[10] Kahn, Kenneth M., "A partial evaluator of Lisp programs written in Prolog," In M. Van Caneghem (ed.): *First International Logic Programming Conference, Marseille, France 1982,* pps 19-25, 1982.

[11] Roylance, Gerald, "Expressing Mathematical Subroutines Constructively," *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* pps 8-13, 1988.

Partial evaluation techniques are actively used by Ungar and Chambers [5] to efficiently compile *SELF*, an object oriented language. They use partial evaluation to specialize programs on the fly as the specialized (they use the word "custom") routines are needed. This work places a heavy emphasis on quickly producing the specialized procedures.

References

[1] Abelson, Harold, "A step towards the automatic analysis of dynamical systems," Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo 1174, Sept 1989.

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

[3] Berlin, Andrew, *A Compilation Strategy for Numerical Programs Based on Partial Evaluation,* Masters Thesis, Massachusetts Institute of Technology, Artificial Intelligence Laboratory Technical Report TR-1144, Cambridge, MA, July 1989.

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

[5] Chambers, Craig, and Ungar, David, "Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programing language," *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* pps 146-160, 1989.

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

[7] M. Halfant and G.J. Sussman, "Abstraction in numerical methods," *Proceedings of the ACM Conference on Lisp and Functional Programming,* pps 1-7, 1988.

[8] Jouppi, Norman, and Wall, David, "Available instruction-level parallelism for superscalar and superpipelined machines," *Proceedings of the Third Internal Conference on Architectural Support for Programming Languages and Operating Systems,* pps. 272-282, 1989.

[9] Jones, N. D., Sestoft, P., and Søndergaard, "Mix: A self-applicable partial evaluator for experiments in compiler generation," Vol 1, Nos 3/4, International Journal of Lisp and Symbolic Computation, Kluwer Publishers, 1988.

8 Summary and Future Research

We have created a system that simultaneously supports very abstract and general programming, while providing better performance than conventional compilers. Our compiler maps high- and mid-level programs into very efficeint low level programs that exhibit large amounts of instruction-level parallelism. Parallelism was exploited by a scheduler that produced close to optimal parallel schedules.

The partial evaluator itself is implemented rather simply: a new type of object called a placeholder was invented, and the implementations of the lowest primitives, such as + and * were changed to accomodate placeholders. The computation then proceeds normally to produce a program that is combined with a prologue and an epilogue to create a specialized program.

Partial evaluation needs to become automatic. Deciding which portions of code to partially evaluate can be cumbersome. We are actively investigating techniques for making partial evaluation be fully automatic. We are also interested in using partial evaluation to specialize other types of computations, such as pattern matching, parsing, compiling, performing inferences over a knowledge base, and conducting large scale database retrievals.

Better and more powerful schedulers for parallel machines need to be designed. We believe our techniques are especially well suited to exploiting the superscalar architectures that are now becoming commercially available. It is also possible to create hardware specifically designed to execute the types of programs the compiler creates. For example, large basic blocks make it feasible to use multiply interleaved memory systems built out of slow and inexpensive components. An entry-point cache would reduce the penalties normally associated with branching in pipelined memory systems.

9 Further Readings in Partial Evaluation

Partial evaluation has many uses and is becoming an active research topic. The first partial evaluators were written by the Artificial Intelligence community for Lisp. Their motivations were much like ours: to automatically remove the costs of abstraction. Ken Kahn [10] has written an excellent paper on the value and application of partial evaluation to Artificial Intelligence research.

Researchers at DIKU at the University of Copenhagen investigate partial evaluation to automatically construct compilers from denotational descriptions of programming languages. They have been making steady progress towards this goal. They report some of their results in [9].

An excellent partial evaluation source book is *Partial Evaluation and Mixed Computation* [4], which is the proceedings of a partial evaluation workshop held in the Summer of 1987. This book also contains a comprehensive bibliography of partial evaluation research.

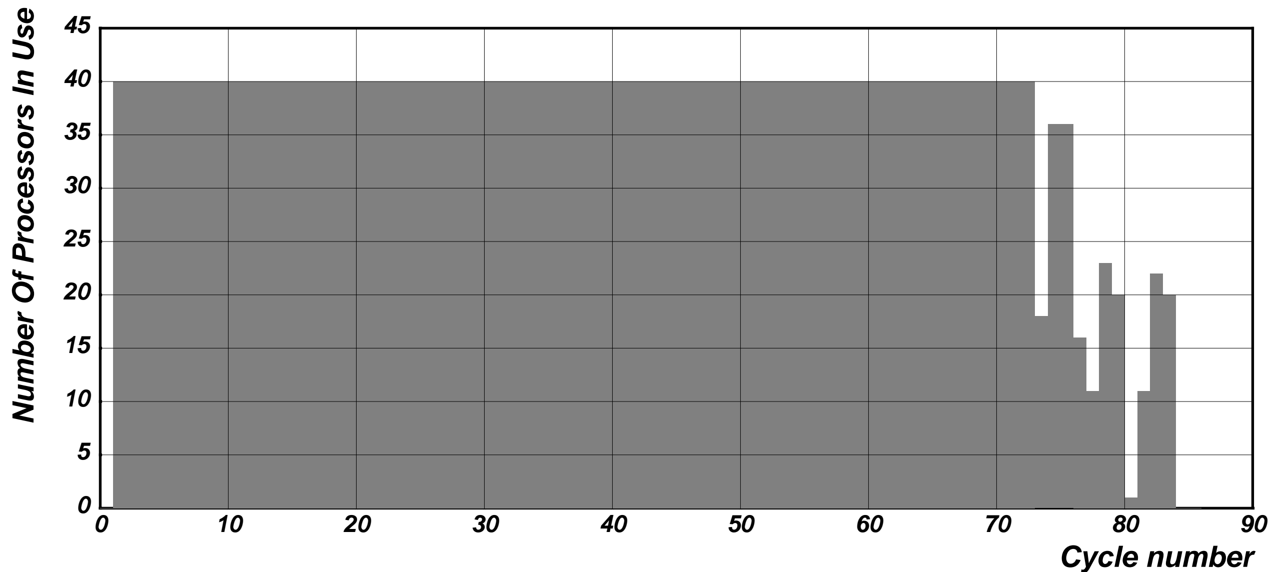


Figure 15: The result of scheduling the 9-body problem onto 40 pipelined processors with a communication latency of one cycle. A total of 85 cycles are required to complete the computation. On average, 36.4 of the 40 processors are utilized during each cycle.

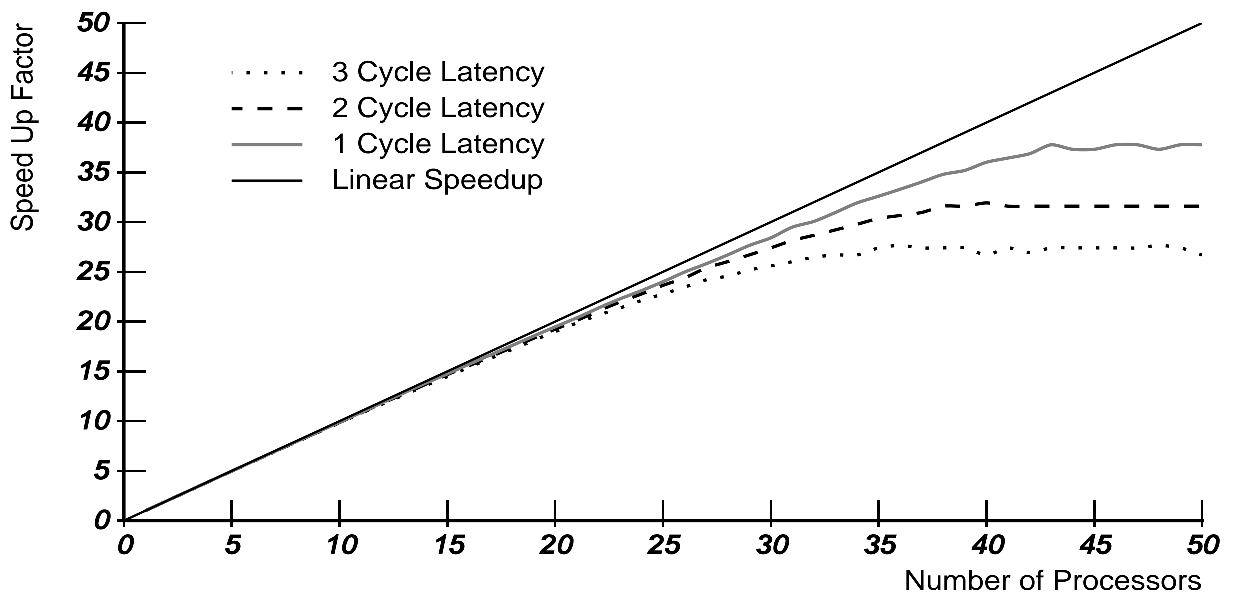


Figure 16: Effects of Communiation Latency on Speedup. The graph shows the speed-up factors over a single pipelined processor. The analysis shown is for a system composed of processors employing a 3-stage pipeline.

- Several heuristics are used to break ties. They use such information 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.

Comparing completion time of the heuristically produced schedule to the theorectically shortest completion time [3] shows that these heuristics are quite effective. On the 9-body problem, the scheduled code provided speed-ups near the theoretical limit.

7.3 Performance Measurements

Figure 15 shows the results of applying the scheduler to the 9-body problem, for a 40 processor system with a 3-stage processor pipeline and a communication latency of one cycle. The parallelism available in the problem has been distributed over the life of the computation, effectively using all 40 processors in most of the cycles. Overall, the performance improved 36-fold over that of a single pipelined processor, 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 we found that communication latency directly affects the maximum speed-up the scheduler provides (Figure 16).

7.4 Relation to Other Parallelization Research

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. Similarly, compilers for VLIW architectures [6] use *trace-scheduling* to guess which way a branch will go, allowing computations beyond the branch to occur in parallel with those that precede the branch. These techniques are limited by their preservation of the user data-structures of the original program: if the original program represented an object as a vector of vectors, the compiled program will do so as well. *Preserving data-structures imposes synchronization requirements that reduce the instruction level parallelism available to the compiler.*

Partial evaluation eliminates data structures and many conditionals to produce numerical dataflow graphs, allowing intermediate results to be used in portions of a program that would not otherwise have been reached even through trace-scheduling. This technique is orthogonal to the trace-scheduling approach: partial evaluation eliminates conditional tests related to data-structures, producing large parallelizable data-independent regions (also known as basic blocks), while trace scheduling optimizes across basic block boundaries.

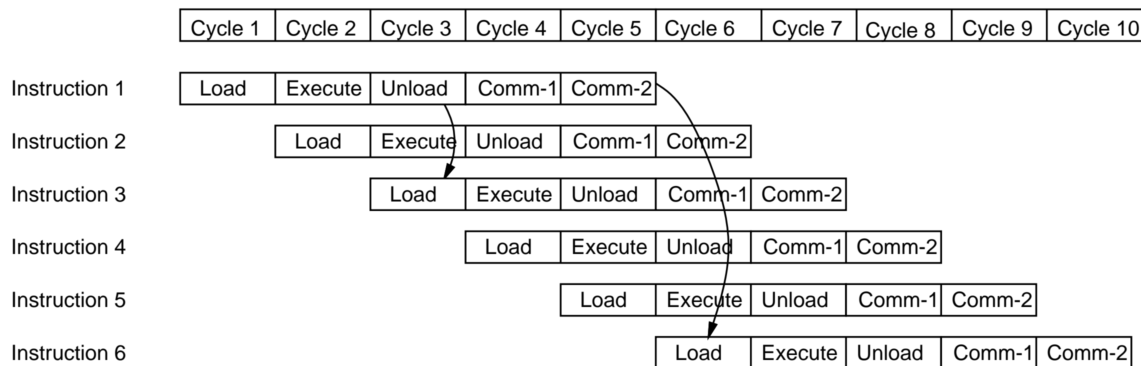


Figure 14: A 3-stage processor pipeline with a communication latency of two cycles. As indicated by the arrows, a result produced by instruction 1 can be used within the same processor by instruction 3, but can not be used by other processors until instruction 6.

---

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.

Depending on the machine model, creating an optimal schedule that completes in the shortest time possible can be an NP-Complete problem. Rather than trying to find an optimal solution to the problem, heuristics are used to select a solution that keeps the processors extremely busy. To give a flavor for the algorithm and heuristics, a brief overview of them is presented.

- A subset O of the operations whose operands have been computed is chosen corresponding to the number of processors that are available. This selection is based on the latency priorities described above.

- The operations in O whose operands have been available long enough to have been transmitted to other processors are given lower scheduling priority than those operations whose operands have been produced recently. This rule 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.

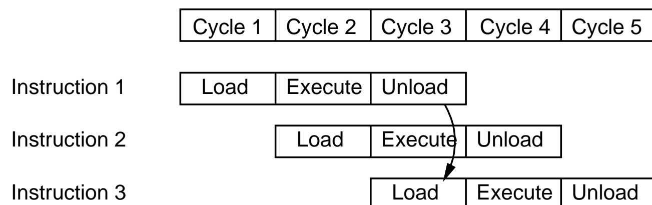


Figure 13: A typical 3-stage processor pipeline. During the `LOAD` stage, the data is loaded into the `ALU`. The result is computed during the `EXECUTE` stage, and unloaded from the `ALU` during the `UNLOAD` stage. The results produced by instruction 1 are not available to be used by instruction 2, but are available to instruction 3.

Despite this increase in the number of cycles required to execute a program, pipelining is advantageous because it reduces the length of each cycle. In addition, parallelism available in the problem can be used to hide the latency imposed by pipelining. Rather than scheduling all available parallel operations into the same cycle on many processors, 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.

Communication Latency

Processors do 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 is illustrated in Figure 14. Just as parallelism can be used to hide the latency in pipelines, parallelism can also hide the latency imposed by communication delays.

7.2 A Scheduler for Parallel Programs

The scheduler searches for a schedule that will keep each processor as busy as possible. It employs heuristics that spread the available parallelism over the processors to hide the latencies imposed by pipeline and communication delays. These heuristics schedule the critical path eagerly and schedule non-critical operations around the critical path. On the 9-body problem, the system was able to utilize 40 pipelined processors with 90% efficiency.

The scheduler operates on the numerical dataflow graph. It first computes 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

to complete the computation. Overall, these two effects tend to cancel each other out.

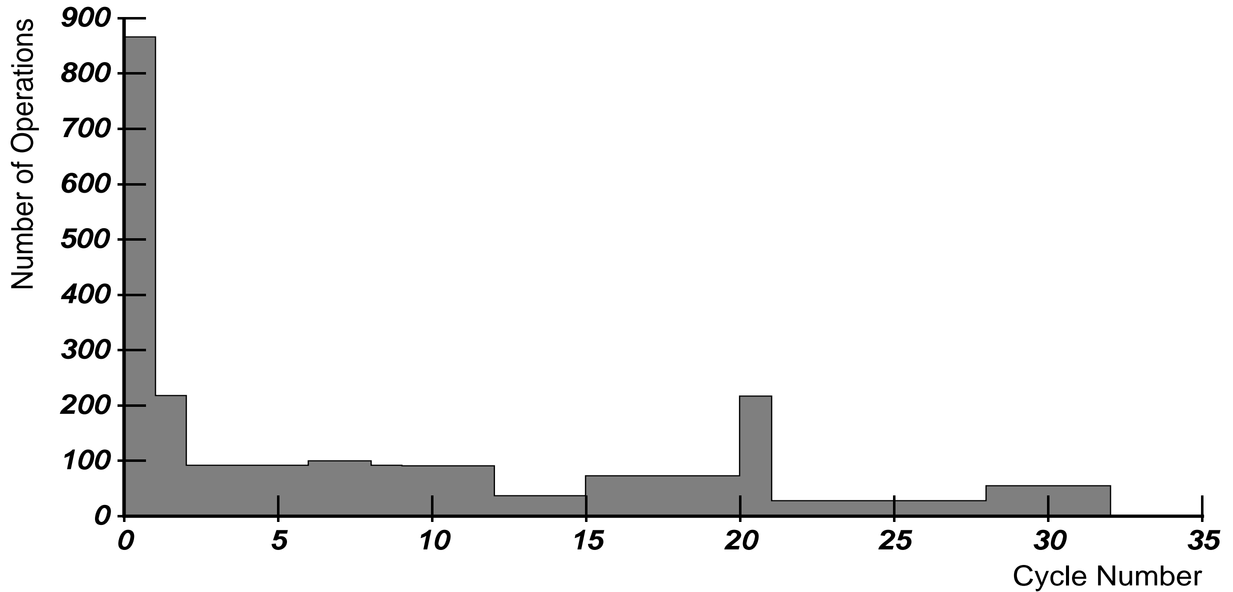


Figure 12: Parallelism profile of the 9-body problem. This graph represents the total parallelism available in the problem, accounting for the latency of numerical operations.

is the number of cycles required for the result of an operation to become available as the source of another operation. Communication latency is the number of cycles required to transfer a result between processors.

Pipelining

Technological considerations often result in pipelined architectures which overlap the execution of successive instructions within a single processor. The parallelism profile presented above was based on the assumption that the result of an instruction that finishes executing in one cycle could be used immediately in the following cycle. Unfortunately, this assumption is not valid in the presence of pipelining. Figure 13 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.[6]

[6]Since some instructions have more latency than others, the processors will sometimes be busy more than half the time. This would make our "twice as many cycles" seem too pessimistic. On the other hand, the estimate also does not consider the additional delay imposed by unloading a result from a processor before it can be loaded into another processor. 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

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
Circ Sim	20.59	2.37	0.026	791	91

Figure 11: Timings of the sample applications (in seconds). It is clear that the specialized routines 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.

The first step in these experiments is to construct a Directed Acyclic Graph (DAG) from the body of the partially evaluated program. Each node in the DAG represents an operation, and there is a directed edge from the producer of a value to each of the consumers of the value.[5] We call the DAG a *numerical dataflow graph.*

Figure 12 presents a parallelism profile for Stormer integration of the 9-body problem. This profile depicts the *maximum* amount of parallel execution that would occur if a computer had an infinite number of processors that could instantaneously communicate. The profile is produced by performing a breadth first search of the numerical dataflow graph, scheduling each operation as soon as it can be performed.

This profile differs from the parallelism profiles that commonly appear in the literature in that it accounts for the different latencies of the different arithmetic operations. (The latencies were based on the Bipolar Integrated Technologies B3110A/B3120A floating point chips.) We discovered that for double precision computations, latency differences are large enough to be of fundamental importance. For our realistic latency measures, there is a factor of 2 difference in the length of the critical path between accounting for latencies and not accounting for latencies.

7.1 Architectural Constraints that Increase Latency

Pipelining and communication delays interfere with efficient execution of numerical dataflow graphs. Both increase the effective time required to complete an operation. In pipelining, the execution of several instructions occurs simultaneously within a processor. Pipeline latency

[5] The graph is created incrementally by the partial evaluator as it constructs the body.

Electrical Circuit Simulation

Partial evaluation was applied to an electrical circuit simulator implemented in Scheme. This simulator was written abstractly to reflect as much of the underlying mathematics of simulation as possible. This abstract structure structure allows experimentation with different simulation algorithms and strategies. Partial evaluation was used to specialize this simulator for the particular circuit of interest, providing a dramatic performance improvement. The experiment we performed simulated a 120 component linear circuit where the integration time step was not specified until run time.

6.2 Performance Measurements

Our compiler was used to generate specialized routines in C for each of the applications described above. The table in Figure 11 presents timings and speed-up factors for each application running in interpreted Scheme, compiled by the Liar Scheme compiler, and compiled by our partial evaluation based compiler. None of these timings include the time required to compile the specialized C routines themselves. For abstract programs, specialization provides dramatic performance improvements.

The performance of our compiler itself has not been investigated. For our experiments, partial evaluation time ranged from tens of seconds to several minutes (all programs and timings were run on the same hardware platform). A particular problem in performing measurement experiments was compiling the specialized programs with a C compiler. The huge basic blocks that appear in specialized programs break many C code optimizers: the optimizers do not seem to terminate. This problem can be solved by generating machine code directly, a task we have not yet persued.

7 Mapping Programs onto Parallel Architectures

Partial evaluation exposes tremendous amounts of instruction-level parallelism. This is very important, as the effective use of superscalar and superpipelined processors often requires program transformations to expose the parallelism needed to keep them completely busy [8]. The first author implemented several analysis and scheduling programs to study and harness this parallelism. For a hypothetical architecture consisting of multiple ALUs and a communication network, experiments were run to measure the effects of pipeline and communication latencies on performance. It was discovered that, at least for the 9-body problem,[4] large numbers of ALUs could be kept continuously busy, thereby efficiently and effectively harnessing the available parallelism.

[4]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.

the other planets, its mass was approximated as *zero*. The partial evaluator propagated this piece of information throughout the program, eliminating numerous computations.

It is interesting to note that the for a given N, the N-body problem is entirely *data-independent*. Measurements were taken for the 6-body problem and for the 9-body problem,[2] using the Runge-Kutta (RK) integration method. We found that when the masses of the planets were provided at compile time, the partially-evaluated programs ran 11% faster than if the masses of the planets are not known until run time.

The Multipole Method Translation Operator

The multipole method approximates force interactions involving a large number of particles. The method divides space into a quadtree-like tree of cubes. Part of the force approximation propagates information up the tree from a cube to its parent. A significant portion of the computation time is spent evaluating translation operators. The translation operator is an entirely *data-independent* computation.

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. Experiments showed that roughly half the numerical operations were eliminated because of algebraic simplification involving these constants. The program was compiled for two different values of a parameter P, which denotes the number of terms in the multipole expansions.[3]

Duffing's Equation

To demonstrate the compilation of programs containing simple loops, 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 work on automatic characterization of the state space of Duffing's equation[1]. It uses an adaptive integration strategy coupled with a control loop that iterates for one period. A declaration was added to the program telling the partial evaluator not to try to unroll the control loop.

---

[2] 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.

[3] $P = 3$ 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 loops could be left intact.

grams produced by the compiler. The application programs were not modified for these experiments, except for the Duffing's equation application, in which a programmer's declaration was added indicating that the main integration loop should be left intact.

The experimental method followed was:

1. Obtain working code from researchers.

2. Select the parts of code to be partially evaluated.

3. Compile the selected code with our compiler, and produce a C program as output.

4. Compile the C program with a conventional compiler, and link it into the MIT-Scheme Lisp system, so that it can be invoked as a subroutine from Lisp.

5. Compile the program using a conventional Lisp compiler.[1]

6. Compare the execution times of the conventionally compiled program with those of the partially evaluated program.

6.1 Applications

The N-body Problem

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 can be modeled as a 10 particle system in which the forces are due to gravitational attraction. An N-body program written in Scheme by Gerry Sussman was used as a starting point for the compilation process. This program 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 in time. Our compiler was used to create a specialized version of the integration-step procedure.

The state of the system includes the planet's positions, velocities, and masses. The data description presented to the compiler left the positions and velocities unknown, but specified the masses, which are virtually time-independent. Many computations involving the planets' masses were performed at compile time. For example, since Pluto is very small relative to

---

[1]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.

5 Limitations of Partial Evaluation

Partial evaluation works best for situations where the structure of the system stays constant and only the state changes. Simulations of circuits, dams, and solar systems fall into this class. It does not work well where the structure changes or the computations are extremely data dependent. For example, partial evaluation does not work for sorting arrays, or inserting elements into balanced trees. Similarly, it is difficult to use these techniques for linear programming, because the choice of pivot is data dependent.

Another drawback of partial evaluation is that the program must be partially evaluated whenever the structure of the problem changes. This is not a problem for simulations that run for a long time, or in applications, such as circuit simulation, where multiple sets of input data and initial conditions need to be run before the system structure is changed. However, on smaller problems, where specialized code is not traversed hundreds of times, the time spent in the partial evaluator may exceed the time saved by the optimizations.

The size of the compiled program can become a problem because loops are expanded at partial evaluation time. Very large datasets and non-linear algorithms result in very large specialized programs. When code size becomes a problem, selected data structures and loops should be left intact. For example, the inner loops that deal with manipulations of a single segment of a large data structure can be partially evaluated, while the outer loop that traverses the data structure can be left intact. Our compiler leaves the choice of which loops to partially evaluate to the programmer's discretion, although decision making could conceivably be automated by the proper heuristics.

Requiring the programmer to decide which regions of a data-dependent program to partially evaluate is a limitation of our technology. Much of the partial evaluation community is investigating automatic methods for partial evaluation that do not require programmer intervention to handle data-dependent programs. Their technologies and methods for full automation have achieved many successes, and are getting more and more powerful, but are not yet able to handle the types of programs and programming styles that our partial evaluator can handle.

6 Experiments

We have applied partial evaluation to several numerically oriented scientific problems. These problems were chosen from active research at MIT and Stanford, 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, the translation operator for the Multipole Method, and an electrical circuit simulator were taken directly from code in use by researchers.

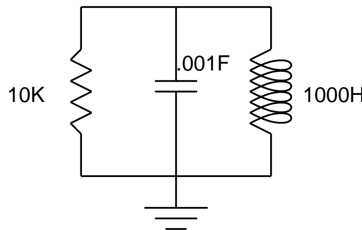
The performance measurements presented below were performed using the C syntax pro-

```
(compile next-state
         rlc-circuit
         (make-circuit-state (voltages (make-placeholder 'floating-point))
                             (currents (make-placeholder 'floating-point)
                                       (make-placeholder 'floating-point)
                                       (make-placeholder 'floating-point))
                             (make-placeholder 'floating-point))
         0.1)

Compiled NEXT-STATE,  Arguments: state

;;PROLOGUE:
Placeholder_1 = (vector-ref (vector-ref state 0) 0)
temp = (vector-ref state 1)
Placeholder_2 = (vector-ref temp 1)
Placeholder_3 = (vector-ref temp 2)
Placeholder_4 = (vector-ref state 2)

;;BODY:
Placeholder_6 = (* Placeholder_1 .00005)
Placeholder_8 = (+ Placeholder_6 Placeholder_2)
Placeholder_9 = (* Placeholder_1 .02)
Placeholder_10 = (+ Placeholder_9 Placeholder_3)
Placeholder_12 = (- Placeholder_10 Placeholder_8)
Placeholder_14 = (* Placeholder_12 49.6277915633)
Placeholder_15 = (- Placeholder_14 Placeholder_1)
Placeholder_17 = (* Placeholder_15 .02)
Placeholder_18 = (- Placeholder_17 Placeholder_3)
Placeholder_19 = (+ Placeholder_14 Placeholder_1)
Placeholder_21 = (* Placeholder_19 .00005)
Placeholder_22 = (+ Placeholder_21 Placeholder_2)
Placeholder_23 = (* Placeholder_12 4.96277915633e-3)
Placeholder_24 = (+ .1 Placeholder_4)

;;EPILOGUE:
(VECTOR (VECTOR Placeholder_14)
        (VECTOR Placeholder_23 Placeholder_22 Placeholder_18)
        Placeholder_24)
```

Figure 10: Compiling `next-state` for an RLC circuit and a fixed time step. The compiler accepts a function and the partial values that describe the function's inputs. The specialized function maps a state at time t into a state at time $t + 0.1$. Constant folding produced constants such as .02 and 49.6277915633.

```
(define (next-state circuit state h)
  (let* ((matrix (create-integration-matrix circuit state h))
         (new-voltages (solve-matrix (trim-ground matrix)))
         (new-currents (compute-b-currents
                         circuit new-voltages state h)))
    (make-circuit-state new-voltages new-currents (+ h (state-time state)))))

(define (create-integration-matrix circuit state h)
  (let ((voltages (state-voltages state)) (currents (state-currents state)))
    (let loop ((components (circuit-components circuit))
               (matrix (create-nxn+1-matrix (circuit-number-of-nodes circuit))))
      (if (null? components)
          matrix
          (loop (cdr components)
                ((component-integration-method (car components))
                 matrix voltages currents h))))))
```

Figure 9: Scheme Code Fragments for Transient Analysis. These two routines constitute the inner loop of transient analysis for linear circuits. The function `next-state` accepts a circuit, state at time t, a time increment h, and returns the state at time $t + h$. It first calculates the node voltages by creating and solving a sparse matrix. Then the branch currents are computed using the node voltages. The function `create-integration-matrix` uses object oriented techniques to add the contributions of each components into the matrix: it retrieves the function for computing an element's contributions from the element iself, and then invokes the function. We show these fragments to emphasize the amount of work the simulator must perform to compute the next state. Figure 10 shows `next-state` specialized for the following RLC circuit.



```
Compiled INNER-PRODUCT,  Arguments: v1, v2

;;PROLOGUE:
Placeholder_1 = (vector-ref v1 0)
Placeholder_2 = (vector-ref v1 1)
Placeholder_3 = (vector-ref v2 0)
Placeholder_4 = (vector-ref v2 1)

;;BODY:
;;from the first iteration of inner-product-loop:
Placeholder_5 = (* Placeholder_1 Placeholder_3) ;;vector elements #0

;;from the second iteration of inner-product-loop:
Placeholder_6 = (* Placeholder_2 Placeholder_4) ;;vector elements #1
Placeholder_7 = (+ Placeholder_5 Placeholder_6) ;;compute sum

;;from the third iteration of inner-product-loop:
Placeholder_8 = (+ Placeholder_7 131.88)

;;EPILOGUE:
Placeholder_8
```

Figure 8: The compiled inner-product program, with the additional prologue and epilogue instructions required to interface to high-level Scheme programs.

not be performed during partial evaluation due to missing data. The prologue *de-structures* the input data structures presented to the program at run time, extracting a numerical value for each input placeholder. Similarly, the *epilogue* constructs the result data structures that the program is expected to return, based on the values of the result placeholders. Our compiler automatically generates the prologue's de-structuring instructions based on the location of the placeholders within the compile-time input data structures. Similarly, the compiler automatically creates the epilogue, based on the location of the placeholders within the result value V produced by partial evaluation.

The compiler also targets the body for a particular machine. For sequential computers, this usually means re-ordering the computation so as to minimize the number of intermediate results that are created, thereby minimizing memory accesses. For parallel computers, scheduling is more complicated, requiring that the computation be partitioned among multiple processors.

Figure 8 shows the inner-product example presented earlier in Figure 5, with the additional prologue and epilogue sections. For an extremely small program like inner-product, the vector references required to interface to the high-level Scheme program represent a significant cost. However, on larger examples, such as the circuit simulation program presented in Figures 9 and 10, the compilation process is far more effective. Although high-level data structure (vector) manipulations remain in the prologue and epilogue, these are insignificant compared to the number of data structure manipulations (such as manipulation of matrices) that were eliminated through partial evaluation.

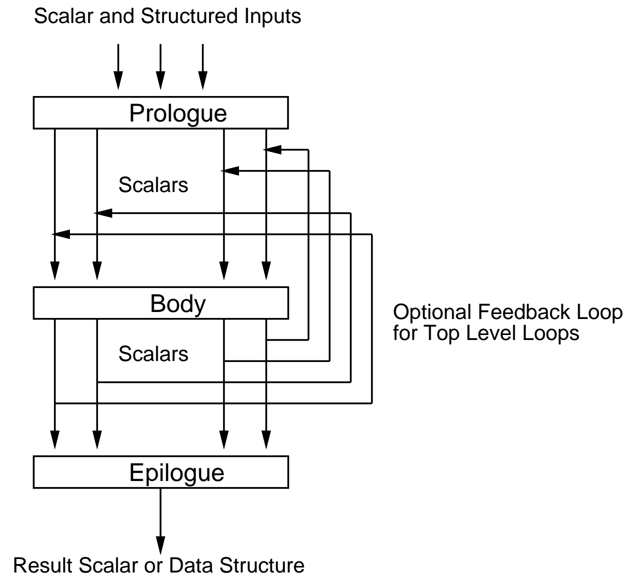


Figure 7: The three stages of the programs produced by the compiler. The *body* is created first using the partial evaluator. The *prologue* destructures the input data structures to produce values for each of the input placeholders. The *epilogue* constructs the output data structure from the placeholder values created by the body. When the body is the body of a loop, it is possible to have the body loop back directly to itself, which saves creating an output data structure and then destructuring it.

4 The Prototype Compiler

We have implemented a prototype compiler that uses partial evaluation. This compiler generates compiled code either in a generic register-transfer language, or in C syntax, or in Scheme syntax. This compiler provides support for invoking partially-evaluated programs as subroutines, in the same manner as the original Scheme programs from which they were derived. Since the original Scheme programs used high-level data structures to receive their inputs and return their results, the compiler generates a set of interface routines to convert between the scalar numerical values manipulated by the partially-evaluated subroutine and the data structures used in the calling program.

The programs produced by our compiler have three stages: a *prologue*, a *body*, and an *epilogue* (Figure 7). Partially evaluating a program yields two values: a result value V, which may be a placeholder or a data structure containing placeholders, and a list of instructions to be executed at run time, which we call the *body*. The body takes as input numerical values for each input placeholder, and performs those remaining numerical calculations that could not

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

Figure 6: 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.

Data-Dependent Programs

Partially evaluating a program via symbolic execution works well for data-independent computations, but runs into problems when applied to data-dependent computations. Most programs contain conditional branches, such as the `IF` statement, in which a predicate is evaluated to determine whether to execute the code associated with the consequent or the code associated with the alternative. For data-independent computations, the predicate can always be evaluated at compile time, since it never depends on the data being manipulated. However, in data-dependent computations, the predicate can depend on values that are not computed until run time.

Certain types of data-dependent conditionals can be partially evaluated by executing (hence generating code for) both the consequent *and* the alternative of the conditional at compile time. A conditional branch is then inserted into the compiled program to choose at run time which set of code to execute. This approach is adequate for simple selection operations, such as those associated with the absolute value, Min, and Max functions, but breaks down when used on recursive functions. Our compiler requires the programmer to explicitly declare (via a program annotation) which data-dependent conditionals can be expanded in this fashion.

There are several ways to get around the problems associated with data dependencies. The simplest method, and the one we take, is to divide the program into data-independent regions, each of which can be partially evaluated. This division of the program limits the scope of the partial evaluation optimizations, since the data structures that act as interfaces between the data-independent regions of the program cannot be eliminated. Fortunately, in scientific codes, 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 operations), with data-dependent conditionals only occuring at the end of these long computations for such operations as convergence checks and strategy selection.

The inner-product-loop is then called recursively, with `sum = <placeholder #5>` and `counter = 1`. The second iteration through the loop creates `<placeholder #6>` and `<placeholder #7>` to represent the results of the multiply and the add operations.

During the third iteration through the inner-product-loop, numerical values for the vector elements are available, allowing the multiply to proceed at compile time. The addition is delayed until run time, creating `<placeholder #8>` to represent the result of the overall computation. The program produced by the partial evaluator contains no data structures, procedure calls, or conditional tests; there are only numerical operations.

```
Inputs: Placeholder_1, Placeholder_2, Placeholder_3, Placeholder_4

;;from the first iteration of inner-product-loop:
Placeholder_5 = (* Placeholder_1 Placeholder_3) ;;vector elements #0

;;from the second iteration of inner-product-loop:
Placeholder_6 = (* Placeholder_2 Placeholder_4) ;;vector elements #1
Placeholder_7 = (+ Placeholder_5 Placeholder_6) ;;compute sum

;;from the third iteration of inner-product-loop:
Placeholder_8 = (+ Placeholder_7 131.88)

Result:
Placeholder_8
```

Figure 5: Result of partially evaluating inner-product. The multiplication of 3.14 times 42.0 to produce 131.88 took place during partial evaluation. All vestiges of the original vectors, of the inner-product loop control structure, and portions of the computation were eliminated by performing them in advance, during partial evaluation.

Traditional compiler optimizations further improve the performance of the partially evaluated program. Optimizations such as algebraic simplification, dead-code elimination, and common subexpression elimination are used to optimize the underlying numerical computation, without interference from compound data structures or abstraction mechanisms. Opportunities for these optimizations often arise when high-level data structure operations are combined, as in the subtract-vectors operation shown in Figure 6. These optimizations are often not noticed by the programmer when the optimizations 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.

Furthermore, the numerical value of the last element of each vector is known during partial evaluation. This information is encoded in the input data structures at compile time:

```
(define (inner-product v1 v2) ;;take 2 vectors as arguments
  (let ((length (vector-length v1)))
    (define (inner-product-loop sum counter)
      (if (< counter length) ;;loop through the vector elements
          (inner-product-loop (+ sum
                                 (* (vector-ref v1 counter)
                                    (vector-ref v2 counter)))
                              (+ counter 1))
          sum))
    (inner-product-loop 0 0)))

(define input-vector-1
  (vector (make-placeholder 'floating-point) ;;placeholder #1
          (make-placeholder 'floating-point) ;;placeholder #2
          3.14))

(define input-vector-2
  (vector (make-placeholder 'floating-point) ;;placeholder #3
          (make-placeholder 'floating-point) ;;placeholder #4
          42.0))

(pe vector-inner-product input-vector-1 input-vector-2)
```

When the inner-product program is run during partial evaluation, execution starts with the call to (vector-length v1), which returns 3. This is the first saving provided by partial evaluation: the vector-length call is executed, and is not included in the compiled program.

Execution continues with the call to inner-product-loop with `sum=0` and `counter=0`. (vector-ref v1 0) returns `<placeholder #1>`, and (vector-ref v2 0) returns `<placeholder #2>`. Again, these vector references are performed during partial evaluation, and will not appear in the compiled program.

`(* <placeholder #1>  <placeholder #3>) ==> <placeholder #5>`

The multiply can not proceed during partial evaluation, because numerical values for the placeholders are not yet available. A multiply instruction is emitted to perform the multiply at run time, and a new placeholder, `<placeholder #5>`, is created to represent the result of the multiply operation.

`(+  sum  <placeholder #5>) ==> <placeholder #5>`

Since sum=0, this operation is able to proceed at compile time, even though the value represented by `<placeholder #5>` is not yet available.

Expression	Result	Instruction emitted
`(+ a b)`	`placeholder_243`	`placeholder_243 := 3 + placeholder_102`
`(+ a c)`	`10`	None
`(+ x y)`	`placeholder_244`	`placeholder_244 := placeholder_243 + 10`

Figure 4: Partially evaluating `(let ((x (+ a b)) (y (+ a c))) (+ x y))`. In this example `a` and `c` are bound to 3 and 7 respectively. `b` is bound to `placeholder_102`. Partially evaluating the expression requires first partially evaluating `(+ a b)`, then `(+ a c)`, and then `(+ x y)`.

```
;; Data structure describing a specific problem.
(define mars
      (make-planet 'mars
       (/ 1 3093500) ;The mass of a planet is known at compile time.
       (3-vector (MAKE-PLACEHOLDER 'mars-position-x 'floating-point) ;p
                 (MAKE-PLACEHOLDER 'mars-position-y 'floating-point)
                 (MAKE-PLACEHOLDER 'mars-position-z 'floating-point))
       (3-vector (MAKE-PLACEHOLDER 'mars-velocity-x 'floating-point) ;v
                 (MAKE-PLACEHOLDER 'mars-velocity-y 'floating-point)
                 (MAKE-PLACEHOLDER 'mars-velocity-z 'floating-point))))
```

When the program is executed at compile time, placeholders are treated just like numbers. For example, they can be aggregated together to form lists, stored in variables or vectors, and passed as arguments to procedures. Anything that manipulates a number will also manipulate a placeholder. This allows all data manipulation operations (e.g., procedure calls and data structure manipulations) to be performed at compile time.

Our implementation of partial evaluation produces two values: a list of instructions, and a result value, which is usually a placeholder or a data-structure containing placeholders. The partial evaluator is built on top of a Scheme interpreter by modifying the behavior of its lowest level numerical operations. During partial evaluation a numerical operation that encounters numeric arguments proceeds normally, returning a numeric result. A numerical operation that encounters placeholders returns a new placeholder as output and delays itself by appending an instruction to the list of instructions (Figure 4). The compiler combines the results of partial evaluation into a specialized program.

3.1 An Example: Inner Product

To illustrate partial evaluation, consider the vector inner-product program shown below. In this hypothetical application, each input vector is known to contain 3 floating-point numbers.

mation 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.

3 Partial Evaluation of Data-Independent Programs

There is a very simple way to derive the underlying numerical computation expressed by a data-independent program: simply execute the program at compile time, and keep track of what it does! The key idea is to capture information about the particular solution that a program uses to solve a particular problem by running the program on input data structures that correspond to the problem statement. Although the actual numerical values for some pieces of data will not be known until run time, their location within the data-structures will be known at compile time. Those numerical values which are not yet available 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.

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 and velocities of the planets, and produces the positions and velocities one time step later. At run time, a typical input datum would be:

```
;; 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
```

Because the planets are in different positions each time the program is run, numerical values for the positions are not known at compile time. Nonetheless, their locations in the data structures, and their types are known, as expressed by the following data structure.

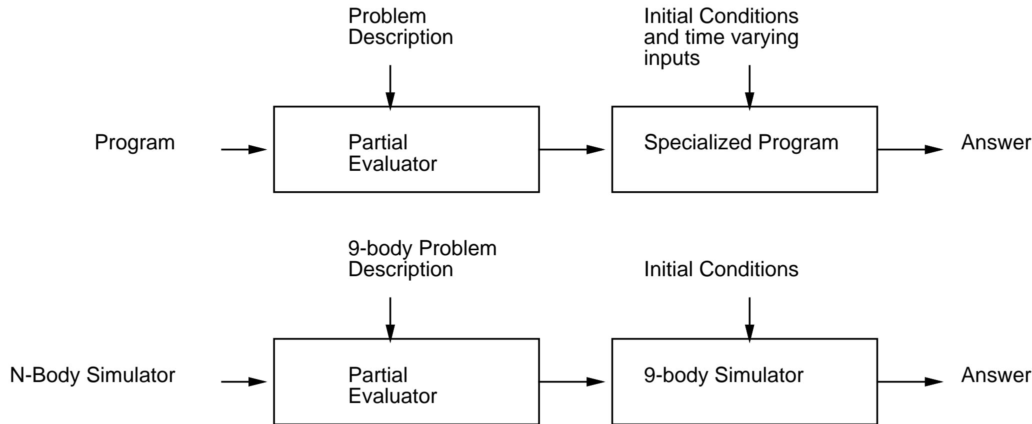


Figure 3: A partial evaluator transforms a general program into one specialized for a given problem. For example, given a high-level program that computes the motion of a collection of planets, together with the fact that the particular problem being studied involves exactly 9 planets, partial evaluation produces a low-level program that computes the motion of a 9-planet solar system for varying initial conditions.

2 Compiling for a Particular Problem

Partial evaluation transforms a general (high or mid-level) program into a specialized (low-level) program by taking advantage of information that is available at compile time about the data structures that the program will be run on (Figure 3). This strategy differs from conventional compilation techniques, in that conventional compilers seek to optimize the execution of procedure calls and data structure manipulations, whereas partial evaluation seeks to eliminate these operations by performing them in advance, at compile time.

Partial evaluation is especially effective on scientific programs because these programs have a special property: they are mostly data-independent. A program is data-independent when the sequence of operations it performs does not depend upon the results of the computation. For example, for any given matrix size, matrix-multiply is data-independent, in that it performs a fixed set of multiplications, regardless of the numerical values of the numbers being multiplied. 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.

Considering information about the data associated with a particular problem is very important, because many data-dependent programs become data-independent once infor-

```
(define (make-integrator F time-step)  ;;;make-integrator takes as arguments
                                       ;;; the function to be integrated, F,
                                       ;;; and the time-step.
  (define (produce-next-state current-state)
    (define k0 (scale-system time-step (F current-state)))
    (define k1 (scale-system time-step
                             (F (add-systems current-state
                                             (scale-system 1/2 k0)))))
    (define k2 (scale-system h
                             (F (add-systems current-state
                                             (scale-system 1/2 k1)))))
    (define k3 (scale-system h
                             (F (add-systems current-state
                                             (scale-system 1/2 k3)))))
    (define new-state
      (scale-system 1/6
                   (add-systems k0
                                (scale-system 2 k1)
                                (scale-system 2 k2)
                                k3)))
    new-state) ;;produce-next-state returns new-state
  produce-next-state) ;;;make-integrator returns produce-next-state
```

Figure 2: Fourth Order Runge-Kutta Integrator. This high-level program composes existing functions to literally construct a new procedure that performs an integration step. The composition is performed at an abstract level, with no attention to how the state of a system is represented, or to how storage is to be managed. The procedure *make-integrator* takes as input the function to be integrated and the integration time-step; it then *creates* and returns a new procedure. When run, this procedure will take as input the current state of the system, and then perform an integration step to produce the system state corresponding to one time-step later. More examples of abstraction in numerical computation appear in [7] and [11].

Overview of Scheme

Scheme is in the Lisp family of languages. All objects, whether they be data structures or continuations, are dynamically created, and have indefinite extent. This means that they can be created at any point, and, once created, they are only reclaimed by the storage system when a program drops all references to them. The primitive datatypes in Scheme include numbers, lists, vectors, and procedures.

The different types of Scheme expressions used in this paper are:

Expression	Meaning
(define <name> <exp>)	Defines <name> to have the value returned by <exp>.
(define (<name> <f1> ... <fn>) <exp>)	Defines a procedure having name <name>, formal parameters <f1> through <fn>, and body <exp>.
(let <binding-list> <exp>)	<binding-list> is a list of name-expression pairs. The expressions are evaluated, then the resulting values are bound to the names, then <exp> is evaluated.
(let* <binding-list> <exp>)	Like let, except that the bindings are processed serially: each name-expression pair is evaluated and bound in turn.
(if <pred> <then> <else>)	First <pred> is evaluated. If it evaluates to true the <then> expression is evaluated, otherwise the <else> expression is evaluated.
(<exp1> <exp2> ... <exp3>)	When the first element of an expression is not a reserved keyword such as if or define, an expression denotes a function call. Each expression is evaluated, and then the result of evaluating the first expression is applied to the other values.

Some of Scheme's built-in functions are:

Function	Description
+, -, *, /	Arithmetic operations
vector	Creates a one dimensional array
vector-ref	Retrieves an element from a one dimensional array
vector-length	Computes the length of a one dimensional array
cons	Creates a pair (a tuple of length two)
car, cdr	Retrieve the first and second element of a pair, respectively.

Figure 1: A very brief overview of Scheme.

particular circuit for different inputs. These programs, because they solve only one problem, are very efficient. Unfortunately, they are rarely worth writing by hand, since their usefulness is limited to one particular problem. At the middle, or conventional level, are programs typically written in C or Fortran that solve a class of problems, such as programs for solving the N-body problem, for analyzing dams, or for simulating circuits. They are more versatile than the lowest class, but are less efficient. At the highest level are programs constructed using advanced abstraction mechanisms such as higher order procedures, automatic storage mechanisms, and object oriented methods. These programs, usually written in Lisp or Smalltalk, embed representation and control choices in the data objects being manipulated. They are the easiest to construct, and the most versatile because they can be adapted and reused more easily than conventional programs. They are the least efficient because of the computational cost imposed by the abstraction mechanisms.

As an example of a high-level program, consider the problem of numerical integration of an unknown function, F, that computes the rate at which a system is changing. Figure 2 shows a program, expressed in Scheme (Figure 1), that, when given a function F, dynamically creates a new procedure that performs the integration. Notice that this program is totally independent of the particular function being integrated, of the data structures used to represent the state of the system, and of the storage allocation strategy. This style of programming is quite flexible, allowing the code for the integrator to be used in many different applications, and making it feasible to create a very general library of numerical techniques that operates independently of data representations and storage maintenance strategies. More detailed and powerful examples of abstraction in numerical computation appear in [7] and [11].

The same flexibility that makes high-level languages expressive also reduces their efficiency. High-level programs are inefficient because maintaining abstraction mechanisms (e.g., dynamic storage allocation, object method dispatching, and higher order procedures) requires computation. Also, the general nature of an individual procedure does not provide enough information for the compiler to predict what computation needs to be performed. For example, efficiently compiling the make-integrator program shown in Figure 2 would be quite difficult – the compiler does not know what function will be integrated, or what kind of data structure "add-systems" will manipulate.

Compilation can improve high-level programs by optimizing references to variables such as "time-step," passing parameters in registers, inlining small functions, and performing interprocedural analysis [2]. Although these optimizations do provide significant performance improvement, the performance of the compiled programs still falls far short of that of the low-level numerical programs that an expert programmer would write: the high-level aspects of the program, such as the procedure calls and data structure manipulations, remain in the compiled program, imposing a performance penalty. This ineffeciency remains because static analysis limits itself to consideration of the code for a program – the instructions for manipulating the data – but does not consider information about the data itself.

Introduction

Scientists are faced with a dilemma: Either they can write abstract 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. Partial evaluation can provide a solution to this dilemma by providing the missing link between the code presented to the compiler and the computation envisioned by the programmer.

Partial evaluation is a technique for converting a high-level program into a low-level program that is specialized for a particular application. The key idea is that rather than just considering a program's code, the compiler can also consider information that is available at compile time about the data structures that the program will manipulate. In scientific applications, there is often enough information available at compile time to allow data manipulation operations to be performed in advance, leaving only the underlying numerical computation to be performed at run time. This approach eliminates nearly all of the programer's control and data abstractions at compile time, producing high-performance code.

We have implemented a prototype compiler that uses partial evaluation. Experiments with our compiler have shown that for an important class of numerical programs, partial evaluation can provide dramatic performance improvements: we have measured speedups over conventionally compiled code that range from seven times faster to ninety one times faster. These experiments have also shown that by eliminating inherently sequential data structure references and their associated conditional branches, partial evaluation exposes the low-level parallelism inherent in a computation. By coupling partial evaluation with parallel scheduling techniques, this parallelism can be exploited for use on heavily pipelined or parallel architectures. We have demonstrated this approach by applying a parallel scheduler to a partially evaluated program that simulates the motion of a nine body solar system.

1 Abstraction and High Level Programs

High-level languages such as Lisp are very powerful in that they allow computations to be expressed in terms of abstract numerical methods and techniques, using abstractions to mirror the way that a person thinks about a problem. This is in contrast to the programming methodology traditionally associated with languages such as Fortran, in which programmers apply the numerical techniques themselves in order to derive the numerical computation required for a particular problem, and then use the programming language only to express the results of their efforts [11].

Programs can be classified according to their versatility and ease of construction. At the lowest level are programs that can be applied to only one problem, such as the 3-body problem, or the analysis of a given dam under different loads, or the transient behavior of a

Compiling Scientific Code using Partial Evaluation

Andrew Berlin and Daniel Weise

Technical Report: CSL-TR-90-422
(Also FUSE Memo 90-2)

March 1990

Andrew Berlin
Massachusetts Institute of Technology
Project MAC
545 Technology Square
Cambridge, Massachusetts 02139

Daniel Weise
Stanford University
Computer Systems Laboratory
Stanford, California 94305-4070

Abstract: Partial evaluation converts a high-level program into a low-level program that is specialized for a particular application. We describe a compiler that uses partial evaluation to dramatically speed up programs. We have measured speedups over conventionally compiled code that range from seven times faster to ninety one times faster. Further experiments have also shown that by eliminating inherently sequential data structure references and their associated conditional branches, partial evaluation exposes the low-level parallelism inherent in a computation. By coupling partial evaluation with parallel scheduling techniques, this parallelism can be exploited for use on heavily pipelined or parallel architectures. We have demonstrated this approach by applying a parallel scheduler to a partially evaluated program that simulates the motion of a nine body solar system. (This report revises and replaces MIT AI Laboratory Memo AIM-1145.)

Key Words and Phrases: Partial Evaluation, Program Transformation, Scientific Computation, Parallel Architectures, Parallelizing Compilers

Compiling Scientific Code using Partial Evaluation

Andrew Berlin
Daniel Weise

Technical Report No. CSL-TR-90-422

March 1990

This report describe research done at the Artificial Intelligence Laboratory of the Massacshusetts Institute of Technology and at the Computer Systems Laboratory of Stanford University. The MIT AI Laboratory's research is supported in part by the Advanced Research Projects Agency of the Department of Defense under ONR contract N00014-86-K-0180. CSL's research is supported in part by Contract N00014-87-K-0828. Daniel Weise is also supported by NSF Contract MIP-8902764.