MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 1145 *July 1989*

Compiling Scientific Code using Partial Evaluation

Andrew Berlin
Project MAC
Massachusetts Institute of Technology

Daniel Weise
Computer Systems Laboratory
Stanford University

Abstract: 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. We have developed a compiler that uses partial evaluation and scheduling techniques to provide a solution to this dilemma.

Where conventional compilers compile a program without any knowledge of the data the program will be run on, our system uses information about the data when transforming the program. This technique, by eliminating nearly all the user's control and data abstractions, produces high performance code. For an important class of numerical programs, partial evaluation dramatically improves performance: we have achieved speedups over conventionally compiled code that range from seven times faster to ninety one times faster.

We also show how partial evaluation can be applied to the programming of parallel computers. By eliminating inherently sequential data structure references and their associated conditional branches, partial evaluation exposes the low-level parallelism inherent in a computation. We present the results of applying a parallel scheduler to a partially evaluated program that simulates the motions of nine bodies under mutual gravitational attraction.

Keywords: Partial Evaluation, Scientific Computation, Parallel Architectures, Parallelizing Compilers

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. Some portions of the first author's work were previously reported in AI TR-1144.

Introduction

This research aims to remove the problem that while the mathematical description of physical phenomena is abstract, concise, and elegant, the code that simulates physical phenomena, and supposedly implements the mathematics, is opaque, verbose, and unwieldy. Code has been this way because programming systems – the language and compiler together – have not collaborated to produce efficient code for abstractly specified computations. We have created a system that allows scientific computations to be expressed using a high-level description analogous to the notation that describes the problem being solved. Our contribution is not in the language, which is Scheme [2], but in the compiler, which consists of a partial evaluator and a scheduler.

Where conventional compilers compile a program without any knowledge of the data the program will be run on, our system uses information about the data when transforming the program (Figure 1). The partial evaluator executes the user's code, removing data structure manipulations and performing optimizations, to create a program specialized for the data. This specialized program is very low level: in the place of data abstractions and control abstractions are explicit scalar data values and primitive numerical operations.

The scheduler uses resource planning and scheduling techniques to map the specialized program onto a particular architecture. The specialized program, internally represented as a dataflow graph, makes explicit the creation and use of each numerical value, allowing the scheduler to decide when and where each value is created and used. This power allows our system to make very efficient use of heavily pipelined and parallel machines, and reduces the need for runtime hardware support such as scoreboards or automatic caches. Our compiler produces high-performance code for serial, parallel, and special purpose architectures.

Programs have been written before to create routines specialized for given inputs. One of the earliest was [12] which would create Fortran code for performing FFT's of given sizes. Barzilai [4] created a program that created specialized digital simulators for circuits, Bryant [7] did the same for switch-level simulators. Early versions of Spice [13] could specialize the code for solving a matrix. Our work differs in that we provide a general mechanism for performing partial evaluation, thereby allowing specialization to occur over a larger portion of the overall problem than would otherwise be practical.

This paper has four sections. The first discusses abstraction and its costs. The partial evaluator, and how it removes abstractions to produce efficient code and expose parallelism, is discussed in the second section. Section 3 presents the results of applying our methods in many different domains. Using partial evaluation to program parallel computers is discussed in Section 4. We conclude with a summary and directions for future research.

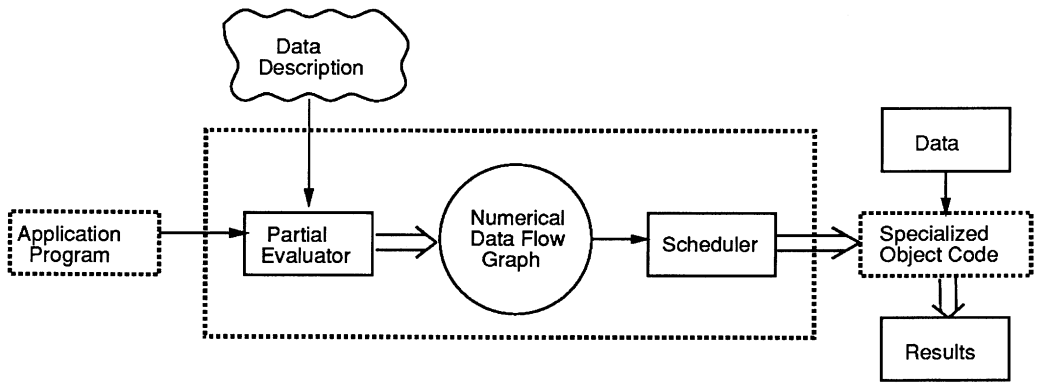


Figure 1: Our compiler consists of a partial evaluator and a scheduler. The partial evaluator accepts a description of the input data set and the program to be evaluated, producing a *numerical dataflow graph.* This graph is composed entirely of numerical operations. The scheduler maps the graph onto the target architecture.

---

1 Programming Abstractions

The purpose of abstraction in programming is to express complex ideas in simple ways. Doing so takes several guises: employing first class aggregate data values, using stateless expressions, creating objects, and inventing idioms wherever possible. Each of these abstraction mechanisms exacts a cost. In this section we elaborate on some of these abstraction mechanisms and discuss their costs.

Capturing computational idioms is very important. Wherever two pieces of code only vary a little, a new idiom should be created to subsume the two special cases. The more general the idiom, the more powerful it is, but the more overhead in getting actual "work" done. For example, consider a fully general routine for summing the results of applying an arbitrary function to a sequence of integers:

```
(define sum
  (lambda (lower upper function)
    (if (= lower upper)
        (function lower)
        (add (function lower)
             (sum (+ lower 1) upper function)))))
```

In this routine `add` is an overloaded function that dispatches on the types of its arguments.

This allows `function` to return any type of object that `add` knows how to add. The cost for such a routine in a conventially compiled language is high: runtime tags must be maintained for `add` to dispatch on, and the dispatch must be done each time through the loop.

Stateless routines reduce the conceptual overhead on the programmer. For example, consider computing a sequence of matrices which are added together. The simplest method might be to use the `sum` procedure defined above where the function passed in delivers each matrix in turn. This incurs a large overhead. Storage for the intermediate matrices must be allocated and reclaimed, and code for adding matrices must be run each time through the loop. To avoid paying the cost of intermediate matrices and matrix operations, a programmer must use a different, and less modular, method that uses a single global matrix that is incrementally added to.

Further examples would show the same symptoms: the program taking on more of the burdens of the programmer, and costing more for it. The costs we just saw included extra tag bits, runtime dispatching, and intermediate structures that must be allocated and deallocated. A system that supports abstraction cannot avoid these costs. We show in the next section how partial evaluation pays these costs exactly once, at partial evaluation time, so that using abstractions bears no runtime cost. Simultaneous with the eviction of the cost of abstraction is the manifestation of all quantities as explicit scalars, exposing all instruction-level parallelism (as we show in the Section 4).

2 Partial Evaluation

Partial evaluation [6] is a technique for compiling and specializing programs. Our partial evaluator converts a program and a symbolic description of the program's eventual inputs into a dataflow graph. The description specifies values that are known at compile time, and uses *symbolic values* to represent those values that are not known. A symbolic value is a data structure, much like a symbol table entry in a standard compiler, that represents a specific piece of missing data. Each symbolic value contains whatever information is available about the piece of data that it represents, including its type.

We have developed a particularly simple technique for performing partial evaluation by executing a program at compile time on the symbolic inputs. A dataflow graph, representing the compiled program, is built up incrementally as symbolic execution proceeds. Whenever all the operands to an operation are known, the operation is simply executed. When one or more inputs are symbolic, a new symbolic value is returned, and a node is added to the dataflow graph. Adding a node to the graph effectively delays the operation until runtime when its operands will be known.

We show examples of the output of partial evaluation, then describe the limitations of

partial evaluation.

2.1 Examples of Partial Evaluation

We now show how optimizations such as loop unrolling, type dispatch, intermediate data structure elimination, and constant folding are automatically achieved. The reader should always be aware that none of these optimizations are explicitly coded: they fall out automatically as a result of partial evaluation. These examples are not exhaustive: wherever the partial evaluator can do work, it will.

Our first example exhibits loop unrolling. Consider a program for summing the elements of a vector together:

```
(define vector-sum
  (lambda (v)
    (sum 1 (vector-length v) (lambda (n) v[n]))))
```

The expression

```
(partially-evaluate vector-sum (make-symbolic-vector 4))
```

creates the specialized version of `vector-sum` for vectors four elements long:

```
(lambda (v)
  (add v[1] (add v[2] (add v[3] v[4])))).
```

The function `make-symbolic-vector` produces a symbolic vector of the specified length. This vector is traversed at compile time. We can successfully apply the produced program to any vector of length four.

One of the most important optimizations is the elimination of short-lived intermediate structures. For example, consider adding complex numbers together, the code for which might be

```
(define complex+
  (lambda (a b)
    (make-complex (+ (complex-real a) (complex-real b))
                  (+ (complex-imag a) (complex-imag b))))).
```

Adding several complex numbers together one at a time will produce several intermediate complex numbers, all of which are represented by vectors. The expression

```
(partially-evaluate vector-sum (make-symbolic-vector 4 'complex))
```

creates (the flowgraph for) the following program

```
(lambda (v)
  (let ((v1 v[1]) (v2 v[2]) (v3 v[3]) (v4 v[4]))
    (make-complex (+ (complex-real v1)
                     (+ (complex-real v2)
                        (+ (complex-real v3)
                           (complex-real v4))))
                  (+ (complex-imag v1)
                     (+ (complex-imag v2)
                        (+ (complex-imag v3)
                           (complex-imag v4)))))))
```

Had this code been compiled using standard techniques, then, at run time, several intermediate structures for complex numbers would be generated and then thrown away. Those structures were only used to route information from one function to the next. Because the routing is done at partial evaluation time, the structures doing the routing do not need to exist at run time. Eliminating intermediate structures this way is a generalization of Wadler's "listless programming" [16].

We now consider constant folding. Consider the inner product function, defined as

```
(lambda (x y)
  (sum 1 (vector-length x) (lambda (k) (times x[k] y[k]))))
```

where `times` does data dependent dispatching, just as `add` did. To create a specialized version of inner product for two vectors of length three where the first element of the first vector is always 1 and the second element of the second vector is always 0, we create the specialized function as follows:

```
(partially-evaluate
  inner-product
  (vector 1.0 (make-symbolic-float) (make-symbolic-float))
  (vector (make-symbolic-float) 0.0 (make-symbolic-float)))
```

which creates

```
(lambda (x y) (float+ y[1] (float* x[3] y[3])))
```

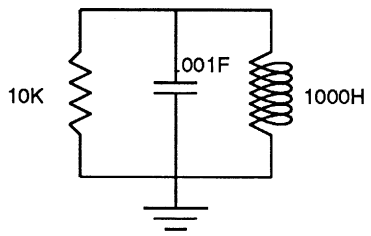
```
(LAMBDA (STATE)
  (LET*
      ((TEMP1 (VREF STATE 1))          (TEMP2 (VREF STATE 0))
       (TEMP3 (VREF TEMP1 1))          (TEMP4 (VREF TEMP2 0))
       (TEMP5 (* TEMP4 .00005))        (TEMP6 (+ TEMP5 TEMP3))
       (TEMP8 (VREF TEMP1 2))          (TEMP9 (* TEMP4 .02))
       (TEMP10 (+ TEMP9 TEMP8))        (TEMP12 (- TEMP10 TEMP6))
       (TEMP13 (VREF STATE 2))         (TEMP14 (* TEMP12 49.6277915633))
       (TEMP15 (- TEMP14 TEMP4))       (TEMP17 (* TEMP15 .02))
       (TEMP18 (- TEMP17 TEMP8))       (TEMP19 (+ TEMP14 TEMP4))
       (TEMP21 (* TEMP19 .00005))      (TEMP22 (+ TEMP21 TEMP3))
       (TEMP23 (* TEMP12 4.96277915633e-3)) (TEMP24 (+ .1 TEMP13)))
    (VECTOR (VECTOR TEMP14) (VECTOR TEMP23 TEMP22 TEMP18) TEMP24)))
```

Figure 2: Compiled code for the transient analysis of a simple RLC circuit.

---

where `float+` and `float*` perform floating point addition and multiplication, respectively. Note that the multiplications of `y[1]` by 1.0 and `x[2]` by 0.0 were performed by the partial evaluator. This example is not as contrived as it seems. One could easily imagine a graphics system that would benefit from specializing affine transformations.

We close this subsection with an example of the program (Figure 2) produced when a circuit simulator that uses nodal analysis and trapezoidal integration is partially evaluated on the simple RLC circuit shown below. The stepsize was chosen at compile time to be 0.1 seconds.



The simulator generates the voltages at time $t + h$ from the state at time t by creating and solving a sparse linear system. From the node voltages at time $t + h$ and the state at time t, it computes the branch currents at time $t + h$. The simulator generates the linear system by summing together the current and conductance contributions of each component. The simulator is object oriented: each time-varying and reactive component carries its own function for computing its contribution to the system M. These functions must be retrieved and invoked at each time step.

The "straight-lineness", compactness, and lack of structured values are the striking at-

tributes of the compiled code. No vestiges of the matrices produced and consumed during compilation, or of the control structures for doing so, or of the gaussian solution, or of the function retrievals and applications, appear in the final code. All address calculations vanish.

2.2 Benefits of Partial Evaluation

There are many algorithms that are specializations of general algorithms. Many of these algorithms are automatically recreated by partial evaluation. For example, consider a system of simultaneous linear equations. The algorithm employed for solution depends upon the shape of the system. If the system is known to be tridiagonal then an algorithm for solving tri-diagonal systems is used. A system with very few non-zero entries is solved using Gaussian elimination. The algorithm for solving tridiagonal systems is a specialization of the general Gaussian elimination algorithm. Partial evaluation produces the same program for a solving a tridiagonal matrix regardless of whether the general or specialized algorithm is partially evaluated. This is an important result: much programmer time can be saved by letting the partial evaluator do the specializations the programmer used to do.

We have found it simpler to write general programs and let the partial evaluator specialize them. This is an example of an important capability provided by partial evaluation: the ability to employ generalized libraries. This capability is particularly powerful in the context of a language such as Scheme that supports first-class procedures. Rather than requiring the programmer to code particular computations, partial evaluation makes it practical to code abstract numerical methods. For example, the Runge-Kutta integration method can be defined independently of the particular function being integrated. A more complex integrator can be created by combining the Runge-Kutta integrator with a quality-control strategy that examines the results produced by runge-kutta. This combination can then be specialized for the particular function being integrated:

```
(define fancy-integrator
  (quality-control-strategy
    (make-runge-kutta-integrator
      (particle-force gravitation))))
```

Halfant [10] shows that by using abstraction and higher order procedures, it is possible to create more powerful problem solving strategies than it is feasible to produce by hand. Partial evaluation makes this approach practical by specializing the particular combination of routines used to express a computation.

2.3 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 as well 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.

One drawback of our approach is that a program must be recompiled when the structure of the problem changes. Recompilation is only a minor inconvenience, as simulations run for a very long time. As long as the recompilation cost is a small fraction of the overall time, partial evaluation is worthwhile. More importantly, initial conditions can be changed and new experiments performed without recompiling.

Because loops are unrolled at partial evaluation time, code blowup can be a problem. In particular, for algorithms that are linear in the size of the data, the code space will also be a linear function of the data. This is not a problem for a simulation of a ten, hundred, or possibly even thousand particle systems, but will cause problems for larger systems. When the algorithm is polynomial, as it is for solving linear systems, the code size can be quadratic or cubic in the data size. When this becomes a problem, we manually choose the parts of the computation to fully unroll and the parts not to unroll.

3 Applications and Results

We have applied these techniques 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 [15], the solution to Duffing's equation [1], the translation operator for the Multipole Method [17], and electrical circuit simulation were taken directly from code in use by researchers. We first discuss the experiments, then present performance measurements.

The experimental method followed was:

1. Obtain working code from researchers.
2. Select the parts of code to be partially evaluated.
3. Compile the code, and produce a C program as output.

4. Compile the C program and link it into the Scheme system[1] as a high-performance subroutine.

3.1 Experiments

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 to the compiler left the positions and velocities unknown, but specified 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 the other planets, its mass was approximated as *zero*. The partial evaluator propagated this piece of information throughout the program, eliminating numerous computations.

Several measurements were taken to determine the effectiveness of our techniques. Tests were run for both the 6-body problem and the 9-body problem,[2] using the Runge-Kutta (RK) integration method. When the masses of the planets are known at compile time, the compiled programs run up to 11% faster.

[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.

[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.

The Multipole Method Translation Operator

The multipole method approximates force interactions involving a large number of particles. The method, as described in [17], involves dividing space up into a quadtree-like tree of cubes. Part of the force approximation involves propagating information up the tree from a cube to its parent. A significant portion of the computation time is spent evaluating translation operators.

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 instructions 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 Hal Abelson's work on automatic characterization of the state space of Duffing's equation. It uses an adaptive integration strategy coupled with a control loop that iterates for one period.

Electrical Circuit Simulation

In Section 2 we presented the result of partially evaluating an electrical circuit simulator on a simple RLC circuit. The simulator performs transient analysis of circuits using nodal analysis and trapezoidal integration. The simulator was written abstractly to reflect as much of the underlying mathematics of simulation as possible. The simulator's simple structure allows experimentation with different simulation algorithms and strategies. The experiment we performed simulated a 120 component linear circuit where the time step was not specified at compile time.

[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.

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 3: 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.

3.2 Performance Measurements

The compiler generates high-performance C programs that are called directly from Scheme programs. The table in Figure 3 presents performance measurements for the applications described above. It presents timings and speedups for each application running interpreted, compiled by the Liar Scheme compiler,[4] and compiled by our compiler. The table clearly shows that specialization provides dramatic performance improvements.

3.3 Standalone Experiment Against Tuned C Programs

As an experiment, a standalone C program was produced for the circuit simulation mentioned above. We then ran Spice3 [14] on the circuit while attempting to maintain the same experimental procedures (number of iterations, size of timestep, etc.) We found that the program our system produced took 12 seconds to execute for 1000 timesteps, whereas Spice3 took 164 seconds. The specialized program used a time step specified at compile time.

We are apparently performing 14 times faster than Spice3. This speedup factor is misleading because Spice3's method are slightly different than ours. It performs one extra matrix solve per timestep, and continuously estimates the integration error. We estimate that a more careful experiment would yield a speedup of around 5, which is still excellent. We are actively working on obtaining more accurate speedup numbers.

[4]The version of Liar we employed close-coded floating point operations. Had it open coded them, as C compilers do, the numerical performance would have increased by a factor of 4. However, this does not significantly affect our measurements because the majority of instructions in standardly compiled code do not use floating point operations.

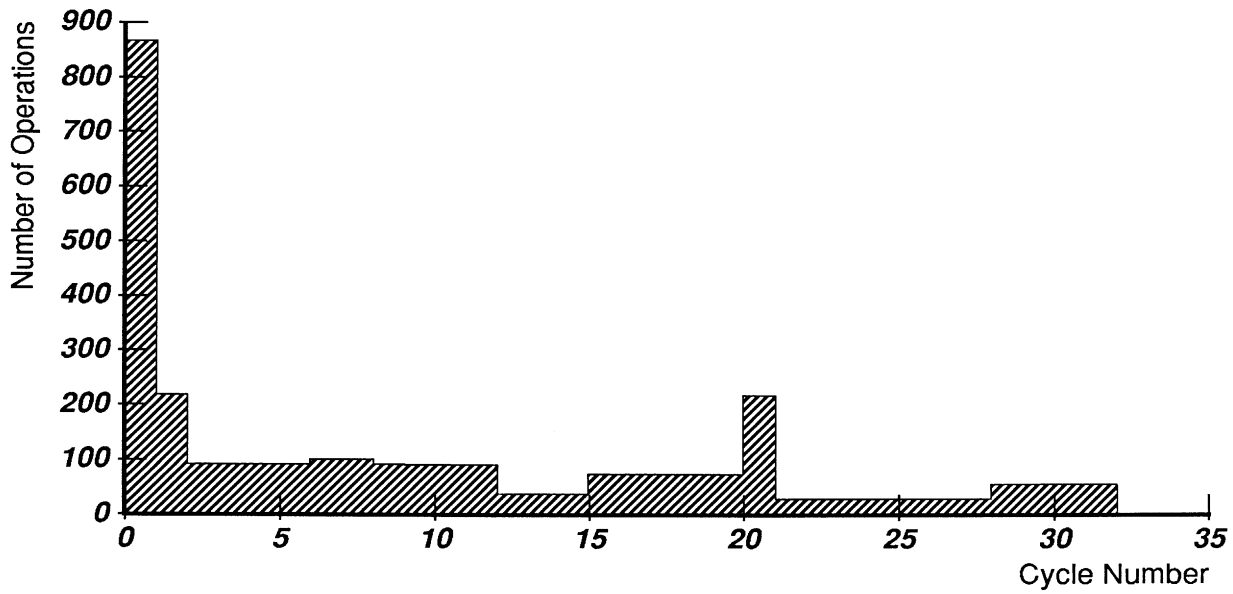


Figure 4: Parallelism profile of the 9-body problem. This graph represents the total parallelism available in the problem, accounting for the latency of numerical operations.

4 Mapping Numerical Dataflow Graphs onto Parallel Architectures

Partial evaluation exposes tremendous amounts of instruction-level parallelism. 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, we measured the effects of pipeline and communication latencies on performance. It was discovered that, at least for the 9-body problem, large numbers of ALUs could be kept continuously busy, thereby efficiently and effectively harnessing all the available parallelism.

The experiments we present were performed using the 9-body problem.[5] We first present graphs that prove large amounts of parallelism are created by partial evaluation. Next we describe the constraints on the target architecture, such as pipelining and communication costs, and show how these constraints limit the amount of parallelism that can be harnessed. We then discuss the scheduler that maps numerical dataflow graphs onto the architecture and present parallelism results. We close with a comparison to other work.[6]

[5]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.

[6]For a more detailed discussion of this research see [5].

4.1 Exposed Parallelism

Figure 4 presents a parallelism profile [3] for Stormer integration of the 9-body problem. The profile describes 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.

4.2 Architectural Constraints that Increase Latency

Pipelining and communication delays work against rapid 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 delay is the number of cycles required for the result of an operation to become available as the source of another operation. Communication delay is the number of cycles spent transferring data between processors. We consider each constraint in turn.

Pipelining

Technological considerations often lead to overlapping 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 5 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.[7]

[7]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 to pessimistic. On the other hand, the estimate also does not consider that a result must first be unloaded from a processor before it can be loaded

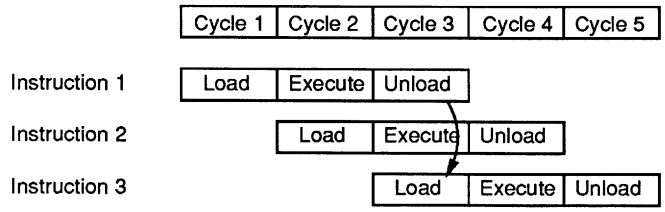


Figure 5: A typical 3-stage processor pipeline. During the LOAD stage, the data is loaded into the ALU. The result is computed during the EXECUTE stage, and unloaded from the ALU during the UNLOAD stage. The results produced by instruction 1 are not available to be used by instruction 2, but are available 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, it is possible to use some of the parallelism available in the problem 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

In practice, processors can not communicate instantaneously. The time required to move a result from one processor to another limits how soon the result can be used by a subsequent instruction. This has an effect that is similar to increasing the length of the pipeline, as illustrated in Figure 6. Just as parallelism can be used to hide the latency in pipelines, parallelism can also hide the latency imposed by communication delays.

4.3 A Scheduler for Parallel Programs

The scheduler finds a schedule that keeps 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 over 90% efficiency.

---

into another one. This creates a one cycle cost to moving data between processors, even when there are no communication delays. This effectively increases the minimum number of cycles required to complete the computation. Overall, these two effects tend to cancel each other out.

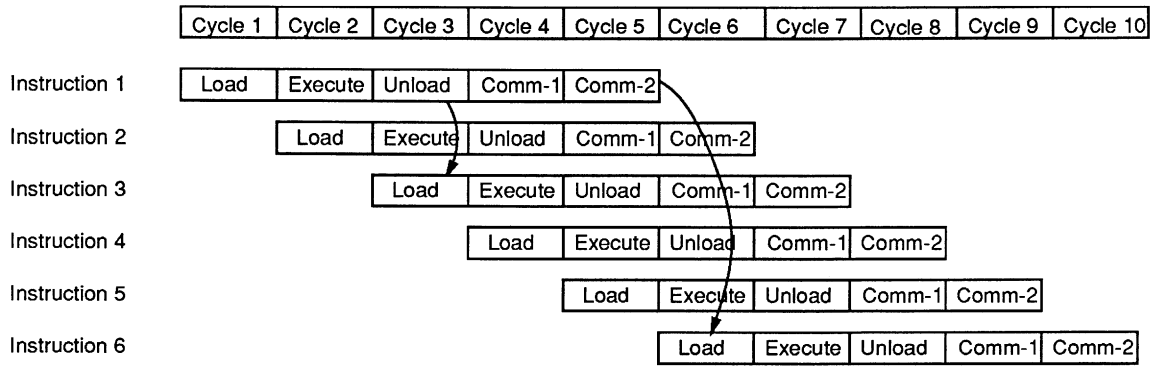


Figure 6: A 3-stage processor pipeline with a communication latency of two cycles. As indicated by the arrows, a result produced by instruction 1 can be used within the same processor by instruction 3, but can not be used by other processors until instruction 6.

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 given to those operations that lie in the critical path of the computation. If all available processors are not needed to work on the most critical path, computations from less critical paths are scheduled.

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

- A set of operations is chosen corresponding to the number of processors that are available. This selection is based on the latency priorities described above.
- Among the "chosen operations", those whose operands have been available long enough to have been transmitted to other processors have lower scheduling priority than those operations whose operands have been produced recently. This gives priority to non-relocatable computations.
- A computation whose operands were produced by a processor will be scheduled in that same processor wherever possible.
- The number of connections between processors is kept to a minimum. When the operands of a computation must be transmitted from one processor to another, the

scheduler attempts to choose a pair of processors that have communicated with each other in the past.

- Several heuristics exist for breaking ties. These take into account such factors as the memory usage within each processor, the number of computations that are waiting for a particular result, and the frequency with which processors use the communication network.

We have found these heuristics to be quite effective. On the 9-body problem, the scheduled code provided speed-ups near the theoretical limit.

4.4 Performance Measurements

Figure 7 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. Notice how 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, these variations are summarized by Figure 8. This graph clearly shows that communication latency directly affects the maximum speed-up the scheduler can provide.

4.5 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 [9] 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.

Our method eliminates data structures and many conditionals to produce numerical dataflow graphs. Intermediate results are used in portions of a program that would not otherwise have been reached even through trace-scheduling. This technique is orthogonal to

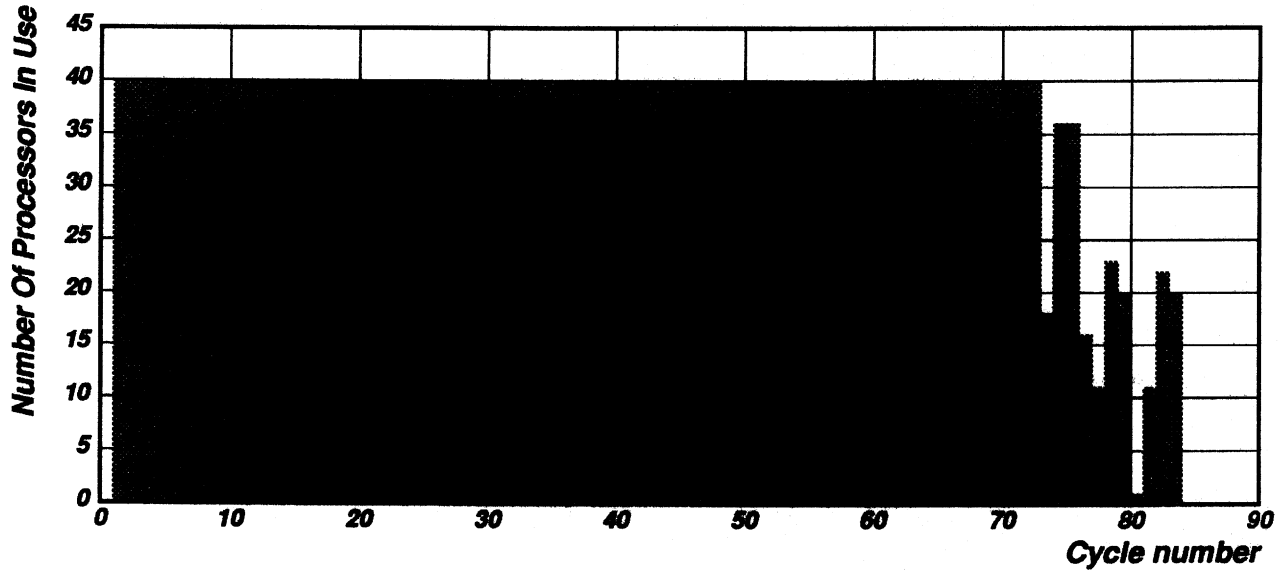


Figure 7: The result of scheduling the 9-body problem onto 40 pipelined processors with a communication latency of one cycle. A total of 85 cycles are required to complete the computation. On average, 36.4 of the 40 processors are utilized during each cycle.

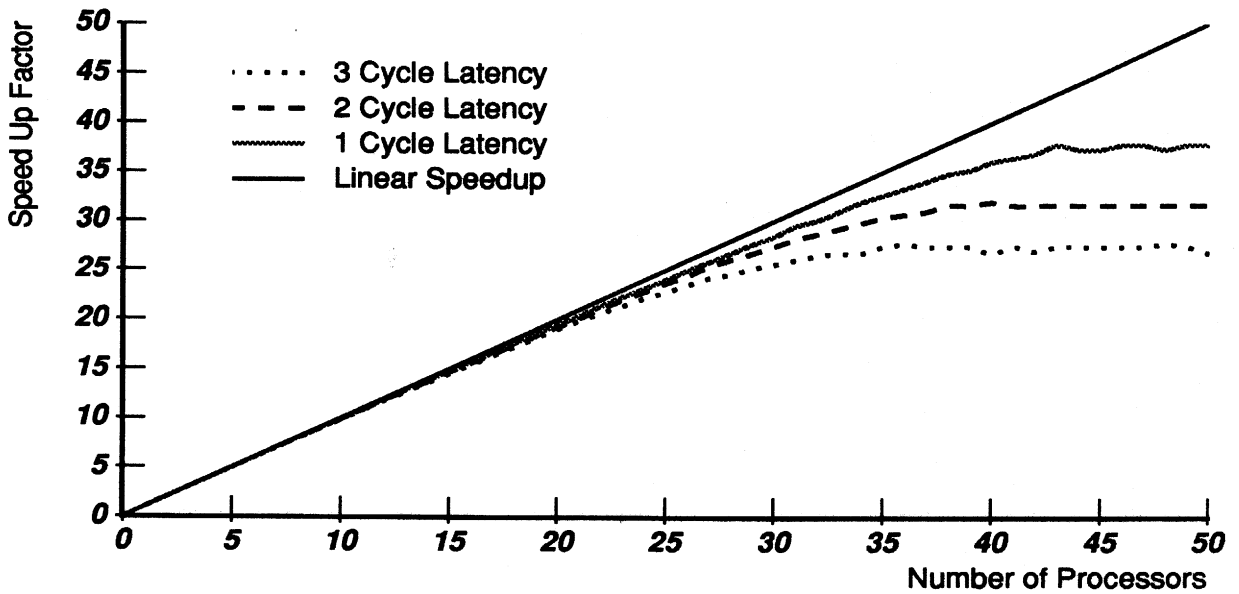


Figure 8: Effects of Communiation Latency on Speedup. The graph shows the speed-up factors over a single pipelined processor. The analysis shown is for a system composed of processors employing a 3-stage pipeline.

the trace-scheduling approach: symbolic evaluation eliminates conditional tests related to data-structures, producing large parallelizable basic blocks, while trace scheduling optimizes across basic block boundaries.

4.6 Suggestions for Future Work

Using symbolic evaluation to create numerical dataflow graphs may be combined with other parallel-programming approaches. For example, this technique can be combined with the *futures* approach of MultiScheme [11] by using symbolic evaluation to parallelize computations within a future, thereby allowing futures to be used to program a collection of superscalar computers. Similarly, our technique may be used in conjunction with hardware that dynamically schedules dataflow graphs: symbolic evaluation can be used to create large statically analyzable nodes within a dynamic dataflow graph. Rather than connecting a collection of relatively simple processors, the parallelism available within each of these statically-analyzable nodes makes it feasible to use dynamic-scheduling hardware to combine a collection of more powerful (parallel) processors.

5 Summary and Future Research

Many tasks lie ahead: expanding our methods to handle different types of scientific computations, extending the partial evaluator, developing code generators for different kinds of parallel architectures, and designing architectures that interact well with our software techniques are a few of the more interesting projects.

We showed that several different types of scientific computations can be helped by our system. Other scientific computations, such as fluid flow, protein folding, or deuterium intake into palladium cathodes, need to be investigated. Also of importance is the development of a library of partially evaluatable scientific routines. These abstract and reusable routines would be automatically specialized for the problem at hand.

The partial evaluator needs to be made more general. Selection of code for partial evaluation can sometimes require intricate massaging of the code. This massaging is an impediment to the general nature of our technique. The partial evaluator can be extended to work on code without any intervention by the coder.

We are interested in using partial evaluation to specialize other types of computations, such as pattern matching [8], parsing, compiling, performing inferences over a knowledge base, or conducting large scale database retrievals. We believe many common special purpose algorithms can be automatically derived via partial evaluation.

We believe our techniques are especially well suited to exploiting the superscalar architectures that are now becoming commercially available. There are also many other different flavors of parallel processors which we have yet to investigate as targets.

Finally, there remains the investigation of architectural features that interact well with our system. 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. A complete system would include lots of memory, ALUs, and high-throughput switches. Supporting hardware such as scoreboards and automatic caches would be extraneous.

We have created a system that simultaneously supports very abstract and general programming, while providing performance competitive with standard compilers operating on conventionally written scientific programs. We have proven that partial evaluation automatically achieves the effects many optimizations a good programmer uses (short of changing the algorithm). We have shown that a scheduler can produce efficient code for either serial or parallel machines. We believe that partial evaluation has an important role to play in scientific computation.

References

[1] Harold Abelson, personal communication, paper in progress.

[2] Harold Abelson, Gerald Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs,* McGraw Hill, Cambridge, MA, 1985

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

[4] Z. Barzilai, J. L. Carter, B. K. rosen, and J. D. Rutledge, "HSS – A High-Speed Simulator," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 6-4, July 1987, pp. 601-617.

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

[6] Dines Bjorner, Andrei P. Ershov, and Niel D. Jones, *Partial Evaluation and Mixed Computation,* Elsevier Science Publishing, 1988.

[7] Randal Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, Thomas Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings of the 24th Design Automation Conference,* June 1987, Miami Beach, Florida, pps 9-16.

[8] C. Consel and O. Danvy, "Partial evaluation of pattern matching in strings," *Information Processing Letters*, 1988.

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

[12] 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.

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

[14] Thomas Linwood Quarrels, *Analysis of Performance and Convergence Issues for Circuit Simulation,* University of California at Berkeley Memorandum number UCB/ERL M89/42, April 1989.

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

[16] P. 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.

[17] Feng Zhao, "An O(N) algorithm for three-dimensional N-body simulations," MIT Artificial Intelligence Laboratory Technical Report TR-995, Cambridge, MA, 1988.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 1 / 12 / 95

Report # AIM-1145

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 21 (27-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form 2(pgs) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1) UNNUMBERED TITLE PAGE	
(2-21) PAGES #'ED 1-20	
(28) SCANCONT	
(23-25) TARGETS	
(26-27) DOD'S	

Scanning Agent Signoff:
Date Received: 1/12/95 Date Scanned: 1/13/95 Date Returned: 1/19/95

Scanning Agent Signature: Michael W. Cook

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 1/13/1995

M.I.T. Libraries
Document Services

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AIM 1145	2. GOVT ACCESSION NO.: AD-A215097
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): Compiling Scientific Code Using Partial Evaluation	5. TYPE OF REPORT & PERIOD COVERED: memorandum
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Andrew Berlin and Daniel Weise	8. CONTRACT OR GRANT NUMBER(s): N00014-86-K-0180, N00014-87-K-0828
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: July 1989 13. NUMBER OF PAGES: 20
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

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

18. SUPPLEMENTARY NOTES

None

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

partial evaluation
scientific computation
parallelizing compilers
parallel architectures

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

Abstract: 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. We have developed a compiler that uses partial evaluation and scheduling techniques to provide a solution to this dilemma.

Where conventional compilers compile a program without any knowledge of the data the program will

(cont. on back)

Block 20 cont.

be run on, our system uses information about the data when transforming the program. This technique, by eliminating nearly all the user's control and data abstractions, produces high performance code. For an important class of numerical programs, partial evaluation dramatically improves performance: we have achieved speedups over conventionally compiled code that range from seven times faster to ninety one times faster.

We also show how partial evaluation can be applied to the programming of parallel computers. By eliminating inherently sequential data structure references and their associated conditional branches, partial evaluation exposes the low-level parallelism inherent in a computation. We present the results of applying a parallel scheduler to a partially evaluated program that simulates the motions of nine bodies under mutual gravitational attraction.