MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1414A May, 1994

Exploiting the Parallelism Exposed by Partial Evaluation

Rajeev Surati
raj@martigny.ai.mit.edu

Andrew Berlin
berlin@parc.xerox.com

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.
The pathname for this publication is: ai-publications/1000-1499/AIM-1414A.ps.Z

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.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 andy by the National Science Foundation under grant number MIP-9001651. Andrew Berlin was also supported in part by an IBM Graduate Fellowship in Computer Science.

1 Introduction

Previous work has shown that partial evaluation is good at breaking down data abstraction and exposing underlying fine-grain parallelism in a program [4]. We have written a novel compiler which couples partial evaluation with static scheduling techniques to exploit this fine-grain parallelism by automatically mapping it onto a coarse-grain parallel architecture.

Partial evaluation eliminates the barriers to parallel execution imposed by the data representation and the control structure of a program by taking advantage of information about the particular problem a program will be used to solve. For example, partial evaluation is able to perform at compile-time most data structure references, procedure calls, and conditional branches related to data structure size, leaving mostly numerical computations to be performed at run time. Partial evaluation is particularly effective on numerically-oriented scientific programs, since they tend to be mostly data-independent, meaning that they contain large regions in which the operations to be performed do not depend on the numerical values of the data being manipulated. For instance, matrix multiplication performs the same set of operations, regardless of the particular numerical values of the matrix elements. We use partial evaluation to produce huge basic blocks from these data-independent numerical regions. These basic blocks often contain thousands of instructions, two orders of magnitude larger than the basic blocks that typically arise in high-level language programs. To benefit from the fine-grain parallelism contained in these huge basic blocks, we schedule the partially-evaluated program for parallel execution primarily by performing the operations within an *individual* basic block in parallel.

In order to automatically map the freshly derived fine-grain parallelism onto a multiprocessor, we developed a technique which coarsens the dataflow graph by selectively aggregating operations together. This technique uses heuristics which take the communication bandwidth, inter-processor communication latency, and processor architecture all into consideration. High inter-processor communication latency requires that there be enough parallelism available to allow each processor to continue to initiate operations, even while waiting for results produced elsewhere to arrive. Limited communication bandwidth severely restricts the parallelism grain size that may be utilized by requiring that most values used by a processor be produced on that processor, rather than being received from another processor. Our approach addresses these problems by tailoring the grain size adjustment and scheduling heuristics to match the communication capabilities of the target architecture.

Our compiler operates in four major phases. The first phase performs partial evaluation, followed by traditional compiler optimizations, such as constant folding and dead-code elimination. The second phase analyzes locality constraints within each basic block, locating operations that depend so closely on one another that it is clearly desirable that they be computed on the same processor. These closely related operations are grouped together to form a higher grain size instruction, known as a *region*. The third compilation phase uses heuristic scheduling techniques to assign each region to a processor. The final phase schedules the individual operations for execution within each processor, accounting for pipelining, memory access restrictions, register allocation, and final allocation of the inter-processor communication pathways.

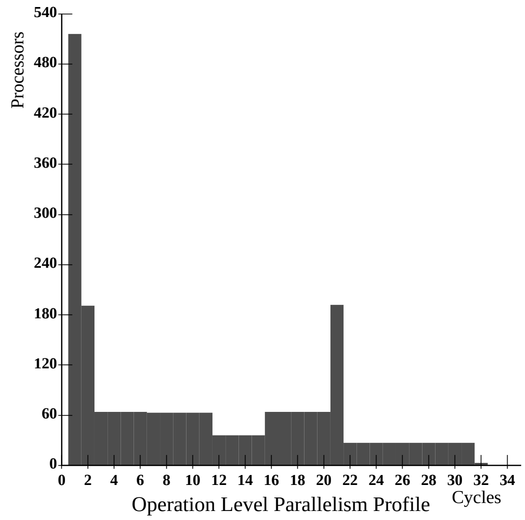

Figure 1: Parallelism profile of the 9-body problem. This graph represents all of the parallelism available in the problem, taking into account the varying latency of numerical operations.

The target architecture of our compiler is the *Supercomputer Toolkit* , a parallel processor consisting of eight independent VLIW processors connected to each other by two shared communication busses [6]. Performance measurements of actual compiled programs running on the *Supercomputer Toolkit* show that the code produced by our compiler for an important astrophysics application[19] runs 6.2 times faster on an eight-processor system than does near-optimal code executing on a single processor. The compilation process of this real world application is used as an example throughout this paper.

2 The Partial Evaluator

Partial evaluation converts a high-level, abstractly written, general purpose program into a low-level program that is specialized for the particular application at hand. For instance, a program that computes force interactions among a system of N particles might be specialized to compute the gravitational interactions among 5 planets of our particular solar system. This specialization is achieved by performing in advance, at compile time, all operations that do not depend explicitly on the actual numerical values of the data.

Many data structure references, procedure calls, conditional branches, table lookups, loop iterations, and even some numerical operations may be performed in advance, at compile time, leaving only the underlying

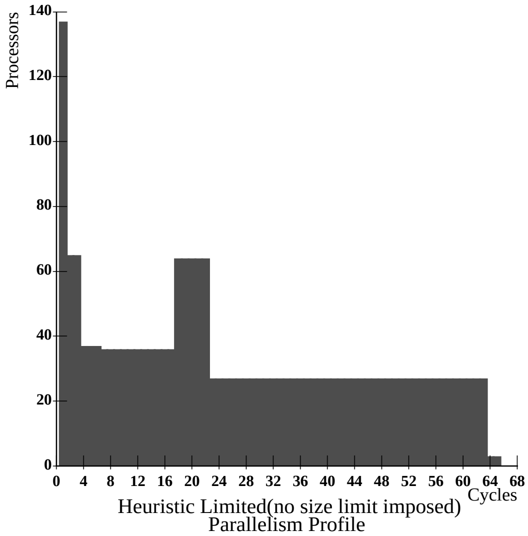

Figure 2: Parallelism profile of the 9-body problem after operations have been grouped together to form regions. Comparison with Figure 1 clearly shows that increasing the grain-size significantly reduced the opportunities for parallel execution. The maximum speedup factor dropped from 69 to 49 times faster than a single processor execution.

numerical operations to be performed at run time

Our compiler exposes fine-grain parallelism using a simple partial evaluation strategy based on a symbolic execution technique described in [5, 4].[1] Despite this technique's simplicity, it works well at exposing fine-grain parallelism. Figure 1 illustrates a parallelism profile analysis of the nine-body gravitational attraction problem of the type discussed in [19].[2] Partial evaluation exposed so much low-level parallelism that in theory, parallel execution could speed up the computation by a factor of 69 over a uniprocessor.

3 Adjusting the Grain Size

Searching for an optimal schedule for a program which exploits fine-grain parallelism is both computationally expensive and difficult to achieve. Rather than do an exhaustive search for the optimal schedule, we developed a heuristic technique to coarsen the exposed fine-grain parallelism to a grain size suitable for critical-path based static scheduling. Prior to initiating critical-path based scheduling, we perform 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 macro-instruction, which we refer to as a *region.*[3] Some regions are large, while others may be as small as one fine-grain instruction. 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.

Since operations within a region will occur on 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 will be large, corresponding to localized computation of intermediate results, while others will be 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. As illustrated in Figure 3, when a value is used by only one instruction, the producer and consumer of that value may be 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 happened to be available. Provided that the maximum region size is chosen appropriately,[4] 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.

An important aspect of grain size adjustment is that the grain size is not increased uniformly. As shown in Table 1, some regions are much larger than others. Indeed, it is important not to forcibly group non-localized operations into regions simply to increase the grain size. For example, it is likely that the result produced by an instruction that has many consumers will be transmitted amongst the processors, since it is not practical to place all of the consumers on the result-producing processor. In this case, creating a large region by grouping together the producer with only some of the consumers increases

[1]More complex partial evaluation strategies that address data-dependent computations may be found in [10, 12, 11].

[2]Specifically, one time-step of a 12th-order Stormer integration of the gravity-induced motion of a 9-body solar system.

[3]The name *region* was chosen because we think of the grain size adjustment technique as identifying "regions" 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.

[4]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 under utilized since the operations within a region do not transmit their results.

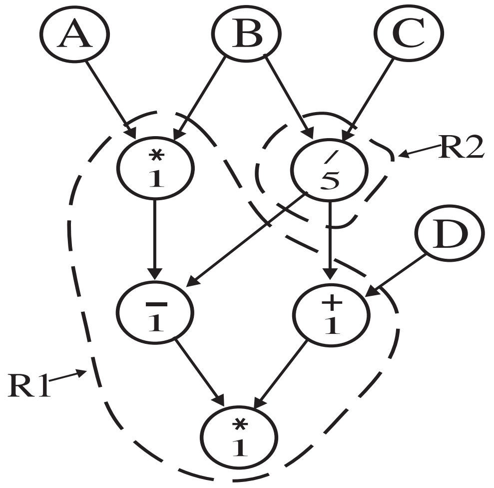

Figure 3: **A Simple Region Forming Heuristic.** A region is formed by grouping together operations that have a simple producer/consumer relationship. This process is invoked repeatedly, with the region growing in size as additional producers are added. The region-growing process terminates when no suitable 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.)

Region Size	Number of Regions
1	108
2	28
3	28
5	56
6	1
7	8
14	36
41	24
43	3

Table 1: The numerical operations in the 9-body program were divided into regions based on locality. This table shows how region size can vary depending on the locality structure of the computation. Region size is measured by computational latency (cycles). The program was divided into 292 regions, with an average region size of 7.56 cycles. The maximal region size used was 43 cycles

the grain size, but does not reduce inter-processor communication, since the result would need to be transmitted anyway. In other words, it only makes sense to limit the scheduler's options by grouping operations together when doing so will clearly reduce inter-processor communication.

4 Parallel Scheduling

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 2 shows the parallelism remaining in the 9-body problem after operations have been grouped into regions. Comparison with Figure 1 shows that increasing the grain size eliminates 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 inter-processor communication channels.

Our compiler schedules operations for parallel execution in two phases. The first phase, known as the region-level scheduler, is primarily concerned with coarse-grain assignment of regions to processors, generating a rough outline of what the final program will look like. The region-level scheduler assigns each region to a processor; determines the source, destinations, and approximate time of transmission of each inter-processor message; and determines the preferred order of execution of the regions assigned to each processor. The region-level scheduler takes into account the latency of numerical operations, the inter-processor communication capabilities of the target architecture, the structure (critical path) of the computation, and which data values each processor will store in its memory. The region-level scheduler does *not* concern itself with finer-grain details such as the pipeline structure of the processors, the detailed allocation of each communication channel, or the ordering of individual operations within a processor. At the coarse grain size associated with the scheduling of regions, a straightforward set of critical-path based scheduling heuristics[5] have proven quite effective. For the 9-body problem example, the computational load was spread so evenly that the variation in utilization efficiency among the 8 processors was only one percent.

The final phase of the compilation process is instruction-level scheduling. The region-level scheduler provides the instruction-level scheduler with an ordered list of regions to execute on each processor along with a list of results that need to be transmitted when they are computed. The instruction-level scheduler chooses the final ordering of low-level operations within each processor, taking into account processor pipelining, register allocation, memory access restrictions, and availability of inter-processor-communication channels. Whenever possible, the order of operations is chosen so as to match the preferences of the region-level scheduler, represented by the ordered list of regions. However, the instruction-level scheduler is free to reorder operations as needed, intertwining operations among the regions assigned to a particular processor, without regard to which coarse-grain region they were originally a member of. This strategy allows the instruction scheduler to maintain a

[5]The heuristics used by the region-level scheduler are closely related to list-scheduling [14]. A detailed discussion of the heuristics used by the region-level scheduler is presented in [1].

schedule similar to the one suggested by the region scheduler, thereby ensuring that the results will be produced at approximately the time that other processors are expecting them, while still taking advantage of fine grain parallelism available in other regions to fill pipeline slots as needed.

The instruction-level scheduler derives low-level pipelined instructions for each processor, choosing the exact time and communication channel for each interprocessor transmission, and determining where values will be stored within each processor. The instruction-level scheduling process begins with a data-use analysis that determines which instructions share data values and should therefore be placed near each other for register allocation purposes. This data-use information is combined with the higher-level ordering preferences expressed by the region-level scheduler, producing a scheduling priority for each instruction. The instruction scheduling process then proceeds one cycle at a time, performing scheduling of that cycle on *all* processors before moving on to the next cycle. Instructions compete for resources based on their scheduling priority; in each cycle, the highest-priority operation whose data and processor resources are available will be scheduled. This competition for data and resources helps to keep each processor busy, by scheduling low-priority operations whose resources are available whenever the resources for higher priority computations are not available. Indeed, when the performance of the instruction-scheduler is measured independently of the region-level scheduler, by generating code for a single *Supercomputer Toolkit* VLIW processor, utilization efficiencies in excess of 99.7% are routinely achieved, representing nearly optimal code.

An aspect of the scheduler that has proven to be particularly important is the retroactive scheduling of memory references. Although computation instructions (such as + or ∗) are scheduled on a cycle-by-cycle basis, memory LOAD instructions are scheduled retroactively, wherever they happen to fit in. For instance, when a computation instruction requires that a value be loaded into a register from memory, the actual memory access operation[6] is scheduled in the past for the earliest moment at which both a register and a memory-bus cycle are available; the memory operation may occur fifty or even one-hundred instructions earlier than the computation instruction. *Supercomputer Toolkit* memory operations must compete for bus access with inter-processor messages, so retroactive scheduling of memory references helps to avoid interference between memory and communication traffic. Figure 4 illustrates the effectiveness of the instruction level scheduler on the nine-body problem example.

[6]On the toolkit architecture, two memory operations may occur in parallel with computation and address-generation operations. This ensures that retroactively scheduled memory accesses will not interfere with computations from previous cycles that have already been scheduled.

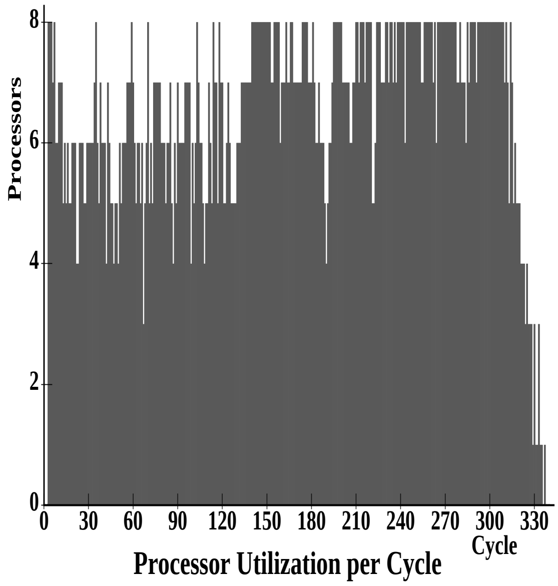

Figure 4: The result of scheduling the 9-body problem onto 8 *Supercomputer Toolkit* processors. Comparison with with the region-level parallelism profile (figure 3) illustrates how the scheduler spread the course-grain parallelism across the processors. A total of 340 cycles are required to complete the computation. On average, 6.5 of the 8 processors are utilized during each cycle.

5 Performance Measurements

The *Supercomputer Toolkit* and our associated compiler have been used for a wide variety of applications, ranging from computation of human genetic pedigrees to the simulation of electrical circuits. The applications that have generated the most interest from the scientific community involve various integrations of the N-body gravitational attraction problem.[7] Parallelization of these integrations has been previously studied by Miller[18], who parallelized the program by using *futures* to manually specify how parallel execution should be attained. Miller shows how one can re-write the N-body program so as to eliminate sequential data structure accesses to provide more effective parallel execution, manually performing some of the optimizations that partial evaluation provides automatically. Others have developed special-purpose hardware that parallelizes the 9-body problem by dedicating one processor per planet.[17] Previous work in partial evaluation [3, 5, 4] has shown that the 9-body problem contains large amounts of fine-grain parallelism, suggesting that more subtle parallelizations are possible without the need to dedicate one processor to each planet.

We have measured the effectiveness of coupling partial evaluation with grain size adjustment to generate code for the *Supercomputer Toolkit* parallel computer, an architecture that suffers from serious inter-processor com-

[7]For instance, [19] describes results obtained using the *Supercomputer Toolkit* that prove that the solar system's dynamics are chaotic.

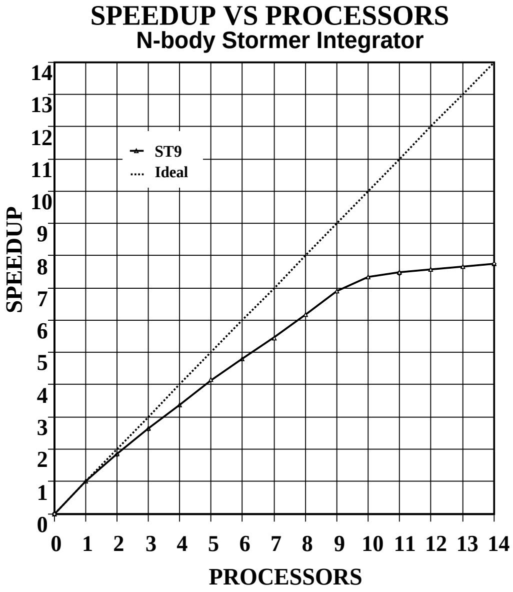

Figure 5: Speedup graph of Stormer integrations. Ample speedups are available to keep the 8-processor *Supercomputer Toolkit* busy, However, the incremental improvement of using more than 10 processors is relatively small.

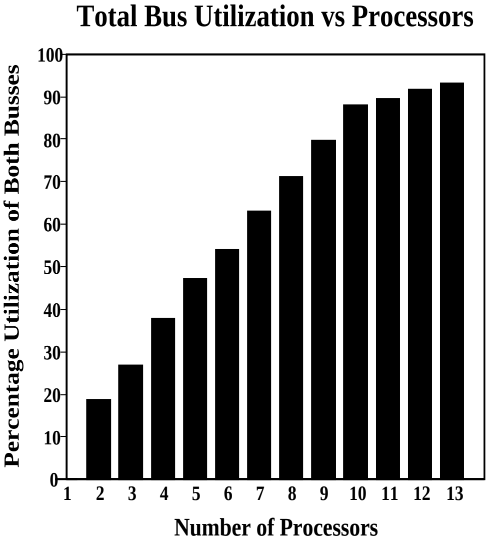

Figure 6: Utilization of the inter-processor communication pathways. The communication system becomes saturated at around 10 processors. This accounts for the lack of incremental improvement available from using more than 10 processors that was seen in Figure 5.

munication latency and bandwidth limitations. Table 2 shows the parallel speedups achieved by our compiler for several different N-body interaction applications. Figure 5 focuses on the 9-body program (ST9) discussed earlier in this paper, illustrating how the parallel speedup varies with the number of processors used. Note that as the number of processors increases beyond 10, the speedup curves level off. A more detailed analysis has revealed that this is due to the saturation of the inter-processor communication pathways, as illustrated in Figure 6. The accuracy of these results was verified by executing the 9-body program on the actual *Supercomputer Toolkit* hardware in an eight processor configuration.

An important drawback to the partial evaluation approach is that it results in the unrolling of loops, which can potentially lead to an explosion in the size of the compiled program. We have found that depending on the size of the data set being manipulated, partial evaluation may reduce the overall size of the program, by eliminating data accesses, branches, and abstraction-manipulation code; or partial evaluation may increase the size of the program by iterating over a large data set. The key to making successful use of the partial evaluation technique is to not carry it too far. For relatively small applications, such as the 9-body integration program, it was practical to partially-evaluate the entire computation; on the other hand, if one was simulating a galaxy containing millions of stars, it would probably be best not to partially-evaluate some of the outermost loops! Our work focuses on achieving efficient parallel execution of the partially-evaluated segments of a program, leaving the decision of which portions of a program should be subjected to this compilation technique up to the programmer.

Program	Single Processor Cycles	Eight Processors Cycles	Speedup
ST6	5811	954	6.1
ST9	11042	1785	6.2
ST12	18588	3095	6.0
RK9	6329	1228	5.2

Table 2: Speedups of various applications running on 8 processors. Four different computations have been compiled in order to measure the performance of the compiler: a 6 particle stormer integration(ST6), a 9 particle stormer integration(ST9), a 12 particle stormer integration(ST12), and a 9 particle fourth-order Runge Kutta integration(RK9). Speedup is the single processor execution time of the computation divided by the total execution time on the multiprocessor.

6 Related Work

The use of partial evaluation to expose parallelism makes our approach to parallel compilation fundamentally different from the approaches taken by other compilers. Traditionally, compilers have maintained the data structures and control structure of the original program. For example, if the original program represents an object as a doubly-linked list of numbers, the compiled program would as well. Only through partial evaluation can the data structures used by the programmer to think about the problem be removed, leaving the compiler free to optimize the underlying numerical computation, unhindered by sequentially-accessed data structures and procedure calls. However, the drawback to the partial-evaluation approach is that it is only highly efffective for applications that are mostly data-independent.

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. Other parallelization techniques include trace-scheduling, software pipelining, vectorizing, as well as static and dynamic scheduling of data-flow graphs.

6.1 Trace Scheduling

Compilers that exploit fine-grain parallelism often employ trace-scheduling techniques [15] to guess which way a branch will go, allowing computations beyond the branch to occur in parallel with those that precede the branch. Our approach differs in that we use partial evaluation to take advantage of information about the specific application at hand, allowing us to totally eliminate many data-independent branches, producing basic blocks on the order of several thousands of instructions, rather than the ten to thirty instructions typically encountered by trace-scheduling based compilers. An interesting direction for future work would be to add trace-scheduling to our approach, to optimize across the data-dependent branches that occur at basic block boundaries.

Most trace-scheduling based compilers use a variant of list-scheduling[14] to parallelize operations within an individual basic block. Although list-scheduling using critical-path based heuristics is very effective when the grain size of the instructions is well-matched to inter-processor communication bandwidth, we have found that in the case of limited bandwidth, a grain size adjustment phase is required to make the list-scheduling approach effective.[8]

6.2 Software Pipelining

Software Pipelining [13] optimizes a particular fixed size loop structure such that several iterations of the loop are started on different processors at constant intervals of time. This increases the throughput of the computation. The effectiveness of software pipelining will be determined by whether the grain size of the parallelism expressed in the looping structure employed by the programmer matches the architecture: software pipelining can not parallelize a computation that has its parallelism hidden behind inherently sequential data references and spread across multiple loops. The partial-evaluation approach on such a loop structure would result in the loop being completely unrolled with all of the sequential data structure references removed and all of the fine grain parallelism in the loop's computation exposed and available for parallelization. In some applications, especially those involving partial differential equations, fully unrolling loops may generate prohibitively large programs. In these situations, partial evaluation could be used to optimize the innermost loops of a computation, with techniques such as software pipelining used to handle the outer loops.

6.3 Vectorizing

Vectorizing is a commonly used optimization for vector supercomputers, executing operations on each vector element in parallel. This technique is highly effective provided that the computation is composed primarily of readily identifiable vector operations (such as dot-product). Most vectorizing compilers generate vector code from a scalar specification by recognizing certain standard looping constructs. However, if the source program lacks the necessary vector-accessing loop structure, vectorizing performs very poorly. For computations that are mostly data-independent, the combination of partial evaluation with static scheduling techniques has the potential to be vastly more effective than vectorization. Whereas a vectorizing compiler will often fail simply because the computation's structure does not lend itself to a vector-oriented representation, the partial-evaluation/static scheduling approach can often succeed by making use of very fine-grained parallelism. On the other hand, for computations that are highly data-dependent, or which have a highly irregular structure that makes unrolling loops infeasible, vectorizing remains an important option.

6.4 Iterative Restructuring

Iterative restructuring represents the manual approach to parallelization. Programmer's write and rewrite their code until the parallelizer is able to automatically recognize and utilize the available parallelism. There are many utilities for doing this, some of which are discussed in [16]. This approach is not flexible in that whenever one aspect of the computation is changed, one must ensure that parallelism in the changed computation is fully expressed by the loop and data-reference structure of the program.

6.5 Static Scheduling

Static scheduling of the fine-grained parallelism embedded in large basic blocks has also also been investigated for use on the *Oscar* architecture at Waseda University in Japan.[7]. The Oscar compiler uses a technique called *task fusion* that is similar in spirit to the grain size adjustment technique used on the *Supercomputer Toolkit*. However, the Oscar compiler lacks a partial-evaluation phase, leaving it to the programmer to manually generate large basic blocks. Although the manual creation of huge basic blocks (or of automated program generators) may be practical for computations such as an FFT that have a very regular structure, it is not a reasonable alternative for more complex programs that require abstraction and complex data structure representations. For example, imagine writing out the 11,000 floating-point operations for the Stormer integration of the Solar system and then suddenly realizing that you need to change to a different integration method. The manual coder would grimace, whereas a programmer writing code for a compiler that uses partial evaluation would simply alter a high-level procedure call.

7 Conclusions

Partial evaluation has an important role to play in the parallel compilation process, especially for largely data-independent programs such as those associated with numerically-oriented scientific computations. Our approach of adjusting the grain size of the computation to match the architecture was possible only because of partial evaluation: If we had taken the more conventional approach of using the structure of the program to detect parallelism, we would then be stuck with the grain size provided us by the programmer. By breaking down the program structure to its finest level, and then imposing our own program structure (regions) based on locality of reference, we have the freedom to choose the grain size to match the architecture. The coupling of

[8] The partial-evaluation phase of our compiler is currently not very well automated, requiring that the programmer provide the compiler with a set of input data structures for each data-independent code sequence, as if the data-independent sequences are separate programs being glued together by the data-dependent conditional branches. This manual interface to the partial evaluator is somewhat of an implementation quirk; there is no reason that it could not be more automated. Indeed, several *Supercomputer Toolkit* users have built code generation systems on top of our compiler that automatically generate complete programs, including data-dependent conditionals, invoking the partial evaluator to optimize the data-independent portions of the program. Recent work by Weise, Ruf, and Katz[10, 11] describes additional techniques for automating the partial-evaluation process across data-dependent branches.

partial evaluation with static scheduling techniques in the *Supercomputer Toolkit* compiler also eliminates the need to write programs in an obscure style that makes parallelism more apparent.

Acknowledgements

Guillermo Rozas was a major contributor to the design of the instruction-scheduling techniques we describe in this paper. We would also like to thank Gerald Sussman and Jack Wisdom for the celestial integrators.

This work is a part of the *Supercomputer Toolkit* project, a joint effort between M.I.T. and Hewlett-Packard corporation.

This report 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] R. Surati, "A Parallelizing Compiler Based on Partial Evaluation," MIT Artificial Intelligence Laboratory Technical Report TR-1377, July 1993

[2] A. Berlin and R. Surati, "Partial Evaluation for Scientific Computing: The Supercomputer Toolkit Experience," *Proc of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation 1994*, Orlando, FL 1994. Also available as MIT Artificial Intelligence Laboratory Memo No. 1487, May 1994

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

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

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

[6] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, G.J. Sussman, and J. Wisdom "The Supercomputer Toolkit: A general framework for special-purpose computing", *International Journal of High-Speed Electronics*, vol. 3, no. 3, 1992, pp. 337–361.

[7] H. Kasahara, H. Honda, and S. Narita "Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR", *Supercomputing 90*, pp 856-864, 1990

[8] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing", *IEEE Software*, Volume 5, No 1, January 1988

[9] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling.", *Journal of Parallel and Distributed Computing*, Volume 10, Number 3, Nov 1990.

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

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

[12] N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generations* Prentice Hall, 1993

[13] M. Lam, "A Systolic Array Optimizing Compiler." Carnegie Mellon Computer Science Department Technical Report CMU-CS-87-187., May, 1987.

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

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

[16] G. Cybenko, J. Bruner, S. Ho, "Parallel Computing and the Perfect Benchmarks." Center for Supercomputing Research and Development Report 1191., November 1991

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

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

[19] G. Sussman and J. Wisdom, "Chaotic Evolution of the Solar System",*Science*, Volume 257, July 1992.