(n)`, `AD(n)`, or `DO(n)`, where `FD` specifies a flow-dependence, `AD` is an anti-dependence, and `DO` is a def-order dependence. The integer n is the distance of the dependence and may be omitted, if unknown. If $n \neq 0$ (i.e., a loop-carried declaration), the name of the recursive function heading the loop must be given. This declaration overrides all dependences identified by the analyzer between the two statements.

6.2.3 Other Declarations

These declarations declare properties of programs that are difficult to infer but are useful for transforming programs for concurrent execution.

- `(the atomic expr)`
  Directs that the expression should execute atomically with respect to other invocations of itself. For example, one way to compute the length of a list is shown in Figure 6.6. The assignment statement does not conflict with itself if it executes atomically since integer addition is associative and commutative.

- `(declare (associative f))` `(declare (commutative f))` `(declare (ac f))`
  Asserts that function `f` is associative, i.e., $f(f(a,b),c) = f(a,f(b,c))$, commutative $f(a,b) = f(b,a)$, or both. These properties are of use in transforming programs since they remove constraints on solving recurrences (see Chapter 3).

```
(defun list-length (lst)
  (let ((length 0))
    (defun count (lst)
      (cond ((null? lst))
            (t
              (the no-conflict
                   (the atomic
                        (set! length (+ length 1))))
              (count (cdr lst)))))

    (count lst)))
```

Figure 6.6: The function `list-length` returns the number of elements in a list.

6.3 Related Work

Declarations of data dependences are rare in both theory and practice. Their paucity is not surprising since their primary role is to facilitate aggressive optimizations that rearrange operations in a program. These optimizations require automatic data-dependence analysis to support the declarations unless the dependences are infrequent enough that a programmer could declare all of them. Analysis of variable references is precise enough that no additional declarations are needed. On the other hand, analysis of aggregate references is difficult enough that it was considered impractical until the advent of parallel computers made these dependences impossible to ignore.

An example of weak dependence declarations used in practice are those in Symbolics' Common Lisp [59]. These declarations, which are undocumented and only used by system programmers, do not distinguish the location in a dependence, but only indicate whether a function produces or is affected by side effects.

The programming language Euclid [51] provides *collections* as a means of specifying the absence of data dependences. A collection is a region of memory that contains dynamically allocated objects of the same type. Pointers to objects specify, as part of their type, the collection to which the objects belong. Pointers to different collections never point to the same object, so some dependences are impossible because of the collections to which accessed variables belong. The drawback of collection is that a programmer must introduce new types and change type declarations to introduce data-dependence information.

A more precise and useful set of declarations is provided by the FX programming language [25] and is discussed in Lucassen's thesis [55]. This work proposes an *effect system* to parallel a language's type system. An effect is a subset of the operations $\{(\text{read } \rho), (\text{write } \rho), (\text{alloc } \rho)\}$. A region, ρ, is a user-defined name for a collection of objects. Lucassen describes an entire type system built on programmer-supplied declarations of the

effects of functions in a Scheme-like language.

These declarations could replace both the dependence analyzer and the declarations proposed in this work. Computing dependences among statements in a fully "effected" system is not difficult, though the precision of the analysis depends on the programmer paying attention to defining separate regions. The main objection to this approach is that a programmer must declare large quantities of information in many places. These declarations promise fewer benefits than type declarations, since they do not help a compiler find errors in a program. They only facilitate optimizing the program. Programmers are unlikely to declare their programs fully for this reason and so may unintentionally limit the areas in which a restructurer can find concurrency.

Is it not strange that desire should
so many years outlive performance?
– William Shakespeare, Henry IV, Part II

Chapter 7

Performance Evaluation

This chapter presents measurements that validate CURARE's runtime system and demonstrate a performance improvement for several transformed programs. The experiments were conducted in Qlisp running on an Alliant FX/8 multiprocessor [27]. Qlisp is a high-quality port of Lucid's commercial Common Lisp system to that multiprocessor.

CURARE's runtime system did not use high-level Qlisp constructs, such as `qlet` or `qlambda`. Instead, it used the low-level primitives for creating and synchronizing processes that underlie Qlisp. The Qlisp-specific code implemented the necessary SPUR Lisp multiprocessor features. It is presented in Appendix A together with the system-independent code.

The Qlisp system permitted stable, repeatable measurements, but it had several flaws. The timing function (`time`) did not function properly when executing concurrently. The only way to time parallel tasks accurately is to time the call on the parallel evaluation function (`qeval`). This function switches into parallel mode, establishes processes on the other processors, evaluates its argument, and cleans up. The overhead of these actions depends on the number of processes, but it is around 40 milliseconds in most cases (see Table 7.1). These overhead times have been subtracted from the reported times in this chapter.

Qlisp's other major flaw was that its synchronization primitive (a spin lock) was expensive, which increased the cost of sends and receives and the barrier synchronization. A `send` to and `receive` from an initially empty mailbox, without memory contention, takes 57.6 μsec. The calls on `acquire-lock` and `release-lock` necessary to synchronize these two operations consume 27.8 μsec. (48%) of this time.

The tests of the runtime system measured the cost of mapping the function `fib` (which computed Fibonacci numbers) over a list of integers. This function was chosen because its execution cost varied greatly depending on its argument. Its sequential execution time was measured by: `(time (mapc #'fib lst))`, which uses the `mapc` function supplied by Qlisp. Parallel times were measured by:

```
(time (qeval (progn (initialize-servers n)
```

Number of Parallel Processes (n)	Time (milliseconds)
0	43
1	43
2	44
3	44
4	45
5	46
6	46
7	47

Table 7.1: Time to execute:
`(qeval (progn (initialize-servers n) (identity t) (terminate-servers *idle-server-queue*)))`
This time measures the overhead required to switch into parallel mode, establish servers on n processors, execute a trivial task, and clean up.

```
(mapc-sync #'fib lst)
(terminate-servers *idle-server-queue*))))
```

The overhead of establishing and terminating the servers was subtracted from the reported times. The detailed measurement code is contained in Appendix B.

Unless otherwise noted, each test executed 100 or 1000 times and the execution times were averaged. The tests were conducted on a lightly-loaded 8 processor Alliant FX/8 using Qlisp Prototype Version 1.0 of January 4, 1989. All timing code was compiled.

The multiprocessor times are compared against two baselines. The first is the time to execute a sequential version of the code in Qlisp. In this mode of operation, Qlisp runs on a single processor, but it still incurs the (unknown) costs of multiprocessor-specific code built into the system. The ratio of the sequential time to the parallel time is the *sequential speed up*. It is larger than 1 when the parallel program runs faster than the sequential one.

The second baseline is the parallel code running in parallel mode on a single processor. This code incurs the cost of CURARE's runtime system but does not gain any advantage from parallelism nor incur any costs from memory contention. The ratio of the parallel time with 1 processor to the parallel time with n processors is the *parallel speed up*.

7.1 Performance of the Runtime System

This section presents measurements of simple test cases that demonstrate that the overhead of the runtime system is small for moderate sized tasks. Because this cost is small, parallel execution can achieve significant performance gains on loops with small bodies and loops that execute only a few times. These measurements do not represent CURARE's performance on real programs; they only show how well it works for isolated loops.

The first test measures the cost of mapping the function `identity` over a list. This

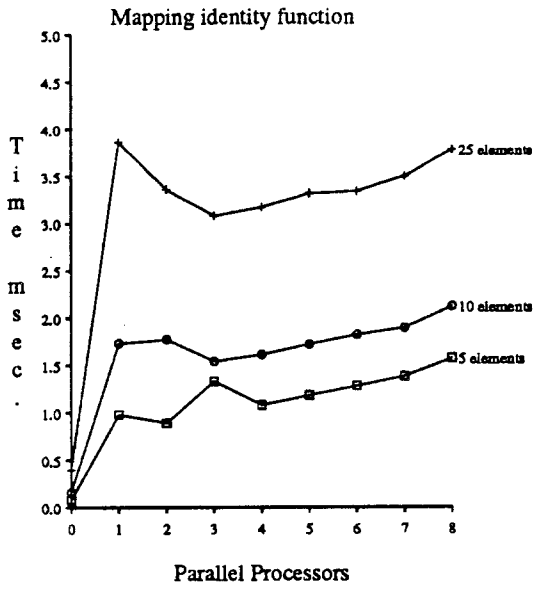


Figure 7.1: Timing results from mapping the function **identity** over lists of 5, 10, and 25 elements. Times are the average of 1000 trials. The sequential time is the one with 0 parallel processors.

function is perhaps the least costly one possible as it simply returns its argument. It executes in 12 μsec. Figure 7.1 shows that the overhead of `mapc-sync` is very large in comparison to `mapc`. The speed-up curves (Figure 7.2) show that two processors execute slightly faster than a single processor. Additional processors do not reduce the execution time because they spend most of their time contending for the task queue. When the cost of a task is much smaller than the cost of dispatching the task, additional tasks do not produce greater concurrency. The dispatch cost must be reduced to allow more loops to execute profitably.

By examining the time to map `identity` over lists of various lengths, we can estimate the overhead of creating and scheduling tasks. With only 1 processor executing `mapc-sync`, it takes $240 + 133n$ μsec. to map `identity` over an n element list. Of the 133 μsec., 57.6 (43%) is spent enqueuing and dequeuing the task.

When each iteration performs more work and is more like a typical loop, the overhead of the runtime system is less important. Figures 7.3 and 7.4 show the cost and performance improvement from mapping `(fib 5)`. This task requires 166 μsec., which is slightly longer than the dispatch time.

Parallel evaluation of this task utilizes upto three processors effectively. Additional processors increase the contention for the queue and prevent each processor from running at maximum efficiency. Again, the task queue is the bottleneck. Note that for lists of more than ten elements, parallel execution with two processors is faster than sequential execution.

Figures 7.5 and 7.6 show similar results from mapping `(fib 10)` over various lists. This computation requires 1.93 msec.—roughly 15 times the loop overhead. With two or

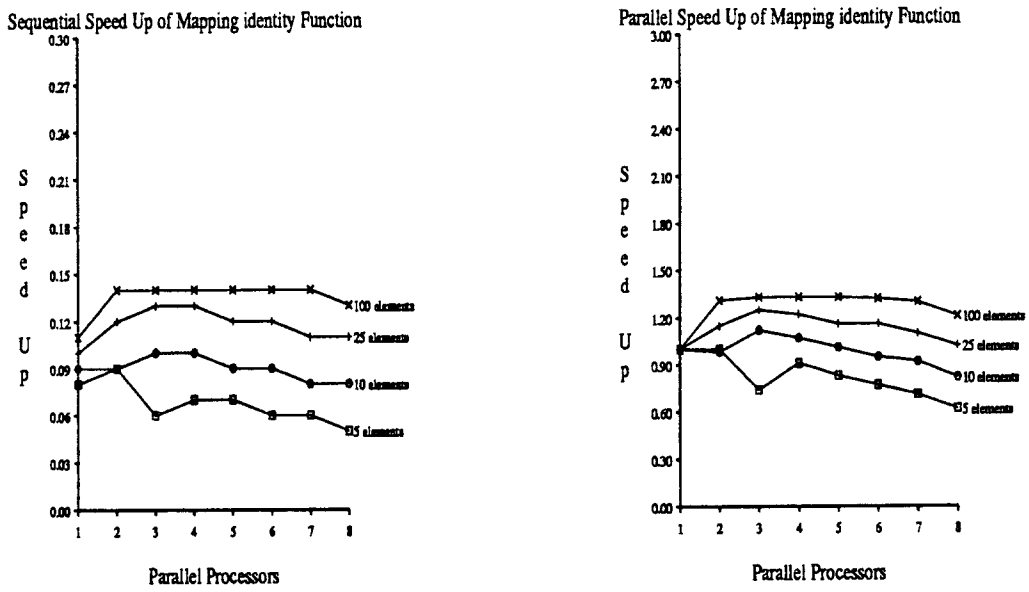


Figure 7.2: Speed-up curves from mapping the function `identity` over lists of 5, 10, 25, and 100 elements.

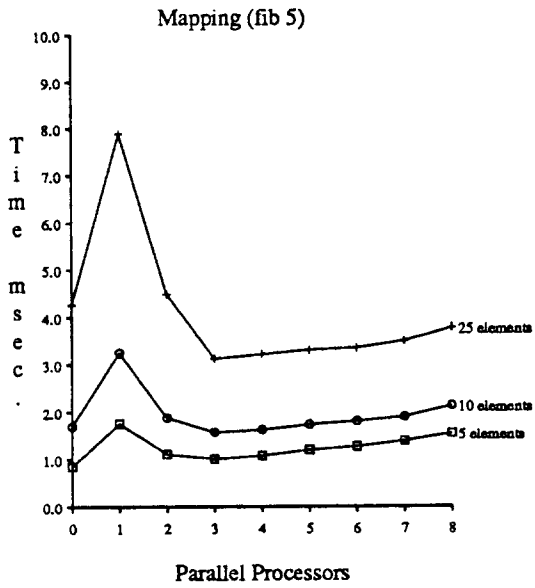


Figure 7.3: Timing results from mapping the function `(fib 5)` over lists of 5, 10, and 25 elements. Result for 100 elements is omitted. Times are the average of 100 trials.

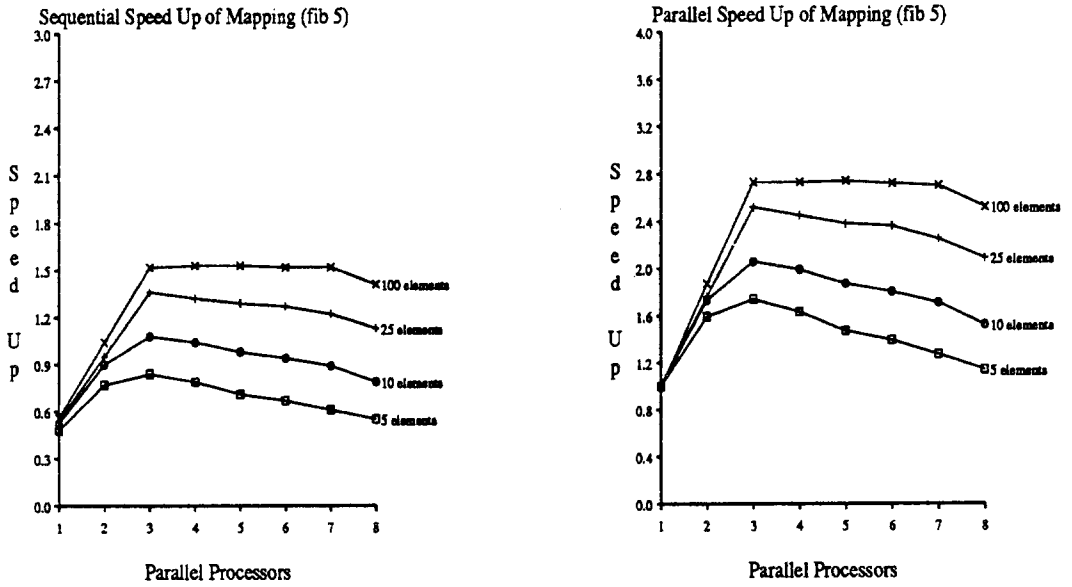


Figure 7.4: Speed-up curves from mapping the function (`fib 5`) over lists of 5, 10, 25, and 100 elements.

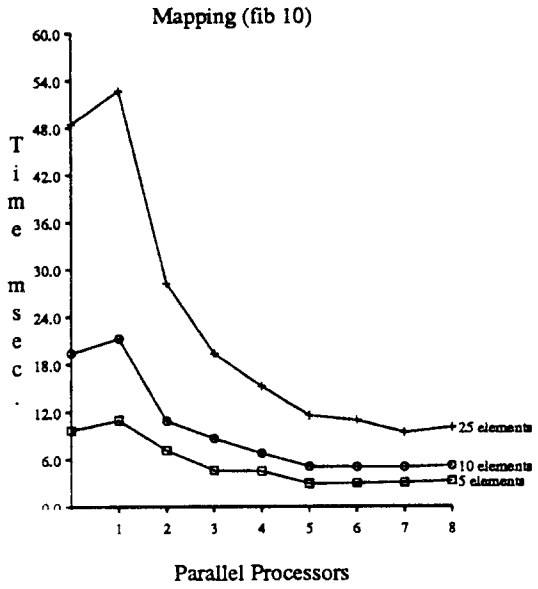


Figure 7.5: Timing results from mapping the function (`fib 10`) over lists of 5, 10, and 25 elements. Result for 100 elements is omitted. Times are the average of 50 trials.

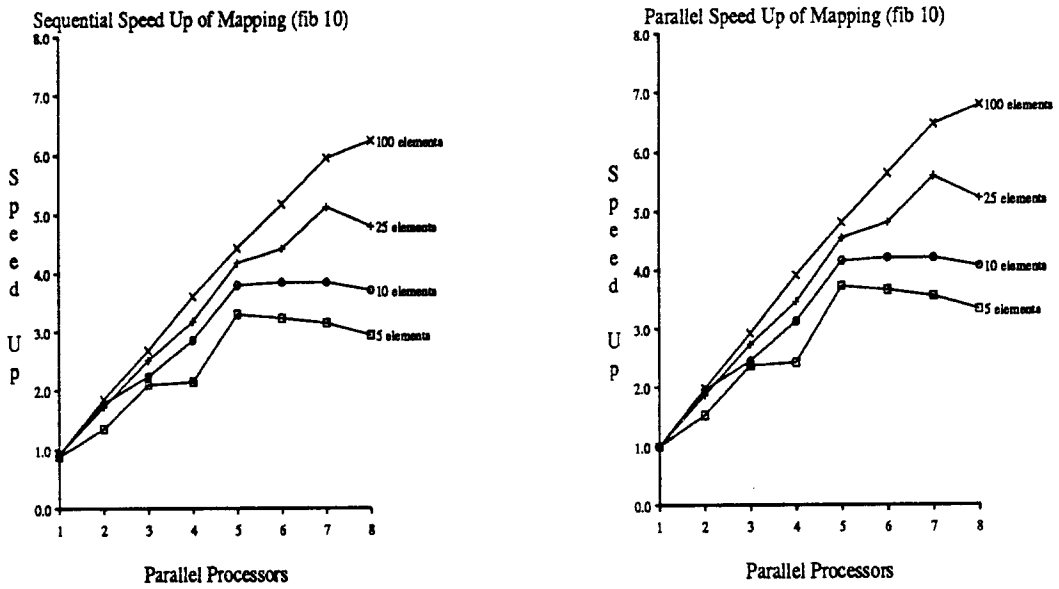


Figure 7.6: Speed-up curves from mapping the function (`fib 10`) over lists of 5, 10, 25, and 100 elements.

more processors, parallel execution is much faster than sequential evaluation and additional processors produce a roughly linear improvement for the longer lists (25 and 100 elements).

Finally, Figures 7.7 and 7.8 illustrate a problem that occurs when the cost of a computation varies greatly between iterations. In this test, we mapped `fib` over the integers from $1 \ldots n$. (`fib 1`) requires 13.7 μsec. and (`fib 25`) requires 2.7 sec. so the cost of each task varies widely. Since $fact(n) = fact(n-1) + fact(n-2)$, each task requires as much execution time as its two predecessors. `mapc` is synchronous so the longest executing task determines the total execution time. Because the last task is so much larger than all other tasks, it—along with the two preceding tasks—dominates the execution and renders more than three or four processors useless.

7.2 Measurements of Programs

This section presents measurements of the execution of several programs restructured by CURARE. These results demonstrate that CURARE can restructure non-trivial programs so their parallel version execute significantly faster than the sequential code. The speed improvements of a factor of 2-4 are excellent for mechanically-applied program optimizations.

The first program is the Boyer-Moore theorem prover from the Gabriel Lisp benchmarks [22]. This program is a simple, rewrite-rule based simplifier and tautology checker. It applies 106 lemmas to rewrite a formula into an identity. CURARE converted the function `rewrite-args` to the destination-passing style and spawned its recursive call concurrently. This function is the heart of the program since most of its execution time is spent applying rewrites. In fact, CURARE introduced parallelism in the same place and manner as did hu-

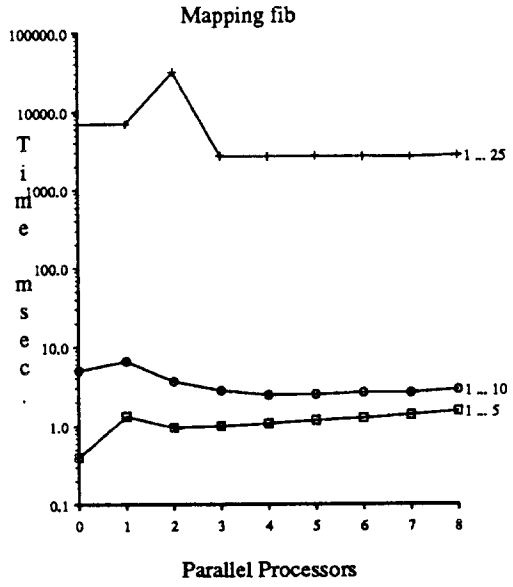


Figure 7.7: Timing results from mapping the function `fib` over the integers from 1 to n, for $n = 5$, 10, or 25. Times are the average of 10 trials. Log scale to accommodate disparity in times.

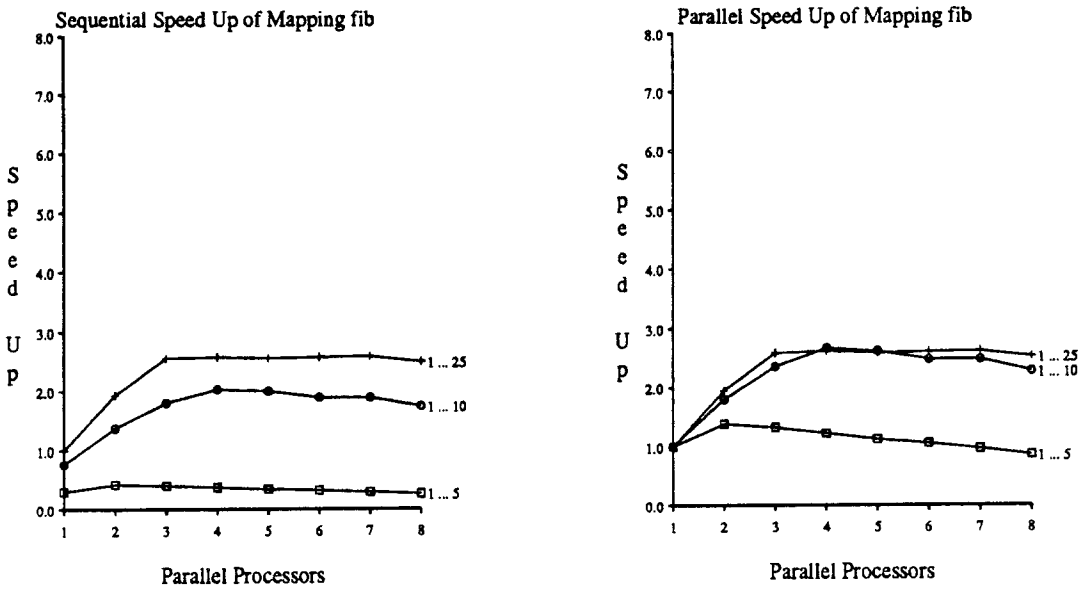


Figure 7.8: Speed-up curves from mapping the function `fib` over the integers from 1 to n, for $n = 5$, 10, 25, or 100.

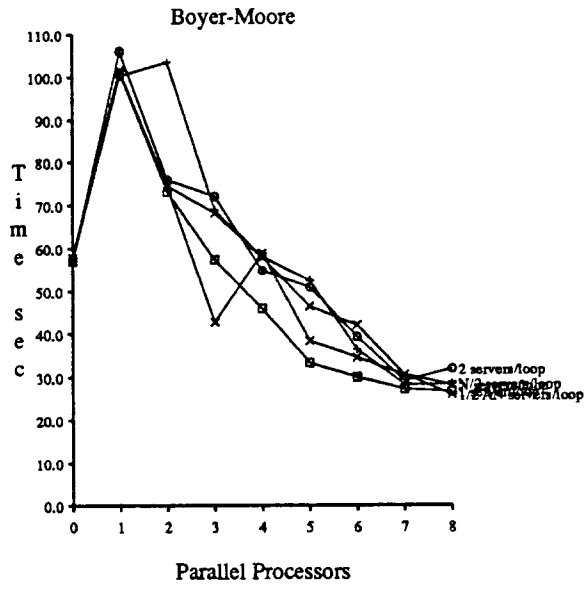


Figure 7.9: Time to execute the Boyer-Moore theorem prover benchmark. The various lines show different strategies for allocating servers to loops. Times are measurements from a single trial.

man researchers demonstrating their parallel Lisp systems [3,27]. CURARE required around a half hour of CPU time on a Sun 3/75 to analyze and restructure this program.

The version of Boyer transformed by CURARE was translated into Scheme by Seth Steinberg of BBN. The only non-trivial change was to collect the set of lemmas in an association list rather than storing them on symbols' property lists (standard Scheme does not provide property lists). Measurements show that this change increased the program's execution time but did not affect its speed up.

Figure 7.9 shows the benchmark's execution time for a variety of policies for allocating servers to loops. Since `rewrite-args` is non-linearly recursive (its call on `rewrite` can also lead to another call on `rewrite-args`), the demand for servers is much greater than the number of processors and many loops execute sequentially. We tried a variety of policies for allocating processors to loops:

- *1 server/loop*. Each loop was given one processor, if available, in addition to the one executing the call on `rewrite-args`.
- *2 servers/loop*. Each loop was given up to two additional processors, if available.
- *1/2 AN servers/loop*. Each loop was given half of the available processors.
- *N/2 servers/loop*. Each loop was given up to half of the total number of processors.
- *N servers/loop*. Each loop was given all available processors.

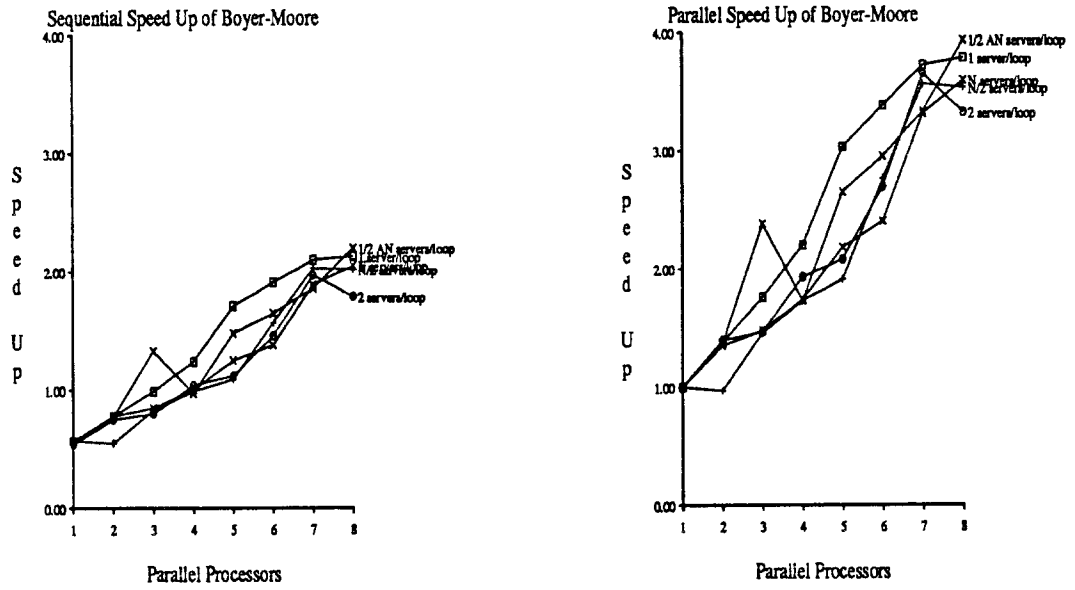


Figure 7.10: Speed-up curves from executing the Boyer-Moore theorem prover benchmark with different numbers of processors.

As can be seen from this data, the best strategy allocated half of the available servers to each loop. This policy gave the loops executed earlier (those higher in the expression tree) most of the servers, which favored the loops with the most expensive bodies.

The speed-up curves (Figure 7.10) show that, for the best allocation strategy, eight processors execute over twice as fast as a sequential processor. Since the parallel version had significant overhead costs (it ran at roughly half the speed of the sequential code), the parallel speed-up was actually much larger—a factor of four. Reducing these overhead costs would improve the sequential speed-up. These costs could be reduced in two ways. The first is to improve the runtime system to reduce the cost dispatching tasks. This improvement requires changes to Qlisp to provide access to machine-level primitives for synchronization. The second (discussed in Section 2.7) is to fall back on the sequential code, which has less overhead, when no parallel processors are available. Figures 7.11 and 7.12 demonstrate that this strategy improves the program's performance.

The parallel speed-up curves in Figure 7.10 or 7.12 compares favorably with Goldman's results for a hand-modified version of the benchmark running in Qlisp. His program produced a similar improvement of 3-4 with seven processors. The program, however, was considerably more complex since it used a depth-based cutoff to limit the spawning of parallel tasks.

The disparity in the rewrites's execution costs limit the potential parallel improvement of the Boyer-Moore benchmark. The simplest rewrites require only a few operations, but the more complex ones traverse large expression trees. Depth-first application of the rewrites compounds the problem since servers are likely to be assigned to small tasks near the leaves

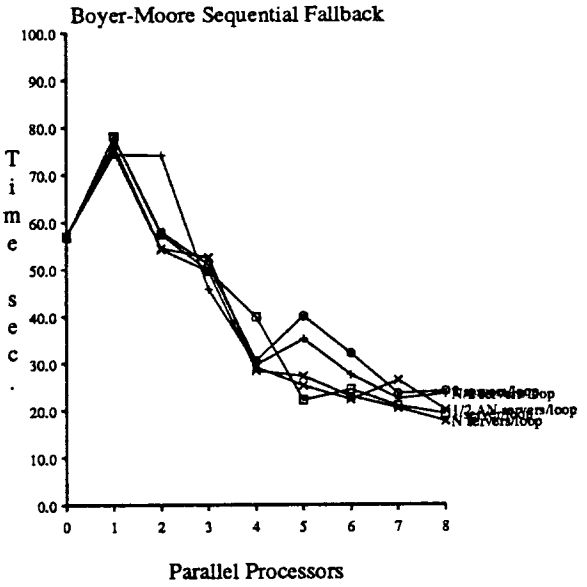


Figure 7.11: Time to execute the Boyer-Moore theorem prover with the optimization of falling back on the sequential code when no parallel processors are available. Times are measurements from a single trial.

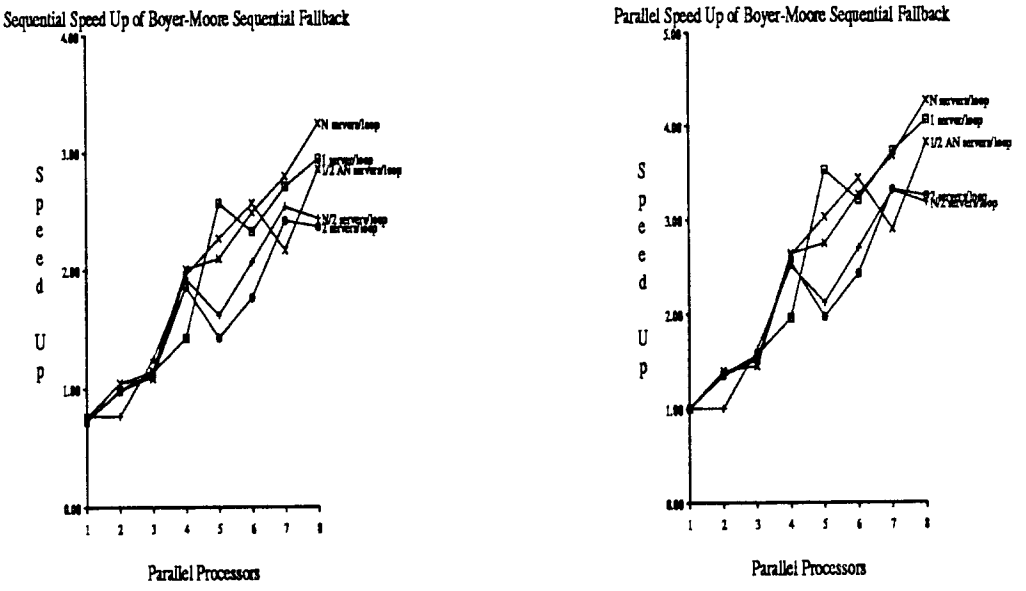


Figure 7.12: Speed-up curves from executing the Boyer-Moore theorem prover with the optimization of falling back on the sequential code when no parallel processors are available.

of the tree. The disparity in costs ensures that some processors are underutilized and limits the gains possible with concurrency.

The other test program is a Scheme version of Gabriel's *frpoly* benchmark. This program raises polynomials to powers. The original program was written for efficiency—with many special variables and goto statements—so we used the polynomial arithmetic operations from Abelson and Sussman [1]. These Scheme routines are similar to Gabriel's code in functionality.

CURARE found parallelism in two places in the polynomial routines. The first was the function:

```
(defun *-term-by-all-terms (t1 l)
   (if (empty-termlist? l)
       (the-empty-termlist)
       (let ((t2 (first-term l)))
         (adjoin-term (make-term (+ (order t1) (order t2))
                                 (*coeff (coeff t1) (coeff t2)))
                      (*-term-by-all-terms t1 (rest-terms l))))))
```

which CURARE changed to the destination-passing style (**adjoin-term** is simply **cons**) and spawned its call recursively. The second parallel loop required a programmer-supplied declaration. In the function:

```
(defun *terms (l1 l2)
  (if (empty-termlist? l1)
      (the-empty-termlist)
      (+terms (*-term-by-all-terms (first-term l1) l2)
              (*terms (rest-terms l1) l2))))
```

the composition operation of polynomial addition (**+terms**) is associative and commutative. With the assistance of a declaration, CURARE transformed the function so it performs a parallel reduction in the manner described in Section 3.3. CURARE required about 15 minutes to analyze and restructure this program.

The test case raised the polynomial $x^{100} + 2x^2 + 1$ to the 20^{th} power. Figure 7.13 shows the time required to compute this result with the server allocation strategies described previously. Unlike Boyer-Moore, the best strategy for this problem allocated all available processors to a loop. This strategy assigned the processors to the outer loop (the addition of subterms in ***terms**) rather than allotting any servers to the other, smaller loop.

Figure 7.14 shows the sequential and parallel speed-up curves for this test. As can be seen, the overhead of parallel execution is small. Unfortunately, the speed improvement is limited by the restructured form of ***terms**, which has a speed up of $n/\log n$ since it is a parallel reduction.

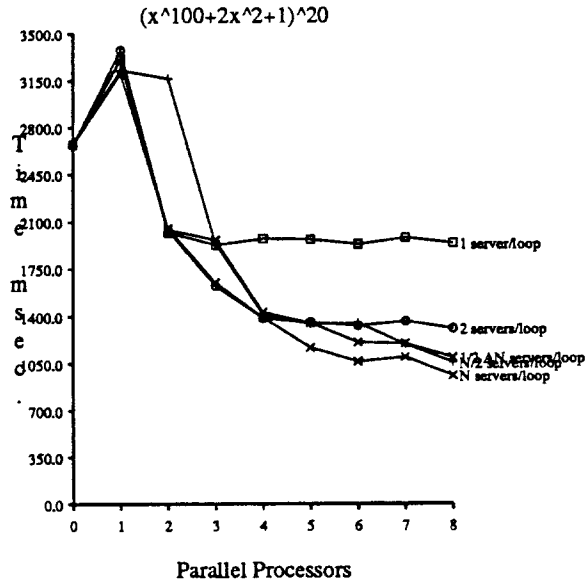


Figure 7.13: Time to compute $(x^{100} + 2x^2 + 1)^{20}$. Times are measurements from a single trial.

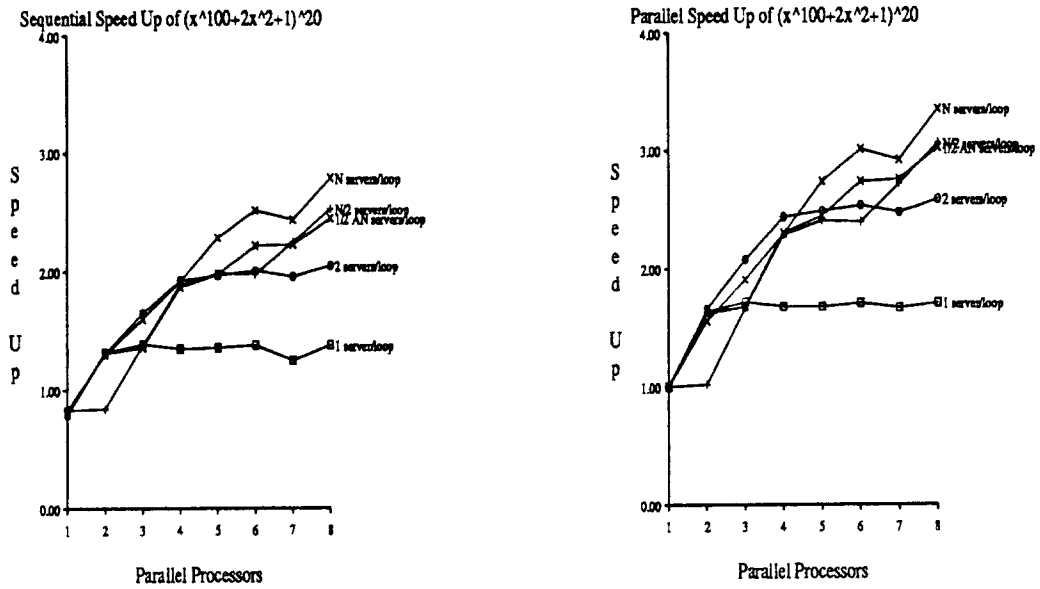


Figure 7.14: Speed-up curves from computing $(x^{100} + 2x^2 + 1)^{20}$.

7.3 Conclusion

The measurements in this chapter show that CURARE can detect and exploit parallelism in real programs. The speed improvements that CURARE produces are large enough to make a multiprocessor attractive to many programmers. The cost of using CURARE is roughly comparable to a conventional compiler and is much smaller than the cost of rewritting a program. More important, the speed improvement produced by CURARE is much larger than the improvement typically produced by switching compilers. Even though the restructured programs do not make optimal use of a multiprocess's parallelism, they take advantage of it.

These measurements also identified several potential bottlenecks that limit a program's concurrent execution.

- The cost of CURARE's runtime system. A restructured loop's parallel behavior is strongly dependent on the cost of scheduling iterations. If each iteration's execution cost is larger than the loop overhead, CURARE can produce effective speed improvements.
- The operations applied by a loop. If the execution times of loop iterations varies greatly, all processors will not be fully utilized and the speed improvement will be less than linear.
- The amount of parallelism inherent in the data manipulated by a program. If a parallel loop traverses short lists, then it will create few tasks and not much speed improvement.
- A restructured loop's contribution to a program's execution time. If the parallel loops only consume a fraction of the time, no amount of parallelism will greatly reduce the total execution time.
- Algorithmic limits on speed improvements. If a loop computes a parallel reduction, additional processors will not produce a linear reduction in the program's execution time.

These factors limit every manual and automatic scheme for introducing parallelism at a low level. From this data, we can conclude that CURARE performs best for programs in which a few loops, with expensive bodies, consume most of the execution time.

Of making many books there is no end;
and much study is a weariness of the flesh.
Let us hear the conclusion of the whole matter....
– Ecclesiastes 12:12-14

Chapter 8

Conclusion

CURARE demonstrates that symbolic programs can be automatically restructured for concurrent execution. The obvious problems—pointers, conditional execution, and recursion—can be resolved efficiently. The restructured programs show a valuable speed improvement because of concurrency. The major remaining question is whether many programs contain dependences and loops that make concurrent execution impossible. If programmers write programs that do not permit concurrent execution, then no restructurer will be able to take advantage of multiprocessors. However, this thesis contains many examples of common idioms and several large program that demonstrate that Lisp can be effectively restructured.

In summary, this thesis describes a simple approach to restructuring Lisp programs. CURARE first analyzes a program to find its control and data dependences. This analysis is most difficult for references to structures connected by pointers. CURARE uses a new data-dependence algorithm, which finds and classifies dependences between structure-accessing statements. This analysis is conservative and may detect conflicts that do not arise in practice. However, a programmer can temper and refine its results with declarations.

Dependences constrain the program's concurrent execution because, in general, two conflicting statements cannot execute in a different order without affecting the program's result. A restructurer must know all dependences in order to preserve them. However, not all dependences are essential to produce the program's result. Some approaches to computing a value cause more conflicts than others and are poorly suited for parallel execution. CURARE attempts to transform the program so it computes its result with fewer conflicts. An optimized program will execute with less synchronization and more concurrency.

CURARE's next examines loops in a program to find those that are unconstrained or lightly constrained by dependences. By necessity, CURARE treats recursive functions as loops and does not limit itself to the narrower class of explicit program loops. Recursive functions offer several advantages over explicit loops since they provide a convenient framework for inserting locks and handling the dynamic behavior of symbolic programs.

Loops that are suitable for concurrent execution are changed to schedule and execute their iterations on a set of concurrent server processes.

These servers are the final part of CURARE. They execute single loop iterations and therefore need to be extremely inexpensive to invoke. This constraint leads us to reject the processes provided by most concurrent Lisp systems and to design non-preemptable servers.

Restructured programs execute significantly faster than the original sequential programs. This improvement is large enough to attract programmers to a multiprocessor, particularly since it requires little effort on their part. Although restructured programs may not make optimal use of a multiprocessor's parallelism, they make good use of a programmer's time.

The major contributions of this work are:

- A clearer understanding of data dependences and their effect on concurrently executed programs. The data-dependence framework in Chapter 4 has not previously been described independent of particular dependence problems and has not been applied to the structure-reference problem. In addition, most program restructuring has not articulated a standard by which the transformed program can be compared to the original program. Conflict equivalence, which is described in Chapter 1, is the only generally-applicable standard and is implicit in most other work in this area.

- A technique for detecting and classifying data dependences among structure-accessing statements. The algorithm presented in Chapter 5 builds on previous work in describing data structures, but it is the first algorithm for detecting and classifying dependences among structure-accessing statements.

- Declarations for refining dependence analysis. This approach has not been necessary in other data-dependence problems, but it is important for structure graphs, which contain more aliases and are more difficult to analyze precisely. Chapter 6 describes the declarations.

- A technique for executing recursive functions concurrently. Previous work either treated recursive functions as conventional loops or used ad hoc techniques for introducing parallelism. The server mechanism described in Chapter 2 efficiently executes functions lightly constrained by dependences.

- Optimizations for removing data dependences from Lisp programs. The transformations in Chapter 3 are not fundamentally new, but they have not been previously applied to improve concurrent Lisp programs. Lisp's dynamic data structures require some major changes to the earlier versions of these transformations.

8.1 Future Work

Experience in applying CURARE to real programs will determine how many programs it can transform and how much concurrency is within this code. There are several areas that need to be explored further.

- Where are the bottlenecks in transformed programs? In particular, do the untransformed loops limit the speed up? Are the transformed loops large enough and do they have enough concurrency to use a multiprocessor effectively?
- Can the precision of the structure-dependence algorithm be further increased?
- Can the execution time required for this analysis be reduced, perhaps by an incremental algorithm that does not need to reanalyze an entire program when a portion of it changes?
- Are there additional transformations that can remove dependences in programs?
- Can the cost of locking be reduced so more functions with dependences can execute concurrently? Or alternatively, is there another approach to synchronization, such as optimistic concurrency, that offers better performance?

8.2 Is This the Way to Go?

It is worthwhile to step back and ask whether program restructuring, as exemplified by CURARE, is the proper way to program multiprocessors or whether it is just a temporary measure until a better technique is found. I believe that restructuring is a valuable technique that will be widely used to prepare programs for multiprocessors.

Programming languages that allow side-effects are natural and expressive and will continue to predominate. Programs written in these languages contain data dependences, which need to be detected and protected during concurrent execution. Programmers have difficulty finding and serializing these conflicts and, in doing so, introduce many errors. Data-dependences analysis, on the other hand, reliably detects all potential conflicts in a wide variety of programs and language features.

CURARE is weaker in areas that require a high-level view of the entire program, such as dividing the program into several large concurrent phases. This is an area in which a programmer can aid a restructurer since the programmer has a much higher-level view of the program and may better understand how to divide a problem. CURARE can assist by scheduling and synchronizing tasks that the programmer identifies. CURARE is also well-suited to introducing fine-grained parallelism within loops. Although a programmer could also transform these functions, there is no need for him to complicate and obscure a program by introducing low-level parallelism.

On the other hand, Lisp has major shortcomings as input to a restructurer. It does not provide an analyzer with much information about the types or effects of operations, so additional declarations must be grafted on to the language. A better language would incorporate this information in a manner that facilitates the analyzer's and programmer's tasks.

In addition, Lisp's fundamental data structure, the linked lists, is poorly suited to parallelism because items are not uniformly accessible. Abstract set operations could replace many uses of lists, thereby reducing explicit dependences and permitting a freer choice of

underlying data structure. Languages such as SETL have demonstrated the power of this abstraction for sequential programs.

From a broader perspective, concurrency complicates a program's semantics at the same time as it improves its performance. These complications can be directly addressed by a programmer who writes in a conventional language augmented with processes and synchronization devices. These programs are difficult to write, to debug, and to maintain because low-level implementation details complicate a program's structure. A better approach is to build tools that translate programs written at a higher semantic level to run on whatever type of computer is most efficiently constructed. These programs can be written to be understood, not just executed.

CURARE changes programs written in a simple, deterministic semantics to run on asynchronous computers. Perhaps more important than the details of its translation process, CURARE demonstrates that the data dependences and execution behavior of symbolic programs do not preclude concurrent execution. Restructuring is not just for numeric programs.

Appendix A

Source Code for Qlisp Runtime System

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: CURARE; -*-

;;; runtime.cl
;;; Copyright (C), James R. Larus, 1988 and 1989.  All rights reserved.
;;;
;;; Runtime package for programs restructured by Curare.


(provide 'runtime)


(in-package 'curare)


(export '(make-task
          server obtain-servers initialize-servers terminate-servers
          find-first

          make-process
          acquire-lock release-lock make-lock
          make-mailbox send receive mb-barrier mb-count
          make-counter wait-counter decf-counter))


(use-package '())


(defvar *runtime-version* "$Revision: 1.6 $")
```

```
;;; This file contains the runtime system for programs restructured by Curare.
;;; It contains a section for each parallel Lisp, which implements the SPUR
;;; parallel features; a section containing system-independent; and some test
;;; cases.

;;; Options:
;;;
;;; #+conservative -- Makes the allocator conservative about handing out
;;; processes.
;;;
;;; #+debug -- Enable debugging printing of values.
;;;
;;; #+spawn-call -- Spawn the recursive calls concurrently.
;;;
;;; #+spawn-head -- Spawn the recursive function's body concurrently.
;;;
;;; #+struct-task -- Tasks are built from structures
;;;
;;; #+closure-task -- Tasks are built from closures


(eval-when (compile load eval)
  (pushnew :spawn-call *features*))


;;; Optimize the code!

(proclaim '(optimize (speed 3) (safety 0) (space 0)))
```

```
;;; Code for Lucid Qlisp.

#+(and lucid qlisp)
(progn

;;; Closures are much faster than structures in Lucid CL.

(eval-when (compile load eval) (pushnew :closure-task *features*))


(when (eql sys:*processes-in-use* 1)
  (sys:change-memory-management :expand 50 :growth-limit 400)
  (sys:change-process-stack-size (* 50 1024)))


(defmacro record-history (&rest args) `(lucid::record-history ,@args))


(defun available-processors () sys:*number-of-processors*)


(defmacro def-global-var (&body body) `(sys:defglobalvar ,@body))


;;; Processes:

(defmacro make-process (body)
  `(sys:spawn t
    (progn
      #+debug
      (record-history "Making Process")
      ,body)))


;;; Locks:

;;; Go directly to the raw primitives.  GET-LOCK and RELEASE-LOCK are too
;;; slow.

(defmacro acquire-lock (lock) `(lucid::%get-lock ,lock 'lucid::lock-lock))


(defmacro release-lock (lock) `(setf (lucid::lock-lock ,lock) 0))


(defun make-lock (state &optional name)
  (declare (ignore name))
```

```
  (let ((lock (sys:make-lock :type :spin)))
    (if (eq state 'locked) (acquire-lock lock))
    lock))


;;; Mailboxes:

(defstruct (mailbox (:conc-name mb-) (:print-function print-mb))
  (lock (sys:make-lock :type :spin))
  (entries-head ())
  (entries-tail ())
  (barrier nil)
  (count nil))


(proclaim '(inline mb-lock mb-entries-head mb-entries-tail mb-barrier
            mb-count))


(defun mb-length (mb) (length (mb-entries-head mb)))


(defun print-mb (mb stream depth)
  (declare (ignore depth))
  (format stream "<Mailbox ~D: ~D items, count: ~D, locked: ~A, barrier: ~A>"
          (lucid::%pointer mb)
          (mb-length mb)
          (mb-count mb)
          (not (sys:check-lock (mb-lock mb)))
          (mb-barrier mb)))


(defun send (item mailbox)
  (let ((entry (cons item nil)))        ; Allocate out of critcal section
    (acquire-lock (mb-lock mailbox))
    (cond ((null (mb-entries-head mailbox)) ; Empty if head -> NIL
           (setf (mb-entries-head mailbox)
                 (setf (mb-entries-tail mailbox) entry)))
          (t
           (setf (mb-entries-tail mailbox)
                 (setf (cdr (mb-entries-tail mailbox)) entry))))
    (release-lock (mb-lock mailbox))))


(defun receive (mailbox)
  (cond ((null (mb-entries-head mailbox))
         (sys:check-need-to-gc)
         (receive mailbox))
```

```
        (t
         ;; Mailbox had something in it.  Get lock and check again.
         (acquire-lock (mb-lock mailbox))
         (cond ((null (mb-entries-head mailbox))
                (release-lock (mb-lock mailbox))
                ;; Wait for something to be enqueued.
                (receive mailbox))

               (t
                ;; Got lock on a full mailbox.  Remove and return first item.
                (let ((value (pop (mb-entries-head mailbox))))
                  (release-lock (mb-lock mailbox))
                  value))))))


;;; Counters:
;;; Can't use events since signals before wait are ignored.

(defun make-counter (n) (cons (sys::make-lock :type :spin) n))


;;; Raw access to counter value.

(defmacro decf-counter* (counter) '(decf (the fixnum (cdr ,counter))))


(defun decf-counter (counter)
  (acquire-lock (car counter))
  (prog1
      (decf-counter* counter)
    (release-lock (car counter))))


(defun clear-counter (counter) (setf (cdr counter) 0))


(defun wait-counter (counter)
  (sys:check-need-to-gc)
  (if (> (the fixnum (cdr counter)) 0)
      (wait-counter counter)))


;;; Use history mechanism for debugging.

(defun replay-history (&optional (stream t))
  (dolist (event (lucid::reset-history-queue))
    (format stream "Process: ~A,  Processor: ~A  ~D~%  ~?~%~%"
```

```
            (lucid::history-process-id event)
            (lucid::history-processor event)
            (lucid::history-timestamp event)
            (lucid::history-format-string event)
            (lucid::history-args event))))


;;; To invoke and time a test.

(defmacro test (n &rest form)
  `(sys:qtime
    (progn
      #+debug
      (lucid::reset-history-queue)
      (initialize-servers ,n)
      (multiple-value-prog1
          ,@form
        (terminate-servers *idle-server-queue*)))))

);; End of Lucid Qlisp
```

```
;;; System-independent code.


;;; A task is a function and its actual arguments.  It is either represented
;;; as a structure or a closure.

#+struct-task
(defstruct (task-object
             (:conc-name task-)
             (:constructor make-task-object (function arguments)))
  function
  arguments)


;;; Enqueue a function and its arguments on a queue.

#+struct-task
(defmacro make-task (queue function &rest arguments)
  `(send (make-task-object ,function ,arguments) ,queue))


;;; Invoke the object from the queue.

#+struct-task
(defmacro invoke-task (task)
  `(apply (task-function task) (task-arguments task)))


#+closure-task
(defmacro make-task (queue function &rest arguments)
  `(send #'(lambda () (funcall ,function ,@arguments)) ,queue))


#+closure-task
(defun task-object-p (obj) (compiled-function-p obj)) ; Better predicate???


#+closure-task
(defmacro invoke-task (task) `(funcall ,task))


;;; List of processes running the servers.

(def-global-var *server-processes* nil)


;;; Idle servers look for work on this queue.  The lock protects the queue, so
```

```
;;; that only one process grabs servers at once.

(def-global-var *server-queue-lock* (make-lock 'unlocked))

(def-global-var *idle-server-queue* (make-mailbox))

(def-global-var *number-of-idle-servers* 0)

(proclaim '(fixnum *number-of-idle-servers*))


;;; A server repeatedly: dequeues a task, applies a function to its arguments,
;;; and looks for more work.  A server quits if passed anything but a task.

(defun server (queue)
  (let ((task (receive queue)))
    (cond ((task-object-p task)
           #+debug
           (record-history "Queue ~A -> task ~A" queue task)
           (invoke-task task)
           #+debug
           (record-history "Task ~A (~A) done" task queue)
           #+allegro
           (mp:process-allow-schedule)
           #+qlisp
           (sys:check-need-to-gc)
           (server queue))
          (t
           #+debug
           (record-history "Quiting (~A) ~A" task queue)))))


;;; This function is executed by a process from the IDLE-SERVER-QUEUE.  It
;;; invokes a server and maintains to bookeeping information for that queue.

(defun idle-server (queue)
  (server queue)
  (acquire-lock *server-queue-lock*)     ; And more processes are idle
  ;; Use raw form to save additional lock/unlock:
  (decf-counter* (mb-barrier queue))    ; One fewer process serving queue
  (incf *number-of-idle-servers*)
  (release-lock *server-queue-lock*))


;;; Obtain between 1 and the desired number of servers.  Create and return a
;;; new queue on which all of the servers are waiting.  The actual number of
;;; servers depends on the number of free processes.
```

```
(defun obtain-servers (&optional desired-number)
  (acquire-lock *server-queue-lock*)
  (let ((n (number-of-servers desired-number)))
    (decf *number-of-idle-servers* n)
    (release-lock *server-queue-lock*)

    (let ((queue (make-mailbox)))
      #+debug
      (record-history "Obtain ~D (asked for ~A) servers" n desired-number)
      (setf (mb-barrier queue) (make-counter n))
      (setf (mb-count queue) (1+ n))    ; Additional for sync process
      (obtain-n-servers n queue)
      queue)))


;;; Assign N tasks from the idle server pool to run servers waiting on a given
;;; queue.

(defun obtain-n-servers (n queue)
  (declare (fixnum n))
  (cond ((= n 0))
        (t
         (make-task *idle-server-queue* #'idle-server queue)
         (obtain-n-servers (- n 1) queue))))


;;; Initialize N processes run the server task on the idle server queue.

(defun initialize-servers (number-of-processes)
  (setq *idle-server-queue* (make-mailbox))
  (setq *number-of-idle-servers* number-of-processes)
  (setq *server-processes* (initialize-n-servers number-of-processes))
  (setf (mb-barrier *idle-server-queue*) (make-counter number-of-processes))
  (setf (mb-count *idle-server-queue*) number-of-processes))


(defun initialize-n-servers (n)
  (declare (fixnum n))
  (cond ((= n 0) ())
        (t
         (cons (make-process (server *idle-server-queue*))
               (initialize-n-servers (- n 1))))))


;;; Terminate all servers for a given queue.  The server return to the idle
;;; queue.

(defun terminate-servers (queue)
```

```
  #+debug
  (record-history "Terminate servers for ~A" queue)
  (terminate-n-servers (mb-count queue) queue))


;;; Terminate the N servers for a queue by enqueuing a task that decrement the
;;; queue's counting counter then enqueues a "killer" value onto the queue.

(defun terminate-n-servers (n queue)
  (declare (fixnum n))
  (cond ((= n 0))
        (t
         (send nil queue)                ; The last task
         (terminate-n-servers (- n 1) queue))))


;;; Find the minimum non-NIL value in a heap.  This function is called when a
;;; value is added to the heap.  The value for slot I+N can either be I,
;;; indicating the clause I returned non-NIL; or N, indicating that clause I
;;; returned NIL.  When clause I is the first non-NIL value, then return I.
;;; Otherwise, return NIL.

(defun find-first (heap i value n)
  (declare (simple-vector heap)
           (fixnum i value n))
  (setf (svref heap i) value)
  (cond ((= i 1)
         value)
        ((evenp i)    ; Left child
         (let ((right-value (svref heap (+ i 1))))
           (if right-value
               (find-first heap (floor i 2) (min value right-value) n)
               (if (= value n)
                   nil
                   (find-first heap (floor i 2) value n)))))
        (t            ; Right child
         (let ((left-value (svref heap (- i 1))))
           (if (and left-value (<= value left-value))
               (find-first heap (floor i 2) value n)
               nil)))))
```

```
;;; Framework for testing runtime system.


;;; Return the number of servers to actually use, given a desirable number.

(defun number-of-servers (desired-number)
  (declare (fixnum desired-number *number-of-idle-servers*))
  (cond ((null desired-number)
         #+conservative (the fixnum (ceiling *number-of-idle-servers* 2))
         #-conservative *number-of-idle-servers*)
        ((<= desired-number *number-of-idle-servers*)
         desired-number)
        (t
         #+conservative (the fixnum (ceiling *number-of-idle-servers* 2))
         #-conservative *number-of-idle-servers*)))


;;; Don't optimize test code.

(proclaim '(optimize (speed 1) (safety 1)))


;;; Write a header for a test to a stream.

(defun display-test-header (stream)
  (multiple-value-bind (second minute hour date month year)
      (get-decoded-time)

    (format stream "Date: ~D/~D/~D ~2,'0D:~2,'0D:~2,'0D~%"
            date month year hour minute second)
    (format stream "~A, ~A on ~A~%"
            (lisp-implementation-type)
            (lisp-implementation-version)
            (machine-instance))
    (format stream "Runtime ~A~%~%" *runtime-version*)))


;;; Body of a test routine.  Test the code sequentially and in parallel with
;;; varying numbers of processors.  Run the tests the specified number of
;;; times.

(defmacro test-body (stream times &key sequential parallel (gc-each nil))
  `(progn
    (if ,gc-each (sys::gc))
    (format ,stream "Sequential: (x ~D)" ,times)
    (time (dotimes (i ,times) ,sequential))
    (format ,stream "~%~%")
```

```
    (dotimes (num-proc (available-processors)) ; Plus one running this code
      (if ,gc-each (sys::gc))
      (format ,stream "1 + ~D Processors: (x ~D)" num-proc ,times)
      (test num-proc (dotimes (i ,times) ,parallel))
      (format ,stream "~%~%"))))


;;; Measure the cost of the test framework.

(defun test-identity (&key (stream t)    ; Stream for output
                           (times 1))    ; Runs of test
  (display-test-header stream)

  (test-body stream
             times
             :sequential (identity 1)
             :parallel (identity 1)))


;;; Make and return a list with N elements.  If the value of the elements is
;;; not specified, the list contains the integers from 1...N.

(defun make-n-list (n &optional value)
  (if (= n 0)
      nil
      (cons (or value n)
            (make-n-list (1- n) value))))


;;; The fibonnaci function.

(defun fib (i)
  (cond ((eql i 0) 1)
        ((eql i 1) 1)
        (t (+ (fib (- i 1)) (fib (- i 2))))))


;;; The factorial function.

(defun fact (i)
  (if (eql i 0)
      1
      (* (fact (- i 1)) i)))


(defun n-time-it (times func arg &key (stream t))
  (format stream "~%~%~D iterations of empty loop~%" times)
```

```
  (time (dotimes (i times) nil))
  (format stream "~%~D iterations of ~A(~A)~% times func arg)
  (time (dotimes (i times) (funcall func arg))))


(defun time-send-receive (q)
  (send 1 q)
  (receive q))


(defun time-lock-unlock (lock)
  (acquire-lock lock)
  (release-lock lock))
```

Appendix B

Source Code for Loop Timings

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: CURARE; -*-

;;; test-map.lisp
;;; Copyright (C), James R. Larus, 1989.  All rights reserved.
;;;
;;; Code to test mapping functions with Curare runtime system.


(require "runtime")

(in-package 'curare)


;;; Don't optimize test code.

(proclaim '(optimize (speed 1) (safety 1)))


;;; Use all processors as servers.

(defun number-of-servers (desired-number)
  (cond ((null desired-number)
         *number-of-idle-servers*)
        ((<= desired-number *number-of-idle-servers*)
         desired-number)
        (t
         *number-of-idle-servers*)))


;;; MAPC:

(defun mapc-sync (f lst)
```

```
  (let ((task-queue (mapc-async f lst)))
    (server task-queue)
    (wait-counter (mb-barrier task-queue)))
  nil)


(defun mapc-async (f lst)
  (let ((task-queue (obtain-servers)))
    (mapc3 task-queue f lst)
    task-queue))


#+spawn-call
(defun mapc3 (task-queue f lst)
  (cond ((null lst)
         (terminate-servers task-queue))
        (t
         (make-task task-queue #'mapc3 task-queue f (cdr lst))
         (funcall f (car lst)))))


#+spawn-head
(defun mapc3 (task-queue f lst)
  (cond ((null lst)
         (terminate-servers task-queue))
        (t
         (make-task task-queue
                    #'(lambda (f lst) (funcall f (car lst)))
                    f
                    lst)
         (mapc3 task-queue f (cdr lst)))))


;;; Measure the cost of sequential and parallel MAPC.

(defun test-mapc (length &key (stream t) ; Stream for output
                         (times 1)       ; Runs of test
                         (value nil)     ; Initial value of list
                         (func #'fact)) ; Mapped function
  (let ((lst (make-n-list length value)))
    (display-test-header stream)
    (if value
        (format stream "MAPC of ~A on ~D ... ~D (~D)~%"
                func value value length)
        (format stream "MAPC of ~A on 0 ... ~D~%" func length))

    (test-body stream
               times
```

```
:sequential (mapc func lst)
:parallel (mapc-sync func lst))))
```

```
;;; MAPCAR:

(defun mapcar-sync (f lst)
  (let* ((dest (cons nil nil))
         (task-queue (mapcar-async dest f lst)))
    (server task-queue)
    (wait-counter (mb-barrier task-queue))
    (cdr dest)))


(defun mapcar-async (dest f lst)
  (let ((task-queue (obtain-servers)))
    (mapcar-dp2 task-queue dest f lst)
    task-queue))


#+spawn-call
(defun mapcar-dp2 (task-queue dest f lst)
  (cond ((null lst)
         (setf (cdr dest) ())
         (terminate-servers task-queue))
        (t
         (let ((tmp (cons nil nil)))
           (make-task task-queue #'mapcar-dp2 task-queue tmp f (cdr lst))
           (setf (car tmp) (funcall f (car lst)))
           (setf (cdr dest) tmp)))))


#+spawn-head
(defun mapcar-dp2 (task-queue dest f lst)
  (cond ((null lst)
         (terminate-servers task-queue)
         (setf (cdr dest) ()))
        (t
         (let ((tmp (cons nil nil)))
           (make-task task-queue
                      #'(lambda (dest f lst)
                          (setf (car tmp) (funcall f (car lst)))
                          (setf (cdr dest) tmp))
                      dest
                      f
                      lst)
           (mapcar-dp2 task-queue tmp f (cdr lst))))))


;;; Test the cost of sequential and parallel MAPCAR.
```

```
(defun test-mapcar (length &key (stream t) ; Stream for output
                           (times 1)    ; Runs of test
                           (value nil)  ; Initial value of list
                           (func #'fact)) ; Mapped function
  (let ((lst (make-n-list length value)))
    (display-test-header stream)
    (if value
        (format stream "MAPCAR of ~A on ~D ... ~D (~D)~%"
                func value value length)
        (format stream "MAPCAR of ~A on 0 ... ~D~%" func length))

    (test-body stream
               times
               :sequential (mapcar func lst)
               :parallel (mapcar-sync func lst))))
```

Bibliography

[1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal of Computing*, 1(2):131–137, June 1972.

[3] Donald C. Allen, Seth A. Steinberg, and Lawrence A. Stabile. Recent Developments in Butterfly Lisp. In *Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence*, pages 2–6, July 1987.

[4] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, October 1988.

[5] John Randel Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[6] Randy Allen and Ken Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[7] Utpal Banerjee. Data Dependence in Ordinary Programs. Technical Report UIUCDCS-R-76-837, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, November 1976.

[8] John P. Banning. An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[9] Jeffrey M. Barth. A Practical Interprocedural Data Flow Analysis Algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[10] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.

[11] Philip A. Bernstein, David W. Shipman, and James B. Rothnie, Jr. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 5(1):18–51, March 1980.

[12] David Billstrom, Joesph Brandenburg, and John Teeter. CCLISP on the iPSC Concurrent Computer. In *Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence*, pages 7–12, July 1987.

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

[14] Daniel G. Bobrow and Douglas W. Clark. Compact Encodings of List Structure. *ACM Transactions on Programming Languages and Systems*, 1(2):266–286, October 1979.

[15] James M. Boyle, Kenneth W. Dritz, M. N. Muralidharan, and Robert J. Taylor. Deriving Sequential and Parallel Programs from Pure LISP Specifications by Program Transformation. In *IFIP WG2.1 Working Conference on Programme Specification and Transformation*, Amsterdam, 1986. North-Holland Publishing Company.

[16] 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.

[17] Robert Cartwright, Robert Hood, and Philip Matthews. Paths: An Abstract Alternative to Pointers. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 14–25, January 1981.

[18] Ron Cytron and Jeanne Ferrante. What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27. Dept. of Electrical Engineering, Penn State Univ., August 1987.

[19] J. Darlington and R. M. Burstall. A System which Automatically Improves Programs. *Acta Informatica*, 6(1):41–60, 1976.

[20] John R. Ellis. Bulldog: A Compiler for VLIW Architectures. Technical Report YALEU/DCS/RR-364, Yale Univ., Dept. of Computer Science, February 1985.

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[22] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[23] Richard P. Gabriel and John McCarthy. Queue-based Multi-processing LISP. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 25–44, August 1984.

[24] Kourosh Gharachorloo, Vivek Sarkar, and John L. Hennessy. A Simple and Efficient Implementation Approach for Single Assignment Languages. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 259–268, July 1988.

[25] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[26] Ron Goldman and Richard P. Gabriel. Preliminary Results with the Initial Implementation of Qlisp. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 143–152, July 1988.

[27] Ron Goldman and Richard P. Gabriel. Qlisp: Experience and New Directions. In *Proceedings of ACM/SIGPLAN PPEALS 1988 (Parallel Programming: Experience with Applications, Languages and Systems)*, pages 111–123, July 1988.

[28] Susan L. Graham and Mark Wegman. A Fast and Usually Linear Algorithm for Global Flow Analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[29] Sharon L. Gray. Using Futures to Exploit Parallelism in Lisp. Master's thesis, MIT, February 1986. MS report.

[30] Vincent A. Guarna, Jr. A Technique for Analyzing Pointer and Structure References in Parallel Restructuring Compilers. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 212–220. Pennsylvania State University Press, August 1988.

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

[32] Luddy Harrison and David A. Padua. PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 527–538, July 1988.

[33] W. Ludwell Harrison, III. Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor. Technical Report CSRD 565, Center for Supercomputing Research & Development, Univ. of Illinois at Urbana-Champaign, March 1986.

[34] 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.

[35] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence Analysis for Pointer Variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[36] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. Technical Report 690, Computer Sciences Dept., Univ. of Wisconsin—Madison, March 1987.

[37] Susan Horwitz, Jan Prins, and Thomas Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, January 1988.

[38] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46, June 1988.

[39] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. Technical Report 756, Computer Sciences Dept., Univ. of Wisconsin—Madison, March 1988.

[40] Gérard Huet and Bernard Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica*, 11(1):31–55, 1978.

[41] Neil D. Jones and Steven S. Muchnick. Flow Analysis and Optimization of Lisp-Like Structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, 1981.

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

[43] Ken Kennedy. A Comparison of Two Algorithms for Global Data Flow Analysis. *SIAM Journal of Computing*, 5(1):158–180, March 1976.

[44] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1973.

[45] Peter M. Kogge. Parallel Solution of Recurrence Problems. *IBM Journal of Research and Development*, 18(2):138–148, March 1974.

[46] Peter M. Kogge and Harold S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[47] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233, June 1986.

[48] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The Power of Parallel Prefix. *IEEE Transactions on Computers*, C-34(10):965–968, October 1985.

[49] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[50] David J. Kuck, Yaichi Muraoka, and Shyh-Ching Chen. On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[51] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek. Report on the Programming Language Euclid. *ACM SIGPLAN Notices*, 12(2):1–79, February 1977.

[52] James R. Larus and Paul N. Hilfinger. Detecting Conflicts Between Structure Accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[53] James R. Larus and Paul N. Hilfinger. Restructuring Lisp Programs for Concurrent Execution. In *Proceedings of ACM/SIGPLAN PPEALS 1988 (Parallel Programming: Experience with Applications, Languages and Systems)*, pages 100–110, July 1988.

[54] Zhiyuan Li and Pen-Chung Yew. Efficient Interprocedural Analysis for Program Parallelization and Restructuring. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–99, June 1988.

[55] John M. Lucassen. Types and Effects: Towards the Integration of Functional and Imperative Programming. Technical Report MIT/LCS/TR-408, MIT Laboratory for Computer Science, August 1987. Ph.D. Dissertation.

[56] Samuel P. Midkiff and David A. Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[57] Samuel Pratt Midkiff. Automatic Generation of Synchronization Instruction for Parallel Processors. Technical Report CSRD Rpt. No. 588, Center for Supercomputing Research & Development, Univ. of Illinois at Urbana-Champaign, May 1986.

[58] James S. Miller. MultiScheme: A Parallel Processing System Based on MIT Scheme. Technical Report MIT/LCS/TR-402, MIT Laboratory for Computer Science, September 1987. PhD thesis.

[59] David A Moon. Electronic Mail Message to Common Lisp Mailing List, May 1988.

[60] Eugene W. Myers. A Precise Inter-Procedural Data Flow Algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

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

[62] David L. Presberg and Neil W. Johnson. The Paralyzer: Ivtran's Parallelism Analyzer and Synthesizer. In *Proceedings of a Conference on Programming Languages and Compilers for Parallel and Vector Machines*, pages 9–16, March 1975. Published in SIGPLAN Notices, V. 10, N. 3, March 1975.

[63] John C. Reynolds. Automatic Computation of Data Set Definitions. In *Information Processing 68*, pages 456–461, Amsterdam, Holland, 1969. IFIP, North-Holland Publishing Company.

[64] Cristina Ruggieri and Thomas P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[65] Jacob T. Schwartz. *On Programming: An Interim Report on the SETL Project.* Computer Science Department, Courant Institute of Mathematical Science, New York University, June 1975.

[66] Olin Shivers. Control Flow Analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[67] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[68] Guy Lewis Steele Jr. and Gerald Jay Sussman. LAMBDA: The Ultimate Imperative. Technical Report AIM 353, MIT AI Laboratory, March 1976.

[69] Mark R. Swanson, Robert R. Kessler, and Gary Lindstrom. An Implementation of Portable Standard Lisp on the BBN Butterfly. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 132–142, July 1988.

[70] Robert Endre Tarjan. A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. *Journal of Computer and System Sciences*, 18(2):110–127, April 1979.

[71] Robert Endre Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, July 1981.

[72] Robert H. Thomas and Will Crowther. The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 245–254. Pennsylvania State University Press, August 1988.

[73] Pete Tinker and Morry Katz. Parallel Execution of Sequential Scheme with ParaTran. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 28–39, July 1988.

[74] Rémi Triolet, François Irigoin, and Paul Feautier. Direct Parallelization of Call Statements. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 176–185, June 1986.

[75] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software Practice & Experience*, 9(1):31–49, 1979.

[76] Philip Lee Wadler. Listlessness is Better than Laziness: An Algorithm that Transforms Applicative Programs to Eliminate Intermediate Lists. Technical Report CMU-CS-85-171, Dept. of Computer Science, Carnegie Mellon Univ., August 1984. PhD thesis.

[77] 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. PhD thesis.

[78] Benjamin Zorn, Kinson Ho, James Larus, Luigi Semenzato, and Paul Hilfinger. Lisp Extensions for Multiprocessing. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, pages 761–770, January 1989.

Index