②

Technical Report 1144

A Compilation Strategy for Numerical Programs Based on Partial Evaluation

Andrew A. Berlin

MIT Artificial Intelligence Laboratory

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1144	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Compilation Strategy for Numerical Programs based on Partial Evaluation		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Andrew Berlin		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE July 1989
		13. NUMBER OF PAGES 78
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

partial evaluation
compilation
parallel programming
symbolic interpretation
parallel scheduling
scientific computation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This work demonstrates how partial evaluation can be put to practical use in the domain of high-performance numerical computation. I have developed a technique for performing partial evaluation by using placeholders to propagate intermediate results, and have implemented a prototype compiler based on this technique. For an important class of numerical programs, this compiler improves performance by an order of magnitude over conventional

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 cont.

compilation techniques. I also show that by eliminating inherently sequential data-structure references, partial evaluation exposes the low-level parallelism inherent in a computation. I have implemented a parallel program generator, as well as several analysis programs that study the tradeoffs involved in the design of an architecture that can effectively utilize this parallelism. I present these results using the 9-body gravitational attraction problem as an example.

A Compilation Strategy for Numerical Programs Based on Partial Evaluation

by

Andrew A. Berlin

Abstract

This work demonstrates how partial evaluation can be put to practical use in the domain of high-performance numerical computation. I have developed a technique for performing partial evaluation by using placeholders to propagate intermediate results, and have implemented a prototype compiler based on this technique. For an important class of numerical programs, this compiler improves performance by an order of magnitude over conventional compilation techniques. I also show that by eliminating inherently sequential data-structure references, partial evaluation exposes the low-level parallelism inherent in a computation. I have implemented a parallel program generator, as well as several analysis programs that study the tradeoffs involved in the design of an architecture that can effectively utilize this parallelism. I present these results using the 9-body gravitational attraction problem as an example. [illegible]

This report is a revised version of a thesis submitted t[illegible]he Department of Electrical Engineering and Computer Science in Februar[illegible]9, in partial fulfillment of the requirements for the degree of Master of Science.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Acknowledgments

I would like to thank my thesis supervisor, Gerry Sussman, for his support and constant encouragement. Gerry is a very special person. Special thanks are also due to Hal Abelson, Tom Knight, and Daniel Weise, who took a special interest in this project, and provided valuable insights.

My interest in program specialization was sparked by a straight-line FFT generator written by Thomas D. Simon. Tom's work was based on a Scheme program written by Gerry Sussman. Tom also helped out with debugging some parts of my compiler, and contributed the Multipole Method program which was used as an example in this report.

I first observed the effects of partial evaluation on available parallelism using a scheduler program written by Steve Codell and myself as part of the Scheme86 project. The parallelism profiles presented in this report were generated using this program.

Jack Wisdom of the Department of Earth and Planetary Sciences, as well as Hal Abelson, Tom Simon, Feng Zhao, Panayotis Skordos, and Gerry Sussman contributed programs to be used as test cases for my compiler, and also provided valuable advice. Feng Zhao, Phil Gree spun, Jim O'Toole, and Daniel Weise also helped out by proof-reading this document.

Special thanks are also due to Chris Hanson, Bill Rozas, and Henry Wu, who had interesting things to say, fixed numerous problems with the alpha-test MIT Scheme system at a moments notice, and generally made the lab a nicer place.

Thanks also to Hewlett-Packard for donating the computer equipment on which this work was performed.

Thanks also to Jim O'Toole for donating 3 GO pieces (white), and for providing dinner entertainment.

Contents

Chapter 1

Introduction

Scientists are faced with a dilemma: Either they can write programs that express their understanding of a problem, but which do not execute efficiently; or they can write programs that computers can execute efficiently, but which are difficult to write and difficult to understand. This report explores how partial and symbolic evaluation can be combined with traditional compilation techniques to provide a solution to this dilemma.

My work demonstrates how partial evaluation can be used to reduce the performance penalties associated with abstraction in numerical programs. I have developed a technique for performing partial evaluation by using placeholders to propagate intermediate results, and have implemented a prototype compiler based on this technique. For an important class of numerical programs, this compiler improves performance by an order of magnitude over conventional compilation techniques. Experiments with this compiler show that by eliminating inherently sequential data-structure references, partial evaluation exposes the low-level parallelism inherent in a computation. I have implemented a parallel program generator, as well as several analysis programs that study the tradeoffs involved in the design of a parallel architecture that can effectively utilize this parallelism.

1.1 Overview

Numerical programs written in high-level languages spend much of their time manipulating data structures rather than performing numerical computations. By using information that is available at compile time about the particular problem that a program will solve, it is possible to perform some of these data structure manipulations in advance, creating a specialized, partially-evaluated program. In many important scientific applications, there is enough information available for the compiler to derive the underlying numerical computation from the high-level program, entirely eliminating compound data structures and abstractions. In Chapter 2, I present a technique for performing the partial evaluation required to create such a specialized program by executing the program symbolically at compile time.

I have implemented a prototype compiler based on this technique. This compiler combines the specialization provided by partial evaluation with traditional compilation techniques, allowing optimizations such as constant folding, symbolic simplification, and dead-code elimination to be performed on the low-level computation itself, without interference from abstraction mechanisms or compound data structures. In Chapter 3, I describe the application of this compiler to several real-world scientific applications, and present performance measurements.

Experiments with the prototype compiler have shown that eliminating abstractions not only improves performance, but also exposes the low-level parallelism inherent in a computation. Eliminating inherently sequential data structure references imposed by the way the programmer thinks about a problem allows the compiler to re-order the computation so as to allow intermediate results to be used immediately in other parts of the computation. I have implemented a parallel program scheduler, and used it to study the nature of the parallelism available in the 9-body gravitational attraction problem. Chapter 4 presents these results, as well as a strategy for effectively utilizing this parallelism by generating a specialized routing network for a particular application.

1.2 Background: Previous Work

1.2.1 Abstraction in Scientific Computation

Recent work[Halfant] has shown that through the use of the abstraction mechanisms of high-level languages, it is possible to create more powerful problem-solving strategies than would otherwise be feasible. Unfortunately, the performance penalties associated with abstraction have precluded the use of high-level languages such as Lisp for computationally intensive numerical applications. The challenge facing high-level language compilers is to reduce these performance penalties, thereby fundamentally changing the way that scientists write programs.

1.2.2 Specialization

High-performance numerical programs often generate specialized routines in order to improve performance. For example, the SPICE analog circuit simulator[Nagel] generates specialized matrix manipulation routines to take advantage of the sparse matrices that arise out of network node equations. Similarly, "straight-line" implementations of the Fast Fourier Transform (FFT) may be generated, such that all array indices are computed in advance. These specialized routines are typically created using a hand-crafted generator program that operates over a relatively small segment of a large problem. I present a generalization of this technique that allows generator programs to be expressed abstractly, enabling specialization to occur over a larger portion of the overall problem than would otherwise be practical.

1.2.3 Partial Evaluation

The process through which specialization is achieved is known as partial evaluation. Partial evaluation has been widely written on, but has not yet found much practical use. In particular, [Schooler] did some interesting work on automatically generating a compiler by partially evaluating the interpreter, and

[Komorowski] investigated applications of partial evaluation to Prolog. Partial evaluation is also used on a relatively small scale by Fortran compilers, at the level of pre-computing constants in individual numerical expressions. In this report, I show how partial evaluation can be put to practical use in the domain of high-performance numerical computation.

Chapter 2

Partial Evaluation

I present a technique for performing partial evaluation by executing a program symbolically at compile time. The key idea is that programs can be divided into *"data-independent"* regions in which the flow of control can be predicted at compile time. The code for these regions may be evaluated in advance, eliminating data abstractions and compound data structures.

2.1 Symbolic Execution

The goal of my compiler is to figure out in advance what numerical operations a program will need to perform to produce its result. Interpreted programs determine this at run time by manipulating abstract data structures. By *tracing* the execution of an interpreted program, one can observe the sequence of numerical operations that it performs. In the special case of a data-independent computation, this sequence of numerical operations is independent of the numerical values of the input data. Thus, one way for a compiler to predict in advance what numerical operations a data-independent program will perform is to simply run the program at compile time.

In order to execute a program at compile time, it is necessary to create the data structures that the program will manipulate. The values for some of the program's numerical inputs will be available at compile time, and can be included directly in their normal places in the data structures. However, numerical values for some pieces of data will not be available until run time. These missing values are represented symbolically using *placeholders.* A placeholder is a data structure that is used to represent a specific piece of missing data. Each placeholder contains whatever information is available about the piece of data that it represents, including its *type* and its point of origin.

During the compile-time execution, a numerical operation that receives numerical arguments executes normally, producing a numerical result. Such an operation does need not be included in the compiled program, since its results have already been computed. However, a numerical operation that receives a placeholder as an argument must be delayed until run time, when the value represented by the placeholder is available. An instruction that will perform the delayed operation at run time is added to the compiled program, and a new placeholder is created to represent the result of the operation. This new placeholder propagates through the compile time data structures, eventually becoming an input to another numerical operation, which will in turn be delayed until run time.

Compile-time interpretation produces result data structures which themselves contain placeholders. These *result placeholders* represent the final results of the computations that have been delayed until run time. The compiled program takes as arguments numerical values for each of the placeholders in the input data structures, and executes the delayed instructions to produce numerical values for the result placeholders.

The power of this approach lies in the fact that placeholders are manipulated as if they are the actual data values that they represent. Placeholders propagate through data abstractions, allowing abstractions to be resolved at compile time. For example, `(cons x y)` will simply create a cons cell, even if x happens to be a placeholder. Later in the program, when this cons cell is referenced using `CAR`, the placeholder will be returned. Thus, data abstractions utilized in the *specification* of a program have no effect on the *run time* performance of the instruction stream generated by the compiler.

2.1.1 An Example

To illustrate the compilation process for data-independent programs, consider the compile time execution of the sum-of-squares program shown below, for an application in which the input is kncwn to be a list of three floating-point numbers, the last of which is always 3.14. The compiled program is shown in Figure 2.1.

```
==> (define (square x) (* x x))
==> (define (sum-of-squares L)
         (apply + (map square L)))

==> (sum-of-squares (list (make-placeholder 'floating-point)
                          (make-placeholder 'floating-point)
                          3.14))
```

```
COMPILED PROGRAM (SPECIALIZED SUM-OF-SQUARES):
;;The numbers refer to the placeholder numbers
(INPUT 1)
(INPUT 2)

(ASSIGN 3 (Floating-Point-* (FETCH 1) (FETCH 1)))
(ASSIGN 4 (Floating-Point-* (FETCH 2) (FETCH 2)))
(ASSIGN 5 (Floating-Point-+ (FETCH 3) (FETCH 4) 9.8596))

(RESULT 5)
```

Figure 2.1: Specialized Code for Sum-of-squares. Notice how the squaring of 3.14 to produce 9.8596 took place at compile time.

At compile-time, the program begins executing normally, as shown below:

```
Entering: (sum-of-squares (<placeholder-1> <placeholder-2> 3.14))
Entering: (apply + (map square
                        (<placeholder-1> <placeholder-2> 3.14)))
Entering: (map square (<placeholder-1> <placeholder-2> 3.14))
Entering:    (square <placeholder-1>)
Entering:       (* <placeholder-1> <placeholder-1>)
```

At this point, the multiply operation can not proceed normally, because its arguments are placeholders. A new placeholder, <placeholder-3> is created to represent the result of the multiplication. Since both arguments to multiply were of type floating-point, the result, <placeholder-3>, will also have type floating-point. A floating-point multiply operation is added to the compiled program as line number 6. Execution resumes with multiply returning <placeholder-3> as its result:

```
Returning:    <placeholder-3> <== (* <placeholder-1> <placeholder-1>)
Returning:  <placeholder-3>  <== (SQUARE <placeholder-1>)
Entering:   (SQUARE <placeholder-2>)
Entering:     (* <placeholder-2> <placeholder-2>)
```

Again, the multiply operation must be delayed until run time. <placeholder-4> is created, and the multiply operation is added to the compiled program as line 7. Execution resumes with multiply returning <placeholder-4> as its result:

```
Returning:   <placeholder-4> <== (* <placeholder-2> <placeholder-2>)
Returning:  <placeholder-4>  <== (square <placeholder-2>)
Entering:   (square 3.14)
Entering:    (* 3.14 3.14)            ;This multiply takes
Returning:   9.8596 <== (* 3.14 3.14) ;place at compile time
Returning:  9.8596 <== (square 3.14)
Returning: (<placeholder-3> <placeholder-4> 9.8596) <== (map ...)
Entering:  (+ <placeholder-3> <placeholder-4> 9.8596)
```

The addition operation is delayed until run time as line 8 in the compiled program. <placeholder-5> is created to represent the result:

```
Returning: <placeholder-5> <== (+ <placeholder-3> ...)
Returning: <placeholder-5> <== (apply + (map square ...))
Returning: <placeholder-5> <== (sum-of-squares ...)
```

<Placeholder-5> represents the result of the computations that will occur at run time. In summary, the partial evaluation process is quite simple: The program executes normally, except that operations whose inputs are not available are delayed until run time by adding instructions to the compiled program.

2.2 Implementation

Compile time interpretation is applicable to a variety of programming languages. The prototype compiler and the examples in this report were implemented for the Scheme dialect of Lisp. Lisp is particularly amenable to these techniques in that all data abstraction mechanisms built-in to the language are type-independent. This allows placeholders to propagate through data structures that would normally have contained numbers. This approach would be difficult to implement in a strictly typed language such as Pascal.

Any Lisp interpreter can be converted for use as a compile time interpreter by adding support for the placeholder data-type. The lowest-level numerical primitives (those that depend on the actual value represented by a placeholder) must be modified[1] to check whether any of their arguments are placeholders. If any of the arguments are placeholders, the instruction adds itself to the list of instructions that must be executed at run time, and then returns a new placeholder to represent its result. Using placeholders to represent intermediate results has proven to be a simple and elegant technique for performing partial evaluation.

2.2.1 Interface to the Lisp System

The compiled data-independent programs are incorporated into the Lisp system as primitive procedures.[2] The Lisp system provides the control structure, executing calls to the optimized data-independent primitives as needed. In this way, the data-dependent portions of a program are handled via the powerful control mechanisms of Lisp, while the data-independent numerical computations are compiled into high-performance primitives.

Whereas the input and output of the original program was via abstract

[1]The **ADVICE** feature provided by **MIT-SCHEME** is a convenient mechanism for implementing this extension. Alternatively, variables such as + can be redefined as extended arithmetic procedures.

[2]Primitive procedures, such as +, are the lowest-level routines provided by the system. In the particular case of CScheme, these procedures are expressed in C and then added to the source code of the Scheme system itself.

Lisp data-structures, the input and output of the compiled primitives is via vectors of numerical values for the input and result placeholders. My compiler automatically generates a set of interface procedures to support the input and output data structures of the original Scheme program. These interface procedures extract values for the input placeholders from the run time data structures, and create result data structures that contain the values computed for the result placeholders.

Programs that manipulate the input and output placeholder vectors directly avoid the performance penalties associated with creating and referencing the interface data structures. To support this style of programming, it is desirable to arrange the ordering of the placeholder values within the input and output vectors such that the *result vector* produced by one primitive can be directly used as the *input vector* for the next primitive to be invoked. The prototype compiler automatically arranges for this ordering in situations where the input and output Lisp data structures have the same format. This has proven to be a particularly effective mechanism for creating high-performance iterative loops, in which the outputs of one iteration serve as the inputs for the next iteration.

2.3 Code Generation

The prototype compiler expresses the compiled program in the form of a low-level C program. *Register allocation* techniques are used to allocate run time storage for placeholder values. If a value is to be used more than once, it is stored in memory until it is no longer needed. However, if a placeholder value is only referenced once, the computation to produce that value is *in-line coded* directly at the place where the reference would have occured. This generates large numerical expressions, giving the C compiler an opportunity to efficiently utilize the hardware registers.

Traditional compilation techniques are used to further improve the performance of the partially evaluated program. From the standpoint of the high-level language program, the partial evaluation process performs many optimizations, including common subexpression elimination of data references and constant propagation. However, the elimination of compound data structures exposes the underlying numerical computation, presenting new opportunities for optimization.

Symbolic manipulation is used to simplify the numerical expressions produced by partial evaluation. For example, expressions such as `(+ y (* -1 x))` are simplified into `(- y x)`. Opportunities for simplifications often arise when high-level data structure operations are combined, as in the subtract-vectors operation shown in Figure 2.2. These optimizations are often not noticed by the programmer when they do not apply uniformly to all elements of a data structure, or when the operations being combined are located in logically separate portions of the program.

```
(define (subtract-vectors a b)
  (add-vectors a
               (scale-vector -1 b)))
```

Figure 2.2: Operations that are built by combining abstract operations can often be simplified through symbolic manipulation of the low-level computation. In this version of subtract-vectors, the addition and scaling operations can be combined to form a subtraction.

It is sometimes possible to determine the result of an operation even before all of the inputs are available. For example, `(* 0 x)` can be easily reduced to 0, which may in turn allow the result of some other operation to be determined. Symbolic simplification increases the opportunities for this sort of optimization, particularly in situations in which 0 can be derived as a result, as in `(- x x)`. *Constant folding* is used to simplify instructions such as `(* 5 x 6)` into `(* 30 x)`.

The instruction simplification process sometimes eliminates all references to the value of a placeholder. For example, simplifying `(- (FETCH 3) (FETCH 3))` might eliminate all references to the value of placeholder 3, allowing the computation of a value for placeholder 3 to be eliminated as well. *Dead code elimination* is used to eliminate those computations which are not needed to produce values for the result placeholders. In some cases, the programmer is able to *declare* that some of the result placeholders are not needed, allowing additional portions of the computation to be eliminated. This becomes particularly important when general tools are employed to solve a specific problem.

2.4 Data-Dependent Control Flow

Conditional branches present a problem for the compile-time interpreter: when the predicate of an IF statement evaluates to a placeholder, the interpreter does not know whether to execute the consequent or the alternative. I refer to computations such as this as *data-dependent* because the flow of control can not be predicted at compile time. Data-dependent computations are generally composed of a set of data-independent regions, which are combined using control constructs that depend on intermediate results of the computation. The techniques described earlier in this Chapter provide a mechanism for identifying and optimizing the data-independent regions of a program. As mentioned earlier, one way to handle data-dependent programs is to compile only the data-independent regions, relying on the LISP system to determine the flow of control.

In some situations it is not practical to use Lisp itself to control program execution. Some control functions can be moved into the compiled code by building mechanisms for handling data-dependent control transfers on top of the basic compile time interpreter. In contrast to the flexibility available in specifying data-independent routines, the mechanisms for specifying control flow are a limited subset of those available in Scheme. Data-dependent control flow is a topic that deserves far deeper consideration, and is an excellent starting point for future work. However, the techniques described here are sufficient to represent most numerically-oriented scientific computations.

2.4.1 Primitive Operations

The simplest technique for handling control transfers is to hide them in primitive numerical operations. This is done for simple comparison operations, such as those associated with **MIN**, **MAX**, and **ABS**. Although these functions require a data-dependent control transfer to compute their result, they are viewed as simple numerical operations. At run time, these operations are supported either by hardware, or by high-performance subroutines.[3] At

[3]In fact, several modern ALU chips are providing operations such as MIN and ABS directly, thereby simplifying instruction sequencing.

compile time, when one of these primitive numerical operations encounters a placeholder, it returns a new placeholder to represent its result, just as any other low-level numerical operation does.

2.4.2 Selection

Mathematical functions are sometimes defined using conditional predicates to divide up the domain. For example, absolute value may be defined using conditional predicates as follows:

$$abs(x) = \begin{cases} x & \text{when } x \geq 0 \\ -x & \text{when } x < 0 \end{cases}$$

To allow the definition of these functions, it is necessary to introduce some form of control over the run time flow of program execution. The conditional control flow required to support division of the domain is relatively straightforward: Provided that the equation is not recursive, domain division can be viewed as *selecting* among various routines. In the prototype compiler, support for these selection operations is provided by augmenting the definition of the `IF` statement. (For the purposes of this document, this augmented version of `IF` is referred to as `RUNTIME-IF`).

The predicate of a `RUNTIME-IF` statement is evaluated at compile time. When the predicate is not a placeholder, `RUNTIME-IF` behaves just as `IF` does, evaluating *either* the consequent or the alternative. However, when the predicate is a placeholder, *both* the consequent and the alternative are evaluated at compile time, generating compiled code for both possibilities. A conditional branch is generated to select at run time whether to execute the code associated with the consequent or the code associated with the alternative.

The execution of *both* the consequent and the alternative at compile time places some restrictions on the structure of the program. Specifically, side-effects performed by both branches of the `RUNTIME-IF` will take place. Ideally, compilation techniques could be used to detect these side-effects and perform them at run time as well. For simplicity, I require that the programmer declare that it is permissable to execute *both* branches at compile time. This is not a significant restriction since selection operations rarely perform side-effects.

2.4.3 Iteration at Run Time

The most important control paradigm in numerical programs is iteration. Many iterative loops can be executed at compile time, replacing the loop with the numerical operations it invokes. However, sometimes it is not practical to evaluate loops at compile time. For example, when the number of iterations to be performed depends upon numerical results of the computation, it is necessary to perform a conditional control transfer at run time. The compile time interpreter has been extended to provide support for iterative control transfers, relaxing the requirement that compiled programs be data-independent.

Partial evaluation is used to specialize *one iteration* of the loop at compile time. The specialized version of the loop is then executed repeatedly. The ordinary procedure-call mechanism of Scheme can not be used directly to control this run-time looping, since procedure calls are eliminated by the partial evaluator at compile time. A new construct, called an *entry-point*, has been introduced to allow programs to perform control transfers at run time. As illustrated in Figure 2.3, defining an entry-point creates a continuation-like procedure, which causes a branch back to the entry-point when invoked.

Several complications are introduced by the entry-point control mechanism. For example, if the *structure* of the data will be different for the next iteration through the loop, then the specialized code generated for the first iteration cannot be re-used to perform the next iteration. More sophisticated compilation techniques could help detect this situation. For now, I simply stipulate that by requesting partial evaluation of the body of a loop, the programmer is implicitly declaring that "the structure of the data present during the first iteration of the loop is identical to the structure that will be present during subsequent iterations."

```
(define-runtime-entry-point ((<var-name1> <initial-value1>)
                             (<var-name2> <initial-value2>)
                             ...)
  (lambda (return-to-entry-point)
    (let ((results (data-independent-computation)))
      (runtime-if (done-yet? results)
         results
         (return-to-entry-point <new-var-value1>
                                <new-var-value2> ...)))))
```

Figure 2.3: Syntax of define-runtime-entry-point. An iterative entry-point is created in the run time instruction stream. Any time the return-to-entry-point procedure is called, the run time flow of program control returns to the start of the loop using the new variable values.

Complex control mechanisms can be created by taking advantage of the fact that *return-to-entry-point* is a first-class Scheme procedure. For example, the EXPonent procedure may be defined to invoke a global *arithmetic-error* procedure whenever its argument is out of range. An adaptive integrator can rebind *arithmetic-error* to be a *return-to-entry-point* procedure that retries the computation using a different step size.[4] This mechanism allows other routines that use EXP to specify a different error behavior, thereby eliminating the need to have different exponent procedures for each error handling methodology, allowing EXP to become a general library function. Use of this technique is illustrated in Appendix B.

[4]The idea of using first-class return-to-entry-point procedures was motivated by the adaptive runge-kutta integrator in the Dynamicist's workbench, a large scientific software system being developed by the MIT Math and Computation Project. This integrator uses Scheme's call-with-current-continuation construct to implement the step-size changing technique.

2.4.4 Conditional Control Transfers: Implementation Details

The variables specified in the entry-point variable list are used to pass information from one iteration of the loop to the next. New placeholders are created to represent each of the loop variables. At run time, when the loop entry-point is first encountered, these variables are assigned their initial values. Each time the loop is restarted, these variables take on new values as specified by the arguments to return-to-entry-point.

Conditional transfers of control are implemented by combining the use of runtime-if with the entry-point control transfer mechanism. Implementing this combination is rather tricky: Runtime-if must execute both the consequent and the alternative in order to generate code for both branches. However, if the *consequent* invokes a return-to-entry-point procedure, the alternative would be bypassed. To avoid this problem, return-to-entry-point only causes a branch back to the entry-point at run time; at compile time, return-to-entry-point returns an abort tag to the nearest runtime-if, allowing the *alternative* to be executed. This form of conditional control transfer provides sufficient functionality to represent most loops that occur in numerical programs. In particular, the full functionality of the traditional `WHILE` and `FOR/NEXT` iteration constructs are supported. Figure 2.4 demonstrates the use of these mechanisms in the implementation of the iterative factorial function.

```
(define (Fac initial-n)
  (define-runtime-entry-point ((n initial-n) (result 1))
    (lambda (fac-loop)
      (runtime-if (= n 0)
                  result
                  (fac-loop (- n 1) (* result n))))))

;;;Compiled Program for (fac <placeholder-1>) :

(INPUT 1) ;placeholder-1 represents the value of initial-n

; Initialize the loop variables
(ASSIGN 2 1) ;placeholder-2 is the loop-variable RESULT
(ASSIGN 3 (FETCH 1)) ;placeholder-3 is the loop-variable N

; The fac-loop entry point itself
LABEL_1:
(ASSIGN 4 (= (FETCH 3) 0)) ;evaluate the runtime-if predicate
(BRANCH (FETCH 4) CONSEQUENT_1) ;select the consequent if predicate true

ALTERNATIVE_1:
(ASSIGN 5 (* (FETCH 2) (FETCH 3))) ;;;(* n result)
(ASSIGN 6 (- (FETCH 2) 1)) ;;;(- n 1)
; Update loop variables for return to entry point
(ASSIGN 2 (FETCH 5)) ;result
(ASSIGN 3 (FETCH 6)) ;N
(GOTO LABEL_1) ; return to entry point

CONSEQUENT_1:

(RESULT 2) ; the result is the value of placeholder 2
```

Figure 2.4: An iterative implementation of the *Factorial* function using runtime-if and entry-points. Invoking the *return-to-entry-point* procedure FAC-LOOP causes a branch back to the entry point to appear in the compiled program. New placeholders are allocated for each loop variable, allowing arguments to be passed.

Chapter 3

Applications

I have applied these techniques to several numerically oriented scientific problems. These problems were chosen from active research at MIT, providing a "real-world" demonstration of the applicability of partial evaluation to scientific computation. Scheme programs implementing the N-body algorithm, the solution to Duffing's equation, and the translation operator for the Multipole Method[Zhao] were taken directly from code in use by researchers. In this chapter, I first describe how my compiler was used in each application, and then present performance measurements showing the performance gains attained by the compiler.

I have implemented all of the partial evaluation techniques described in Chapter 2, including runtime-if and entry points.[1] In addition, the prototype compiler includes a constant folder, a dead-code eliminator, and a storage allocator.[2] The compiler generates low-level C programs which are incorporated into the Scheme system as high-performance primitive operations.

[1]Type propagation within placeholder values has not been fully implemented. For now, the values represented by placeholders are assumed to be of type 'floating-point.

[2]The dead-code eliminator and storage allocator used in the prototype compiler are restricted to purely data-independent programs. Eventually, I plan to merge these partial evaluation techniques into an existing Lisp compiler, thereby providing the full range of conventional compiler optimizations.

3.1 The N-body Problem

3.1.1 Overview

The N-body problem involves computing the trajectories of a collection of N particles which exert forces on each other. This very important problem arises in particle physics, astronomy, and space travel. For example, our solar system is a 10 particle system in which the forces are due to gravitational attraction. An N-body program written in Scheme by Gerry Sussman is used as a starting point for the compilation process.[3] This program, provided in Appendix C, makes liberal use of abstraction mechanisms, including higher-order procedures, lists, vectors, table lookups, and set operations.

In order to simulate future particle motion, the program integrates the forces that the particles exert on each other over time. The *integration-step* routine takes an initial state of the planets, and produces a new state that corresponds to one time-step later. This routine is then repeated, thereby advancing the system forward in time. The prototype compiler was used to create a specialized version of the integration-step procedure.

3.1.2 Describing the problem to the compiler

The input data structures for the integration-step procedure were created at compile time. Since the positions and velocities of the planets are not known at compile time, these were represented using placeholders. The mass of each planet is known and does not vary appreciably with time, allowing numerical values for the masses to be included directly in the compile time data-structures. The compiler took advantage of the availability of these numerical values by performing some parts of the computation in advance. For example, since Pluto is very small relative to the other planets, its mass was approximated as *zero* in the compile time data-structures. The compiler propagated this piece of information throughout the program, eliminating

[3] A variant of this program was used in [Miller] to demonstrate parallel computing using *futures*.

numerous computations.

3.1.3 Measurements

Several measurements were taken to determine the effectiveness of the various code-generation optimizations. Tests were run for both the 6-body problem and the 9-body problem,[4] using both the runge-kutta (RK) and the Stormer (ST) integration methods. As shown in Figure 3.1, knowing the masses of the planets in advance allows the compiler to eliminate hundreds of instructions. This savings comes in part from performing portions of the computation in advance, and in part from taking advantage of the fact that Pluto's mass is zero.

Variation in Number of Instructions

Problem Description	Mass Unknown	Mass Known	Savings
6-body RK	2125	2005	120 (5.6%)
9-body RK	4645	4357	288 (6.2%)
6-body ST	1663	1465	198 (11.9%)
9-body ST	2845	2521	324 (11.4%)

Figure 3.1: Selecting the masses of the planets at compile time allows some computations to take place in advance, reducing the number of instructions that need to take place at run time. (The time-step was also chosen at compile time.)

In Figure 3.2, I present measurements that were taken to determine the effects of the various post partial-evaluation optimizations. The results show that for this set of problems, traditional compiler optimizations had little effect. The only significant improvement was in the case of the runge-kutta

[4]In astronomy, the 6-body and 9-body problems are of particular interest. The 6-body problem is interesting because it includes only the outer planets and the sun, allowing questions of the long-term stability of the solar system to be investigated. The 9-body problem describes the motion of our solar system, excluding Mercury. Mercury is excluded because its high eccentricity necessitates the use of an extremely small integration step-size that makes long-term integrations impractical.

integrator with the time-step chosen at run time. This improvement might have resulted from the simplification process noticing that the derivative of the mass is always zero, allowing dead-code elimination to remove parts of the computation. However, this is purely supposition as I have not investigated the source of this improvement.

The prototype compiler generates low-level programs expressed in the language C. Wherever possible, operations are in-line coded so as to allow the C compiler to make effective use of the processor registers. Storage cells are used to store results which will be re-used elsewhere in the program, as well as the inputs and outputs of the computation. Figure 3.3 shows the number of operations which were in-line codable, as well as the number of storage cells which were needed. The Stormer integrator used more storage cells than the the runge-kutta integrator because the Stormer integrator takes as input the derivatives obtained during the thirteen previous integration steps. Notice that for the runge-kutta integrator, register allocation techniques were sufficiently effective that the number of storage cells required is far less than the number of results that are actually stored in them.

Effectiveness of Optimizations			
Problem Description	Initial Operation Count	Final Operation Count	Savings due to Optimizations
Time-step chosen at run time			
6-body RK	2206	2011	195 (8.4%)
9-body RK	4732	4363	369 (7.8%)
6-body ST	1480	1477	3 (0.2%)
9-body ST	2542	2539	3 (0.1%)
Time-step chosen at compile time			
6-body RK	2008	2005	3 (0.15%)
9-body RK	4360	4357	3 (0.07%)
6-body ST	1466	1465	1 (0.06%)
9-body ST	2522	2521	1 (0.03%)

Figure 3.2: Measuring the effectiveness of optimizations. RK denotes the runge-kutta integration method; ST denotes the stormer integration method. (The masses were chosen at compile time.)

Code Generation Statistics				
Problem Description	Number of Operations	# of In-line Codable opns	# of re-used results	# of Storage Cells Required
Time-step chosen at run time				
6-body RK	2011	1392 (69%)	619 (31%)	216
9-body RK	4363	3057 (70%)	1306 (30%)	369
6-body ST	1477	1311 (89%)	166 (11%)	635
9-body ST	2539	2196 (86%)	343 (14%)	973
Time-step chosen at compile time				
6-body RK	2005	1386 (69%)	619 (31%)	215
9-body RK	4357	3051 (70%)	1306 (30%)	368
6-body ST	1465	1305 (89%)	160 (11%)	628
9-body ST	2521	2187 (87%)	334 (13%)	965

Figure 3.3: Code Generation Statistics for the N-body problem.

3.2 The Multipole Method Translation Operator

3.2.1 Overview

The multipole method is used to approximate force interactions involving a large number of particles. The method, as described in [Zhao], involves dividing space up into a quadtree-like tree of cubes. Part of the force approximation involves propagating information up the tree from a cube to its parent. A significant portion of the computation time is spent evaluating translation operators.

The prototype compiler was used to compile a specialized version of the translation operator. A Scheme implementation of this operation was taken from a program written primarily for people to understand. As such, the program does not take advantage of special cases in the multipole expansions, such as terms that are known to have exponents of zero or one. As shown in Figure 3.4, the compilation optimizations were able to detect these special cases, providing a significant performance improvement. The parameter P, which denotes the number of terms in the multipole expansions, was chosen at compile time.[5] In contrast to the N-body problem, post partial-evaluation optimizations provided significant improvements for the Translation Operator.

3.3 Duffing's Equation

To demonstrate the compilation of data-dependent programs, an adaptive runge-kutta integrator was used to integrate a one period evolution of the variations and derivatives of Duffing's equation. This program was taken from Hal Abelson's work on automatic characterization of the state space of

[5] $P = 3$ is commonly used for benchmark purposes. For large P (above 10), the growth in code size makes compilation of the entire translation operator impractical. For large P, either a smaller segment could be compiled, or else some of the loops could be left intact.

Effectiveness of Optimizations: Translation Operator			
Problem Description	Initial Operation Count	Final Operation Count	Savings due to Optimizations
Time-step chosen at run time			
Translation, P=3	292	114	178 (60%)
Translation, P=6	3040	1698	1342 (44%)

Figure 3.4: Measuring the effectiveness of optimizations on the translation operator of the multipole approximation method. The compiler detected special cases such as exponentiation to the zero power, making it feasible for the program to be expressed in an elegant fashion that does not include checks for these special cases.

Code Generation Statistics: Translation Operator				
Problem Description	Number of Operations	# of In-line Codable opns	# of re-used results	# of Storage Cells Required
Translation, P=3	114	90 (79%)	24 (21%)	23
Translation, P=6	1698	1582 (93%)	116 (7%)	115

Figure 3.5: Code generation statistics for the Multipole Method translation operator.

Duffing's equation[6]. The program, provided in Appendix B uses runtime-if and entry-points to express both the adaptive integrator and a control loop that iterates for one period.

As I did not fully implement dead-code elimination and register allocation for data-dependent computations, only constant-folding and a few symbolic simplifications were performed. Partial evaluation produced 694 instructions, 12 (1.7%) of which were eliminated by compiler optimizations. It is interesting to note that of the 682 remaining instructions, 639 (93.7%) form a data-independent region. The relatively large size of this data-independent region suggests that my focus on the optimization of data-independent regions is appropriate.

[6]Duffing's equation is presented in Appendix A

3.4 Performance Measurements

The programs generated by the prototype compiler were integrated into the MIT Scheme system[7] as primitive operations. The chart in Figure 3.6 presents performance measurements for the applications described above.[8] For comparison, the table also shows the speed-up factor of these primitives relative to the original Scheme programs that they were generated from. The table clearly shows that specialization can provide dramatic performance improvements.[9]

It is also worth noting that the version of the Liar compiler which I was using does not open-code floating-point computations. In some cases, it is possible that implementation of this optimization could improve Liar's floating-point performance by as much as a factor of four.[10] However, performing this optimization would require the availability of type information, obtained either through type-inference or from declarations made by the programmer. This represents a fundamental difference between the two approaches: the conventional approach has the programmer put the declarations on *variables in the program*, imposing restrictions on the types of data that a routine can operate on, whereas the partial-evaluation approach allows general routines to be written, and then automatically specializes them based on declarations regarding the structure of the *data* for the particular problem at hand.

[7]Specifically, MIT CScheme release 7 with Liar compiler version 4.38, running on a Hewlett-Packard 9000 Series 350 with 16 Megabytes of memory. The timings presented do not include garbage collection time.

[8]Timings for Stormer integration of the N-body problem are not provided. A bug in the unix C compiler prevented the compilation of these C primitives. It is not clear why this problem arose, considering that the Stormer primitives are actually *smaller* than the primitives that use the runge-kutta integration method. The runge-kutta primitives had no trouble compiling.

[9]The Duffing's equation measurements for the specialized primitives are slow partially because in-line coding was not performed on the computation.

[10]It is possible that if Liar performed this optimization, the performance of the special-purpose primitives would also improve, as it would no longer be necessary to "unbox" the input floating-point numbers, or to "box" the outputs.

Performance Measurements					
Problem Desc.	Interpreted CScheme	Compiled CScheme	Specialized Primitive	Speed-Up over Interpreted	Speed-up over Compiled
6-Body RK	1.7	0.76	0.020	85	38
9-Body RK	3.4	1.50	0.038	89	39
Xlate P=3	0.26	0.022	0.002	130	11
Xlate P=6	2.76	0.28	0.011	250	25
Duffing	26.1	4.04	0.53	49	7.6

Figure 3.6: Timings of the sample applications. It is clear that the specialized primitives are significantly faster than the Scheme programs they were generated from. For the N-body problem, both the time-step and the masses of the planets were chosen at compile time. Note that in-line coding was not performed on the Duffing's equation program.

Chapter 4

Parallel Computation

Partial evaluation can play an important role in programming parallel computers. By eliminating inherently sequential data structure references and conditional branches, partial evaluation exposes the low-level parallelism inherent in a computation. I have implemented several analysis and scheduling programs that detect and analyze this parallelism in data-independent computations. Without having to change the way that the algorithm is specified by the programmer, my programs have been able to detect virtually all of the parallelism available in the N-body problem. My programs also analyze some of the trade-offs involved in the design of a parallel architecture that can effectively utilize this parallelism.

I demonstrate the effectiveness of these techniques by using the 9-body problem[1] as an example. I first present measurements of the theoretical maximum amount of parallel execution that can be attained on this problem, and then show how real-world design constraints such as pipelining and the cost of inter-processor communication limit the amount of parallelism that can be effectively exploited.

[1]Specifically, 12th-order Stormer integration of the 9-body gravitational attraction problem, with masses chosen at compile time, and time-step chosen at run time.

4.1 Parallelism Profiles

Figure 4.1 presents a parallelism profile[Arvind] for Stormer integration of the 9-body problem. This profile describes the *maximum* amount of parallel execution that would occur if a computer had an infinite number of processors that could communicate with each other instantaneously. The profile was produced by re-ordering the numerical operations produced by the partial evaluator, such that all computations that can occur in parallel are grouped together into a single machine cycle. Without partial evaluation, sequential data-structure references would be intermixed with the numerical operations, hiding some of the parallelism.

This profile differs from the parallelism profiles that commonly appear in the literature in that it accounts for the fact that some operations require more cycles to complete than others.[2] To illustrate this distinction, a conventional parallelism profile that ignores the latency differences between numerical operations is presented in Figure 4.2. I have found that for computations involving double-precision floating-point operations, the latency differences between numerical operations are large enough to be of fundamental importance.

[2]I derived relative timings for the various arithmetic operations from the latencies of the B3110A/B3120A floating-point chips manufactured by Bipolar Integrated Technologies.

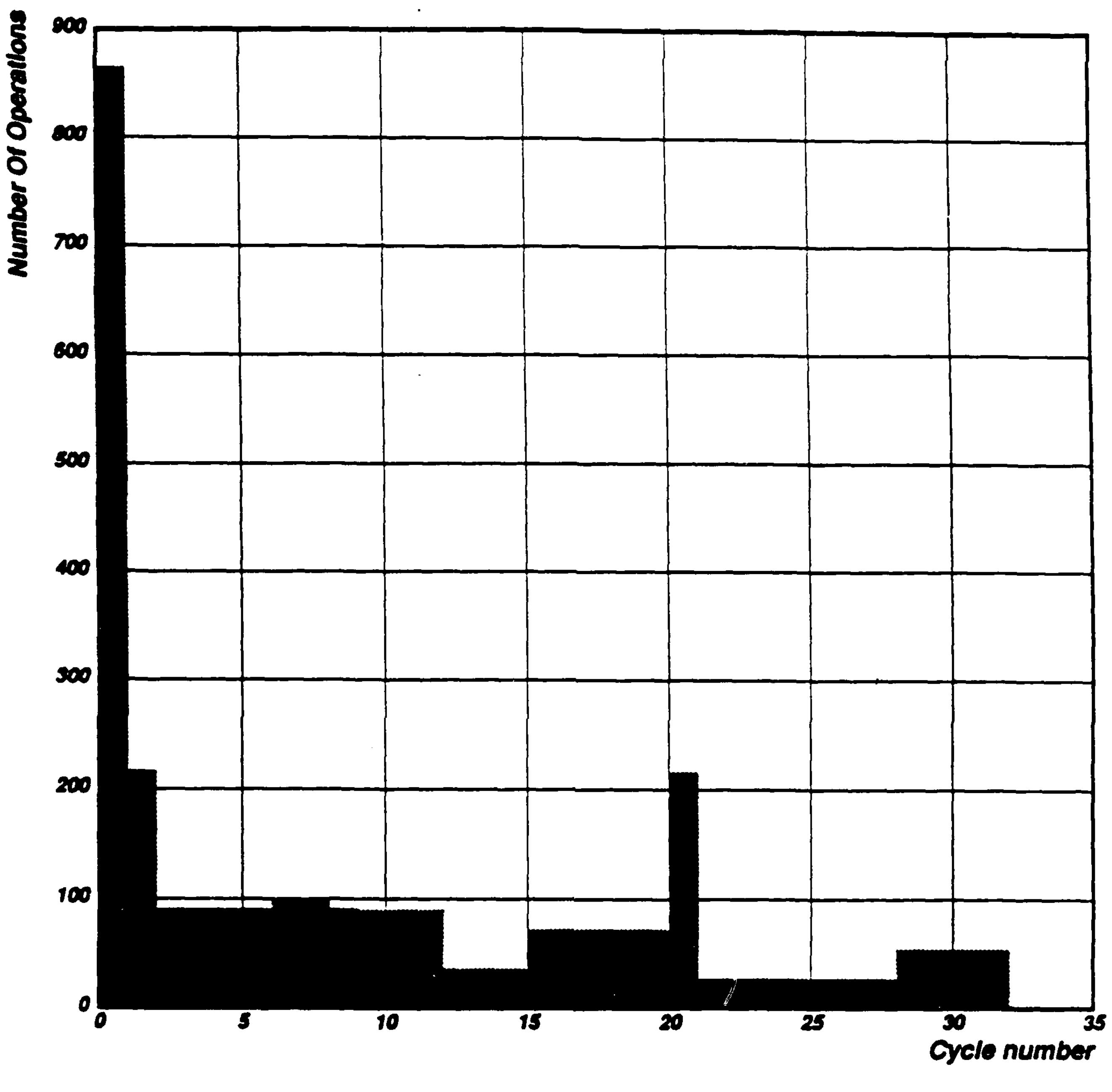

Figure 4.1: Parallelism profile of the 9-body problem. This graph represents the total parallelism available in the problem, accounting for the latency of numerical operations.

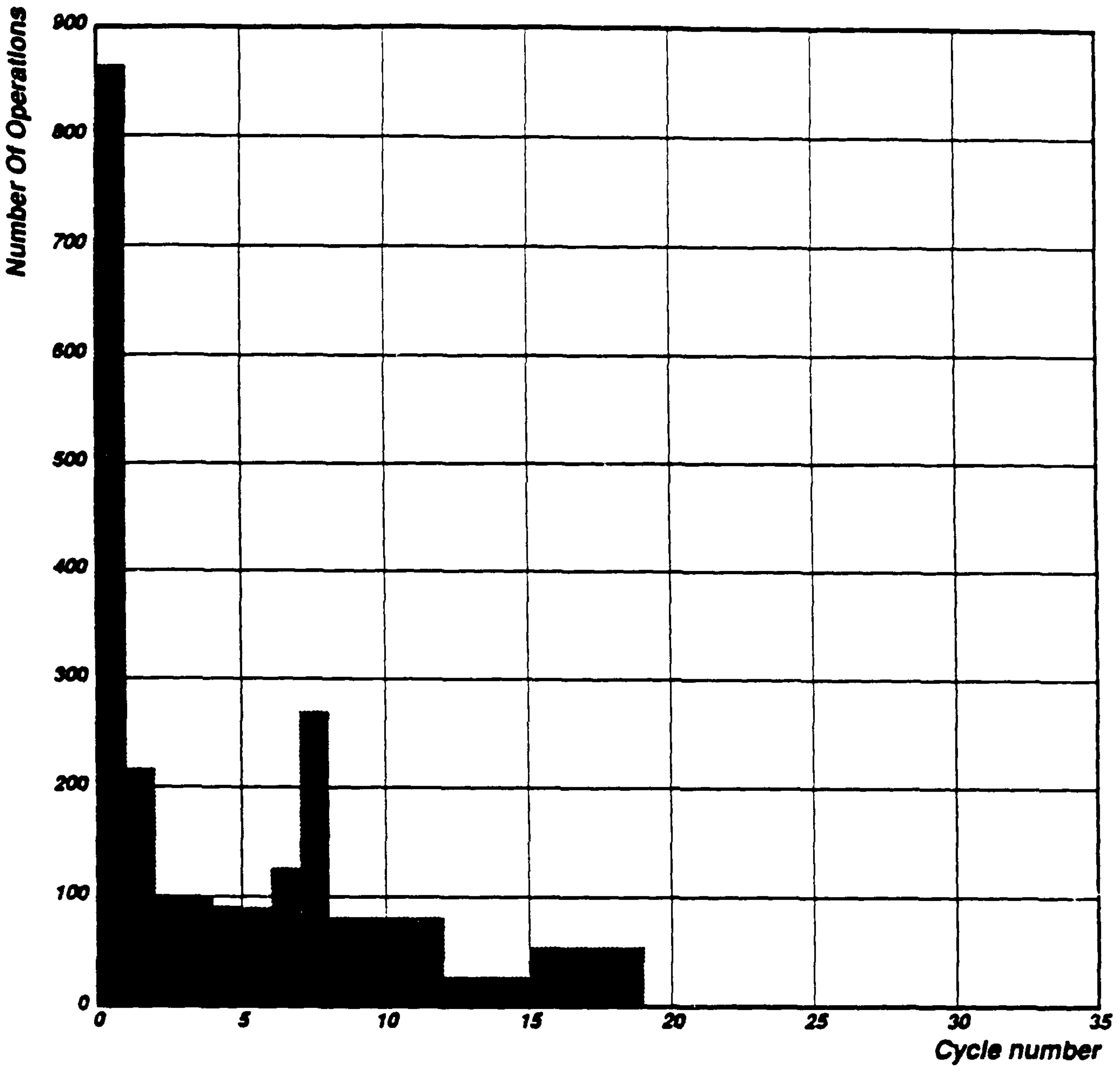

Figure 4.2: Conventional parallelism profile of the 9-body problem. This graph does not account for the varying latency of numerical operations. It is clear from comparison with Figure 4.1 that the latency of individual numerical operations is an important factor that must be accounted for.

4.2 Practical Considerations

4.2.1 Efficiency

For the 9-body problem, fully utilizing parallelism in the fashion suggested by the parallelism profile would require 865 processors. Using only one processor, this computation requires 3094 cycles, whereas using 865 processors, the computation requires only 32 cycles. Thus in the ideal world where communication is free, parallelism improves performance by a factor of 97. However, in the 32 cycles it took to perform the 9-body computation, those same 865 processors could have performed 27,680 computations. In other words, these processors were used with only about 11% efficiency. From a cost vs. performance standpoint, this approach does not make much sense.

A more efficient approach is to distribute the parallelism evenly among the machine cycles. For example, in the parallelism profile of the 9-body problem, there is a great deal of parallelism available in the first cycle, but not very much in some of the later cycles. The parallelism can be distributed more uniformly by moving some of the operations from the first cycle into later cycles. As long as these delayed operations are performed before their results are actually needed, the program will execute in the same amount of time, but will require fewer processors. In the next section, I describe a program that I have implemented to perform this parallelism redistribution.

4.2.2 Pipelining

Technological considerations often lead to overlapping the execution of successive instructions within a single processor. The parallelism profile analysis presented above was based on the assumption that the result of an instruction that finishes executing in one cycle can be used immediately in the following cycle. Unfortunately, this assumption is not valid in the presence of pipelining. Figure 4.3 shows that for a 3-stage pipeline, the result of an instruction which is initiated in cycle 1 will not be available to the instruction that is initiated during cycle 2. Thus even with an infinite number of processors and no communication delays, a machine composed of 3-stage pipelined processors will require about *twice as many cycles* to execute a computation as a non-pipelined machine would.[3]

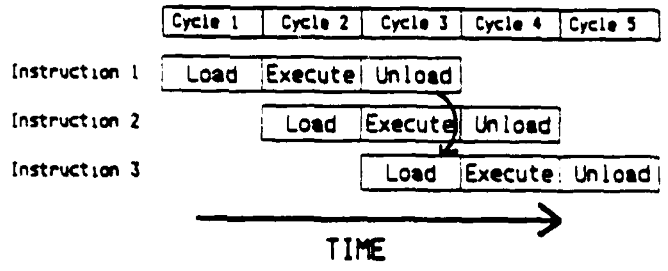

Figure 4.3: A typical 3-stage processor pipeline. During the LOAD stage, the data is loaded into the ALU. The result is computed during the EXECUTE stage, and unloaded from the ALU during the UNLOAD stage. The results produced by instruction 1 are not available to be used by instruction 2, but are available to instruction 3.

Despite this increase in the number of cycles required to execute a program, pipelining is advantageous because it can make it possible to reduce the length of each cycle. In addition, it is possible to use some of the parallelism available in the problem to keep processors from falling idle. In other words, rather than scheduling all available parallel operations into the same cycle on many processors (parallelism in space[WU 87]), it is possible to use a smaller number of processors, and schedule some of the operations during the next cycle (parallelism in time) in order to keep the pipeline busy. This utilizes the individual processors more effectively.

[3]Since some instructions require more than one EXECUTE cycle to compute their result, the processors will sometimes be busy more than half the time. My "twice as many cycles" estimate assumes that operations require one EXECUTE cycle to complete.

On the other hand, the estimate also does not account for the fact that a result must first be unloaded from a processor before it can be loaded into another one. This creates a one cycle cost to moving data between processors, even when there are no communication delays. This effectively increases the minimum number of cycles required to complete the computation. Overall, these two effects tend to cancel each other out.

4.2.3 Communication Latency

In practice, processors can not communicate instantaneously. The time required to move a result from one processor to another limits how soon the result can be used by a subsequent instruction. This has an effect that is similar to increasing the length of the pipeline, as illustrated in Figure 4.4. To counter this effect, some of the parallelism available in the problem can be used to keep the processors busy while they are waiting to receive the results of previous computations.

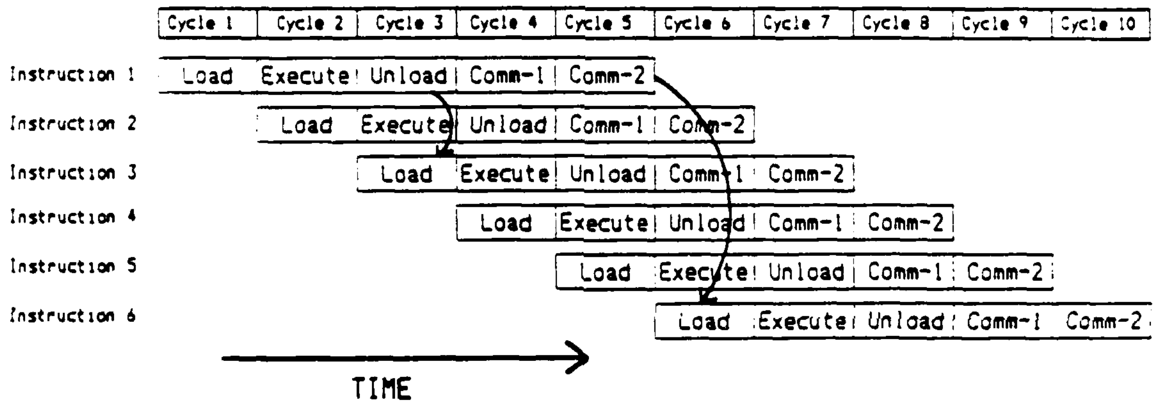

Figure 4.4: A 3-stage processor pipeline with a communication latency of two cycles. As indicated by the arrows, a result produced by instruction 1 can be used within the same processor by instruction 3, but can not be used by other processors until instruction 6.

4.3 A Scheduler for Parallel Programs

I have implemented a program that schedules parallel operations onto multiple processors. Accounting for the effects of pipelining and communication latency, my program increases processor utilization by delaying operations which are not in the critical path of the computation. In other words, if the result of a computation is not needed right away, that computation is delayed until a processor is free to handle it.

4.3.1 Performance Measurements

Figure 4.5 shows the results of applying my scheduling program to the 9-body problem, for a 40 processor system with a 3-stage processor pipeline and a communication latency of one cycle. The figure shows how the parallelism available in the problem has been distributed over the life of the computation, so as to effectively utilize all 40 processors in most of the cycles. Overall, the performance improvement over that of a single *pipelined processor* was a factor of 36, indicating that the processors were used with approximately 90% efficiency.

The ability of the scheduler to effectively utilize the available processors varies with both the number of processors in the system and the communication latency. For the 9-body problem, these variations are summarized by Figures 4.6 and 4.7. These graphs clearly show that communication latency has a direct effect on the maximum speed-up that the scheduler can provide.

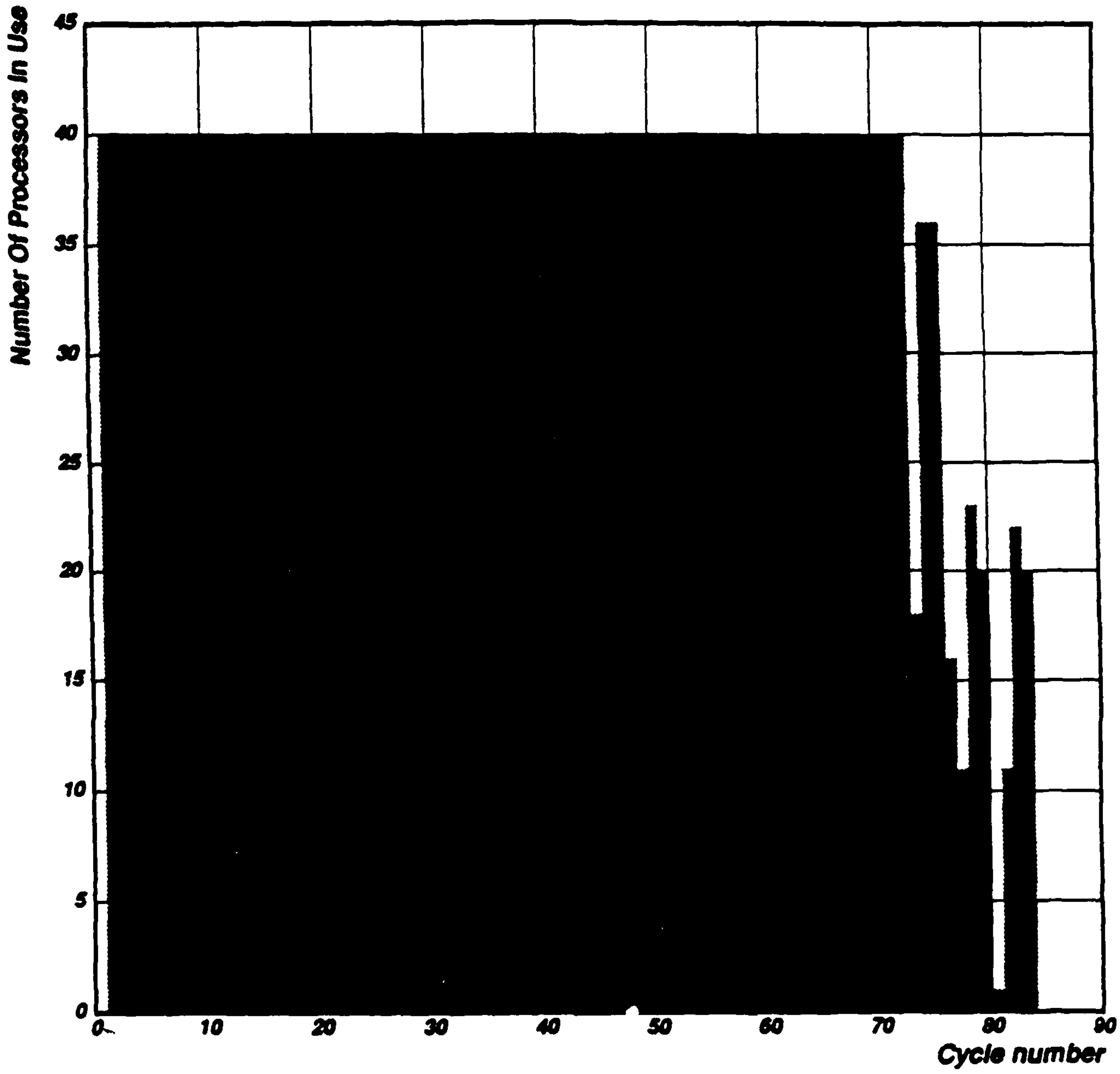

Figure 4.5: The result of scheduling the 9-body problem onto 40 pipelined processors with a communication latency of one cycle. A total of 85 cycles are required to complete the computation. On average, 36.4 of the 40 processors are utilized during each cycle.

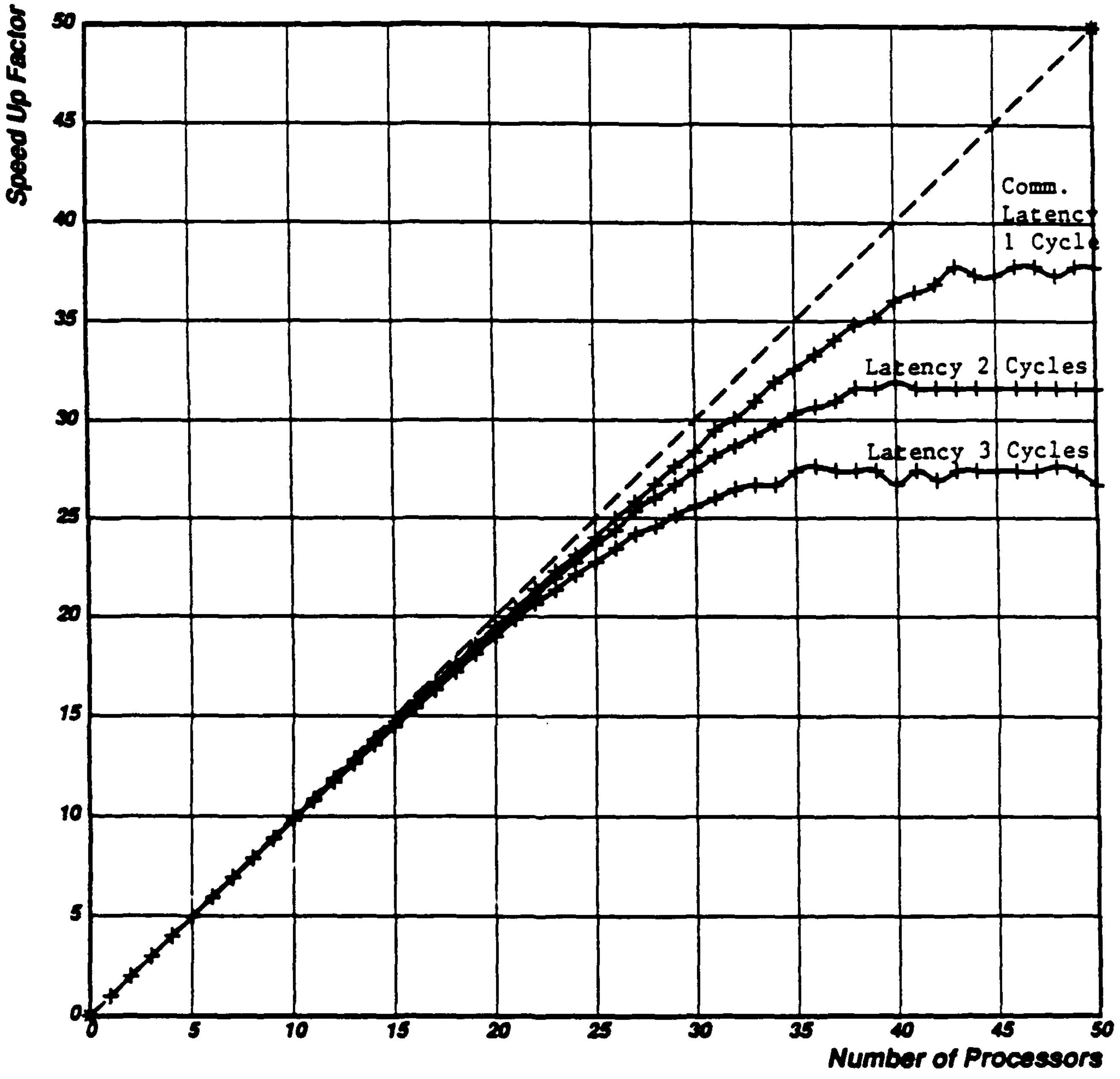

Figure 4.6: Scheduling of the 9-body problem: The graph shows the speed-up factor over a single pipelined processor. The analysis shown is for a system composed of processors employing a 3-stage pipeline.

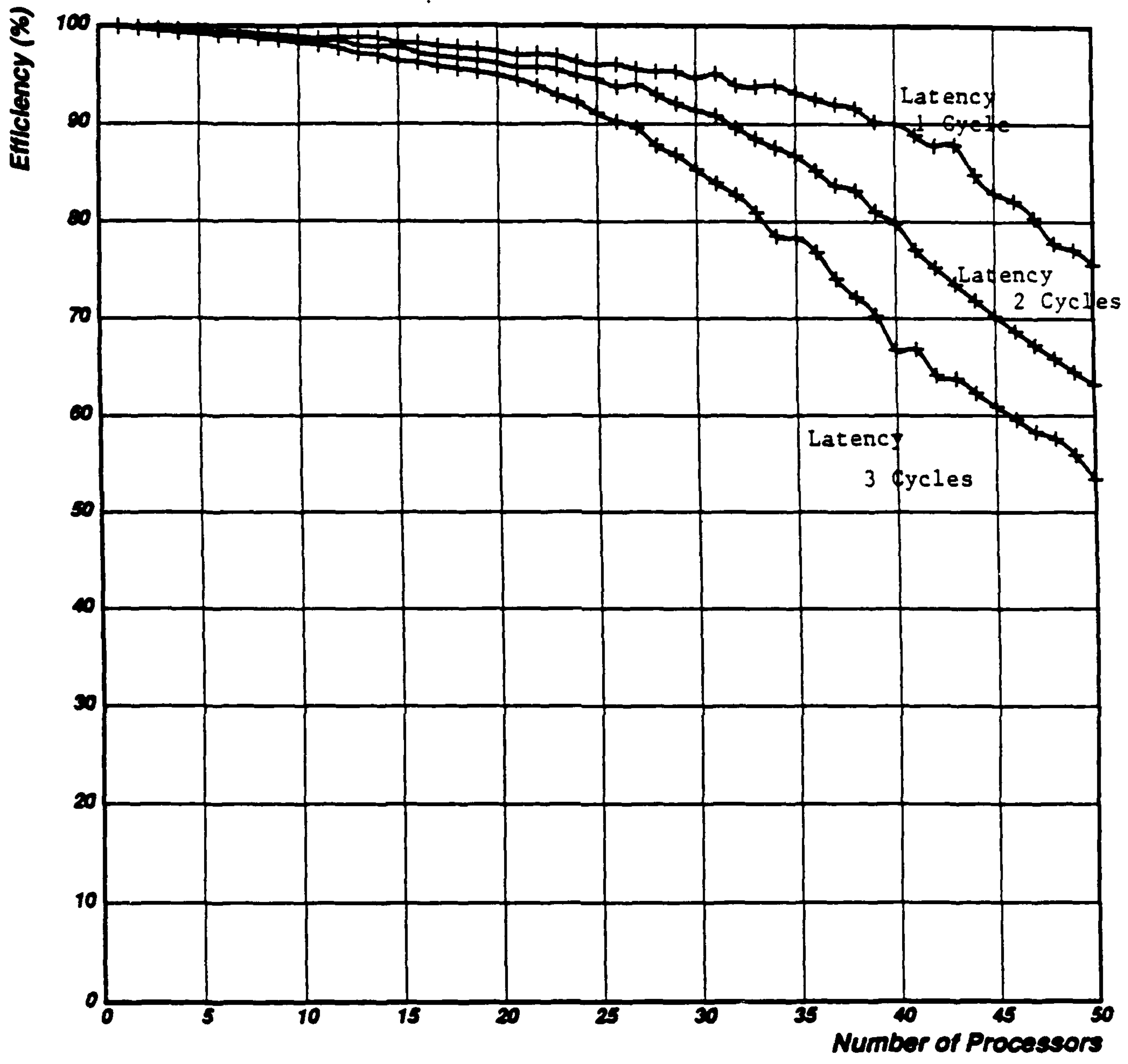

Figure 4.7: Scheduling of the 9-body problem: The graph shows how processor utilization efficiency varies with the number of processors in the system and with communication latency. Efficiency is defined as the ratio of speed-up factor to the total number of processors. The analysis shown is for a system composed of processors employing a 3-stage pipeline.

4.3.2 Implementation Details

The scheduler operates by manipulating a data-flow graph. It starts by computing the latency of every possible path through the graph. These paths are then sorted, allowing the critical-path of the computation to be identified. When the operations are scheduled, priority is given to those operations that lie in the critical-path of the computation. If all available processors are not needed to work on the most critical-path, computations from less critical paths are scheduled.

The problem of scheduling every operation onto the "best processor" at the "best time" is extremely difficult. Rather than trying to find an optimal solution to the problem, I developed a heuristic designed to select a "pretty good" solution. To give a flavor for the algorithm, a brief overview is presented below:

- A set of operations is chosen corresponding to the number of processors that are available. This selection is based on the latency priorities described above.
- Among the "chosen operations", those whose operands have been available long enough to have been transmitted to other processors have lower scheduling priority than those operations whose operands have been produced recently. This gives priority to non-relocatable computations.
- A computation whose operands were produced by a processor will be scheduled in that same processor wherever possible.
- The number of connections between processors is kept to a minimum. When the operands of a computation must be transmitted from one processor to another, the scheduler attempts to choose a pair of processors that have communicated with each other in the past.
- Several heuristics exist for breaking ties. These take into account such factors as the memory usage within each processor, the number of computations that are waiting for a particular result, and the frequency with which processors use the communication network.

4.3.3 Does the scheduler do a good job?

In section 4.2.2 I showed that even with no communication latency, a collection of 3-stage pipelined processors can at best expect to complete a computation in approximately twice the number of cycles shown in the parallelism profile. In the case of the 9-body problem, the parallelism profile has 32 cycles, so in a world of zero-cost communication and an infinite number of processors, we can expect the program to complete in 64 cycles, plus 2 cycles for loading and unloading the pipeline at the start and end of the computation.

For comparison purposes, I ran the scheduler on the 9-body problem for 1000 processors with a communication latency of zero cycles. The resulting program required 66 cycles, which agrees exactly with my estimate of the maximum possible performance. Although this represents an overall speed-up over a single pipelined processor of a factor of **46**, it is rather inefficient to use hundreds of processors to attain this improvement.

In the real world, it takes time to communicate, and this can be expected to decrease the maximum attainable speed-up. In the zero communication cost situation, 1339 values were moved among the processors, 378 of which were used immediately. In other words, the zero-cost communication was taken advantage of in about 28% of the cycles. It is reasonable to expect that increasing the communication latency will decrease the maximum attainable speed-up below a factor of 46 by slowing down the arrival of messages in the "critical-path" of the computation. For a communication latency of one cycle, the scheduler is still able to provide a speed-up factor of **37**, using only 40 processors. Given that the theoretical maximum for this problem is somewhere below a factor of 46, I consider the attained speed-up factor of 37 to be a quite good, especially considering that this speed-up was attained while making good use of the processors (90% efficiency).

4.4 Routing Considerations

The scheduler places computations in processors with a view towards minimizing the routing requirements. However, unlike conventional parallel schedulers, it does not make any assumptions about the topology of the routing network. Although a processor transmits at most one result per processor per cycle, this result can have multiple destinations. After the placement has been performed, the required communication can be mapped onto an existing routing network, making local changes in the placement as needed.

For small numbers of processors (up to about 16), it is feasible to fully interconnect the processors, allowing all communication specified by the scheduler to be implemented directly with a latency of only one cycle. Pipelined multi-stage networks can also be used, but as shown in the speed-up graphs, the latency of these networks is a critical factor limiting the maximum speed-up that can be attained.

4.4.1 Synthesizing a Routing Network

The 9-body problem is of sufficient importance to justify constructing special-purpose computers to solve it. One such computer was in fact constructed several years ago in a joint Caltech/MIT project[Applegate]. Some problems of current interest to astronomers require more computation power than is provided by the existing orrery, leading to the consideration of new approaches.

I propose that a low-latency special-purpose routing network can be constructed to match the routing specifications derived by the scheduling program. A full interconnection of N processors requires N^2 switch-points. Since the placement algorithm attempts to minimize the number of destinations for each processor, it is advantageous to delete those switch-points that correspond to communication paths that will never be used. Figure 4.8 summarizes the number of switch-points that would be required to implement the

9-body problem.[4] This approach provides high-speed, low-latency interconnection at a reasonable price, but at the cost of generality.

[4]This analysis does not account for the communication required to route the outputs back to the inputs for the next iteration. I do not expect this to be a significant problem.

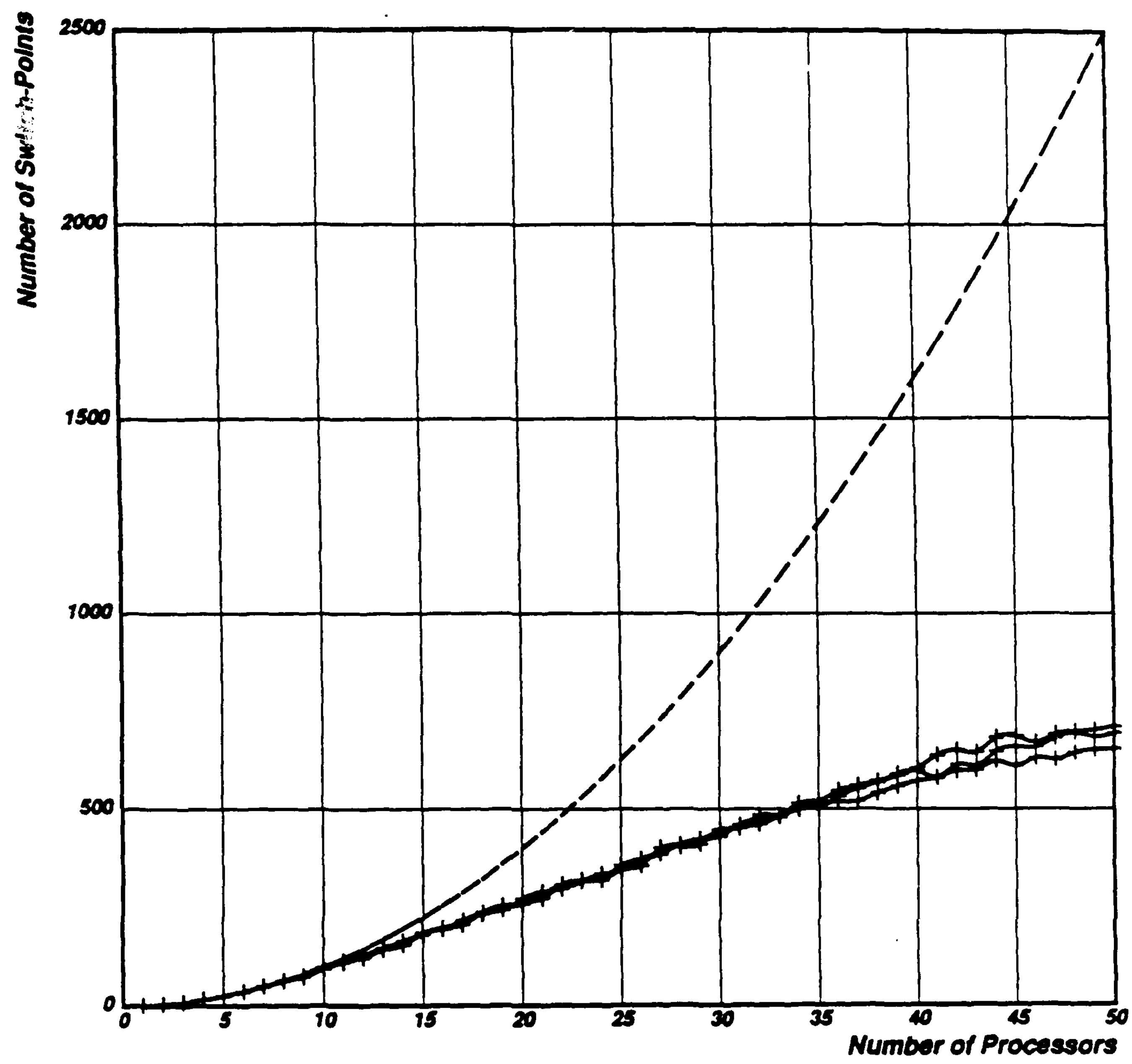

Figure 4.8: Number of switch-points required to implement a special-purpose routing network for the 9-body problem. As the number of processors increases, specialization provides a dramatic reduction in routing network size. The dashed line indicates the number of switch-points in a fully interconnected network, while the three solid lines indicate that the number of switch-points is relatively independent of routing latency.

4.5 Summary

I have shown that partial evaluation can be used to eliminate the barriers to parallel execution, exposing the low-level parallelism inherent in a computation. For the 9-body problem, I have shown that this parallelism can in fact provide significant performance improvements, and have presented a strategy for effectively utilizing this parallelism by scheduling the computation onto a collection of pipelined processors. I have also proposed a cost-efficient technique for constructing high-performance special-purpose computers. The combination of these contributions comprises a strategy for automatically generating and programming a special-purpose computer based on a high-level language specification.

4.5.1 Relation to Previous Work

Many compilers for high-performance architectures use program transformations to exploit low-level parallelism. For instance, compilers for vector machines unroll loops to help fill vector registers.[Padua] Similarly, compileis for VLIW architectures use *trace-scheduling*[Ellis] to guess which way a branch will go, allowing computations beyond the branch to occur in parallel with those that precede the branch. The effectiveness of both of these techniques is limited by their preservation of the user data-structures of the original program. In other words, if the original program represented an object as a vector of vectors, the compiled program will do so as well. This preservation of user data-structures imposes synchronization requirements which reduce the low-level parallelism available to the compiler.

My work focuses on parallelizing those regions of a program which it is feasible to evaluate symbolically at compile-time. This compile-time evaluation allows user data-structures and many conditionals to be eliminated, producing purely numerical programs that contain extremely large basic-blocks. This approach makes it possible to use intermediate results immediately in portions of a program that would not otherwise have been reached even through trace-scheduling, due to conditional tests related to user data-structures. This technique is orthogonal to the trace-scheduling approach: symbolic-evaluation can be used to eliminate conditional tests related to data-structures, producing large parallelizable basic-blocks, while trace-scheduling can be used to optimize across basic-block boundaries.

4.5.2 Suggestions for Future Work

The technique of using symbolic-evaluation to expose parallelism within data-independent regions may be combined with other parallel-programming approaches. For example, this technique can be combined with the *futures* approach of MultiScheme [Miller] by using symbolic evaluation to parallelize computations within a future, thereby allowing futures to be used to program a collection of parallel computers. Similarly, this technique may be used in conjunction with hardware that does dynamic scheduling of data-flow graphs: symbolic-evaluation can be used to create large statically-analyzable nodes within a dynamic data-flow graph. Rather than connecting a collection of relatively simple processors, the parallelism available within each of these statically-analyzable nodes makes it feasible to use dynamic-scheduling hardware to combine a collection of more powerful (parallel) processors.

Chapter 5

Conclusions

Partial evaluation has an important role to play in the compilation of numerically-oriented scientific computations. When sufficient information is available at compile time, the specialization and data abstraction elimination capabilities of partial evaluation provide significant performance improvements over conventional compilation techniques. These improvements leave the programmer free to write abstract, elegant programs that express ideas, leaving the optimization of special cases as a task for the compiler.

Using placeholders to propagate intermediate results has proven to be a simple and elegant technique for performing partial evaluation. The prototype compiler I implemented does a fairly good job of optimizing data-independent regions. A good starting point for future work would be to improve the techniques for combining data-independent regions to form data-dependent programs. For example, by using lexical analysis techniques, a compiler could identify the potential side-effects and lexical references of a program. Those data structures which are side-effected could be created and referenced at run time. Placeholders could then be used to represent run-time data structures as well as numerical values.

Partial evaluation exposes the low-level parallelism inherent in a computation. I have implemented a scheduler that takes advantage of this parallelism to make efficient use of a collection of pipelined processors. I have also shown that communication latency is a critical factor that limits the maximum speedup that can be attained through parallel execution. I have shown that a special-purpose routing network can efficiently provide the low-latency communication required for a particular problem at a reasonable cost. A good starting point for future work would be to investigate routing networks that provide for a mixture of low and high latency communication. For example, by exploiting locality, it is possible for a network to provide some single cycle latency communication paths, together with some multiple cycle paths for long-distance messages.

The combination of the performance gains provided by parallel execution with the order-of-magnitude performance improvement provided by program specialization promises a fundamental improvement in the performance of an important class of abstractly specified numerical programs. I believe that a production quality compiler based on the techniques that I have described would lead to a fundamental change in the way that scientists write programs.

Bibliography

[Applegate] Douglas Applegate et. al, "A Digital Orrery". In *Lecture Notes in Physics #267* - Use of supercomputers in Stellar Dynamics, Springer Verlag, 1986. Reprint of IEEE Trans. on Computers article.

[Arvind] Arvind, David E. Culler, and Gino K.Maa. "Assessing the benefits of fine-grain parallelism in dataflow programs". In *International Journal of Supercomputer Applications*, Vol. 2, No. 3, 1988, pp. 10-36.

[Berlin] A. Berlin and H. Wu, "The Scheme86 Project: A system for interpreting Scheme", Proceedings of the ACM Conference on Lisp and Functional Programming, 1988.

[Ellis] John R. Ellis, *Bulldog: A Compiler for VLIW Architectures,* MIT Press, Cambridge, MA, 1986.

[Halfant] M. Halfant and G.J. Sussman, "Abstraction in numerical methods", Proceedings of the ACM Conference on Lisp and Functional Programming, 1988.

[Komorowski] Henryk Jan Komorowski, "A Specification of an Abstract Prolog Machine and its application to Partial Evaluation". Linkoping Studies in Science and Technology Dissertations, No. 69., 1981, Linkoping University

[Miller] James S. Miller, "Multischeme: A Parallel Processing System Based on MIT Scheme". MIT Laboratory For Computer Science technical report no. TR-402. September, 1987.

[Nagel] Laurence Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Electronics Research Laboratory *Report No. ERL-M520,* University of California, Berkeley, May 1975.

[Padua] Padua, David A., Wolfe, Michael J., *Advanced Compiler Optimizations for Supercomputers*, Communications of the ACM, Volume 29, Number 12, December 1986.

[Schooler] Richard Schooler, "Partial Evaluation As A Means Of Language Extensibility". MIT Laboratory For Computer Science technical report no. TR-324.

[Skordos] P. Skordos, "Multistep methods for integrating the solar system". TR-1055, MIT Artificial Intelligence Laboratory.

[Sussman] G.J. Sussman and J. Wisdom, "Numerical evidence that the motion of Pluto is chaotic". In *Science*, Volume 241, 22 July 1988.

[WU 87] Henry M. Wu, "Performance Evaluation of the Scheme86 and HP Precision Architectures". S.M. thesis, Department of Electrical Engineering and Computer Science, MIT. May 1987. Soon to be published as TR-1103, MIT Artificial Intelligence Laboratory.

[Zhao] Feng Zhao, "An O(N) algorithm for three-dimensional N-body simulations". TR-995, MIT Artificial Intelligence Laboratory.

Appendix A

The Duffing's Equation Program

```
;;;UEDA-VAR-DER-K is the function being integrated in the examples
;;;in Chapter 3.

(define (ueda-var-der-k b k)
  (let ((k- (- k)))
    (lambda (state)
      (let ((t (vector-ref state 0))
            (x (vector-ref state 1))
            (u (vector-ref state 2))
            (del/x/x (vector-ref state 3))
            (del/u/x (vector-ref state 4))
            (del/x/u (vector-ref state 5))
            (del/u/u (vector-ref state 6))
            (dx/dk (vector-ref state 7))
            (du/dk (vector-ref state 8)))
        (let ((g1 (* -3 x x)))
          (vector
           1
           u
           (- (* B (cos t)) (* k u) (* x x x))
del/u/x
(+ (* k- del/u/x) (* g1 del/x/x))
del/u/u
(+ (* k- del/u/u) (* g1 del/x/u))
du/dk
(+ (- u) (* k- du/dk) (* g1 dx/dk))))))))
```

```
;;; The input data structure for the Duffing's Equation example:

(define varder-k-state-vector
  (vector (make-placeholder 't)
          (make-placeholder 'x)
          (make-placeholder 'u)
          (make-placeholder 'del/x/x)
          (make-placeholder 'del/u/x)
          (make-placeholder 'del/x/u)
          (make-placeholder 'del/u/u)
          (make-placeholder 'dx/dk)
          (make-placeholder 'du/dk)))

;;; The actual sequence of code that was compiled:

(end-state (ueda-var-der-k b k)
            varder-k-state-vector)
```

Appendix B

Quality Controlled Runge-Kutta Integrator

```
;;;The following code is an example of the use of runtime-if and
;;;entry-points to implement an adaptive runge-kutta integrator.

;;;It was derived from an integrator written by Hal Abelson and
;;;Gerry Sussman as part of the dynamicists workbench.

;;;The library of vector operations is omitted.


;;;One runge-kutta integration step
(define (rkstep der dydx state h)
  (let* ((h* (scale-vector h))
         (k0 (h* dydx))
         (k1 (h* (der (add-vectors state (1/2* k0)))))
         (k2 (h* (der (add-vectors state (1/2* k1)))))
         (k3 (h* (der (add-vectors state k2)))))
    (add-vectors state
                 (1/6* (add-vectors (add-vectors k0 (2* k1))
                                    (add-vectors (2* k2) k3))))))
```

```
(define (quality-control stepper order)
  (let ((2^order (expt 2 order))
        (error-scale (/ -1 (+ order 1))))
    (let ((halfweight (scale-vector (/ 2^order (- 2^order 1))))
          (fullweight (scale-vector (/ 1 (- 2^order 1)))))
      (lambda (der)
        (lambda (state h-init continue)
          (let ((dydx (der state)))
            (define (loop h)
              (DEFINE-ITERATIVE-ENTRY-POINT ((h h))
                (lambda (return-to-loop)
                  (fluid-let (((access arithmetic-error circuit-package)
                               (lambda () ;;;IF OVERFLOW, TRY SMALLER STEPSIZE
                                 (RETURN-TO-LOOP (/ h 4)))))
                    (let* ((h/2 (/ h 2))
                           (fullstep (stepper der dydx state h))
                           (halfstep (stepper der dydx state h/2))
                           (2halfsteps (stepper der (der halfstep) halfstep h/2))
                           (diff (sub-vectors 2halfsteps fullstep))
                           (err (/ (maxnorm
                                    ((elementwise (lambda (y d)
                                                    (/ d
                                                       (+ qc-error-stop
                                                          (abs y)))))
                                     2halfsteps
                                     diff))
                                   qc-allowable-error)))
                      (RUNTIME-IF
                       (> err 1)
                       (RETURN-TO-LOOP
                        (* qc-safety h (expt err error-scale)))
                       (let ((newh (* qc-safety h (expt err error-scale)))
                             (new-state
                              (sub-vectors (halfweight 2halfsteps)
                                           (fullweight fullstep))))
                         (continue new-state newh)))
                      )))))
            (loop h-init))))))
(define (integrate-for-interval system-derivative
                                initial-state
                                interval
                                h
                                . integ)
  (let ((integrator (if (null? integ) default-integrator (car integ))))
    (let ((integration-step (integrator system-derivative))
         (final-time (+ interval (state-vector-time initial-state))))
      (DEFINE-ITERATIVE-ENTRY-POINT ((h-left interval)
                                     (h-next h)
                                     (state initial-state))
        (lambda (STEPLOOP)
          (RUNTIME-IF (<= h-left 0)
                      state
                      (integration-step
                       state
                       (min h-next h-left)
                       (lambda (next-state next-h)
                         (steploop (- final-time
                                      (state-vector-time next-state))
                                   (RUNTIME-IF (< next-h h)
                                               next-h
                                               h)
                                   next-state))))))))))
```

```
;;;The Integrator used on duffing's equation is a quality controlled runge-kutta-4:

(define rkqc (quality-control rkstep 4))

(set! default-integrator rkqc)
```

Appendix C

The N-body Program

```
;;; This program solves the N-body Problem using the gravitation force
;;;   law with a runge-kutta-4 integrator.  It was written by G.J. Sussman.

;;; Code for the stormer integrator is not shown.

;;; This program was originally written to allow execution on
;;; parallel processors - no assumptions are made as to the ordering
;;; of elements within a list.  It has become a fairly standard benchmark
;;; in the local community.
```

```
;;;; Integration of orbits

;;; INTEGRATE-SYSTEM takes an initial system state and a time step, H.
;;; it produces a stream of future states.  A system state is a time,
;;; and a set of particles.  Each particle has a name, a mass, a position
;;; and a velocity.

(define (integrate-system initial-state h)
  (let ((integrator (runge-kutta-4 (particle-force gravitation) h)))
    (define (next state)
      (cons state
            (delay (next (integrator state)))))
    (next initial-state)))

(define (integration-step initial-state h)
  (let ((integrator (runge-kutta-4 (particle-force gravitation) h)))
       (integrator initial-state)))

;;; Runge-Kutta takes a function, F, that produces a system derivative
;;; from a system state.  Runge-Kutta produces a function that takes a
;;; system state and produces a new system state.

(define (runge-kutta-4 f h)
  (define h* (scale-system h))
  (define 2* (scale-system 2))
  (define 1/2* (scale-system (/ 1 2)))
  (define 1/6* (scale-system (/ 1 6)))
  (lambda (y)                               ; y is a system state
    (define k0 (h* (f y)))
    (define k1 (h* (f (add-systems y (1/2* k0)))))
    (define k2 (h* (f (add-systems y (1/2* k1)))))
    (define k3 (h* (f (add-systems y k2))))
    (add-systems y (1/6* (add-systems k0 (2* k1) (2* k2) k3)))))
```

```
;;; Given a system state PARTICLE-FORCE produces a system derivative.
;;; A system derivative is a time increment and a set of differential
;;; particles.  A differential particle has a name, a differential of
;;; mass, a differential of position and a differential of velocity.

(define (particle-force force-law)
  (define (accelerations bodies)
    (let ((p (car bodies)) (r (cdr bodies)))
      (if (null? (cdr r))                  ;2-body problem
          (let ((inc (force-law p (car r))))
            (list (car inc) (cdr inc)))
          (let ((incs
                 (map (lambda (other)
                        (force-law p other))
                      r)))
            (cons (reduce add (map car incs))
                  (map add
                       (map cdr incs)
                       (accelerations r)))))))
  (lambda (system-state)
    (make-system 1                         ; dt/dt
      (map (lambda (p a)
             (make-particle (name p)
                            0              ; dm/dt
                            (velocity p)
                            a))
           (particles system-state)
           (accelerations (particles system-state))))))

(define (gravitation p1 p2)
  (let ((dx (sub (position p1) (position p2))))
    (let ((rcube (cube (sqrt (reduce + (map square dx))))))
      (cons (scalar (/ (* -G (mass p2)) rcube) dx)
            (scalar (/ (* G (mass p1)) rcube) dx)))))
```

```
;;;; Orbital mechanics data structures

(define type car)

;;; A system state is a time and a set of particles.
;;;  Each particle has a name, a mass, and a dynamical description
;;;  in either rectangular or element coordinates.

(define (make-system time particles)
  (list '(system-state) time particles))
(define time cadr)
(define particles caddr)

(define (operate-system f system)
  (make-system (time system) (map f (particles system))))

(define (operate-planets f system)
  (operate-system (lambda (s)
                    (if (eq? (name s) 'sun) s (f s)))
                  system))

(define (find-particle n system)
  (lookup (lambda (p) (eq? (name p) n)) (particles system)))

(define (scale-system scale-factor)
  (define sp (scale-particle scale-factor))
  (lambda (system)
    (make-system (* scale-factor (time system))
                 (map sp (particles system)))))

(define (add-systems . systems)
  (make-system (reduce + (map time systems))
               (map (lambda (bunch) (reduce add-particles bunch))
                    (group name (map particles systems)))))
```

```
;;; Particles

(define (scale-particle factor)
  (define factor* (scale factor))
  (lambda (particle)
    (assert (memq 'rectangular (type particle)) "Not rectangular")
    (list (type particle)
          (name particle)
          (* factor (mass particle))
          (factor* (position particle))
          (factor* (velocity particle)))))

(define (add-particles . particles)
  (for-each (lambda (p)
              (assert (memq 'rectangular (type p)) "Not rectangular"))
            particles)
  (list (type (car particles))
        (name (car particles))
        (reduce + (map mass particles))
        (reduce add (map position particles))
        (reduce add (map velocity particles))))
```

```
;;;;  Math library

;;; This is a standard math library - not all of the functions are
;;; used by the N-body program.

(define (acos w)
  (define epsilon 1.0e-13)
  (cond ((< (abs w) 1)
         (atan (sqrt (- 1 (square w))) w))
        ((> (- (abs w) 1) epsilon)
         (error "ACOS -- argument > 1: " w))
        ((>= w 1) 0)
        (else pi)))

;;; Functional operators

(define ((bracket . fl) . x)
  (map (lambda (f) (apply f x))
       fl))

(define ((apply-to-all f) x)
  (map f x))

(define (reduce proc args)
  (if (null? (cdr args))
      (car args)
      (proc (car args)
            (reduce proc
                    (cdr args)))))
```

```
(define (reduction f make-f-identity)
  (define (reduce l)
    (cond ((null? l) (make-f-identity))
          ((= (length l) 1) (car l))
          (else (f (car l) (reduce (cdr l))))))
  (lambda l
    (reduce l)))

(define ((compose . fl) x)
  (if (null? fl)
      x
      ((car fl) ((apply compose (cdr fl)) x))))

(define (fixed-point f start)
  (define epsilon 5.0e-13)
  (define (good-enough? new start)
    (< (/ (abs (- new start))
          (max (abs new) (abs start) epsilon))
       epsilon))
  (let loop ((old start) (new (f start)))
    (if (good-enough? new old)
        new
        (loop new (f new)))))

(define (for-each-distinct-pair proc list)
  (if list
      (let loop ((first (car list)) (rest (cdr list)))
           (for-each (lambda (other-element)
                       (proc first other-element))
                     rest)
           (if rest (loop (car rest) (cdr rest))))))
```

```
;;; Matrices

(define make-matrix list)

(define (matrix*vector m v)
  (map (lambda (v2) (dot v v2)) m))

(define (transpose ll)
  (apply map list ll))

(define (2-matrix-multiply m1 m2)
  (let ((t (transpose m2)))
    (map (lambda (v1)                      ;(matrix*vector t v1)
           (map (lambda (v2) (dot v1 v2)) t))
         m1)))

(define (matrix-multiply . m)
  (reduce 2-matrix-multiply m))

(define (2-rotation angle)
  (let ((s (sin angle)) (c (cos angle)))
    (make-matrix (3-vector c (- s))
                 (3-vector s    c))))


;;; Geometry

(define (make-x-rotation angle)
  `((1                 0                0)
    (0                 ,(cos angle)     ,(- (sin angle)))
    (0                 ,(sin angle)     ,(cos angle))))

(define (make-y-rotation angle)
  `((,(cos angle)      0       ,(sin angle))
    (0                 1       0)
    (,(- (sin angle))  0       ,(cos angle))))

(define (make-z-rotation angle)
 `((,(cos angle)       ,(- (sin angle))    0)
   (,(sin angle)       ,(cos angle)        0)
   (0                  0                   1)))

;;; Angles

(define pi (* 4 (atan 1 1)))
(define 2pi (* 2 pi))
(define pi/2 (/ pi 2))

(define radians-per-arc-second
  (/ 2pi 60 60 360))

(define radians-per-degree
  (/ 2pi 360))

(define (principal-value angle)
  (let ((na (/ angle 2pi)))
    (* 2pi (- na (round na)))))

;;; Vectors

(define (square x) (* x x))
(define (cube x) (* x x x))

(define 3-vector list)
(define x-coord car)
(define y-coord cadr)
(define z-coord caddr)

(define (map-vector f)
  (define (mapv . vl)
    (if (pair? (car vl))
        (apply map mapv vl)
        (apply f vl)))
  mapv)


(define zero-vector (3-vector 0 0 0))
```

```
(define add (map-vector +))
(define sub (map-vector -))
(define (scale c) (map-vector (lambda (x) (* c x))))

(define (scalar c v)
  ((scale c) v))

(define (magnitude v)
  (sqrt (apply + (map square v))))

(define (dot v1 v2)
  (apply + (map * v1 v2)))

(define (cross v1 v2)
  (3-vector
   (- (* (y-coord v1) (z-coord v2)) (* (y-coord v2) (z-coord v1)))
   (- (* (z-coord v1) (x-coord v2)) (* (z-coord v2) (x-coord v1)))
   (- (* (x-coord v1) (y-coord v2)) (* (x-coord v2) (y-coord v1)))))

(define (vector-angle v1 v2)
  (let ((epsilon 1.0e-13) (dp (dot v1 v2)) (t1 (cross v1 v2)))
    (let ((m (magnitude t1)))
      (cond ((and (< m epsilon) (< dp epsilon))
             (error "Vector angle is numerically ill-defined." v1 v2))
            (else (atan m dp))))))

(define (signed-vector-angle v1 v2 z)
  ;; this computes the signed angle from v1 to v2.
  (let ((epsilon 1.0e-13) (dp (dot v1 v2)) (t1 (cross v1 v2)))
    (let ((m (magnitude t1)))
      (cond ((and (< m epsilon) (< dp epsilon))
             (error "Vector angle is numerically ill-defined." v1 v2 z))
            ((< (dot z t1) 0) (- (atan m dp)))
            (else (atan m dp))))))
```

```
;;; Stream procedures.

(define head car)
(define (tail s) (force (cdr s)))
(define empty-stream? null?)
(define the-empty-stream '())

(define (apply-to-stream f . streams)
  (let loop ((s streams))
    (if (not (apply *or (map empty-stream? s)))
        (sequence
          (apply f (map head s))
          (loop (map tail s))))))

(define (map-stream f . streams)
  (let loop ((s streams))
    (if (not (apply *or (map empty-stream? s)))
        (cons (apply f (map head s))
              (delay (loop (map tail s))))
        the-empty-stream)))

(define (multiples-of n h)
  (cons (* n h)
        (delay (multiples-of (1+ n) h))))
```

```
;;;; General utilities

(define false (= 1 0))
(define true (= 0 0))

(define (*or . disjuncts)
  (cond ((null? disjuncts) false)
        ((car disjuncts) true)
        (else (apply *or (cdr disjuncts)))))

(define (*and . conjuncts)
  (cond ((null? conjuncts) true)
        ((car conjuncts) (apply *and (cdr conjuncts)))
        (else false)))

(define (assert x? . m)
  (if (not x?)
      (apply error m)))

(define (lookup pred? l)
  (cond ((null? l) '())
        ((pred? (car l)) (car l))
        (else (lookup pred? (cdr l)))))

(define (translate compare from-list to-list key)
  (let loop ((from from-list) (to to-list))
    (cond ((null? from) (error "Can't find" key "in" from-list))
          ((compare key (car from)) (car to))
          (else (loop (cdr from) (cdr to))))))
```

```
;;;; Multisets

;;; GROUP forms the set of groups of elements from SETS such that all
;;; of the elements in each group have the same value of the KEY.  It is
;;; presumed that each set has a unique element with each key and each key
;;; appears in every set.

(define (group key sets)
  (let ((n (- (length sets) 1)))
    (define (new-frob frob sets accum)
      (let ((bin (lookup (lambda (x)
                           (eq? (key frob) (key (car x))))
                         accum)))
        (if bin
            (if (= (length bin) n)
                (adjoin (cons frob bin)
                        (scan-out sets (delete bin accum)))
                (scan-out sets
                          (adjoin (cons frob bin)
                                  (delete bin accum))))
            (scan-out sets (adjoin (list frob) accum)))))
    (define (scan-out sets accum)
      (if (empty? sets)
          (empty-set)
          (if (empty? (first sets))
              (scan-out (rest sets) accum)
              (new-frob (first (first sets))
                        (adjoin (rest (first sets)) (rest sets))
                        accum))))
    (scan-out (apply make-set sets)     ; coerce list to multiset.
              (empty-set))))


;;; MAP-DISTINCT-PAIRS applys a function, F, to every distinct pair
;;; of values chosen from the multiset, M, producing a multiset of the
;;; results.

(define (map-distinct-pairs f mset)
  (map (lambda (p) (apply f p))
        (distinct-pairs mset)))

(define (distinct-pairs mset)
  (if (empty? mset)
      (empty-set)
      (let ((f (first mset))
            (r (distinct-pairs (rest mset))))
        (let loop ((left (rest mset)))
             (if (empty? left)
                 r
                 (adjoin (list f (first left))
                         (loop (rest left))))))))


(define make-set list)
(define first car)
(define rest cdr)
(define adjoin cons)

(define empty? null?)
(define (empty-set) '())
```

```
;;; Initial conditions commonly used as a starting point
;;;   for solar system integrations

(define SS
  (let ()
    ;; Heliocentric equatorial rectangular coordinates 1950.0
    ;; From Schubart and Stumpff (1966)
    ;; Note that velocities are entered in AU/40 days but converted to
    ;; AU/day on system construction.  Positions are in AU.  Masses are in Msun.

    (define sun
      (make-rectangular-heliocentric 'sun (+ 1 (/ 1 6000000))      ; +Mercury
        (3-vector 0 0 0)
        (3-vector 0 0 0)))

    (define venus
      (make-rectangular-heliocentric 'venus (/ 1 408000)
       (3-vector -.5113942959 -.4780976854 -.1830874810)
       (3-vector .5663768182 -.5120871589 -.2664978745)))

    (define earth
      (make-rectangular-heliocentric 'earth (/ 1 329390)
       (3-vector -.2614989917 .8696237687 .3771652157)
       (3-vector -.6746573183 -.1700948008 -.07377431920)))

    (define mars
      (make-rectangular-heliocentric 'mars (/ 1 3093500)
       (3-vector -1.295477589 -.8414136141 -.3513513446)
       (3-vector .3440042605 -.3696674843 -.1789373952)))

    (define jupiter
      (make-rectangular-heliocentric 'jupiter (/ 1 1047.355)
       (3-vector 3.429472643 3.353869719 1.354948917)
       (3-vector -.2228647739 .2022768826 .09223051780)))

    (define saturn
      (make-rectangular-heliocentric 'saturn (/ 1 3501.6)
       (3-vector 6.641453441 5.971569844 2.182315015)
```

```
   (3-vector -.1662288590 .1462712350 .06765636470)))

(define uranus
  (make-rectangular-heliocentric 'uranus (/ 1 22869)
   (3-vector 11.26304125 14.69525888 6.279605833)
   (3-vector -.1301308165 .07588051779 .03508967792)))

(define neptune
  (make-rectangular-heliocentric 'neptune (/ 1 19314)
   (3-vector -30.15522934 1.657000860 1.437858110)
   (3-vector -.009619598984 -.1150657040 -.04688875226)))

(define pluto
  (make-rectangular-heliocentric 'pluto 0
   (3-vector -21.12383780 28.44651101 15.38826655)
   (3-vector -.07074485007 -.08655927220 -.005946850713)))

(define (convert-to-AU/day system)  ; only for rect helio
  (operate-system (lambda (particle)
                    (make-rectangular-heliocentric
                     (name particle)
                     (mass particle)
                     (position particle)
                     ((scale (/ 1 40)) (velocity particle))))
                  system))

(convert-to-AU/day
 (make-system 0
   (make-set sun venus earth mars jupiter saturn uranus neptune pluto))))
```