

PARCEL and MIPRAC: Parallelizers for Symbolic and Numeric Programs*

Williams Ludwell Harrison III and Zahira Ammarguella

May 7, 1992

1 Introduction

PARCEL is a compiler and run-time system that automatically parallelizes a Scheme program for execution on a shared-memory multiprocessor; it is described at length in [Har89]. PARCEL was a success in several respects: its interprocedural analysis makes it capable of extracting high-level parallelism from complex applications; it introduced a number of program transformations that are especially useful in parallelizing symbolic computations; and its run-time system makes good use of multiple procedure versions to achieve parallelism without undue inefficiency. However, PARCEL suffers from a software engineering defect common to parallelizing compilers: its intermediate form is complex, and is specific to the source language it accepts, making it impossible to adapt the compiler to a new source language, and difficult to add new passes. This defect, along with the desire to extend the power of PARCEL's interprocedural analysis, and to apply the techniques of PARCEL to a broad class of programming languages (rather than to a single source language), have motivated the design of MIPRAC. MIPRAC is an interprocedural, parallelizing compiler that operates upon a semantically-oriented intermediate form. It attempts to fuse the techniques of PARCEL (for symbolic programs) with those developed for Fortran (for numeric programs) in a single compilation system. Programs from a variety of source languages, including Scheme, Common Lisp, C, and Fortran have a straightforward translation to MIPRAC's intermediate form. Moreover, the intermediate form obeys a number of simple theorems that make it easy to analyze and transform. In this paper we will describe the strengths and weak-

*This work was supported in part by the **National Science Foundation** under Grant No. NSF MIP-8410110, the **U.S. Department of Energy** under Grant No. DE-FG02-85ER25001, the **Office of Naval Research** under Grant No. ONR N00014-88-K-0686, the **U.S. Air Force Office of Scientific Research** under Grant No. AFOSR-F49620-86-C-0136, and by a donation from the **IBM Corporation**.

nesses of PARCEL, and outline the design of MIPRAC in light of our experience with PARCEL.

2 PARCEL

The input to the PARCEL compiler is a sequential Scheme program, and its output is an object code for the Alliant FX/8 or Cedar multiprocessor. The compiler operates in four phases: interprocedural analysis, standard transformations, parallelizing transformations, and code generation. From every procedure of the input program, PARCEL produces two code versions, one sequential, one parallel. The PARCEL run-time system selects between these versions during execution. The parallel versions are executed when the processors of the target machine are underutilized, and the sequential versions are used when the machine is saturated.

2.1 Strengths

2.1.1 Interprocedural analysis

The technology of automatic parallelization originated with vectorizing Fortran compilers, which were concerned principally with innermost loops, or with loops nested but a few levels deep, and where the nesting was static (that is, where the nesting did not arise because of procedure calling). Interprocedural analysis, if performed at all, was ordinarily done to strengthen dependence analysis of loops, to infer aliasing among parameters, and to fold constants across procedure boundaries, but seldom with the goal of extracting high-level parallelism from a Fortran program. (There are exceptions; see [Tri84, BC86].) For this reason, such compilers are typically quite capable of extracting fine-grained parallelism, but do less well in extracting large-grained, high-level parallelism.

A typical Lisp program contains many more, and much smaller, procedures than its Fortran counterpart, and while it is sensible to extract parallelism from innermost iterative or recursive constructs, it is also apparent that in many an application, the richest source of parallelism is not the innermost level, but intermediate or even outermost layers in the program's control flow. In the presence of side-effects (and both the Scheme and Common Lisp standards permit every variety of side-effect, both upon variables and upon compound data), to extract such parallelism requires an understanding of the interprocedural visibility of side-effects. That is, given a procedure **f** that directly accesses (uses or modifies) a memory location, we must ascertain to which procedures that call **f** this access is visible, in order to correctly construct dependence graphs of those calling procedures, so that their control-flow structures can be parallelized.

In PARCEL, this is accomplished by an abstract interpretation that monitors the interprocedural movements made by objects, from their points of creation,

```

(define list-of-sums (lambda (l)
  (if (null l) '()
      (cons (sum (car l) (counter 0))
            (list-of-sums (cdr l))))))
(define sum (lambda (l ctr)
  (if (null l) 0
      (begin (sum (cdr l) ctr)
              (ctr)))))
(define counter (lambda (x)
  (lambda () (set! x (1+ x)) x)))

```

Figure 1: An example of PARCEL’s interprocedural analysis

to the points at which they are used and modified. On the basis of these movements, the analysis reveals to which procedures every (direct) side-effect within a program is visible. In essence, the observation applied by this analysis is the following: the visibility of a side-effect is restricted to the subtree of computation that contains the lifetime of the object affected. (The interested reader will find this analysis described and proven correct in [Har89].) From this information, PARCEL constructs dependence graphs of individual procedures, and thereafter parallelizes their control-flow structures regardless of whether they are “outermost” or “innermost” procedures (indeed, such a characterization may be meaningless, as a procedure may be invoked in many different contexts, as well as recursively).

PARCEL’s analysis is also used in memory management. In the simplest application, the analysis decides for every dynamically instantiated object (lexical variable, cons cell, etc.) whether the object can be stack-allocated, or whether it must be heap-allocated. This lifetime information can also be used to guide the placement of objects in the hierarchical shared memory of a machine like Cedar. These ideas are developed in detail in [Har89].

As an example of this analysis, consider the program in Figure 1. The procedure `counter` returns an object (a closure that increments a free variable `x` and returns its updated value). The analysis reveals that the procedure `sum` has a side-effect upon (an instance of) `x` as result of using such a counter. However, the analysis also reveals that `list-of-sums` has no such side-effect, despite that it calls `sum`. Therefore, there is no dependence between the expressions

(sum (car l) (counter))

and

(list-of-sums (cdr l))

even though the former has a side-effect. Therefore, the compiler is justified in rewriting the procedure `list-of-sums` as in Figure 2. The construct `plet` is

```

(define list-of-sums (lambda (l)
  (if (null l) '()
      (plet ((a (sum (car l) (counter)))
            (b (list-of-sums (cdr l))))
            (cons a b))))))

```

Figure 2: A parallelized version of `list-of-sums`

identical to `let`, except that it evaluates in parallel the forms `(sum (car l) (counter))` and `(list-of-sums (cdr l))`, rather than sequentially as `let` would do.

2.1.2 Automatic parallelization of control structures

Just as the role of interprocedural analysis in vectorizing compilers was dictated by the objective of those compilers (generate vector code from innermost loops) and their input programs (loop-laden numeric code), so were the types of transformations they performed dictated by these considerations. Consequently, parallelizing compilation has been largely a synonym for loop transformations; to name a few, we have loop distribution, loop fusion, loop interchange, loop unrolling, loop blocking, etc. And for good reason: if the iterations of a substantial loop are independent, an enormous source of parallelism is latent.

These loop transformations were much the easier because of the regular structure of Fortran’s `DO` loop, where the number of iterations is computed prior to entrance to the loop, and is not infrequently a constant. Lisp programs differ markedly from Fortran ones in this respect: there is no iterative construct so simple in Lisp as Fortran’s `DO`; rather, a typical iterative computation is expressed using tail-recursion, or a generalized iterative facility that resembles a `WHILE` or `REPEAT` loop. The number of iterations of such a construction is not available prior to its entrance, but rather the termination condition is computed, along with other quantities, during the iterations themselves.

Indeed, many Lisp computations are not iterative at all, but are described by recursive procedures (not tail-recursive ones). These must be, to a parallelizing Lisp compiler, what `DO` loops are to a parallelizing Fortran compiler, if the principal parallelism within applications is to be discovered automatically.

For these reasons, `PARCEL` makes use of two transformations called “exit-loop parallelization” and “recursion splitting.” The former parallelizes iterative structures (control-flow cycles with exits) that may have no simple translation to `DO` loop form; the latter translates a self-recursive procedure into two loops, the first of which performs the “downward” portion of the procedure (that portion of the procedure body between the entrance to the procedure and the self-recursive calls), the second of which performs the “upward” portion (between the return from the self-recursive calls and the exits from the procedure). These loops are, in turn, parallelized by exit-loop parallelization and other loop parallelization

techniques.

Exit-loop parallelization and recursion splitting require that the compiler be able to recognize the recurrence relations that govern the exit of an iterative or recursive computation, and to transform such a recurrence into parallel code. For this reason, PARCEL is equipped to recognize and solve a variety of symbolic and numeric recurrences, including recurrences over s-expressions.

2.1.3 Run-time support for automatically parallelized codes

An object code produced by the PARCEL compiler has directives for parallelism (parallel loops and calls to parallel recurrence solvers), and invocations of Scheme's intrinsic procedures (`car`, `cons`, `call/cc`, etc.), in addition to assembly language instructions. It is linked to the PARCEL run-time system to produce an executable object.

As mentioned above, the object code contains two versions of every compiled procedure, one sequential and one parallel. These versions have quite different purposes, and therefore result from quite different restructuring processes. The sequential version creates no additional threads of activity, but rather is intended to complete its computation as quickly as possible on a single processor, with a minimum of disturbance to the shared memory of the computer. The parallel version is intended to create additional threads of activity, and makes less efficient use of memory than the sequential version, in order to do so (see the example below). As the program executes, parallel threads of activity are created, and these in turn may create parallel threads of activity, and so forth. When the nesting of these parallel threads is judged to be sufficient to saturate the machine (this judgement is made according to one of several experimental strategies; see [Cho90]), the sequential versions of procedures are invoked, so that no further creation of parallel activity will occur beyond that nesting depth.

A traditional microtasking environment [Cra82, EHJP90] associates a stack with every processor. When a processor initiates a parallel loop, the continuation of that loop is on its stack; it must wait for all iterations of the initiated loop to terminate, before it executes the continuation.

We observed, however, first that the parallel loops in PARCEL's object codes often have very few iterations (2 or 3 being common), and second that the running times of these iterations may vary radically one from another. The latter is likely precisely because PARCEL often creates parallel loops that are quite "high" in a program's calling graph, so that the computation that is performed in each iteration may entail many procedure calls and conditional branches. To address this irregularity in the execution time of its parallel loops, we broke the traditional binding between processors and stacks. The PARCEL run-time system has more stacks than processors. When the processor that initiates a parallel loop finds that others are still executing iterations of the loop (but that there are no further iterations to begin), it relinquishes its stack,

```

(define copy-integer
  (lambda (n) (copy-integer-aux 1 n)))
(define copy-integer-aux
  (lambda (i s)
    (if (>= i s)
        i
        (copy-integer-aux (1+ i) s))))
(define copy
  (lambda (x)
    (if (atom? x) (if (integer? x) (copy-integer x) nil)
        (cons (copy (car x)) (copy (cdr x))))))
(write (copy (read)))

```

Figure 3: The procedure `copy` and its auxiliary procedures

allocates another, and turns to the microtask queue to find available work. The last processor to finish work on the parallel loop will return its stack to the pool, seize the stack abandoned by the initiating processor, and execute the continuation of the loop. This scheme is described in [Cho90].

As an example of the transformations performed in PARCEL, consider the program in Figure 3. The procedure `copy` is defined, to copy a tree of `cons` cells. When the procedure finds an integer at a leaf of the tree, it copies it by calling `copy-integer`, which counts from 1 until an integer is reached that is greater than or equal to the leaf. Else, when the procedure finds an atomic leaf that is not an integer, it returns nil, and finally when it finds a `cons` cell it recursively copies the `car` and `cdr` of the cell into a fresh cell.

From this definition of `copy`, PARCEL arrives by a sequence of transformations to the version in Figure 4. The syntax of this figure requires explanation. First, the list of variables

```
<i t-33 t-36 t-37 t-38 t-39 t-40 t-41 t-42>
```

are the local variables bound by the procedure. They are uninitialized until assigned. The syntax

```

(exit-block EXPR
  (l1: EXPR1)
  (l2: EXPR2)
  ...
  (ln: EXPRn))

```

denotes a loop that runs until it is exited by a branch (`go` form) to one of the labels `l1:` to `ln:`. The symbol `#self-closure#` denotes the innermost closure (procedure + environment) in which the symbol appears. A `return` form exits the innermost procedure that contains the form, and provides its argument as the return value of the procedure.

```

(lambda nil
  <copy t-43 t-44 t-45>
  (set!
    copy
    (lambda (x)
      <i t-33 t-36 t-37 t-38 t-39 t-40 t-41 t-42>
      (set! t-37 (atom? x))
      (cond
        ( t-37
          (set! t-38 (integer? x))
          (cond
            ( t-38
              (set! i 1)
              (exit-block (repeat (set! t-33 (>= i x))
                (if t-33 (go 1-210:)
                  (set! i (1+ i))))
                (1-210: (set! t-36 i))) )
            ( else
              (set! t-36 #f) ) ) )
          ( else
            (set! t-41 (car x))
            (set! t-39 (#self-closure# t-41))
            (set! t-42 (cdr x))
            (set! t-40 (#self-closure# t-42))
            (set! t-36 (cons t-39 t-40)) ) )
          (return t-36) ))
      (set! t-45 (read))
      (set! t-44 (copy t-45))
      (set! t-43 (write t-44))
      (return t-43) )

```

Figure 4: The sequential version of copy

```

(lambda nil
  <copy t-43 t-44 t-45>
  (set!
    copy
    (lambda (x)
      <i t-33 t-36 t-37 t-38 t-39 t-40 t-41 t-42 t-47
        i-48 i-49 i-50 i-51 i-52 t-53 t-54 i-55 t-56>
      (set! t-47 (length x))
      (set! x (allocate x t-47))
      (set! t-39 (allocate #f t-47))
      (set! t-41 (allocate #f t-47))
      (do (i-51 t-47) (set! x[i-51.1] (cdr x[i-51.0])))
      (doall (i-49 t-47)
        (set! t-41[i-49.1] (car x[i-49.0]))
        (set! t-39[i-49.1] (#self-closure# t-41[i-49.1])))
      (set! x (restore x t-47))
      (set! t-38 (integer? x))
      (cond
        ( t-38
          (set! i 1)
          (exit-block (repeat (set! t-33 (>= i x))
            (if t-33 (go l-210:)
              (set! i (1+ i))))
            (l-210: (set! t-36 i))) )
        ( else
          (set! t-36 #f) ) )
      (set! t-56 (allocate-r #f t-47))
      (do (i-55 t-47)
        (set! t-56[i-55.1] (cons t-39[[i-55.1]] t-56[i-55.0])))
      (set! t-56 (restore-r t-56 t-47))
      (set! t-36 (append2 t-56 t-36))
      (return t-36) ))
    (set! t-45 (read))
    (set! t-44 (copy t-45))
    (set! t-43 (write t-44))
    (return t-43) )

```

Figure 5: copy after Recursion Splitting

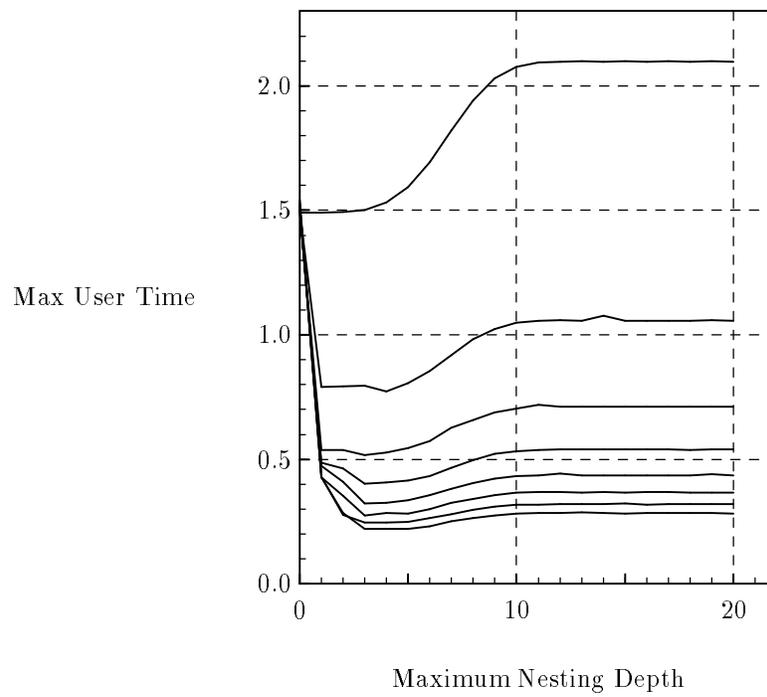


Figure 6: Execution profile for copying a tree of 10000 cons cells

boyer: max user time vs max nesting depth (number of active processors)

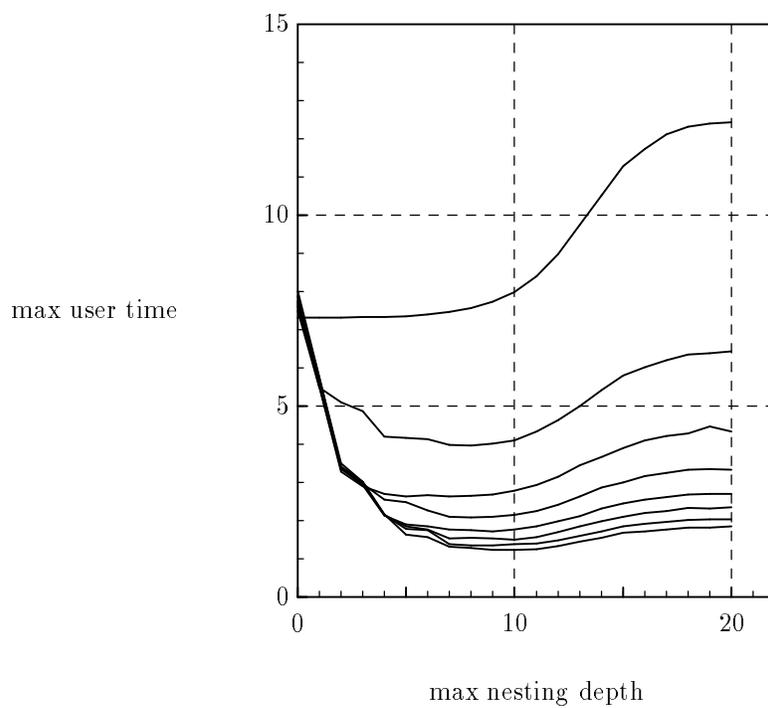


Figure 7: Running Time of BOYER

boyer: speedup vs max nesting depth (number of active processors)

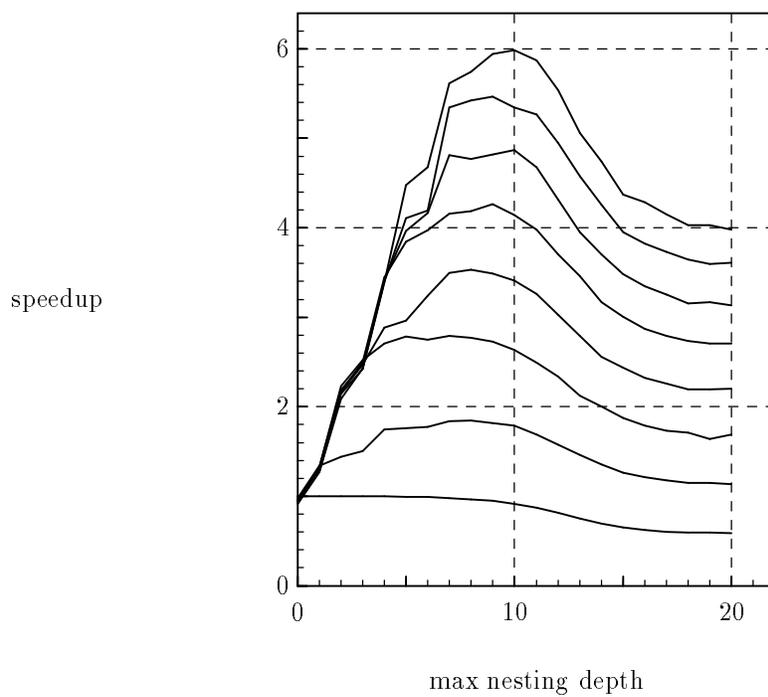


Figure 8: Speedup of BOYER

We can see from this figure that all of the procedures from Figure 3 are merged into a single procedure body. Also note that the values of all subexpressions are captured in temporary variables; this gives the compiler maximum flexibility in code motion, common subexpression elimination, and so on.

In Figure 5 we have illustrated the program after a single transformation, recursion splitting, has been performed on the program of Figure 4. This transformation breaks the definition of `copy` into two halves. The first half contains all of the code from the entrance to the procedure, until the recursive call

```
(set! t-40 (#self-closure# t-42)),
```

and the second half contains all the code from this recursive call to the exit of the procedure. The first `do` and `doall` loops of the figure perform the *downward* portion of the recursion through this call site, and the final `do` loop of the figure performs the *upward*, unwinding portion of the recursion. The procedures `allocate` and `allocate-r` create vectors on the stack, and the procedures `restore` and `restore-r` return the value stored in the last position of one of these vectors. The notation `t-41[i-49.0]` is used to access the (i-49)'th position of the vector pointed to by `t-41`, and the notation `t-41[i-49.1]` is used to access the (i-49 + 1)'th position of the vector pointed to by `t-41`. The notation `t-39[[i-55.0]]` is used to access the (i-55)'th position *from the end* of the vector pointed to by `t-56`. It can therefore be seen that iteration space of the final `do` loop of Figure 5 runs, intuitively, in the opposite direction of the first `do` and `doall` loops of the figure.

From the version of `copy` in Figure 5, PARCEL will further parallelize the inner loop (the `exit-block`), and the recurrence relations described by the two `do` loops. The PARCEL run-time system is presented with both a sequential version of the procedure (near to that in Figure 4) and a parallel version. According to the degree of parallelism at any point during execution, the run-time system selects either the sequential or parallel version to be executed, when the procedure is called.

In Figure 6, we have presented an execution profile of this example program, when copying a tree of about 10000 cons cells, where each leaf is an integer with an (average) value of 50. The program was executed on an eight-processor Alliant FX/8; there is one curve in the figure for each value of the number of active processors, from one to eight. (When two processors are active, the others are in the idle state, not accessing the memory of the machine.) On the horizontal axis is shown the nesting depth of parallelism that is achieved before the sequential version of `copy` is invoked. If, for example, the nesting depth is 10, then the `doall` loop of Figure 5 is nested (by recursive procedure calls) ten levels deep, before the sequential version of `copy` is invoked. On the vertical axis is given the user cpu time (the maximum of the user cpu times over all of the processors of the machine). The program achieves a maximum speedup of around 6.5, at a nesting depth of about 4, with 8 processors active. We are experimenting with a number of strategies for automatically selecting between

the parallel and sequential procedure versions during execution; the results of these experiments are reported in [Cho90].

In Figure 7, a similar execution profile is given for the BOYER benchmark [Gab85], a simple rewriting system designed as a benchmark for prediciting the performance of the Boyer-Moore theorem prover. In Figure 8, the same profile is shown, except that the speedup, rather than the cpu time, is plotted on the vertical axis. Speedup in this case is the ratio of the running time of the sequential version of BOYER (a nesting depth of 0) running on one processor, to the running time of the program for a given nesting depth (horizontal axis) and number of active processors (one of the eight curves).

2.2 Shortcomings

While PARCEL was successful in a number of respects, including those mentioned above, it also has a number of shortcomings, in its interprocedural analysis, its parallelizing transformations, and its run-time system.

2.2.1 Interprocedural analysis

Intrinsic procedures Scheme has a fairly large collection of built-in (intrinsic) procedures, including `car`, `cons`, `putprop`, `equal?`, etc. Some of these have side-effects, and virtually all are restricted in the types of objects they accept or return. The properties of an intrinsic procedure are important in several ways during automatic parallelization and compilation. First, during interprocedural analysis, we must take into account any side-effects performed by the built-in procedure for the sake of correctness. Second, any type restrictions upon parameters or return value can be used to sharpen the analysis. Third, we would like to generate in-line code from an invocation of an intrinsic procedure, whenever such an invocation is discovered by the interprocedural analysis (such an analysis is necessary to discover that, for example, the expression `(foo x)` is really an invocation of the procedure `car`, and whereas the expression `(cdr x)` is *not* an invocation of the intrinsic procedure `cdr`, because the program has modified the global variable `cdr`).

In PARCEL, special knowledge concerning each built-in procedure was built into the interprocedural analysis. This made the analysis sharp: the types accepted and returned by each intrinsic procedure, as well as any side-effects it might perform, were written directly into the analysis. This unfortunately meant that many thousands of lines of code were dedicated to the treatment of intrinsic procedures, and therefore that the analysis could not be retargeted to another input language without rewriting most of the analysis, despite that the analysis is essentially language-independent. Thus, for example, PARCEL cannot be easily adapted for the interprocedural analysis of Common Lisp programs, much less of C or SmallTalk programs.

```

(define house (lambda (color owner)
  (lambda (x y)
    (cond ((eq? x 'color!) (set! color y))
          ((eq? x 'owner!) (set! owner y))
          ((eq? x 'color) color)
          ((eq? x 'owner) owner))))))
(define house-color (lambda (x) (x 'color #f)))
(define house-color! (lambda (x y) (x 'color! y)))
(define house-owner (lambda (x) (x 'owner #f)))
(define house-owner! (lambda (x y) (x 'owner! y)))

```

Figure 9: The structure `house` represented using closures

This difficulty would have been largely alleviated if the intrinsic procedures had been written in Scheme itself, where possible, and presented to the compiler in this form, along with the user-defined procedures. To do so, however, would have reduced the sharpness of the interprocedural analysis slightly (not a major problem). More significantly, a number of the intrinsic procedures have no straightforward representation in Scheme. These include, for example, `putprop`, `vector-ref`, and a number of intrinsic procedures added to `PARCEL` for the support of parallelism. The difficulty here is that Scheme is too high-level a language for the direct expression of such procedures. As a consequence, the dependence consequences of these procedures were treated specially and directly by the interprocedural analysis.

Compound objects A second problem with `PARCEL`'s analysis is that it treats all mutable, compound data objects (vectors, user structures, etc.) as though they were lexical closures over free variables. For example, a two-element structure called `house`, expressed in terms of nested lexical contours, is illustrated in Figure 9. An instance of `house` is represented as an instance of the inner procedure in this figure; it is applied to a symbol (one of `color!`, `owner!`, `color`, or `owner`), and a value. If the symbol is `color!` or `owner!`, then the `color` or `owner` field of the structure is assigned the value, otherwise the value of one of these fields is simply returned (with no update occurring). The procedure `house-owner` is used when reading a field of a `house`, and an expression like `(set! (house-owner x) y)` is macro-expanded to the form `(house-owner! x y)`.

While this analogy (between structures and closures) is quite correct, it leads to a number of difficulties. First, the interprocedural analysis must be highly sensitive to correctly notice that `house-owner!` uses and modifies an instance of the variable `owner`, whereas `house-owner` only uses this variable (does not modify it). Second, the representation (within the interprocedural analysis) of closures contains much information that could be compressed or eliminated,

if one were to represent structures like `house` directly. These two facts make the analysis considerably more costly to perform than need be. Finally, while structures can be represented quite precisely as closures, the representation given to vectors for the purpose of analysis is much less precise; consequently, the analysis is much less sharp concerning side-effects upon vectors than upon user structures or lexical variables. In particular, the analysis cannot benefit from even the simplest test for independence among subscript expressions [Ban76, Ban79].

Connectivity Analysis Finally, PARCEL's analysis allows one to build a dependence graph of each procedure based upon the side-effects attributed to the expressions of that procedure; this was illustrated in the example of Figure 1 above, where it was found that there are no dependences between

```
(sum (car l) (counter))
```

and

```
(list-of-sums (cdr l))
```

because the intersection of their visible side-effects is empty. In short, the observation employed here is that *the visibility of side-effects upon an object is restricted to the subcomputation that contains the lifetime of the object*. This observation is effective in identifying coarse-grained parallelism from the program, but is not generally effective in extracting medium- and fine-grained parallelism from traversals that modify data structures, because it does not make use of information concerning the connectivity of pointers between objects (even though such information is provided by the analysis), but simply characterizes side-effects by the lifetimes of the objects involved.

2.2.2 Automatic parallelization of intermediate form

Unstructured intermediate form PARCEL represented its input programs as control-flow graphs for the purpose of transformations, and because these graphs were transformed rather radically from their original form, they were often less structured than a Scheme program would be. For example, in the example of Figure 4, we see an `exit-block` form. This is a printed representation of a sequential loop as PARCEL views it. Such a loop is simply a cycle of control flow, with a single entrance and (possibly) many exits. In the case of Figure 4, there is but one exit, a branch to the label `1-210`. In general, such an exit might simultaneously escape from several, nested loops. PARCEL's loop-parallelization algorithms accept as input a sequential loop in this form, and attempt to render it as a `do` loop, before further parallelizing it. This process is somewhat ad-hoc, and in any event is very complex, owing to the generality of form of input loop it treats. The representation given to the program is likewise quite complex, as it is dictated by the generality of structure of the loops.

Intrinsic procedures and dependences In all phases of PARCEL, Scheme's intrinsic procedures (including those added to the run-time system for support of parallelism) proved to be a nagging difficulty. During program transformation, PARCEL had to account for the particular dependence consequences of a number of intrinsic procedures. As in the case of interprocedural analysis, this difficulty would have been alleviated had the intrinsic procedures been written in Scheme itself and presented to the compiler along with the user-defined procedures; but as mentioned above, a number of the intrinsic procedures, including many added to PARCEL for support of parallelism, have no straightforward representation in Scheme.

2.2.3 Code Generator and Run-time system

PARCEL's code generation was, in most respects, quite successful. The compiler produces a high-level assembly program, that contains macro invocations for data movement, for invocations of intrinsic procedures, for accessing lexical variables, for entering and exiting lexical contours, etc. To target PARCEL for a particular machine means to expand these macros into the assembly language of the target machine. The only difficulty here is, once again, that the intrinsic procedures must be hand-crafted for each target machine. Moreover, the compiler requires several versions of each intrinsic procedure, according to whether its arguments will arrive in registers or in memory, and according to where its return value is to be placed. Like the other problems caused by the intrinsic procedures, this would have been solved had they been written in Scheme itself, which proved impossible for many of the low-level intrinsics.¹

3 MIPRAC

MIPRAC is a successor to PARCEL; it is a parallelizing compiler for mixed symbolic / numeric computations. By *symbolic* computations, we mean those that make extensive use of *locations* and *procedures*. Locations may occur as values (pointers), and are the means by which memory is read and written. Procedures may likewise occur as values (e.g. `lambda` expressions in Scheme, function pointers in C), and are a principal means by which control is transferred. By *numeric* computations we mean those that make extensive use of *numbers*, integers or floating point.

MIPRAC is multilingual: it accepts C, Scheme, Common Lisp, and Fortran programs, which are translated into an intermediate form called MIL. A critical property of MIL is that the variety of control structures that occur in the source languages, is compressed into only a few control structures in MIL; similarly, the

¹By the time code generation is reached in PARCEL, the program has calls to a number of parallel recurrence solvers, and procedures that manipulate the stack; these procedures simply have no natural expression in Scheme.

variety of constructs by which memory is accessed and modified in the source languages, is compressed into only a few constructs for manipulating storage in MIL. This is accomplished in two steps: a syntax-directed translation from the source language, and a subsequent control-flow normalization [Amm90]. In the end, the only control structures that remain are sequencing (**begin**), conditional execution (**if**), and procedure calls. This gives MIPRAC a procedure orientation, in contrast to the loop orientation of conventional parallelizers. The bulk of the run-time support for MIL programs (procedures provided by the source language, formatted input/output, etc.) is written in MIL itself, or in one of the source languages. This, together with the compactness of MIL, makes MIPRAC an easy compiler to retarget.

The object of MIPRAC is to parallelize programs automatically, for shared-memory multiprocessors. This means creating parallel activity to keep multiple processors busy, and placing data in a hierarchical shared memory, so that access times are minimized. Our target architectures are machines with many processors and a deep memory hierarchy. For this reason, two kinds of analysis are needed: an interprocedural analysis of dependences (to extract coarse-grain parallelism), and of object lifetimes (to manage the memory over large units of computation). In MIPRAC, this is accomplished by an interprocedural, abstract interpretation over domains in which side-effects, dependences, object lifetimes, and structure sharing (aliasing among pointers) have a natural expression [Har90].

The code generator and run-time support for MIPRAC on Cedar is being adapted from the same components of PARCEL.

3.1 The Intermediate Language MIL

The core of the intermediate language used by MIPRAC, called MIL, is given by the following grammar.

$$\begin{array}{l}
Expr \Rightarrow \quad (\mathbf{begin} \ Expr \ \dots \ Expr) \\
\quad | \quad (\mathbf{if} \ Expr \ Expr \ Expr) \\
\quad | \quad (\mathbf{read} \ Expr) \\
\quad | \quad (\mathbf{read} \ [Expr \ Const]) \\
\quad | \quad (\mathbf{write} \ Expr \ Expr) \\
\quad | \quad (\mathbf{create} \ Expr \ Id) \\
\quad | \quad (\mathbf{call} \ Expr \ \dots \ Expr) \\
\quad | \quad (\mathbf{call} \ [Expr \ Const] \ Expr \ \dots \ Expr) \\
\quad | \quad (+ \ Expr \ Expr) \\
\quad | \quad (* \ Expr \ Expr) \\
\quad | \quad (= \ Expr \ Expr) \\
\quad | \quad (\leq \ Expr \ Expr) \\
\quad | \quad Proc \\
\quad | \quad Id \\
\quad | \quad Const \\
\quad | \quad [Const \ Const] \\
Proc \Rightarrow \quad (\mathbf{procedure} \ Params \ Locals \ Expr) \\
Params \Rightarrow \quad (\mathbf{parameter} \ Decl \ \dots \ Decl) | \epsilon \\
Locals \Rightarrow \quad (\mathbf{local} \ Decl \ \dots \ Decl) | \epsilon \\
Decl \Rightarrow \quad Id \\
\quad | \quad [Id \ Const]
\end{array}$$

Expr represents legal MIL expressions. *Const* represents (integer) constant expressions. *Id* represents identifiers. The square brackets that appear in the productions of **read** and **call** expressions and constants, are concrete syntax; that is, a **read** expression may take the form

$$(' \ \mathbf{read} \ ' [Expr \ Const] ')'.$$

There are three types of values in MIL: *locations*, *closures*, and *integers*. A location refers either to a dynamically allocated object, or to a variable bound by a procedure activation. A location may be used to read or write memory; it may be stored in memory (as a value); it may be passed to or returned from a procedure call; it may be added to an integer (to yield another location); and it may be compared to another value.

A closure is a procedure along with a lexical environment. The environment is used in mapping the free variables of the procedure to locations. A closure may be applied to arguments (that is, the closed procedure may be called); it may be stored in memory; it may be passed to or returned from a procedure call; and it may be compared to another value.

An integer may be added to a location or another integer; it may be multiplied by another integer; it may be stored in memory; it may be passed to or returned from a procedure call; and it may be compared to another value.

Each type of value is assumed to have a size, the number of bytes of storage it occupies. Locations have size **L**, closures have size **C**, and integers have (default) size **Z**. These constants act as parameters to the semantics of MIL, and allow us to describe programs that manipulate values of various sizes.

Every expression in a MIL program returns an object when evaluated. The object is a collection of values (locations, closures, and integers). Frequently an object contains just one value, and when we speak informally of the value returned by an expression, we mean that the expression returns an object containing exactly one value.

The semantics of each type of MIL expression is given informally below.

- (**begin** $\nu_1 \cdots \nu_l$): ν_1 through ν_l are evaluated in order and the object returned by ν_l is returned from the **begin** expression as a whole.
- (**if** $\nu_1 \nu_2 \nu_3$): ν_1 is evaluated. It must return an integer. If the integer is non-zero ν_2 is evaluated, else ν_3 is evaluated; the object returned by this evaluation is returned from the **if** expression.
- (**read** ν_1) or (**read** [$\nu_1 \ \kappa$]): ν_1 is evaluated. It must yield a location. An object of κ bytes is read from this location, if κ is specified; otherwise, an object of **Z** bytes is read. The object is returned.
- (**write** $\nu_1 \nu_2$): ν_1 is evaluated. It must yield a location. ν_2 is evaluated; let us suppose that the object it returns is k bytes in length. This object is written into the k bytes that begin at the location given by ν_1 , and is returned from the **write** expression.
- (**create** $\nu_1 \iota$): ν_1 is evaluated. It must yield an integer, call it n . The location of a newly allocated object of n bytes is returned. ι specifies in which *area* of memory the object is to be allocated. This area may be obtained by applying a special operator to the location returned by **create**. In this way, tagged data can be conveniently implemented.
- (**call** $\nu_1 \cdots \nu_l$) or (**call** [$\nu_1 \ \kappa$] $\nu_2 \cdots \nu_l$): ν_1 is evaluated, and it must yield a closure. ν_2 through ν_l are evaluated, and the objects they return are concatenated into a single object. The formal parameters and local variables of the called procedure are bound to fresh locations. The formal parameters are initialized by reading from the object that contains the actual parameters. The body of the procedure is evaluated, and an object is obtained. The first κ bytes of the object are read, if κ is given in the **call** expression; otherwise, **Z** bytes of the object are read, and the resulting object (of κ or **Z** bytes) is returned from the **call** expression.
- (**+** $\nu_1 \nu_2$): ν_1 and ν_2 are evaluated, and at least of them must return an integer; the other may return a location or an integer. If both yield integers, they are added and the result is returned. Otherwise, one of

them yields an location; the integer returned by the other is added as a displacement (in bytes) to the location, and the resulting location is returned.

- $(*\nu_1 \nu_2)$: ν_1 and ν_2 are evaluated; both must return integers. The integers are multiplied and returned.
- $(= \nu_1 \nu_2)$: ν_1 and ν_2 are evaluated. If they yield equal locations, or equal closures, or equal integers, then the integer 1 is returned; otherwise, 0 is returned.
- $(\leq \nu_1 \nu_2)$: ν_1 and ν_2 are evaluated. Both must yield integers, or both must yield locations within a single object. If both yield integers, they are compared. The integer 1 is returned if the outcome of the comparison is true; otherwise 0 is returned. If ν_1 and ν_2 yield locations within a single object, the integer 1 is returned if the first location is no farther displaced from the base of the object than the second; otherwise 0 is returned.
- **(procedure *Params Locals* μ)**: A closure is created, that specifies this procedure and the current lexical environment.
- ι : The *location* to which this identifier is bound, is returned. (To obtain the contents of a variable \mathbf{x} one writes **(read \mathbf{x})** or **(read [$\mathbf{x} \kappa$])**; \mathbf{x} yields the location of the variable, and the **read** expression reads memory beginning at this location.)
- κ_1 or $[\kappa_1 \kappa_2]$: An integer constant is returned. By default, this constant is \mathbf{Z} bytes in length, but if κ_2 is given, the size of the constant is taken to be κ_2 instead.

The following examples may clarify the informal semantics of MIL. Figures 10, 12, and 14 show programs in C, Pascal, and Scheme, and Figures 11, 13, and 15 show their translations in MIL. The **for** loop in figure 12 becomes a call to a tail-recursive procedure in figure 13. In the next section we will discuss the control-flow normalization that is necessary to effect such a transformation. In Figure 15 it is assumed that $\mathbf{Z} = \mathbf{C} = \mathbf{L}$; this means that every variable may accommodate any type of value (Z , C , or L).

3.2 Control-flow Normalization

The variety of control-flow constructs among the source languages accepted by MIPRAC is overwhelming, and to allow such a number of different structures to be present in MIPRAC's intermediate form would be to invite disaster. We handle this difficulty in two steps. First, the front-ends for MIPRAC rewrite all iterative structures in terms of **gotos** and labels. Second, a phase of control-flow normalization [Amm90] rids the program of **gotos** entirely, replacing them with

```

struct foo {
    int info;
    struct foo *next;
};
struct foo *make_list(n)
    int n;
{
    struct foo *s, *t;
    if (n != 0)
    {
        t = (struct foo *) malloc(sizeof (struct foo));
        t->info = n;
        s = make_list(n-1);
        t->next = s;
        return(t);
    }
    return(NULL);
}
main()
{
    make_list(15);
}

```

Figure 10: A C Program

```

(call (procedure (local [make_list C])
  (write make_list (procedure (parameter n)
    (local [s L] [t L])
      (if (read n)
        (begin (write t (create (+ Z L) foo1))
          (write (+ (read [t L]) 0) (read n))
          (write s (call [(read [make_list C]) L]
            (+ (read n) -1)))
          (write (+ (read [t L]) Z)
            (read [s L]))
          (read t))
        [0 L])))
  (call (read [make_list C]) 15)))

```

Figure 11: The Program of Figure 10 Rewritten in MIL

```

procedure f;
  var a : array [1 : 10] of integer;
      i : integer;
  begin
    for i = 1 to 10 do
      begin
        a[i] = i;
        g(a[i]);
      end
    end
  end
procedure g(var i : integer);
  begin
    i = i + 1;
  end
begin
  f;
end.

```

Figure 12: A Pascal Program

```

(call (procedure (local [f C] [g C])
  (write f (procedure (local [a (* 10 Z)]
    [i Z]
    [loop C])
    (write loop (procedure
      (if (< (read i) 10)
        (begin
          (write (+ a (* (read i) Z)) (read i))
          (call (read [g C]) (+ a (* (read i) Z)))
          (write i (+ i 1))
          (call (read [loop C])))
        0)))
      (write i 1)
      (call (read [loop C])))))
  (write g (procedure (parameter x)
    (write (read [x L])
      (+ (read (read [x L])) 1))))
  (call (read [f C]))))

```

Figure 13: The Program of Figure 12 Rewritten in MIL

```

(define f (lambda (x)
  (lambda (y)
    (set! x (+ x y))
    x)))
(define g (f 2))
(define h (f 3))
(g 10)
(h 20)

```

Figure 14: A Scheme Program

```

(call (procedure (local f g h )
  (begin (write f (procedure (parameter x)
    (procedure (parameter y)
      (write x (+ (read x) (read y))))))
    (write g (call (read f) 10))
    (write h (call (read f) 20))
    (call (read g) 10)
    (call (read h) 10))))))

```

Figure 15: The Program in Figure 14 Rewritten in MIL

```

struct pair {
  int info
  struct pair *link
};
struct pair *g(l,m)
  struct pair *l;
  struct pair *m;
  {
  struct pair *next;
  goto LAB2;
  LAB1:
  if (next != 0) {
    l = next;
    LAB2:
    next = l->link;
    goto LAB1;
  }
  else {
    l->link = m;
    goto LAB3
  }
  LAB3:
  return l;
}

```

Figure 16: A C Program In Need of Restructuring

while loops, properly nested **if** forms, and **begin** forms. Finally, the **while** loops of the program are rewritten as tail-recursive procedures. The result is that the only remaining control structures are properly nested **ifs**, **begins**, and procedure calls.

An example will make the matter clear. Figure 16 shows a C program, containing a loop described using **gotos**. Figure 17 shows the program translated into a MIL program, that also contains branch instructions. Figure 18 shows the program after the **gotos** have been replaced by a **while** structure, and Figure 19 shows the program after the **while** loop have been replaced by procedure calls.

```

(:write g (:procedure (:parameter l m)
  (:local next)
  (:label LAB1 LAB2 LAB3)
  (:go LAB2)
  LAB1
  (:if (:read next)
    (:begin
      (:write l (:read next))
      LAB2
      (:write next (:read (+ (:read l) 1)))
      (:go LAB1))
    (:begin
      (:write (+ (:read l) 1) (:read m))
      (:go LAB3)))
  LAB3
  (:return (:read l))))

```

Figure 17: The C Program, Translated into MIL

```

(procedure (parameter l m) (local next return-63 pred-64)
  (label lab1 lab2 lab3 proc-exit-62 true-branch-65
    false-branch-66 if-exit-67 proc-entry-73)
  (begin (while (begin (write next
    (read (+ (read l) 1)))
    (write pred-64 (read next))
    (if (read pred-64)
      (write l (read next)) 0)
    (read pred-64)))
    (write (+ (read l) 1) (read m))
    (write return-63 (read l))
    return-63))

```

Figure 18: After Control Normalization

```

(procedure
  (parameter l m)
  (local next return-63 pred-64 while-75)
  (label lab1 lab2 lab3 proc-exit-62 true-branch-65
    false-branch-66 if-exit-67 proc-entry-73)
  (begin (write while-75
    (procedure
      (if (begin (write next (read (+ (read l) 1)))
        (write pred-64 (read next))
        (if (read pred-64)
          (write l (read next)) 0)
        (read pred-64))
        (call (read while-75))
      0)))
    (call (read while-75))
    (write (+ (read l) 1) (read m))
    (write return-63 (read l))
    return-63))

```

Figure 19: After While Elimination

	ϵ
call α	α^d
call β	$\alpha^d \beta^d$
call γ	$\alpha^d \beta^d \gamma^d$
return γ	$\alpha^d \beta^d \gamma^d \gamma^u$
call β	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d$
call γ	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d$
return γ	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d \gamma^u$
return β	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d \gamma^u \beta^u$
return β	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d \gamma^u \beta^u \beta^u$
call α	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d \gamma^u \beta^u \beta^u \alpha^d$
return α	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d \gamma^u \beta^u \beta^u \alpha^d \alpha^u$
return α	$\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d \gamma^u \beta^u \beta^u \alpha^d \alpha^u \alpha^u$

Figure 20: Procedure Strings

3.3 Interprocedural analysis

MIPRAC attempts to parallelize by a deep, interprocedural semantic analysis. This analysis takes the form of an abstract interpretation. Two kinds of information are gleaned from this analysis: dependences, and object lifetimes. Two kinds of dependence information are collected: the interprocedural visibility of side-effects, and structure sharing (pointer aliasing). The analysis is an extension and refinement of the analysis implemented in PARCEL. Like PARCEL’s analysis, this one revolves around *procedure strings* and their abstraction.

3.4 Procedure Strings

Procedure strings are a means of “objectifying” interprocedural control-flow, so that it can be examined and abstracted straightforwardly. Imagine that a program has procedures α , β , and γ . We denote a call to procedure α by the term α^d . The d indicates that control moves *downward* into α . We denote a return from procedure β by the term β^u . The u indicates an upward movement from β . A sequence of such terms is called a *procedure string*. We can record the interprocedural history of an execution of the program by accumulating the procedure calls and returns that occur in a procedure string. Figure 20 shows the procedure calls and returns made by a hypothetical execution of the program and the corresponding procedure strings after each interprocedural movement.

Every time a procedure α is called, we say that an *activation* of α is created. This activation is identified (uniquely) by a procedure string of the form $\dots \alpha^d$. We call this procedure string the *birthdate* of the activation. For example, in figure 20 there are two activations of procedure γ . The first has birthdate

$\alpha^d \beta^d \gamma^d$ and the other $\alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d$.

There are several operations over procedure strings that we will find useful. If p_1 and p_2 are procedure strings, then $p_1 + p_2$ is the concatenation of p_1 and p_2 . The operations $-$ and \cdot are defined by $p_3 - p_1 = p_2$ and $p_3 \cdot p_2 = p_1$, where $p_3 = p_1 + p_2$. For example, if $p_1 = \alpha^d \beta^d \gamma^d$ and $p_2 = \gamma^u \beta^d \gamma^d$ then $p_3 = p_1 + p_2 = \alpha^d \beta^d \gamma^d \gamma^u \beta^d \gamma^d$, $p_3 - p_1 = \gamma^u \beta^d \gamma^d$, and $p_3 \cdot p_2 = \alpha^d \beta^d \gamma^d$.

If p is a procedure string, then $Net(p)$ is the procedure string obtained by deleting all substrings of the form $\alpha^d \alpha^u$ (for all $\alpha \in Proc$) from p , until no further such deletions are possible. $Trace(p, \alpha)$ is obtained by deleting all terms other than α^d or α^u from p . For example, if $p = \alpha^u \beta^d \gamma^d \gamma^u \beta^u \alpha^d \beta^d \gamma^d \gamma^u$ then $Net(p) = \alpha^u \alpha^d \beta^d$ and $Trace(p, \beta) = \beta^d \beta^u \beta^d$.

Side-effects, object lifetimes, and structure sharing all have a natural expression in terms of procedure strings. Let p_b be the procedure string of the state in which an object X is created (allocated), and let p_r be the procedure string of a later state in which X is modified. It can be shown that if and only if $Net(p_r - p_b)$ contains a term α^d , then the procedure α has a side-effect as a result of this reference. By “has a side-effect”, we mean that the caller of α can observe that the modification has occurred. Similar theorems can be written which characterize the lifetimes of objects (e.g., whether or not they may be stack-allocated), and the structure sharing among objects, in terms of procedure strings.

3.5 Standard, Instrumented, and Abstract Semantics for MIL

The abstract interpretation that is MIL’s interprocedural analysis algorithm is derived in several steps. We begin with a standard semantics over the following domains:

S	=	$Id \rightarrow Z \rightarrow B$	stores
B	=	$(Z \times Y) \rightarrow V$	objects
V	=	$L + C + Z$	values
L	=	$Id \times Z \times Z$	locations
C	=	$Proc \times E$	closures
E	=	$Id \rightarrow Z$	environments
Z			integers
Y	\subset	Z	sizes
T			booleans
$Proc$			procedures
$Expr$			expressions
Id			identifiers

The domains Z , Y , T , $Expr$, $Proc$, and Id are flat: each has a least element ($-z$, $-T$, etc.), and its other elements are incomparable. The structure of

the non-bottom elements of *Expr* and *Proc* is given by the grammar of MIL above. None of the above domains is reflexive, since each is defined in terms of those strictly below it in the table. Said another way, the equation for each can be written (finitely) as products (\times), sums ($+$) and function spaces (\rightarrow) of the flat domains Z , T , *Expr*, *Proc*, and *Id*. For example, B is equal to $(Z \times Y) \rightarrow ((Id \times Z \times Z) + (Proc \times (Id \rightarrow Z)) + Z)$.

Whenever a procedure is called, we say that a procedure *activation* is created. In the standard semantics, a count of procedure calls is maintained as part of the state of evaluation. The count is incremented at every procedure call, and the incremented count is called the *birthdate* of the procedure activation that is created. The birthdate is used to distinguish the parameters and local variables bound by the procedure from other instances of the same variables. Similarly, when a **create** expression is evaluated, the value of the count of procedure calls is used to distinguish the new object from the objects allocated by other instances of the same **create** expression. The count associated with a variable or dynamically allocated object is likewise called its birthdate.

The meaning of each of the domains is as follows:

- $S = Id \rightarrow Z \rightarrow B$ (stores). A member s of S denotes a store. It maps an identifier ι and an integer z to an object. ι names an object (variable or dynamically allocated object). z is the birthdate of the object.
- $B = (Z \times Y) \rightarrow V$ (objects). A member b of B denotes a block of values in contiguous locations. b maps a pair of an offset (an integer displacement from the base of the object) and a size (an integer) to the value, if there is one of the given size, stored at that offset. By the meaning that we give to the writing of data into an object, every position in it contains (part of) at most one value. That is, if we store a long integer (4 bytes) at offset 6, and subsequently store a character (1 byte) at offset 7, the value of the long integer is destroyed; reading a long integer from position 6 after the write will return an undefined value. This is consistent with the conventions of C: we may read and write overlapping locations by the use of unions, casting of pointers, and pointer arithmetic, but writing to a location that overlaps another destroys the contents of the latter.

In addition to denoting regions of addressable memory, objects are used to denote the blocks of values returned by the evaluation of expressions. For example, a **read** or **call** expression may return a structure or vector, and similarly a **write** expression may copy such an object into the store. The object returned by an expression has constant size (see the function *Size* below) and frequently contains only a single value.

- $V = L + C + Z$ (values). A member v of V denotes a value. A value may be either a location (a member of L), a closure (a member of C), or an integer (a member of Z).

- $L = Id \times Z \times Z$ (locations). A member l of L denotes a location in the store. It specifies a position within an object, either a variable or a dynamically allocated object. The object is named by the identifier $l \downarrow 1$. The integer $l \downarrow 2$ is the birthdate of the object. The integer $l \downarrow 3$ is a displacement from the beginning of the object. For example, if $l = \langle \iota, z_b, z \rangle$, then l points to the instance of ι whose birthdate is z_b ; l points z bytes from the base of this object. z_b is called the *birthdate* of the object pointed to by l .
- $C = Proc \times E$ (closures). When a **procedure** expression is evaluated, it yields a member c of C , a closure, that captures the procedure along with the lexical environment at the point of evaluation. If $c = \langle \alpha, e \rangle$, then α is the procedure and e is the lexical environment that is used to map the free variables in α to locations in the store.
- $E = Id \rightarrow Z$ (environments). A member e of E is used in mapping an identifier to a location in the store, so that the value of the variable named by the identifier may be obtained or altered. e does not map identifiers directly to locations, however; rather, it maps an identifier to the birthdate of the variable it names. For example, suppose that $e(\iota) = z_b$. Then z_b is the birthdate of the instance of ι that is visible in the environment e , and $\langle \iota, z_b, 0 \rangle$ is the location associated with ι in the environment e .
- Z (integers). Z is the flat domain of integers.
- Y (sizes). Y is the flat domain of value sizes. Y contains at least the elements $-z$, \mathbf{L} , \mathbf{C} , and \mathbf{Z} . In addition, it contains all of the sizes that integers are permitted to have. For example, if $\mathbf{Z} = \mathbf{L} = \mathbf{C} = 4$, and integers of 1, 2, 4, and 8 bytes are permitted, then $Y = \{-z, 1, 2, 4, 8\}$. Y , \mathbf{L} , \mathbf{C} , and \mathbf{Z} are parameters to the semantics of MIL; they determine the meaning of memory access and pointer arithmetic.²
- T (booleans). T is the flat domain $\{-, true, false\}$.

We define a standard evaluator for MIL, that takes an expression, a count of procedure calls (an integer), a lexical environment, and a store, and returns a store, and object, and a count of procedure calls. That is, the standard evaluator has type $Expr \times Z \times E \times S \rightarrow S \times B \times Z$.

Next, we derive an instrumented evaluator from the standard one, by replacing the count of procedure calls, by a procedure string. That is, rather than to count the procedure calls that occur during a program execution, the instrumented evaluator maintains a procedure string that accumulates the history of calls and returns. The birthdates of objects are no longer integers; they are procedure strings. It is therefore possible to write theorems that describe all the

²Similarly, we may think of the `sizeof` operator of C as a parameter to the semantics of C.

side-effects, dependences, object lifetimes, and structure sharing of a program, in terms of its instrumented semantics.

The domains of the instrumented semantics are as follows:

S	$=$	$Id \rightarrow P \rightarrow B$	stores
B	$=$	$(Z \times Y) \rightarrow V$	objects
V	$=$	$L + C + Z$	values
L	$=$	$Id \times P \times Z \times P \times P$	locations
C	$=$	$Proc \times E$	closures
E	$=$	$Id \rightarrow P$	environments
P	$=$	$(Proc^d Proc^u)^*$	procedure strings
Z			integers
Y	\subset	Z	sizes
T			booleans
$Proc$			procedures
$Expr$			expressions
Id			identifiers

The instrumented evaluator has type $Expr \times P \times E \times S \rightarrow S \times B \times P$.

Finally, an abstract interpretation is derived from the instrumented one, by projecting procedure strings and integers onto finite domains. All of the other domains in the instrumented semantics are built from non-reflexive domains P , Z , T , Id , $Proc$, and $Expr$. Since the syntactic domains are finite (from the perspective of a single program), when P and Z are made finite, all of the domains become finite. It is therefore easy to construct an abstract semantics, and in fact it is easy to construct a variety of them, with differing degrees of approximation to the instrumented semantics.

The domains of the abstract semantics are as follows:

\hat{S}	$=$	$Id \rightarrow \Delta \rightarrow \hat{B}$	abstract stores
\hat{B}	$=$	$(\hat{Z} \times Y) \rightarrow \hat{V}$	abstract objects
\hat{V}	$=$	$\hat{L} \times \hat{C} \times \hat{Z}$	abstract values
\hat{L}	$=$	$\hat{Id} \times \hat{P} \times \hat{Z} \times \hat{P} \times \hat{P}$	abstract locations
\hat{C}	$=$	$Proc \times \hat{E}$	abstract closures
\hat{E}	$=$	$Proc \rightarrow \hat{P}$	abstract environments
\hat{P}	$=$	$Proc \rightarrow 2^\Delta$	abstract procedure strings
\hat{Z}	$=$	$\{-, 0, \dots, \ell - 1, \top\}$	abstract integers
\hat{T}	$=$	$\{-, true, false, \top\}$	abstract booleans
Δ	$=$	$\{\epsilon, \mathbf{d}, \mathbf{dd}^+, \mathbf{u}, \mathbf{uu}^+, \mathbf{u}^+ \mathbf{d}^+\}$	movement tokens
\hat{Proc}	$=$	2^{Proc}	abstract procedures
\hat{Id}	$=$	2^{Id}	abstract identifiers

Just as theorems were written that described dependences, object lifetimes, and structure sharing exactly in terms of the instrumented semantics, theorems

that describe these things approximately are written in terms of the abstract semantics. The interprocedural analysis is then the following problem: to find the abstract meaning of a program (a fixpoint problem), and to apply theorems to that meaning, that describe the dependences, lifetimes, and structure sharing of the program.

This analysis is described in detail in [Har90]. It has been implemented in C, and is itself being parallelized to run on an Alliant or on the Cedar, because it is the most expensive phase of MIPRAC. For example, the analysis requires 77 cpu seconds of time on a Sparc-1 to analyze the example program shown in Figures 21 and figure 22. The analysis reveals that all of the procedures of the program are side-effect free, but that the expression

```
(:write (:+ (:read t) 1)
        (:call (:read make_inf_list)
              (:- (:read n) 1)
              (:read t)))
```

induces a cycle in the data structure it modifies.

4 Parallelization and Code Generation

After the interprocedural analysis of a program is performed, MIPRAC attacks the problem of parallelizing automatically, by two strategies. The first is to introduce `cobegin` forms when the interprocedural analysis of side-effects has revealed that two expressions, guessed to be significant by the compiler, may be evaluated simultaneously. Because all iterative computation is rewritten by this point as recursive procedure invocation, this strategy alone turns up significant parallelism, depending (entirely) upon the accuracy of the interprocedural analysis. The second strategy operates when a simple freedom from side-effects is not found. An interprocedural recurrence recognizer is used to determine what pattern of memory access is being made by a recursive or tail-recursive procedure. For example, the parameter `l` of the procedure `count_inf_lists` above describes the recurrence $n_{i+1} = (\text{read } (+ (\text{read } l) 1))$. *The terms of this recurrence relation are locations in memory.* The recurrence relation is examined in light of the interprocedural analysis of structure sharing, to determine if the terms of the recurrence refer to distinct locations in memory. If they do, modifications to those locations may be performed in parallel.

5 Conclusion

PARCEL was arguably the first complete system for the automatic parallelization of Lisp programs. It was quite successful in several respects: it introduced a sharp interprocedural semantic analysis that computes the interprocedural visibility of side-effects, and allows the placement of objects in memory according to

```

struct cell {
    int info;
    struct cell *next;
};
struct cell *make_infl_list(n,last)
    int n;
    struct cell *last;
    {
    struct cell *t;
    t = (struct cell *) malloc(sizeof(struct cell));
    t->info = n;
    if (n != 0)
        t->next = make_infl_list(n-1,t);
    else
        t->next = last;
    return t;
    }
int count_infl_list(l)
    struct cell *l;
    {
    if (l->info != 0)
        return (l->info + count_infl_list(l->next));
    else
        return 0;
    }
int make_and_count_infl_lists(m)
    int m;
    {
    struct cell *c;
    if (m != 0) {
        c = make_infl_list(m,NULL);
        return count_infl_list(c) + make_and_count_infl_lists(m-1);
    }
    else
        return 0;
    }
main()
    {
    make_and_count_infl_lists(10);
    }

```

Figure 21: An Input to Interprocedural Analysis

```

(:write make_inf_list
  (:procedure (:parameter n last)
    (:local t)
    (:begin
      (:write t (:create 2 0))
      (:write (:+ (:read t) 0) (:read n))
      (:if (:read n)
        (:write (:+ (:read t) 1)
          (:call (:read make_inf_list)
            (:- (:read n) 1)
            (:read t)))
        (:write (:+ (:read t) 1) (:read last)))
      (:read t))))
(:write count_inf_list
  (:procedure (:parameter l)
    (:if (:read (:+ (:read l) 0))
      (:+ (:read (:+ (:read l) 0))
        (:call (:read count_inf_list)
          (:read (:+ (:read l) 1))))
      0)))
(:write make_and_count_inf_lists
  (:procedure (:parameter m)
    (:local c)
    (:if (:read m)
      (:begin
        (:write c (:call (:read make_inf_list) (:read m) 0))
        (:+ (:call (:read count_inf_list) (:read c))
          (:call (:read make_and_count_inf_lists)
            (:- (:read m) 1))))
        0)))
(:call (:read make_and_count_inf_lists) 10)

```

Figure 22: The Example, In MIL

their lifetimes; it introduced several restructuring techniques tailored to the iterative and recursive control structures that arise in Lisp programs; and it made use of multiple procedure versions with a flexible microtasking mechanism for efficient parallelism at run-time. PARCEL had several shortcomings however: the intrinsic procedures of Scheme, and those added to PARCEL for support of parallelism, were embedded in its interprocedural analysis, transformations, code generation and run-time system, making the system difficult to adapt for other source languages; its interprocedural analysis handled compound, mutable data only indirectly (by analogy to closures), making it less accurate and more expensive than necessary; and its representation of programs as general control-flow graphs made the implementation of complex transformations difficult.

MIPRAC is a successor to PARCEL, in which we are extending the techniques of PARCEL, and applying them to a broad class of procedural languages. MIPRAC operates upon a compact intermediate form in which the operational semantics of several source languages may easily be expressed. This form is normalized so that the programs MIPRAC analyzes and parallelizes have a very simple control structure. A deep interprocedural analysis is performed, to determine dependences, object lifetimes, and structure sharing. Automatic parallelization is done by a twofold strategy of simple `cobegin` insertion based on interprocedural side-effect information, and a more sophisticated parallelization of recursive procedures that modify data structures, by examination of the recurrence relations they describe as they traverse memory.

References

- [Amm90] Zahira Ammarguellat. An algorithm for control-flow normalization and its complexity. Technical report, Center for Supercomputing Research and Development, University of Illinois, July 1990.
- [Ban76] Uptal D. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, November 1976.
- [Ban79] Uptal D. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, University of Illinois at Urbana-Champaign, October 1979.
- [BC86] Michael Burke and Ronald G. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 162–175. Association for Computing Machinery, July 1986.
- [Cho90] Jhy-Herng Chow. Run-time support for automatically parallelized lisp programs. Master's thesis, University of Illinois at Urbana-Champaign, 1990.

- [Cra82] Cray Research, Mendota Heights, MN. *Cray X-MP Series Mainframe Reference Manual (HR-0032)*, 1982.
- [EHJP90] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar fortran and its compiler. Technical Report 966, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, jan 1990.
- [Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, Massachusetts, 1985.
- [Har89] Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation: an International Journal*, 2(3/4):179–396, 1989.
- [Har90] Williams Ludwell Harrison III. Semantic analysis for automatic parallelization of symbolic programs. Technical Report Work In Progress, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October 1990.
- [Tri84] Remi Triolet. *Contributions to Automatic Parallelization of Fortran Programs with Procedure Calls*. PhD thesis, University of Paris VI (I.P.), 1984.