Partial Evaluation Applied to Numerical Computation

Andrew A. Berlin
Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Abstract

There have been many demonstrations that the expressive power of Lisp can greatly simplify the process of writing numerical programs, but at the cost of reduced performance.[10][16] I show that by coupling Lisp's abstract, expressive style of programming with a compiler that uses partial evaluation, data abstractions can be eliminated at compile time, producing extremely high-performance code. For an important class of numerical programs, partial evaluation achieves order-of-magnitude speed-ups over conventional Lisp compilation technology. This approach has proven to be especially effective when used in conjunction with schedulers for VLIW and highly pipelined architectures, because the elimination of data structures and procedural abstractions exposes the low-level parallelism inherent in a computation.

Introduction

Most modern compilers focus on optimizing a program's instructions, without regard for the particular problem that the program will be used to solve. This limited perspective forces scientific programs to be taken more literally than the programmer intended. For example, a scientist writing a complex program builds up layers of data abstractions, describing a computation in terms of high-level data structures; at run time, the computer actually creates and manipulates these data structures. Unfortunately, this is rarely what the scientist wants – the real goal is to perform a *numerical* computation; data structures are merely a convenient way of specifying that computation. Thus to achieve high performance, scientists typically do not program by combining high-level procedures, but instead hand-code specialized versions of these procedures for the particular application at hand.

I introduce a simple technique for exposing the underlying numerical computation expressed by a high-level program. This technique is based on the observation that numerical programs are mostly data independent. In other words, a routine such as matrix-multiply performs a fixed set of multiplications, regardless of the numerical values of the numbers being multiplied. Even when numerical programs are not data-independent, they typically contain extremely large data-independent regions, with only a few data-dependent branches included for such things as convergence checking and strategy selection. Data-independence is important because it makes it possible to predict what operations a program will perform, even before actual numerical values for its inputs are available. This allows data manipulation operations to be performed in advance, at compile time, leaving only the underlying numerical computation to be performed at run time.

The key idea is to use *partial evaluation* to create a specialized version of a program for the particular application at hand. Partial evaluation uses information about the application to evaluate portions of a program in advance, thereby creating a specialized program. For example, given a program that computes force interactions among N particles, together with the fact that the particular application of interest involves only 9 particles, partial evaluation creates a program specialized to handle exactly 9 particles. In the very special case of a data-independent computation, there is enough information available at compile time for the partial evaluator to perform *all* data manipulation operations in advance, producing a compiled program that consists entirely of numerical operations.

Considering information about the data associated with a particular problem is very important, because many data-dependent programs become data independent once information is available about the particular problem that the program will be used to solve. For example, a general version of matrix-multiply, in which the size of the matrix is not known at compile time, would be data-dependent, since the sequence of operations would vary depending upon the size of the matrices being manipulated. This would prevent the matrix reference operations from being performed at compile time, requiring that the matrix data-structures be manipulated at run time. However, by considering information about the particular matrices associated with a given problem, the matrix size can be determined at compile time, transforming matrix-multiply into a data-independent program.

```
;; Typical data at run time:
(define mars
  (make-planet 'mars
               (/ 1 3093500) ;mass
               (3-vector -1.295477589 -.8414136141 -.3513513446)   ;position
               (3-vector .3440042605 -.3696674843 -.1789373952)))  ;velocity


;; Data structure as created at compile time:
(define mars
  (make-planet 'mars
   (/ 1 3093500) ;The mass of a planet is known at compile time.

   (3-vector                                ;position
    (MAKE-PLACEHOLDER 'mars-position-x 'floating-point)
    (MAKE-PLACEHOLDER 'mars-position-y 'floating-point)
    (MAKE-PLACEHOLDER 'mars-position-z 'floating-point))

   (3-vector                                ;velocity
    (MAKE-PLACEHOLDER 'mars-velocity-x 'floating-point)
    (MAKE-PLACEHOLDER 'mars-velocity-y 'floating-point)
    (MAKE-PLACEHOLDER 'mars-velocity-z 'floating-point))))
```

Figure 1: The program's input data structures are created at compile time. Notice how placeholders are used to represent those numerical values which will not be available until run time.

Partial Evaluation of Data-Independent Programs

There is a very simple way to figure out what numerical computations a data-independent program will perform: simply execute the program at compile time, and keep track of what it does! The trick is to create the input data structures for a particular application at compile time. Although the actual numerical values for some pieces of data will not be available until run time, their locations within the input data-structures are known at compile time. These missing values are represented symbolically using a data-structure known as a *placeholder*. Placeholders can also be used to hold additional information about a missing number, such as its type or its range of possible values.

For example, consider the input data structures for a program that integrates the motion of the solar system. The program takes as input the current positions of the planets, and produces a new set of positions corresponding to one time step later. Since the planets will be in different positions each time the program is run, numerical values for the positions are not known at compile time. However, we do know the locations of these missing values within the input data structures, and we know that they will be of type floating-point. Figure 1 shows how placeholders are used to embed this information in the compile-time input data structures.

Partial evaluation is accomplished by executing the program symbolically at compile time using the placeholder based data structures as input. During symbolic execution, placeholders are treated just like numbers: they can be consed together to form lists, be stored in variables, be passed as arguments to procedures, etc. Anything that would move a number around will also move a placeholder around. This allows all data manipulation operations – procedure calls, data structure manipulations, etc., to be performed at compile time – the only operations executed at run time are numerical.

Modifying a Lisp implementation to handle the placeholder data structure is trivial: the definitions of the lowest level numerical operations (such as +, *, -, /) are modified,[1] such that if an operand is a placeholder, the operation is delayed until run time, when the actual numerical value represented by the placeholder is available. As illustrated in Figure 2, delaying an operation until run time is achieved by adding an instruction to the compiled program, and creating a new placeholder to represent the not-yet-computed result.

Languages such as Lisp are especially well suited to the use of placeholders, because there are no restrictions on what type of object can be placed a data structure, enabling placeholders to be treated exactly like numbers. Using this technique in a strictly typed language such as PASCAL would be somewhat more difficult, because the type checking mechanism prohibits placing a placeholder into a data structure that expected a number.

[1] Changing the definitions of the low level operations to handle placeholders can be trivially achieved through the use of an ADVICE mechanism, or by redefining the procedures +, *, etc.

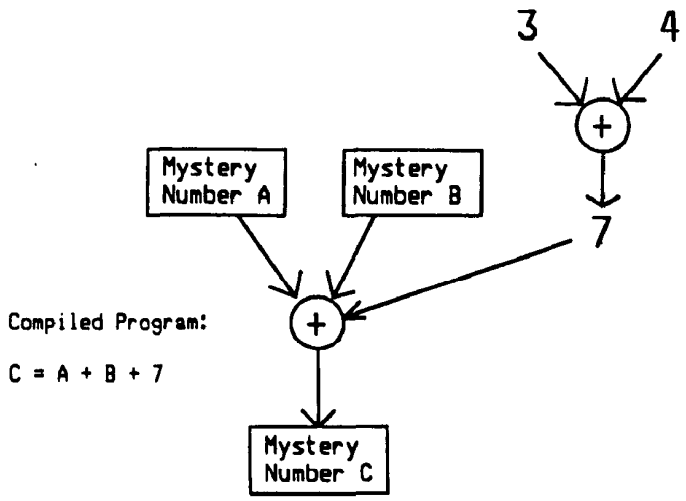

Figure 2: The program is executed at compile time. Placeholders are used to represent values which are not yet available. Those operations for which numerical values are available proceed normally, producing numerical results. Those operations whose values are not available are delayed until run time by adding an instruction to the compiled program.

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

==> (define input-data
      (list (make-placeholder 'floating-point) ;placeholder #1
            (make-placeholder 'floating-point) ;placeholder #2
            3.14))

==> (sum-of-squares input-data) ;Execute the program at compile time


   COMPILED PROGRAM (SPECIALIZED SUM-OF-SQUARES):

   (INPUT 1) ;;numerical value for placeholder #1
   (INPUT 2) ;;numerical value for placeholder #2

   (ASSIGN 3 (floating-point-multiply (FETCH 1) (FETCH 1)))
   (ASSIGN 4 (floating-point-multiply (FETCH 2) (FETCH 2)))
   (ASSIGN 5 (floating-point-add (FETCH 3) (FETCH 4) 9.8596))

   (RESULT 5)
```

Figure 3: Specialized code for Sum-of-squares. Notice how the squaring of 3.14 to produce 9.8596 took place at compile time. Annotating the placeholders with the fact that the input numbers would be of type *floating-point* enabled the generic arithmetic dispatch to be performed at compile time, allowing floating-point (rather than generic arithmetic) routines to be used in the compiled program.

Examples

Sum of Squares

To illustrate the compilation process, consider the sum-of-squares program shown in Figure 3. In this hypothetical application, the input is known to be a list of three floating-point numbers, the last of which is always 3.14. This information is encoded in the input data structures at compile time. Partial evaluation is then performed by symbolically executing the program on the placeholder-based input data, producing the compiled program shown in Figure 3. Notice that in the compiled program, all data structures and procedure calls have been eliminated, leaving only numerical operations to be performed at run time.

The Sine Function

At the 1988 Lisp and Functional Programming Conference, Roylance [16] showed how the expressive power of Lisp can be used to construct an implementation of the sine function that mirrors our understanding of the mathematical concepts being employed. This is in contrast to conventional implementations of sine, which are based on "concrete arithmetic expressions that include many mysterious numerical constants."[2] Roylance goes on to complain that the concrete implementations execute significantly faster, primarily because Lisp compiler technology is not yet sufficiently advanced to perform the program transformations necessary to produce code that is competitive with the traditional programming style. Partial evaluation provides an elegant solution to this problem.

[2] L&FP 1988 [16], Page 8.

Roylance's code computes the sine function by combining several higher-order procedures. He starts by setting up an infinite Taylor expansion, then truncates it to an appropriate number of terms depending on the number of digits of precision required. My compiler took this collection of procedure calls, and specialized them for the particular case of computing the sine function. The result is a low-level procedure that mirrors the conventional implementation of sine: A complex expression full of mysterious numbers, but which executes very quickly (Figure 4).

Further Optimizations

Traditional compiler optimization techniques[2] can further improve the performance of a partially evaluated program. The most important of these are numerical optimizations, such as constant folding, sign targetting, common subexpression elimination, dead code elimination, and symbolic simplifications. Although some optimizations, such as constant folding, are performed incrementally by the partial evaluation process itself, further improvements can often be obtained once the entire computation exposed by the partial evaluation process is available. For instance, a symbolic simplification may produce opportunities for constant folding, which in turn may produce opportunities for common-subexpression elimination. In Roylance's sine-half-9 procedure, these optimizations were able to combine multiple calls to `EXPT`, such that the work they had in common could be shared. As shown in the compiled code, `(EXPT X 5)` is computed based on work already accomplished by `(EXPT X 3)`. This was not requested by the original high-level program; it was automatically derived by the compilation process.

Data-Dependent Programs

Performing partial evaluation by symbolically executing a program works well for data-independent computations, but runs into problems when applied to data-dependent computations. When the partial evaluator reaches a data-dependent conditional, such as an `IF` statement, it can not decide whether to evaluate the *consequent* or the *alternative* until run time. In the general case, this stops the compile-time execution process: the program might do vastly different things depending on which branch of the conditional is taken. Fortunately, in numerical programs, considering information about the particular problem that a program will solve usually makes the data-independent regions of a program extremely large (often several thousand of operations), with data-dependent conditionals only occuring at the end of these long computations for such things as convergence checks and strategy selection.

For programs that are mostly data-independent, partial evaluation can be used to generate highly efficient code for each data-independent region. By incorporating these partially evaluated routines into the Lisp system as high-performance subroutines, the control mechanisms of Lisp can be used to handle the data dependencies. In practice, this division of the program into data-independent regions has proven to be effective, but has the drawback that it limits the scope of the partial eval-

```
;;The original program is composed of high-level procedures:
(define sine-half-9
  (lambda (x)
    (termlist-eval
     (truncated-series-eps sine-term sine-mono
                           1.0e-9 (/ pi 2))
     x)))

(sine-half-9 (make-placeholder 'X))

;;COMPILED PROGRAM:
(define (sine-half-9 x)
  (let ((node-2 (* x (* x x))))                    ;X^3
    (let ((node-3 (* x (* x node-2))))             ;X^5
      (let ((node-4 (* x (* x node-3))))           ;X^7
        (let ((node-5 (* x (* x node-4))))         ;X^9
          (let ((node-6 (* x (* x node-5))))       ;X^11
            (let ((output-1
                   (+
                    (* 1.6059043836821613e-10 (* x (* x node-6)))
                    (+
                     (* -2.505210838544172e-8 node-6)
                     (+
                      (* 2.7557319223985893e-6 node-5)
                      (+
                       (* -1.984126984126984e-4 node-4)
                       (+
                        (* 8.333333333333333e-3 node-3)
                        (+ (* -.16666666666666666 node-2) x))))))))
              output-1)))))))
```

Figure 4: Compiled version of Roylance's sine-half-9 procedure. This procedure computes the sin function for values of x from -pi/2 to pi/2.

uation optimizations, since the high-level data structures that act as interfaces between data-independent regions cannot be eliminated.

[3] describes certain situations in which it is possible to partially evaluate beyond data-independent regions. For example, in simple selection operations that do not change the structure of the data, as in the absolute value function, it is possible to symbolically evaluate *both* the consequent and the alternative, generating compiled code for both possibilities. A data-dependent branch is included in the compiled program to choose among these two different code segments at run time. [3] and [4] describe situations in which the data-dependent conditionals used to control loop termination can be partially evaluated as well.

Code Size

Another potential problem with using partial evaluation is the size of the compiled program. Partial evaluation, as described in this paper, expands loops associated with traversing data structures. However, in applications that manipulate very large amounts of data, some loops really ought to be left intact. For example, consider a piece of code that traverses a large data structure, such as a quadtree. Partially evaluating the entire execution of a program that maps over the quadtree would not be practical – there are too many operations. It would make more sense to compile the inner loops that deal with manipulations on one section of the quadtree, while leaving intact the outermost loop that traverses the tree. This is in fact the approach that was used to compile the multipole translation operator application, described below. The operation of the translation operator on a single cube of space was compiled, while the outer loop that maps this operator over all such cubes was left intact.

The problems posed by code size are not nearly as serious as those posed by data dependencies. For instance, one can envision heuristics that would monitor the amount of code produced by a loop, and when necessary, inform the partial evaluator to only unroll the body of the loop if too much code is being generated. On the other hand, for applications involving small amounts of data, elimination of data structure manipulation instructions often causes the partially evaluated programs to be smaller than the high-level programs they were generated from.

Prototype Compiler

I have implemented a prototype compiler based on these ideas. This compiler uses the placeholder-based symbolic execution technique to perform partial evaluation. The partially evaluated program is represented as a data flow graph, which is then optimized using traditional

compiler optimizations. Straightforward transformations are used to map the resulting data-flow graph into Lisp syntax, into C syntax, or into register-transfer language. As illustrated in the sine program (Figure 4), when expressing a program in Lisp syntax, LET statements are used to store results that are referenced multiple times, whereas an instruction whose result is only referenced once is in-line coded at the point where its result is used. Similarly, when expressing a program in C syntax, an array of temporary memory locations is used to store results that are used more than once, while operations that are referenced only once are in-line coded.

The partially evaluated programs are invoked from Lisp as subroutines. Where the original programs used high-level data structures to receive their inputs and transmit their outputs, the partially evaluated subroutines take as input numerical values for the input placeholders, and produce as output numerical values for the result placeholders. My compiler *automatically* generates a set of interface routines that extract numerical values for the input placeholders from the input data structures, and that construct the result data structures based on the values computed for the result placeholders. Program's that manipulate that low-level input and output placeholder values directly can avoid the overhead associated with creating and referencing the high-level interface data structures.

The prototype compiler does not provide support for automatically detecting whether it can continue past a conditional branch. By default, the compiler will partially evaluate only a data-independent subroutine, leaving data-dependencies to be handled by the high-level program that invokes the subroutines. However, if requested by the programmer, the partial evaluator will use the techniques described in [3] to continue partial evaluation beyond the data-dependency, thereby generating data-dependent branches in the partially evaluated program.

Measurements

To measure the effectiveness of partial evaluation on numerical computations, I compared the execution speeds of the specialized programs produced by my prototype compiler against those of the same programs compiled using the LIAR Scheme compiler.[3] Measurements were performed on a variety of scientific programs obtained from researchers at MIT. These programs were taken from actual research in progress, and were not modified for the purposes of this comparison.

The experiments were conducted by compiling each sample application using the prototype compiler. The compiler's output was expressed in C syntax[4] and compiled using the GNU C compiler. The resulting compiled program was then linked in to the MIT Scheme system so that it could be called from Lisp as a high-performance subroutine.

Sample Applications

A Scheme implementation of a program that computes force interactions among a set of particles (the N-body problem) was obtained from Gerry Sussman. This important application arises in particle physics, astronomy, and space travel. The program itself is written very abstractly, making liberal use of abstraction mechanisms, including higher-order procedures, lists, vectors, table lookups, and set operations. Two specialized versions of the program were compiled: one for a six body solar system, and one for a nine body solar system. In both cases, the force law (gravitation) and the integration method (runge-kutta) were chosen at compile time, and included in the input data structures.

The second program tested was a translation operator from the multipole method of force approximation.[20] This approximation method is practical for use in fluid-flow applications, and in simulations involving millions of particles. The source program was written in Lisp, primarily to help people understand the numerical methods being used. As such, it does not take advantage of special cases in the expansions, such as terms that are known to have exponents that are zero or one. My compiler was able to take advantage of these special cases, providing significant performance improvement. The variable P, which determines the number of terms in the multipole expansions, was chosen at compile time.

The last program tested was an adaptive integration of Duffing's equation, a small, non-linear differential equation. This program was taken from Hal Abelson's work on automatic characterization of the state space of dynamical systems[1]. The innermost routine of this program integrates for one time step. A control-loop invokes this routine repeatedly until a data-dependent conditional indicates that a single period of the function has been integrated. A declaration was added to the original program, indicating to the partial evaluator that it should use the techniques described in [3] to include the data-dependent branch and control loop in the partially evaluated program.

Results

Table 5 shows the execution times (measured in seconds) for the original program in MIT Scheme; for the program after having been compiled using the LIAR Scheme compiler; and for the specialized version of the program produced by my prototype compiler.

An additional experiment was conducted to measure the performance of the specialized version of Roylance's sine routine (Figure 4). In this experiment, the specialized sine program produced by partial evaluation was expressed in Scheme, and then compiled using the Liar Scheme compiler.[5] The performance of the specialized routine was compared against that of Roylance's high-level program, which was also compiled using the Liar Scheme compiler. Since no floating-point declarations were provided, both programs compute sine using

[3] 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.

[4] C was chosen merely for convenience. The programs could have been expressed in Scheme and then compiled using the Liar Scheme compiler. However, the optimization algorithms that the Scheme compiler uses take a very long time to execute when applied to the already optimized straight-line segments of code generated by the partial evaluator. This problem has been corrected in later versions of the Liar compiler.

[5] This experiment was performed using MIT CScheme release 7.1 with Liar compiler version 4.70.

Performance Measurements					
Problem Desc.	Interpreted CScheme	Compiled CScheme	Specialized Program	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 5: 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.

generic arithmetic functions. This experiment showed that the specialized version of the sine routine executes 17 times faster than the high-level program from which it was derived.

Exposing Parallelism

Partial evaluation has an important role to play in the programming of parallel computers. Parallel compiling involves two challenges: identifying the parallelism that is available in a program, and deciding how to divide the parallel operations among multiple processors. Partially evaluating a program can greatly simplify both of these tasks, because parallelizing the underlying numerical computation is much easier than parallelizing the original high-level program. For instance, opportunities for parallel execution are often masked by inherently sequential data structure references, such as cdr-chaining through a list, which can often be eliminated through partial evaluation. Eliminating data structures also eliminates synchronization points: computations that reference one part of a data structure frequently have to wait for the entire data structure to be created before they can begin their work. To illustrate this approach, figure 6 shows a parallelism profile for a partially evaluated version of the 9-body program.[6] This profile represents the maximum amount of parallel execution that could occur if an infinite number of processors were available, with no communication or pipeline delays.

Partial evaluation also simplifies the task of scheduling computations onto multiple processors. In practice, communication does take time, and some of the parallelism must be devoted to keeping processor pipelines full. Conditional branches make it difficult to account for these factors, since the amount of time a computation will take to complete is not known until the direction that the branch will take is known. Static schedulers, which schedule instructions at compile time, often use a technique known as trace-scheduling[9] to guess which way a conditional branch will go, allowing computations beyond the branch to be performed in parallel with those that precede the branch. Another approach is dynamic scheduling, which delays the scheduling decisions until run time, but imposes a computational overhead at run time. Partial evaluation simplifies the scheduling problem by eliminating conditional branches that relate to the structure of the data, thereby allowing static scheduling to be used effectively on large sections of a program.

I have implemented a static scheduler, described in detail in [3], that maps a data-independent computation onto multiple pipelined processors. The explicitly numerical nature of the partially evaluated programs greatly simplified this task: partial evaluation made the data flow patterns explicit, allowing the critical paths of the computation to be identified and given scheduling priority. For a hypothetical system composed of 40 processors, and accounting for pipelining and communication delays,[7] this scheduler is able to achieve a speed-up factor of **36.4** on the 9-body problem, utilizing the processors with 90% efficiency (Figure 7).

Scheduling for highly pipelined architectures

Partial evaluation based compiler technology is being used as the basis for the compiler of the MIT-HP Supercomputer Toolkit project. This project, a joint research effort between MIT and Hewlett-Packard, is building a set of long-instruction word computers, which are to be interconnected by the user to match the needs of a particular problem. The processors themselves can perform two memory references, two integer operations, numerous register accesses, and a floating-point operation, *in every machine cycle.* Programming these processors efficiently is quite challenging: the computation must be mapped onto the data paths in such a way that enough data arrives at the floating-point chips to enable them to initiate a new operation in every cycle.

Partial evaluation aids this scheduling task by producing programs that contain extremely long sequences of numerical operations (often several thousand operations long), with no intervening branches. This makes it feasible for the scheduler to re-order the program to account for pipeline delays, allowing the floating-point unit to be fully utilized. In addition, this predictability allows data motion instructions, such as memory fetches, to be initiated far in advance of the numerical

[6] 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.

[7] In this case, the scheduler is assuming a 1-cycle communication delay. This is measured from just after the data has left the pipeline of the transmitting processor, until just before the data enters the pipeline of the receiving processor. Results for different communication latencies are presented in [3] and [8]. This analysis does not consider limitations on communication bandwidth.

operation that needs the data. Work on the Supercomputer Toolkit compiler has progressed to the point where we can schedule the 6-body program in such a way as to keep one processor fully utilized.[8] We are now working on generalizing this approach to schedule code for multiple toolkit processors.

Relation to Previous Work

Specialized Computation

The idea of creating specialized programs has been around for a long time. For example, early versions of SPICE[14] generated specialized sparse matrix manipulation routines to handle the particular set of node equations of the circuit being simulated. Similarly, "straight-line" implementations of the FFT have be generated, such that all array indices are computed in advance.[13] These routines are typically created using a problem-specific hand crafted generator program. In contrast, partial evaluation is a problem-independent technique for transforming the original program into a specialized program, allowing specialization to occur over a larger portion of the overall problem than it would be practical to handcraft a generator program for.

Partial Evaluation

Partial evaluation is also an idea that has been around for a long time. A thorough review of the field can be found in [5], including a complete bibliography until 1988. With the exception of its use as a means of providing programming language extensibility[17], partial evaluation has not found much practical use. Komorowski [12] has investigated the application of partial evaluation to Prolog, and Danvy[7] has done some interesting new work on applying partial evaluation to the compilation of pattern matching programs. The primary contribution of my work is to show that partial evaluation is basically simple to implement, and can be put to practical use for compiling numerical programs.

The symbolic execution technique has proven to be effective for mostly data-independent programs, but needs to be made more automatic. A significant recent development in partial evaluation technology is Jones's[11] technique of analyzing the binding times of source programs. This technique enables the partial evaluator to be applied to itself, which is important for use in automatic compiler generation. Bondorf and Danvy[4] describe techniques for automating some of the decisions inherent in the partial evaluation process, such as which conditionals and loops should be expanded at partial evaluation time, and which should be left residual in the partially evaluated program. Their partial evaluation techniques can handle data dependencies better and more automatically than my symbolic evaluator can, but require that programs be expressed in a particular "well-staged" form, due to the approximations of their binding time analysis. Weise[19] describes how to extend the symbolic execution process to allow placeholders to represent structured data as well as numbers.

[8] This is a result obtained through simulation of the Toolkit processor. This processor is being fabricated by Hewlett-Packard's Information Architecture Group. We expect to have a fully operational prototype by April of 1990.

A good starting point for future work would be to investigate ways to merge some of these recent developments in partial evaluation technology with the symbolic execution technique.

Other Optimization Techniques

Partial evaluation is closely related to several other optimization techniques. The loop-jamming technique proposed by Burge[6] seeks to eliminate intermediate data structures by applying transformation rules which rearrange a program's instructions so as to combine the producer and consumer of a data structure. Wadler's listless programming[18] has a similar goal, analyzing the source code of a program to eliminate intermediate data structures. Partial evaluation differs in that it involves partial execution of the portions of a program that are relevant to a particular application, eliminating data structures by producing and consuming them at partial evaluation time.

Parallel Programming

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.[15] Similarly, compilers for VLIW architectures use *trace-scheduling*[9] to guess which way a branch will go, allowing computations beyond the branch to occur in parallel with those that precede the branch. However, the effectiveness of both of these techniques is limited by their preservation of the user data structures of the original program, which impose synchronization requirements. Many of the branches that trace-scheduling seeks to optimize can simply be eliminated through the use of partial evalualation. The two approaches are orthogonal: partial 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 these basic-block boundaries.

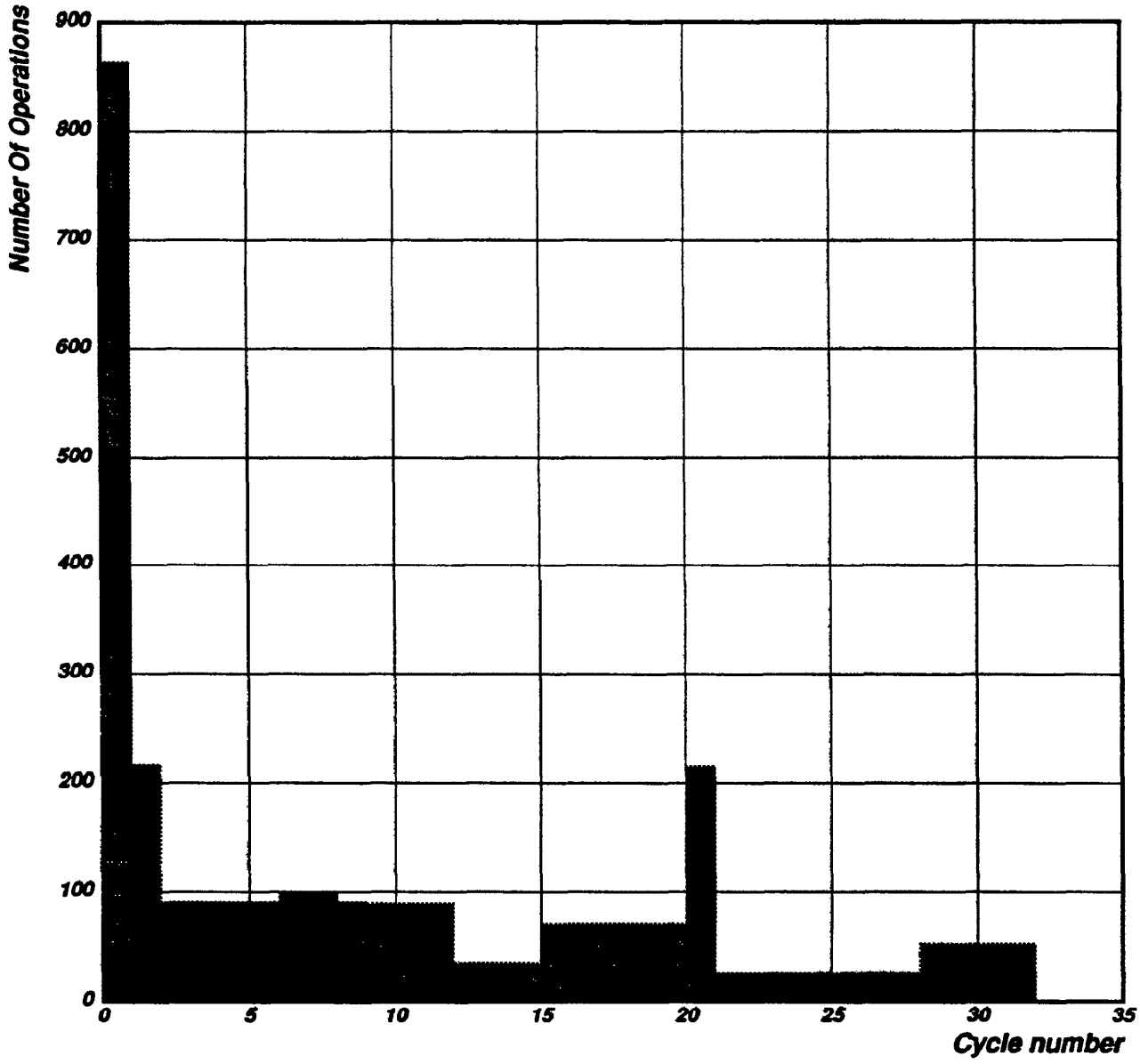

Figure 6: Parallelism profile of the 9-body problem. This graph represents the total parallelism available in the problem, accounting for the latency of numerical operations. Exploiting parallelism in this way would result in a speed-up factor of 97 over a single-processor implementation, but would require 865 processors, which would be used with only about 11% efficiency.

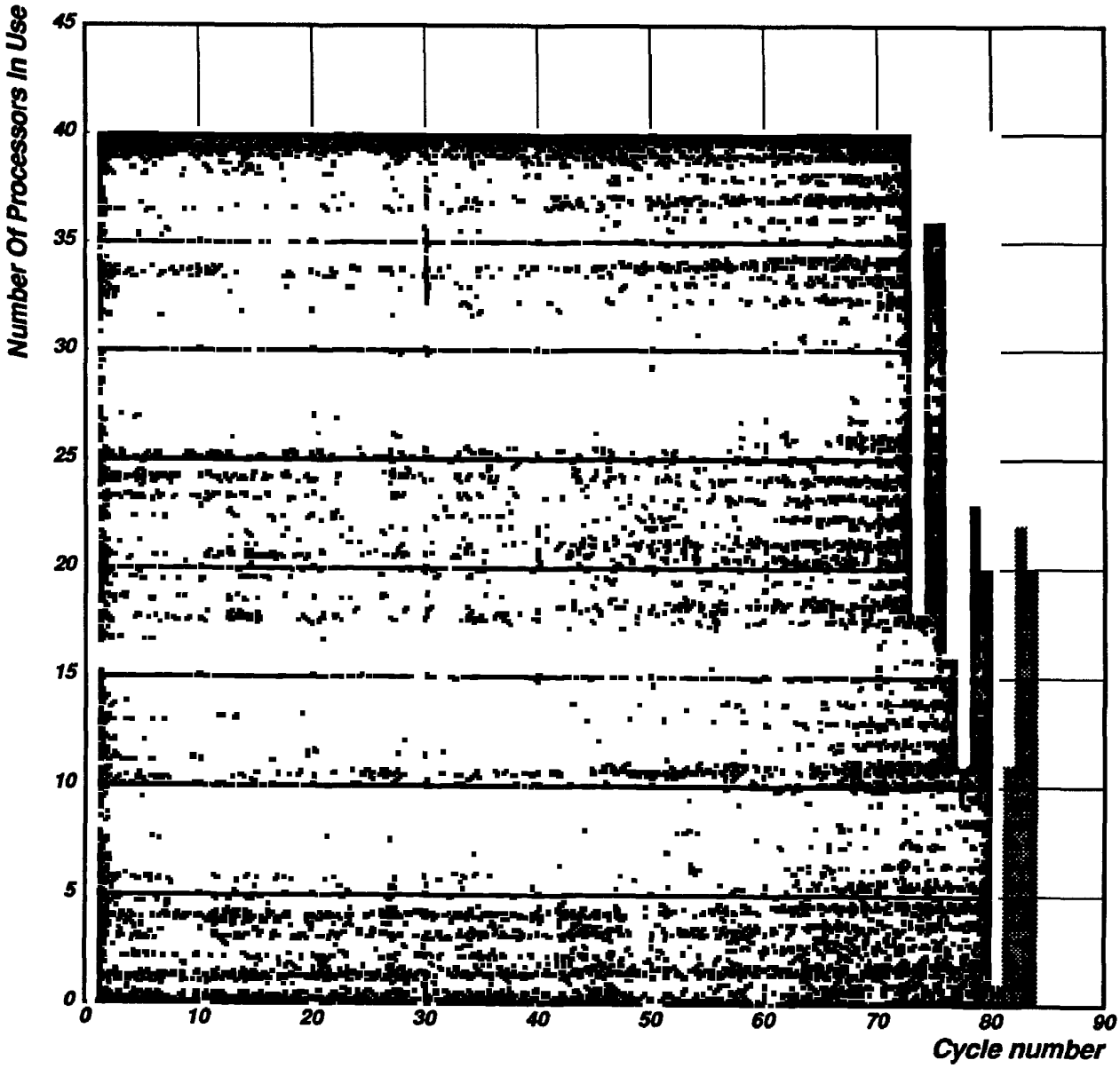

Figure 7: The result of scheduling the 9-body problem onto 40 pipelined processors with a communication latency of one cycle. A total of 85 cycles are required to complete the computation. On average, 36.4 of the 40 processors are utilized during each cycle.

Conclusions

Partial evaluation is an important technique that provides significant performance improvements for an important class of numerical programs. Implementing partial evaluation using the placeholder technique is adequate for data-independent computations, but needs to be made more general, particularly in the area of automatically deciding which loops and data structures should be partially evaluated, and which should be left for run time evaluation.[9]

The combination of the performance gains available from the elimination of data abstractions with those available from exploiting the low-level parallelism exposed by partial evaluation have the potential to provide a performance improvement of 2 to 3 orders of magnitude over current technology. This will fundamentally change the nature of many scientific computations. For example, modern analog circuit simulators simulate a single instantiation of a circuit. With the performance gains available from partial evaluation, it will become feasible to have the simulator perform a search of the circuit's parameter space, recommending new values for circuit elements such as resistors and capacitors.

The most exciting result of this work is the ability of partial evaluation to make abstractly specified programs execute efficiently. One of the most frustrating tasks in scientific programming is transforming an application into a form that can make use of existing library routines. Partial evaluation will allow the library routines to be specialized to match the program, rather than requiring the programmer to transform the program to match the library routines. This should lead to a new generation of generalized scientific library routines.

Acknowledgements

This work would not have been possible without the contributions of many people. Bill Rozas wrote the common subexpression eliminator used to optimize the partially evaluated programs. Bill also wrote the register allocator for the Supercomputer Toolkit. Hal Abelson, Tom Simon, Gerry Sussman, Jack Wisdom, and Feng Zhao contributed scientific programs to be used as test cases. Much was also learned as a result of discussions with Daniel Weise of Stanford University. Daniel and I are currently cooperating to combine our research results. Thanks are also due to Olivier Danvy for his insightful comments.

References and Bibliography

References

[1] Harold Abelson, "The Bifurcation Interpreter: A step towards the automatic analysis of dynamical systems," MIT Artificial Intelligence Laboratory Memo 1174, Cambridge, MA., September 1989.

[2] Al Aho; Ravi, Sethi; Jeff, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1985.

[9] Daniel Weise's group at Stanford has made some progress in this area.

[3] A. Berlin, "A compilation strategy for numerical programs based on partial evaluation," MIT Artificial Intelligence Laboratory Technical Report TR-1144, Cambridge, MA., July 1989.

[4] Anders Bondorf and Olivier Danvy, "Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types", DIKU research report 90/04, University of Copenhagen, Denmark, January 1990.

[5] Bjørner, D., Ershov, A. P., and Jones, N. D., (eds.), *Partial Evaluation and Mixed Computation*, North Holland, 1988.

[6] William H. Burge, "An Optimizing Techniques for High Level Programming Languages", IBM Thomas J. Watson Research Center Report RC5834 (#25271), Yorktown Heights, New York, February 1976.

[7] Olivier Danvy, "Semantics-Directed Compilation of Non-Linear Patterns", Technical Report 303, Indiana University, Bloomongton, IN., January 1990.

[8] A. Berlin and D. Weise, "Compiling Scientific Programs using Partial Evaluation", MIT Artificial Intelligence Laboratory Memo AIM-1145, Cambridge, MA., July 1989.

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

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

[11] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, Torben Æ. Mogensen: "A Self-Applicable Partial Evaluator for the Lambda-Calculus", *proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages*, New Orleans, Louisiana, USA (March 12-15, 1990)

[12] 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

[13] L. Robert Morris, "Automatic generation of time efficient digital signal processing software," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-25, No. 1, pps. 74–79, February 1977.

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

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

[16] G. Roylance, "Expressing Mathematical Subroutines Constructively", Proceedings of the ACM Conference on Lisp and Functional Programming, 1988.

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

[18] Philip Wadler, "Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time," in *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[19] Weise, Daniel, "Graphs as an Intermediate Representation for Partial Evaluation", Computer Systems Laboratory, Stanford University, Submitted for publication, March 1990.

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