Partial Evaluation for Scientific Computing: The Supercomputer Toolkit Experience

Andrew A. Berlin
Xerox Palo Alto Research Center
berlin@parc.xerox.com

Rajeev J. Surati
M.I.T. Artificial Intelligence Laboratory
raj@zurich.ai.mit.edu

Abstract: *We describe the key role played by partial evaluation in the Supercomputer Toolkit, a parallel computing system for scientific applications that effectively exploits the vast amount of parallelism exposed by partial evaluation. The Supercomputer Toolkit parallel processor and its associated partial evaluation-based compiler have been used extensively by scientists at M.I.T., and have made possible recent results in astrophysics showing that the motion of the planets in our solar system is chaotically unstable.*

1 Introduction

In 1989, researchers at M.I.T. and Hewlett-Packard began a joint effort to create the *Supercomputer Toolkit* , a set of hardware and software building blocks to be used for the construction of special-purpose computational instruments for scientific applications. Earlier work ([6],[7]) had shown that partial evaluation of numerical programs that are mostly data-independent converts a high-level, abstractly specified program into a low-level, special-purpose program, providing order-of-magnitude performance improvement and exposing vast amounts of low-level parallelism. A central focus of the Supercomputer Toolkit project was to find a way to exploit this extremely fine-grained parallelism. By combining the performance improvements available from partial evaluation with novel parallel compilation techniques and a parallel processor architecture specifically designed to execute partially evaluated programs, the *Supercomputer Toolkit* system enabled scientists to run an important class of abstractly-specified programs approximately three orders of magnitude faster than a conventionally compiled program executing on the fastest available workstation.

This paper presents an overview of the role played by partial evaluation in the *Supercomputer Toolkit* system, describes the novel parallelism grain-size adjustment technique that was developed to make effective use of the fine-grained parallelism exposed by partial evaluation, and summarizes the various real-world scientific projects that have made use of the *Supercomputer Toolkit* system.

2 Motivation

Scientists are faced with a dilemma: They need to be able to write programs in a high-level language that allows them to express their understanding of a problem, but at the same time they need their programs to execute very quickly, as their problems often require weeks or even months of computation time. In the astrophysics community, the situation had become critical: programs would be written in a few days in a high level language, only to have weeks or even months invested in reexpressing the problem so that it could make better use of a vectorizing subroutine library; rewriting the entire program in assembly language; or in extreme cases, constructing special-purpose hardware to solve the problem. ([16]) Although partial evaluation promised to provide a solution to this dilemma for an important class of numerically-intensive programs, the parallel hardware and compilation technology required to take full advantage of the potential of partial evaluation did not exist.

Much of the design of the *Supercomputer Toolkit* was based on the observation (See [7]) that numerical applications are special in that they are for the most part *data-independent*, meaning that the sequence of numerical operations that will be performed is independent of the actual numerical values being manipulated. For instance, matrix multiply performs the same sequence of numerical operations regardless of the actual numerical values of the matrix elements. Partial evaluation of a data-independent program has the effect of removing all data abstractions and program structure, producing a purely numerical program that fully exposes the low-level parallelism inherent in the underlying computation.

For the scientific applications we were targeting, such as orbital mechanics calculations, partial evaluation of data-independent calculations produced purely numerical programs containing several thousands of floating-point operations, with the potential for parallel execution of 50 to 100 operations simultaneously. However, the parallelism exposed by partial evaluation is difficult to exploit, because it is extremely fine-grained, at the level of individual numerical operations.

3 The Supercomputer Toolkit System

The *Supercomputer Toolkit* is a parallel processor consisting of eight independent processors connected by two independent communication busses. The Toolkit system makes effective use of the parallelism exploited by partial evaluation in two ways. First, within each processor, fine-grain parallelism is used to keep the pipeline of a floating-point chip set fully utilized. Second, multiple operations can execute in parallel on multiple processors.

The compilation process consists of four major phases. The first phase begins by using partial evaluation to convert each data-independent section of a program into a data-flow graph that consists entirely of numerical operations. This is followed by traditional compiler optimizations, such as constant folding and dead-code elimination. The second phase analyzes locality constraints within the data-flow graph and groups fine-grain operations together to form higher grain-size instructions known as *regions*. In the third phase, critical-path based heuristic scheduling techniques are used to assign each coarse-grain region to a processor. Finally, the region boundaries are broken down, and instruction-level scheduling is performed to assign computational resources to the fine-grain operations that have been assigned to each processor. A very detailed discussion of the compiler and all of its phases can be found in [3] and [5].

Before discussing the details of the *Supercomputer Toolkit* architecture and compilation techniques, we present a set of measurements intended to provide an idea of the relative importance of the various sources of performance improvement achieved by the Toolkit system, using a 9-body orbital mechanics program[1] as an example.

1 The performance improvement provided by using **partial evaluation** to convert a high-level, data-independent program into a low-level, purely numerical data-flow graph was measured by expressing the data-flow graph in an rtl-style program expressed in the C programming language, by using a C vector to store the numerical value produced by each node in the dataflow graph. Comparison of this low-level (partially evaluated) C program to the original Scheme program (compiled by the LIAR Scheme compiler) revealed speedups which typically ranged from 10 to 100 times faster. In the case of the 9-body program, partial evaluation provided a speedup factor of 38x. This speedup factor can be realized through execution in C on traditional sequential machines as well as through execution on the *Supercomputer Toolkit*.

2 The performance improvement provided by the ability of each Toolkit processor to make effective use of fine-grain parallelism to keep the floating-point pipeline full was measured by comparing the sustained rate attained by each Toolkit processor (12.9 Mflops) to the sustained rate attained by the fastest workstation available at the time[2] (which happened to make use of the same floating-point chip set as the *Supercomputer Toolkit* processor) executing hand-optimized code expressed in Fortran (2 Mflops). Thus the Toolkit's processor architecture achieved approximately a 6x performance improvement by enabling multiple fine-grained instructions to execute in parallel within the floating-point chip set.

3 The effectiveness of the static scheduling and grain-size adjustment parallel compilation techniques to make use of multiple toolkit processors simultaneously was measured by comparing the execution time of the 9-body program executing on eight Toolkit processors in parallel to a virtually optimal uniprocessor implementation of the 9-body program. A factor of 6.2x performance improvement was attained by making use of eight processors in parallel.

[1] Specifically, five time-steps of a 12th-order Stormer integration of the gravity-induced motion of a 9-body solar system.

[2] An HP9000/835

The speedups available from partial evaluation, from the use of fine-grain parallelism within each processor, and from multiprocessor execution are orthogonal. Thus from the "black box" point of view of our scientific user community, the 9-body program executed in parallel on the *Supercomputer Toolkit* **1413x** faster than did the traditionally compiled high-level Scheme program executed on a high performance workstation. Of this speedup, a factor of 38 resulted directly from partial evaluation and could have been achieved by executing the partially-evaluated program in C on a workstation, while a factor of 37.2 of the speedup resulted from the ability of the *Supercomputer Toolkit* hardware to make use of the parallelism exposed by partial evaluation.

4 Design Goal: Optimization of Data-Independent Programs

The *Supercomputer Toolkit* system was designed based on the observation that in the scientific applications we were most interested in, such as the integration of ordinary differential equations, the data-dependent portions of a program tend to be very small, typically taking the form of error checks or "Is it good enough yet?" style loops, with the vast majority of the computation occuring in the data-independent portions of the program. This focus on data-independent programs was carried to an extreme, leading to a system that provided extraordinary performance on data-independent code, but which required that code containing data-dependent branches be left residual.

In most partial evaluation systems, the partially-evaluated program is expressed in the same programming language as the source program, allowing code that is left residual to intermingle with code that is partially-evaluated. However, in our system, partially-evaluated code is executed on a specialized numerical processor that does not support the original source language. Each piece of code that is not partially evaluated must be converted (either by hand or by an application-specific program generator) into the low-level assembly language of each Toolkit processor. Thus in order to use the *Supercomputer Toolkit* compiler on a data-dependent program, the program must first be divided up into data-independent subprograms, each of which are then compiled (via partial evaluation and parallel scheduling) to form a high-performance subroutine.

For the numerical applications the toolkit was intended to be used for, such as the integration of ordinary differential equations, the division of programs into data-independent subprograms did not pose a major problem, as the complexity inherent in these problems tends to be isolated in one or two well-defined data-independent subprograms. However, when people from communities outside of the Toolkit's originally intended user base began to use the Toolkit for problems exhibiting greater data dependence, the poor handling of data-dependent branches posed a serious obstacle.

It is important to note that there is no technical obstacle that prevented better handling and limited partial-evaluation of data-dependent branches. Indeed, our original intention was to implement a compilation process that combined aggressive partial evaluation-based optimization of data-independent subprograms with traditional code generation techniques that would handle the data-dependent branches. However, this integration with traditional techniques was never completed: as soon as the portion of the compiler that handles data-independent programs became operational, the allure of the dramatic performance increases available motivated scientists to start using the system immediately, using a few lines of assembly language to implement the residual data-dependencies, and invoking the compiled data-independent subprograms from assembly language as subroutines. Eventually, a number of the users built on top of the Toolkit compiler their own application-specific program generators that automatically created the few lines of assembly-language instructions required to implement the data-dependent branches of their programs.

5 The Partial Evaluator

The *Supercomputer Toolkit* compiler performs partial evaluation of data-independent programs expressed in the Scheme dialect of Lisp by using the symbolic execution technique described in previously published work by Berlin ([6]). Using this technique, the input data structures for a particular problem are provided at compile time, using placeholders to represent those numerical values that will not be available until execution time. Partial evaluation occurs by executing the program symbolically at compile time, creating and accessing data-structures as necessary, and performing numerical operations whenever possible. The partial evaluator only leaves as residual those operations whose numerical input values will not be available until execution time. The partially-evaluated program consists entirely of numerical operations: the execution of all loops, data-structure references and creations, and procedure manipulations occurs at compile time.

Our partial evaluation strategy proved quite effective on the ordinary differential equation style applications we originally envisioned that the Toolkit would be used for. As a wider scope of applications began to develop, the most serious deficiency in our system proved to be the lack of support for leaving selected data-structure operations residual in the partial evaluation process. For instance, although users might want an operation such as matrix multiply to be completely unrolled, they might still want the resulting data to be stored in a particular matrix format. Our system eliminated *all* data-structures, making it difficult to perform certain programming tricks that rely on the location of a piece of data in memory, and requiring a data-rearrangement when interfacing with subroutines that had particular memory-storage expectations.

6 The Toolkit Processor Architecture

Each *Supercomputer Toolkit* processor is a Very Long Instruction Word (VLIW) computer. The processor architecture is designed to make effective use of the fine-grain parallelism exposed by partial evaluation by keeping a pipelined high-performance floating-point chip set fully utilized. In general, the floating-point chip set produces a 64-bit result during every cycle, and requires two 64-bit inputs during each cycle. Constructing a processor that can move around enough data to keep the floating-point chips busy required the inclusion within each processor of two independent memory systems, as illustrated in Figure 1. Each memory system has its own dedicated integer ALU and register file for generating memory addresses, while a third integer ALU handles

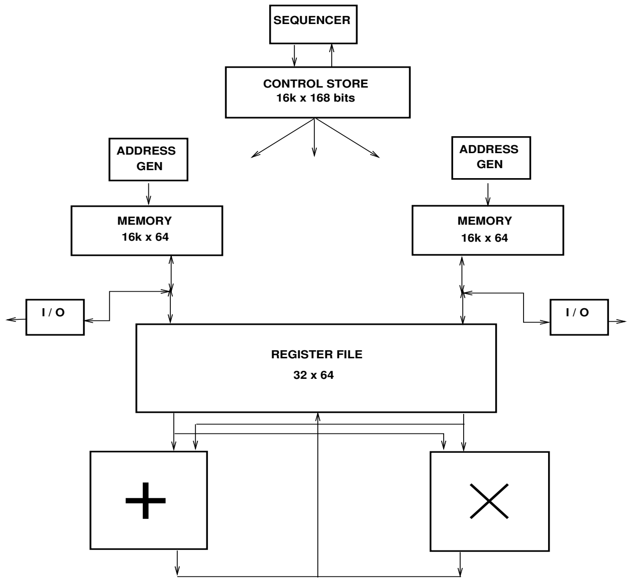

Figure 1: This is the overall architecture of a *Supercomputer Toolkit* processor node, consisting of a fast floating-point chip set, a 5-port register file, two memories, two integer alu address generators, and a sequencer.

program-counter sequencing operations. To support inter-processor communication, each processor has two high-speed Input/Output ports attached directly to its main register files. For a more detailed description of the *Supercomputer Toolkit* processor architecture, see [2].

Since partial evaluation eliminated all data-structures and higher-order procedure calls, the compiler was able to predict the data needs of the floating-point chips at compile time, giving it the freedom to decide which of the two memory systems each result would be stored in, and to begin the data movement necessary to support a particular floating-point operation many cycles in advance of the actual start of the operation. Due to the pipeline structure of the floating-point chip set, it is possible to initiate an operation during each cycle, but the result of that operation is often not available for use by the next operation. By utilizing the parallelism exposed by partial evaluation, the Toolkit compiler was able to schedule operations during these intermediate cycles, thereby keeping the floating-point chip set fully utilized. Indeed, on a wide variety of applications, the *Supercomputer Toolkit* compiler was able to sustain floating-point unit usage rates in excess of 99%.

In theory, up to twelve Toolkit processors may be combined to form a parallel computing system, although the largest system ever constructed is an eight processor system. Each Toolkit processor has its own program-counter and is capable of independent operation. Special synchronization and branch control hardware provide the program-counters of the various processors with the ability to track one another, effectively allowing a single program to make use of multiple processors simultaneously. The experimental results presented in this paper were performed on an eight processor *Supercomputer Toolkit*, configured so that two independent inter-processor communication channels were shared by all eight processors.

7 Parallel Compilation Technology

We have developed parallel compilation software that automatically distributes a data-independent computation for parallel execution on multiple processors. Dividing up the computation at compile time is practical only because partial evaluation eliminates the uncertainty about what numerical operations the compiled program will perform, by evaluating conditional branch instructions related to data-structures and strategy selection at compile time. In other words, all branches of the form "Have we reached the end of the vector yet?" and "Have we been through this loop 5 times yet?", are eliminated at compile time, leaving for run-time execution only those branches that actually depend on the numerical values of the results being computed. Thus the partial evaluation process is similar to loop unrolling, but is much more extensive, as partial evaluation also eliminates inherently sequential procedural abstractions and data structures, such as lists, that would otherwise act as barriers to parallel execution.

In the compiler community, a sequence of computation instructions ending in a conditional branch is known as a basic block. The largest basic blocks produced by traditional compilers are usually around 10-30 instructions in length, and reflect the calculations expressed within the innermost loop of a program. In contrast, the basic blocks of a partially evaluated program are usually several thousand instructions in length. For example, the basic-block associated with the 9-body program mentioned earlier consisted of 2208 floating-point instructions. A limitation of the partial evaluation approach is that for programs that manipulate large amounts of data, the basic blocks may actually get too long to fit in memory, at which point it is necessary for the programmer to declare that certain data-independent branches, such as outermost loops, should be left intact, limiting the scope of partial evaluation.

Each basic block produced by partial evaluation may be represented as a data-independent (static) data-flow graph whose operators are all low-level numerical operations. Previous work ([6]) has shown that this graph contains large amounts of low-level parallelism. For instance, the parallelism profile for the 9-body program, illustrated in Figure 2, indicates that partial evaluation exposed so much low-level parallelism that in theory, parallel execution could speedup the computation by a factor of 69x faster than a uniprocessor execution. However, achieving this theoretical maximum speedup factor would require using 516 non-pipelined processors capable of instantaneous communication with one another.[3]

In practice, much of the available parallelism must be used within each processor to keep the floating-point pipeline full, it does take time (latency) to communicate between processors. As the latency of inter-processor communication increases, the maximum possible speedup decreases, as some of the parallelism must be used to keep each processor busy

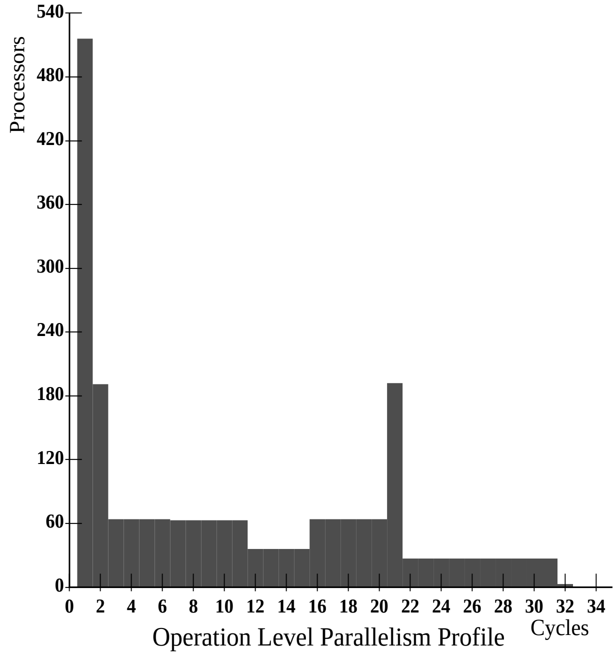

Figure 2: Parallelism profile of the 9-body problem. This graph represents all of the parallelism available in the problem, taking into account the varying latency of numerical operations.

while awaiting the arrival of results from neighboring processors. Bandwidth limitations on the inter-processor communication channels further restrict how parallelism may be used by requiring that most numerical values used by a processor actually be produced by that processor.

8 Parallel Scheduling Techniques

Previously published work by Berlin and Weise ([4]) suggested the use of critical-path based parallel scheduling techniques to take advantage of the low-level parallelism exposed by partial evaluation. Critical-path based techniques, which give priority to the longest computations in a program, are very effective at overcoming latency limitations, but do not consider bandwidth limitations at all. In other words, a critical-path based scheduler will seek to schedule a non-critical path operation on any processor that happens to be available, without regard to the fact that the operands and result of that operation may need to be transmitted between processors. This approach is only effective in situations where a large amount of inter-processor communication bandwidth is available, making it feasible for many results to be transmitted between processors.

Each of the *Supercomputer Toolkit* 's two inter-processor communication channels can accept one result every other cycle. As a result of this communication bandwidth limitation, on an eight processor system, only one out of every eight results produced by a processor can be transmitted to other processors. Thus on the Toolkit system, roughly seven out of every eight numerical results used by a processor must be produced by that processor. We first attempted to generate parallel code for the *Supercomputer Toolkit* using critical-path based scheduling techniques similar to those suggested

[3]We originally chose the 9-body program as an example to ease comparison with previously published work that also studied this program, including [11], [6], and [4]. However, there are numerical discrepancies between the theoretical speedup factors published in this paper and those presented in our previously published work, due to improvements that were made to the constant-folding phase of our compiler. As a result of these improvements, the data-flow graph of the 9-body program being discussed in this paper has fewer operations than the data-flow graph used in [6] and [4]. All graphs and statistics presented in this paper, including the parallelism profile, have been updated to account for this change.

by Berlin and Weise. Due to communication bandwidth limitations, the results were dismal: On the 9-body program, a speedup factor of only 2.5x was achieved using eight processors.

9 Grain-Size Adjustment

To overcome the scheduling difficulties associated with limited communication bandwidth, we developed a technique that adjusts the grain-size of the fine-grain parallelism exposed by partial evaluation to match the inter-processor communication capabilities of the architecture. Prior to initiating critical-path based scheduling, we perform a locality analysis that groups together operations that depend so closely on one other that it would not be practical to place them in different processors. Each group of closely interdependent operations forms a larger grain-size instruction, which we refer to as a *region*.[4] In essence, grouping operations together to form a region is a way of simplifying the scheduling process by deciding in advance that certain opportunities for parallel execution will be ignored due to limited communication capabilities. Critical-path based scheduling is performed and works effectively at the region level, assigning regions to processors, rather than assigning fine-grain instructions to processors.

Since all operations within a region are guaranteed to be scheduled onto the same processor, the maximum region size must be chosen to match the communication capabilities of the target architecture. For instance, if regions are permitted to grow too large, a single region might encompass the entire data-flow graph, forcing the entire computation to be performed on a single processor! Although strict limits are therefore placed on the maximum size of a region, regions need not be of uniform size. Indeed, some regions are large, corresponding to localized computation of intermediate results, while other regions are quite small, corresponding to results that are used globally throughout the computation.

We have experimented with several different heuristics for grouping operations into regions. The optimal strategy for grouping instructions into regions varies with the application and with the communication limitations of the target architecture. However, we have found that even a relatively simple grain-size adjustment strategy dramatically improves the performance of the scheduling process. For instance, as illustrated in Figure 3, when a value is used by only one instruction, the producer and consumer of that value are grouped together to form a region, thereby ensuring that the scheduler will not place the producer and consumer on different processors in an attempt to use spare cycles wherever they happen to be available. Provided that the maximum region size is chosen appropriately,[5] grouping operations together based on locality prevents the scheduler from making gratuitous use of the communication channels, forcing it to focus on scheduling options that make more effective use of the limited communication bandwidth.

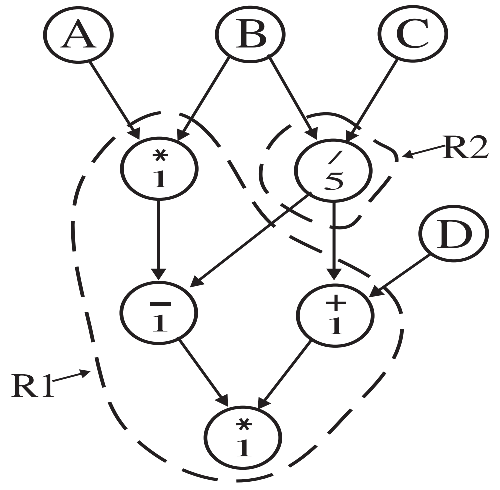

Figure 3: **A Simple Region Forming Heuristic.** A region is formed by grouping together operations that have a simple producer/consumer relationship. This process is invoked repeatedly, with the region growing in size as additional producers are added. The region-growing process terminates when no suitable producers remain, or when the maximum region size is reached. A producer is considered suitable to be included in a region if it produces its result solely for use by that region. (The numbers shown within each node reflect the computational latency of the operation.)

Exploiting locality by grouping operations into regions forces closely-related operations to occur on the same processor. Although this reduces inter-processor communication requirements, it also eliminates many opportunities for parallel execution. Figure 4 shows the parallelism remaining in the 9-body problem after operations have been grouped into regions. Comparison with Figure 2 shows that increasing the grain-size eliminated about half of the opportunities for parallel execution. The challenge facing the parallel scheduler is to make effective use of the limited parallelism that remains, while taking into consideration such factors as communication latency, memory traffic, pipeline delays, and allocation of resources such as processor buses and interprocessor communication channels.

10 Performance Measurements

The final result of compiling the 9-body program using the *Supercomputer Toolkit* compiler is shown in Figure 5.[6] Notice how the compiler was able to take the available parallelism shown in Figure 4 and spread it across the processors. By utilizing eight processors in parallel, the compiler was able to achieve a speedup factor of approximately 6.2x faster than a nearly optimal implementation of this program running on a single Toolkit processor.

[4] The name *region* was chosen because we think of the grain-size adjustment technique as identifying "region" of locality within the data-flow graph. The process of grain-size adjustment is closely related to the problem of graph multisection, although our region-finder is somewhat more particular about the properties (shape, size, and connectivity) of each "region" sub-graph than are typical graph multisection algorithms.

[5] The region size must be chosen such that the computational latency of the operations grouped together is well-matched to the communication bandwidth limitations of the architecture. If the regions are made too large, communication bandwidth will be underutilized since the operations within a region do not transmit their results.

[6] This figure represents a single time step of the integration, on which the compiler achieved a speedup factor of 6.5x using eight processors. The more conservative speedup factor quoted throughout this document for the 9-body problem refers to five integration time steps, thereby including the overhead of moving data around to restart the computation after each time step.

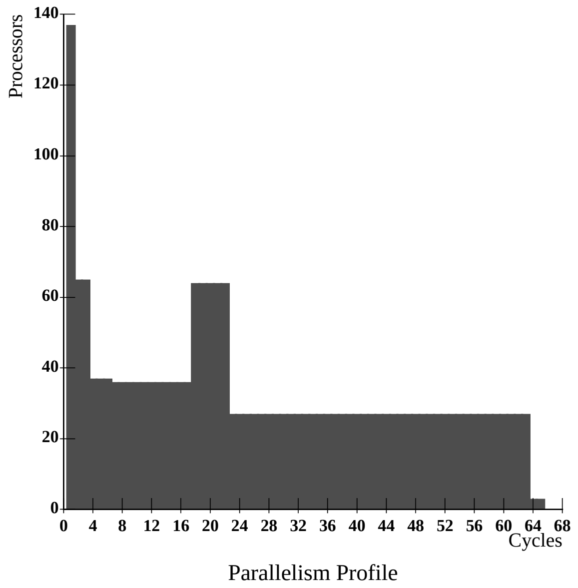

Figure 4: Parallelism profile of the 9-body problem after operations have been grouped together to form regions. Comparison with Figure 2 clearly shows that increasing the grain-size significantly reduced the opportunities for parallel execution. In particular, the maximum speedup factor dropped from 69 times faster to only 34.5 times faster than a single processor.

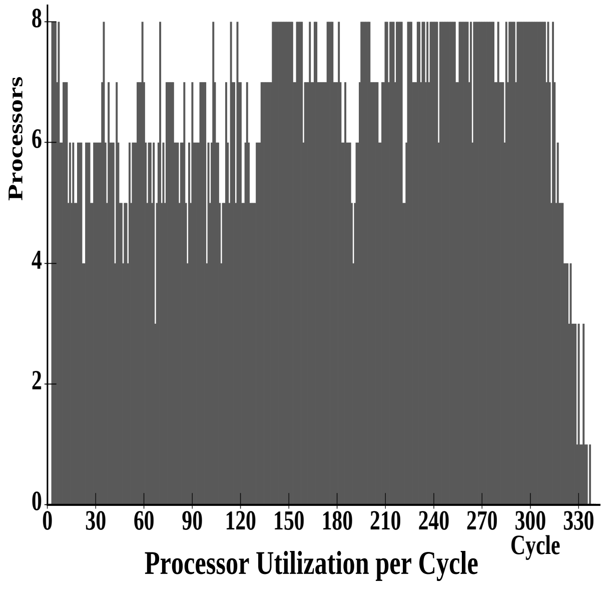

Figure 5: The result of scheduling the 9-body program onto eight *Supercomputer Toolkit* processors. Comparison with with the region-level parallelism profile (figure 4) illustrates how the scheduler spread the coarse-grain parallelism across the processors. A total of 340 cycles are required to complete the computation. On average, 6.5 of the 8 processors are utilized during each cycle.

11 Applications

A variety of scientific applications made use of the *Supercomputer Toolkit* system, ranging from numerical integration of the solar system to clinical genetic counseling. Some applications utilized only a single Toolkit processor, while others ran the same program on multiple processors simultaneously, or used the automatic parallelization features of the compiler to execute a single program on eight processors in parallel. We present an overview of these applications, focusing on the role played by partial evaluation, and on the advantages and difficulties encountered.

Chaos in the Solar System:
The *Supercomputer Toolkit* application having the most scientific importance was a 100-million-year integration of the entire Solar System, incorporating a post-Newtonian approximation to General Relativity and corrections for the quadrupole moment of the Earth-Moon system. The longest previous such integration ([21]) was for about 3 million years. The integration performed on the *Supercomputer Toolkit* confirmed that the evolution of the Solar system as a whole is chaotic with a remarkably short time scale of exponential divergence of about 4 million years. A complete analysis of the integration results appears in [1].

A novel type of symplectic integration strategy was developed by Wisdom and Holman for use in this application, and was expressed in the Scheme language using an abstract programming style. Partial evaluation specialized this integration strategy for use on the solar system problem with a particular force law (gravitation) and a particular solar system configuration. The 100-million-year integration used eight Toolkit processors running in parallel. The computation was arranged so that each processor simulated a single solar system, but with each processor starting with slightly different initial conditions. Chaos was observed by comparing the differences between the states that evolved from the slightly varying initial conditions. The Toolkit compiler was used to generate code for each processor independently. The compiled code for a single processor contains almost 10,000 Toolkit instructions for each integration step, more than 98% percent of which correspond to floating-point operations.

This application posed somewhat of a challenge to our partial evaluation system, as it violated our simple model of programs as consisting of data-independent inner loops surrounded by data-dependent branches. Specifically, the new integration strategy took advantage of the elliptical nature of the planetary orbits, making extensive use of selection operations and scientific subroutines, some of which were heavily data-dependent. Thus this program had data-dependencies at the very core of its innermost loops.

We chose to handle these innermost data dependencies by providing a mechanism for leaving subroutines residual. In our hybrid system, this amounted to allowing a partially-evaluated program to include a call to a data-dependent hand-coded routine, such as sin. By developing a small library of code that could be left "residual", that included the trigonometric functions as well as a few selection operations such as "return the second argument if the first argument is greater than 0", we were able to abstract away these innermost data dependencies, effectively burying them inside of rather simple subroutines.

Note that an alternative approach would have been to use techniques for extending the placeholder-based partial evaluation strategy to allow it to generate code that contains selection-style conditional branches, as described in [7]. We did indeed add these techniques to our front-end partial evaluator, but have not extended the code generation back-end to handle conditional branches, primarily because demand for this functionality from our scientific users dropped off once the subroutine library of selection operations became available.

Orrery Verification Experiment:
Another astrophysics application involved verifying results that had been obtained in 1988 by G. Sussman and J. Wisdom using the Digital Orrery to demonstrate that the long-term motion of the planet Pluto, and by implication the dynamics of the Solar System, is chaotic ([15]). The Digital Orrery was a special-purpose parallel computer designed explicitly to integrate the solar system. Computations run on the Orrery were parallelized and programmed in microcode by hand, with one processor devoted to each planet. In contrast, the program that executed on the *Supercomputer Toolkit* was written in Scheme, and automatically compiled using the Toolkit's partial evaluation-based compiler.

The Orrery integration required integrating the positions of the outer planets for a simulated time of 845 million years (note that this is only 6 planets, rather than the 9 in the whole solar system), which required running the Orrery continuously for more than three months. The same integrations utilizing a 6-body stormer integrator were performed on a single toolkit processor, showing that *each* toolkit processor coupled with the compiled partially evaluated code was about 3 times faster than the entire multiple processor Digital Orrery.

This program mapped nearly perfectly onto the Toolkit system. The only data-dependent branches were located at the outermost "is it done yet?" loop. With the exception of this single instruction end-test, the entire program was partially evaluated. The abstract programming style enabled by partial evaluation permitted quad-precision floating-point operations to be substituted for double-precision operations with the simple replacement of a few procedure definitions.

The Orrery verification experiment ran on a single Toolkit processor, since the automatic parallelization portion of the Toolkit compiler was not yet operational at the time the experiment was performed. Once the automatic parallelizer was completed, we compiled a Stormer integration of a full 9 planet solar system, generating a program that utilized eight processors in parallel to achieve a factor of 6.2x speedup over the single processor Toolkit program. This program, which we refer to as an example earlier in this paper, was the first to take full advantage of the parallelism exposed by partial evaluation, and to the best of our knowledge constituted the fastest integration of the solar system ever achieved.

Circuit Simulation:
Hal Abelson, Jacob Katznelson, and Ognen Nastov wrote several programs that utilized the toolkit to perform simulation of circuits like phase locked loops. Some of the problems they studied utilized a runge-kutta integrator, which was well suited to the Toolkit environment, including a Voltage Controlled Oscillator and a Phase Locked Loop. Both simulations when compiled by the toolkit compiler were shown to run approximately 6 times faster on a toolkit processor than on the best floating point workstation available at the time, an HP835 running a Fortran version of the same program.

Partial evaluation was used to specialize the circuit simulator and integration method for the particular circuit being simulated. When a straightforward integration strategy such as 4th-order runge-kutta was used, the application was almost entirely data-independent, mapping very well onto the Toolkit architecture. However, simulation of many of the circuits studied required the integration of a stiff system of differential equations, using a complex and highly data-dependent Gear integration technique. The Gear integration technique uses a sparse linear equation solver, which involves significant data-dependent control flow.

It was possible to utilize the Toolkit compiler to produce code for the data-independent portions of these simulations, including the code that implements the dynamic equations of the circuit itself, but implementation of the highly data-dependent portions of the GEAR integrator had to be performed by hand in assembly language. This required the assembly language programer to have knowledge of the storage allocation strategy used by the compiler to store results in memory, which led to a fairly complex and not very well organized set of interactions. A much needed enhancement to our system would be to provide a way for the programmer to request that the compiler adhere to a particular data storage strategy, such as maintaining a particular data representation for a matrix, rather than the strategy used by our current implementation which leaves the compiler free to store data values in any place that is convenient, including processor registers.

Interestingly, despite the use of partial evaluation, circuit simulations involving the Gear integrator ran slowly compared to other circuit simulators. Later investigation revealed that this was primarily because this simulator, and the Gear integrator in particular, did not employ some implementation tricks that are used by other circuit simulators such as SPICE. However, another factor limiting the performance of this application is that the interface between the compiled code implementing the circuit dynamics and the hand-written code implementing the Gear integrator involved a lot of copying of data. A better interface that allows the compiler to take the ultimate destination of a value into account would provide noticeable performance improvement.

Computation of Lyapunov Exponents:
The toolkit was used in an experiment by Shyam Parekh to compute the Lyapunov exponents of non-linear systems. Lyapunov exponents characterize the divergence of the distance between two trajectories in a dynamical system and can serve as an indicator of chaotic behavior. The *Supercomputer Toolkit* system was used to do parameter space scans of chaotic circuits such as the double scroll circuit. These theoretical scans were compared against actual scans performed using a real circuit. The results and implementation details of these experiments can be found in [19].

An Integration System for Ordinary Differential Equations:
Sarah Ferguson built a software system on top of the toolkit compiler that takes an equation as input, and automatically generates a Scheme program to integrate it. Sarah's system uses the partial-evaluation features of the Toolkit compiler to specialize the integrator for the particular equation being integrated, and to generate code for the main body of the integration. Her system also generates a few lines of Toolkit assembly language that implement a data-dependent branch that adjusts the integration step size based on how much in-

tegration error is being encountered. This system performed quite well, with the data-dependent branches playing a minor role that did not significantly affect system performance.

Elizabeth Bradley used Sarah Ferguson's integration system to perform dynamical simulations of chaotic systems as part of her research on control of chaotic systems ([20]), including the Lorenz system and the double pendulum system. These systems were a perfect match for both our partial evaluation technology and the Toolkit architecture, and executed extremely quickly. Unfortunately, the Toolkit was designed to support applications that run for a long time before producing a result, whereas Elizabeth Bradley needed to capture the intermediate results that were being produced rapidly. Although the computationally expensive integration routines mapped very well onto the Toolkit architecture, the symbolic routines that analyzed the numerical results could not be executed on the numerically-oriented Toolkit system and had to be run on the workstation host. The program thus became I/O limited, with the Toolkit computer producing data far more quickly than it could be transferred to the workstation host. A faster I/O connection to the Toolkit that would have solved this problem was designed, but was never constructed.

Clinical Genetic Counseling:
Finally, a program to calculate the probabalistic relationships over a Bayesian Network like a pedigree was written by Minghsun Liu. This program was designed to be used to answer the "What if?" types of questions that arise in genetic counseling when determining the probability that a potential child may have a particular defect. The computation time grows exponentially with the number of "unknown" nodes in the probability tree. However, if certain assumptions are made about the relative independence of some of these "unknown" nodes, partial evaluation can play an important role, significantly reducing the size of the computation, as described in more detail in [17] and [18]. For any particular program invocation this program performed well. However, for successive invocations, execution speed was hampered by lack of the ability to perform *incremental* partial evaluation, so that the structure of the network could be locally changed without triggering the need to recompile entire probability network.

12 Conclusions and suggestions for future work

To the best of our knowledge, the *Supercomputer Toolkit* system is the first to make effective use of the vast amount of low-level parallelism exposed by partial evaluation. Partial evaluation proved effective in virtually all of the applications encountered during the *Supercomputer Toolkit* project. In some cases, the Toolkit and its compiler created new opportunities to produce important results in science. In other cases, mostly due to shortcomings in the implementation of the compilation system, the applications did not map well onto the Toolkit.

The range of applications that could be run on the *Supercomputer Toolkit* would have been greatly expanded had the Toolkit's compiler provided a way of leaving selected data-dependent branches and data-structures residual. In this way, heavily data-dependent applications such as the Gear integrator, that require the existence of data-structures in a particular format (sparse matrices) on the Toolkit itself could have been written without the need for hand-coding in Toolkit assembly language.

The symbolic execution technique for performing partial evaluation of data-independent programs was simple to implement and worked well. We have already developed some ways (see [7]) to extend this technique to handle certain types of data-dependent branches, and can envision extending it to permit certain data-structures to be left residual.

With recent developments in partial evaluation technology, the Toolkit's partial evaluator for data-independent programs may appear somewhat primitive. However, a key design goal of our system was to be able to take existing highly complex and abstract Scheme programs from scientists, unaltered, and run them on the *Supercomputer Toolkit.* These programs often included global state, side-effects, manipulation of complex data structures such as streams, and the storing of higher-order procedures within data-structures. Such program features pose serious challenges to partial evaluation technology. It is remarkable that a partial evaluation system such as ours, capable of handling only data-independent programs, could have so large an impact on science.

As hardware technology evolves, the use of partial evaluation to expose parallelism will play an increasingly important role. As processor clock speeds increase, pipeline lengths will grow longer, and will require significant amounts of parallelism to keep them full. But more importantly, as it becomes possible to build multiple processors on a single chip, the vast amount of parallelism exposed by partial evaluation will play a key role in computation, affecting programming language and library design as well as the compilation process itself.

13 Acknowledgements

The *Supercomputer Toolkit* project was the result of a joint effort between M.I.T. and Hewlett-Packard. The following people were all involved in the effort to create and utilize this system: Hal Abelson, Andrew A. Berlin, Joel Birnbaum, Jacob Katzenelson, William H. McAllister, Guillermo J. Rozas, Gerald Jay Sussman, Jack Wisdom, Dan Zuras, Rajeev J. Surati, Karl Hassur, Dick Vlach, Albert Chun, David Fotland, Marlin Jones, John Shelton, Sam Cox, Robert Grimes, Bruce Weyler, Elizabeth Bradley, Shyam Parekh, Sarah Ferguson, Ognen Nastov, Mingsun Liu, Peter Szolovitz, Carl Heinzl, Darlene Harrell, and Rosemary Kingsley.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology and at Hewlett-Packard corporation. Support for the M.I.T. laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 and by the National Science Foundation under grant number MIP-9001651. Andrew Berlin's work was supported in part by an IBM Graduate Fellowship in Computer Science.

References

[1] G. J. Sussman and J. Wisdom, "Chaotic Evolution of the Solar System," *Science*, Volume 257, pp. 256-262, July 1992.

[2] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, G. Sussman, "The Supercomputer Toolkit and its Applications,"

MIT Artificial Intelligence Laboratory Memo 1249, Cambridge, Massachusetts.

[3] R. Surati and A. Berlin, "Exploiting the Parallelism Exposed by Partial Evaluation", In *Proc. of the IFIP WG10.3 International Conference on Parallel Architectures and Compilation Techniques*, Elsevier Science, 1994 also available as MIT Artificial Intelligence Laboratory Technical Report no 1414, April 1993.

[4] A. Berlin and D. Weise, "Compiling Scientific Code using Partial Evaluation," *IEEE Computer* December 1990.

[5] R. Surati, "A Parallelizing Compiler Based on Partial Evaluation", S.B. Thesis, Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology, May 1992. Also available as TR-1377, MIT Artificial Intelligence Laboratory, July, 1993.

[6] A. Berlin, "Partial Evaluation Applied to Numerical Computation," *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice France, June 1990.

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

[8] E. Ruf and D. Weise, "Opportunities for Online Partial Evaluation", Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA. 1992.

[9] E. Ruf and D. Weise, "Avoiding Redundant Specialization During Partial Evaluatio," In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluationand Semantics-Based Program Manipulation,* New Haven, CN. June 1991.

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

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

[12] M. Katz and D. Weise, "Towards a New Perspective on Partial Evaluation," In *Proceedings of the 1992 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation,* San Francisco, June 1992.

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

[14] J.A. Fisher, "Trace scheduling: A Technique for Global Microcode Compaction." *IEEE Transactions on Computers*, Number 7, pp.478-490. 1981.

[15] G.J. Sussman and J. Wisdom, "Numerical evidence that the motion of Pluto is chaotic," *Science*, (to appear). Also available as MIT Artificial Intelligence Laboratory Memo no. 1039, April 1988.

[16] J. Applegate, M. Douglas, Y. Gürsel, P. Hunter, C. Seitz, G.J. Sussman, "A Digital Orrery," *IEEE Trans. on Computers*, Sept. 1985.

[17] P. Szolovits, "Compilation for Fast Calculation Over Pedigrees," *Cytogenet Cell Genet* Vol. 59, pgs 136 - 138, 1992

[18] P. Szolovits and S. Pauker, "Pedigree Analysis for Genetic Counseling," *Medinfo 92*

[19] S. Parekh, "Parameter Space Scans for Chaotic Circuits" S.B. Thesis, MIT 1992.

[20] E. Bradley, "Taming Chaotic Circuits" Ph.D. Thesis, MIT 1992.

[21] T. Quinn, S. Tremaine, and M. Duncan "A Three Million Year Integration of the Earth's Orbit," *Astron. J.,* vol. 101, no. 6, June 1991, pp. 2287–2305.

[22] C. Heinzl, "Functional Diagnostics for the Supercomputer Toolkit MPCU Module", S.B. Thesis, MIT, 1990.