MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1414 April 1993

Exploiting the Parallelism Exposed by Partial Evaluation

Andrew A. Berlin and Rajeev J. Surati

Abstract

We describe an approach to parallel compilation that seeks to harness the vast amount of fine-grain parallelism that is exposed through partial evaluation of numerically-intensive scientific programs. We have constructed a compiler for the *Supercomputer Toolkit* parallel processor that uses partial evaluation to break down data abstractions and program structure, producing huge basic blocks that contain large amounts of fine-grain parallelism. We show that this fine-grain parallelism can be effectively utilized even on coarse-grain parallel architectures by selectively grouping operations together so as to adjust the parallelism grain-size to match the inter-processor communication capabilities of the target architecture.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology.Support for this 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.

1 Introduction

One of the major obstacles to compiling parallel programs is the question of how to automatically identify and exploit the underlying parallelism inherent in a program. We have implemented a compiler for parallel programs that uses novel techniques to detect and effectively utilize the fine-grained parallelism that is inherent in many numerically-intensive scientific computations. Our approach differs from the current fashion in parallel compilation, in that rather than relying on the structure of the program to detect locality and parallelism, we use partial evaluation[5] to remove most loops and high-level data structure manipulations, producing a low-level program that exposes all of the parallelism inherent in the underlying numerical computation. We then use an operation-aggregating-technique to increase the grain-size of this parallelism to match the communication characteristics of the target parallel architecture. This approach, which was used to implement the compiler for the *Supercomputer Toolkit* parallel computer[1], has proven highly effective for an important class of numerically-oriented scientific problems.

Our approach to compilation is specifically tailored to produce efficient statically scheduled code for parallel architectures which suffer from serious interprocessor communication latency and bandwidth limitations. For instance, on the eight processor *Supercomputer Toolkit* system in operation at M.I.T., six cycles[1] are required before a value computed by one processor is available for use by another, while bandwidth limitations allow only one value out of every eight values produced to be transmitted among the processors. Despite these limitations, code produced by our compiler for an important astrophysics application[2] runs 6.2 times faster on our eight-processor system than does near-optimal code produced for a uniprocessor system.[3]

Interprocessor communication latency and bandwidth limitations pose severe obstacles to the effective use of multiple processors. High 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.[4] 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. We overcome these obstacles by combining partial evaluation, which exposes large amounts of extremely fine-grained parallelism, with an operation-aggregating-technique that increases the grain-size of the operations being scheduled for parallel execution to match the communication capabilities of the target architecture.

[1] A "cycle" corresponds to the time required to perform a floating-point multiplication or addition operation.

[2] Stormer integration of the 9-body gravitational attraction problem

[3] The code produced for the uniprocessor was also partially evaluated, to ensure that the factor of 6.2 speedup is entirely due to parallel execution.

[4] [5] (page 35) describes how the effect that interprocessor communication latency has on available parallelism is similar to that of increasing the length of an individual processor's pipeline. In order to continue to initiate instructions on a heavily pipelined processor, there must be operations available that do not depend on results that have not yet emerged from the processor pipeline. Similarly, in order to continue to initiate instructions on a parallel machine that suffers from high communication latency, there must be operations available that do not depend on results that have not yet been received.

2 Our Approach

We use partial evaluation to eliminate the barriers to parallel execution imposed by the data representations and control structure of a high-level program. Partial evaluation is particularly effective on numerically-oriented scientific programs since these programs 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.[5] As a result of this data-independence, partial evaluation is able to perform in advance, at compile time, most data structure references, procedure calls, and conditional branches related to data structure size, leaving only the underlying numerical computations to be performed at run time. The underlying numerical computations form huge sequences of purely numerical code, known as basic blocks. Often, these basic blocks contain several thousand instructions. The order in which basic blocks are invoked is determined by data-dependent conditional branches and looping constructs.

We schedule the partially-evaluated program for parallel execution primarily by performing the operations within an *individual* basic block in parallel. This is practical only because the basic blocks produced by partial evaluation are so large. Were it not for partial evaluation, the basic blocks would be two orders of magnitude smaller, requiring the use of techniques such as software pipelining and trace scheduling, that seek to overlap the execution of multiple basic blocks. Executing a huge basic block in parallel is very attractive since it is clear at compile time which operations need to be performed, which results they depend on, and how much computation each instruction will require, ensuring the effectiveness of static scheduling techniques. In contrast, parallelizing a program by executing multiple basic blocks simultaneously requires guessing the direction that conditional branches will take, how many times a particular basic block may be executed, and how large the data structures will be.

Our approach of combining partial evaluation with parallelism grain size selection was used to implement the compiler for the Supercomputer Toolkit parallel processor.[6] [1] The Toolkit compiler operates in four ma-

[5] For instance, matrix multiplication performs the same set of operations, regardless of the particular numerical values of the matrix elements.

[6] See Appendix for a brief overview of the architecture of the Supercomputer Toolkit parallel processor"

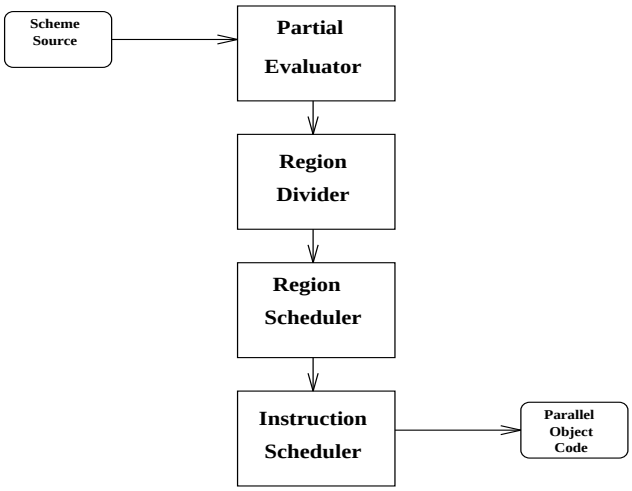

Figure 1: Four phase compilation process that produces parallel object code from Scheme source code.

jor phases, as shown in Figure 1. 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. 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.

3 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 numerical operations to be performed at run time.

The Toolkit compiler performs partial evaluation using the symbolic execution technique described in [4]. The partial evaluator takes as input the program to be compiled, as well as the input data structures associated with a particular application. Some numerical values within the input data structures will not be available at compile time; these missing numerical values are represented by a data structure known as a *placeholder*. The data-independent portions of the program are executed symbolically at compile time, allowing all operations that do not depend on missing numerical values to be performed in advance, leaving only the lowest-level numerical operations to be performed at runtime. This process is illustrated in Figure 2, which shows the result of partially evaluating a simple sum-of-squares program.

```
HIGH-LEVEL PROGRAM:

(define (square x) (* x x))

(define (sum-of-squares L)
  (apply + (map square L)))

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

(partial-evaluate (sum-of-squares input-data))

PARTIALLY-EVALUATED PROGRAM:

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

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

(RESULT 5)
```

Figure 2: Partial evaluation of the sum-of-squares program, for an application where the input is known to be a list of three floating-point numbers, the last of which is always 3.14. Notice how the squaring of 3.14 to produce 9.8596 took place at compile time, and how all list-manipulation operations have been eliminated.

Although partial evaluation is highly effective on the data-independent portions of a program, data-dependent conditional branches pose a serious obstacle. Data-dependent conditional branches interrupt the flow of compile time execution, since it will not be known until runtime which branch of the conditional should be executed. Fortunately, most numerical programs consist of large sequences of data-independent code, separated by occasional data-dependent conditional branches.[7] We partially evaluate each data-independent segment of a

[7] Some typical uses of data-dependent branches in scientific programs are to check for convergence, or to examine the accumulated error when varying the step-size of a numerical integrator. These uses usually occur after a long sequence of data-independent code. Indeed, the only significant exception to this usage pattern that we have encountered is when

program, leaving intact the data-dependent branches that glue the data-independent segments together.[8] In this way, each data-independent program segment is converted into a sequence of purely numerical operations, forming a huge basic block that contains a large amount of fine-grain parallelism.

4 Exposing Fine-Grain Parallelism

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 has shown that this graph contains large amounts of low-level parallelism. For instance, as illustrated in Figure 3, a parallelism profile analysis of the 9-body gravitational attraction problem[9] indicates that partial evaluation exposed so much low-level parallelism that in theory, parallel execution could speed up the computation by a factor of 69 times faster than a uniprocessor execution.

Achieving the theoretical speedup factor of 69 for the 9-body problem would require using 516 non-pipelined processors capable of instantaneous communication with one another. In practice, much of the available parallelism must be used to keep processor pipelines full, and 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 while awaiting the arrival of results from neighboring processors. Communication bandwidth limitations further restrict how parallelism may be used by requiring that most values used by a processor actually be produced by that processor.

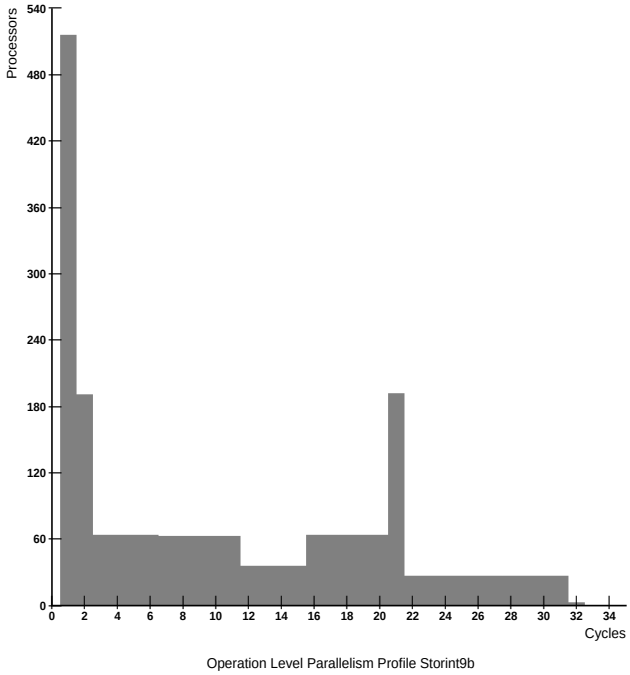

Figure 3: Parallelism profile of the 9-body problem. This graph represents all of the parallelism available in the problem, taking into account the varying latency of numerical operations.

5 Grain Size vs. Bandwidth

We have found that bandwidth limitations make it impractical to use critical path based scheduling techniques to spread fine-grain parallelism across multiple processors. In the latency-limited case investigated by Berlin and Weise [5], it is feasible to schedule a fine-grain operation for parallel execution whenever there is sufficient time for the operands to arrive at the processor doing the computing, and for the result to be transmitted to its consumers. Hence it is practical to assign non-critical-path operations to any available processor. Bandwidth limitations destroy this option by limiting the number of values that may be transmitted between processors, thereby forcing operations that could otherwise have been computed elsewhere to be computed on the processor that is the ultimate consumer of their results. Indeed, on the Supercomputer Toolkit architecture, which suffers from both latency and bandwidth limitations, heuristic techniques similar to those used by Berlin and Weise achieved a dismal speedup factor of only 2.5 using 8 processors. One possible solution to the bandwidth problem is to modify the critical-path based scheduling approach to make a much more careful and computationally-expensive decision regarding which results may be transmitted between processors, and which processor a particular result should be computed in. Although this modification could be achieved by adding a backtracking heuristic that searched for different ways of assigning each fine-grain instruction to a processor,[10]

a matrix solver examines the numerical values of the matrix elements in order to choose the best elements to use as pivots. [3] describes additional techniques for partially evaluating data-dependent branches, such as generating different compiled code for each possible branch direction, and then choosing at run-time which set of code to execute. Although techniques of this sort can not overcome large-scale control flow changes, they have proven quite effective for dealing with localized branches such as those associated with the selection operators MIN, MAX, and ABS, as well as with piecewise defined equations.

[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 seperate 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[19, 20, 13] describes additional techniques for automating the partial-evaluation process across data-dependent branches.

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

[10] Indeed, one possibility would be to design the backtrack-

this optimization based approach seems computationally prohibitive for use on the huge basic blocks produced by partial evaluation.

6 Adjusting the Grain Size

Rather than extending the critical-path based approach to handle bandwidth limitations by searching for a globally acceptable fine-grain scheduling solution, we seek to hide the bandwidth limitation by increasing the grain-size of the operations being scheduled. 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.*[11] Some regions will be 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 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 will be large, corresponding to localized computation of intermediate results, while other regions 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. For instance, as illustrated in Figure 4, 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,[12] 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.

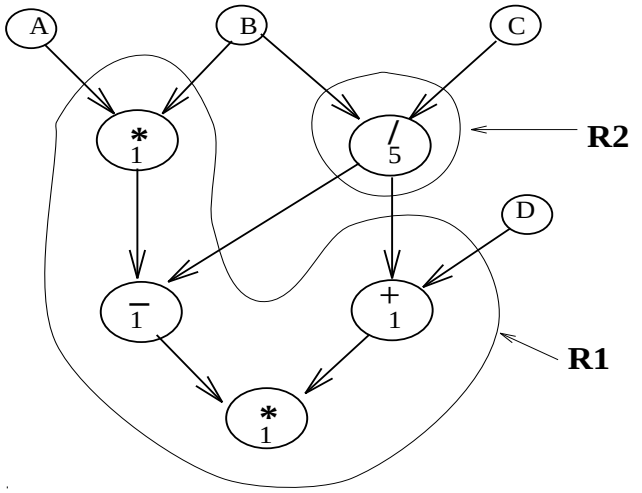

Figure 4: **A Simple Region Forming Heuristic.** A region is formed by grouping together operations that have a simple producer/consumer relationship. This process is invoked repeatedly, with the region growing in size as additional producers are added. The region-growing process terminates when no suitable 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.)

An important aspect of grain-size adjustment is that the grain-size is not increased uniformly. As shown in Figure 5, 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 would not be 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 would increase the grain-size, but would 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 reduce inter-processor communication.

7 Parallel Scheduling

Exploiting locality by grouping operations into regions forces closely-related operations to occur on the same

ing heuristic based on a simulated annealing search of the scheduling configuration space.

[11] 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.

[12] 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.

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

Figure 5: 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 as measured by computational latency (cycles). The program was divided into 292 regions, with an average region size of 7.56 cycles.

processor. Although this reduces inter-processor communication requirements, it also eliminates many opportunities for parallel execution. Figure 6 shows the parallelism remaining in the 9-body problem after operations have been grouped into regions. Comparison with Figure 3 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 inter-processor communication channels.

The *Supercomputer Toolkit* 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. However, 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[13] have proven

[13]The heuristics used by the region-level scheduler are closely related to list-scheduling [8]. A detailed discussion of the heuristics used by the region-level scheduler is presented in [22].

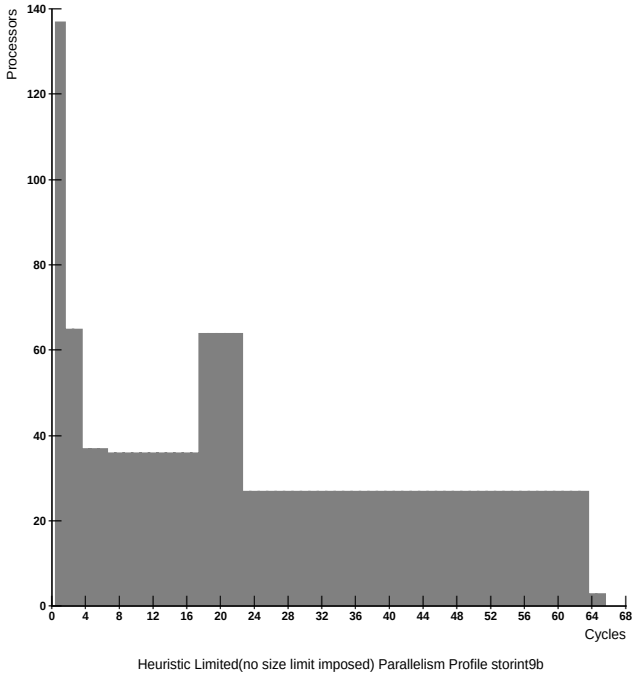

Figure 6: Parallelism profile of the 9-body problem after operations have been grouped together to form regions. Comparison with Figure 3 clearly shows that increasing the grain-size significantly reduced the opportunities for parallel execution. In particular, the maximum speedup factor dropped from 98 times faster to only 49 times faster than a single processor.

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 1%.

The final phase of the compilation process is instruction-level scheduling. The region-level scheduler provides the instruction-level scheduler with a set of operations to execute on each processor, along with a set of preferences regarding the order in which those operations should be computed, and a list of results that need to be transmitted among the processors. The instruction-level derives low-level pipelined instructions for each processor, chooses the exact time and communication channel for each inter-processor transmission, and determines where values will be stored within each processor. The instruction-level scheduler chooses the final ordering of the operations within each processor, taking into account processor pipelining, register allocation, memory access restrictions, and availability of interprocessor-communication channels. Whenever possible, the order of operations is chosen so as to match the preferences of the region-level scheduling phase. However, the instruction-level scheduler is free to reorder operations as needed, intertwining operations without regard to which coarse-grain region they were originally a member of.

The instruction-level scheduler begins by performing a data-use analysis to determine which instructions share data values and should therefore be placed near each other for register allocation purposes. The scheduler combines the data-use information with the instruction-ordering preferences provided by the region-level scheduler to produce a scheduling priority for each instruction. The scheduling process is performed one cycle at a time, performing scheduling of a 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. Due to this competition for data and resources, operations may be scheduled out of order if their resources happen to be available, in order to keep the processor busy. Indeed, when the performance of the instruction-scheduler is measured independently of the region-scheduler, by generating code for a single 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[14] 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 50 or even 100 instructions earlier than the computation instruction. Since on the Supercomputer Toolkit, memory operations must compete for bus access with interprocessor messages, retroactive scheduling of memory references helped to avoid interference between memory and communication traffic.

[14] 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.

8 Performance Measurements

The Supercomputer Toolkit and its 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.[15] Parallelization of these integrations has been previously studied by Miller[15], 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 to each planet.[2] Previous work in partial evaluation [3, 4, 5] has shown that the 9-body problem contains large amounts of fine-grain parallelism, making it plausible 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 interprocessor communication latency and bandwidth limitations. Figure 7 shows the parallel speedups achieved by our compiler for several different N-body interaction applications. Figure 9 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 interprocessor communication pathways, as illustrated in Figure 10.

9 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 represented an object as a doubly-linked list of numbers, the compiled program would as well. Only through partial evaluation can the

[15] For instance, [23] describes results obtained using the *Supercomputer Toolkit* that prove that the solar system is chaotic.

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

Figure 7: 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. Speedup is the single processor execution time of the computation divided by the total execution time on the multiprocessor.

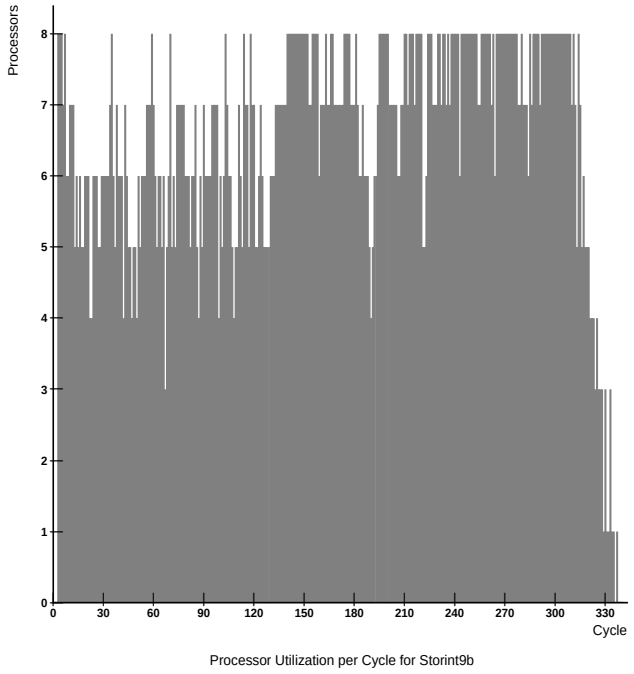

Figure 8: The result of scheduling the 9-body problem onto 8 *Supercomputer Toolkit* processors. Comparison with with the region-level parallelism profile (figure 6) illustrates how the scheduler spread the course-grain parallelism across the processors. A total of 340 cycles are required to complete the computation. On average, 6.5 of the 8 processors are utilized during each cycle.

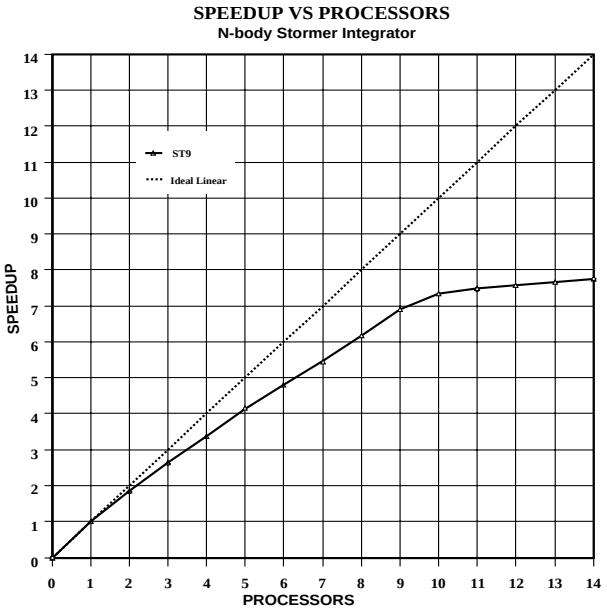

Figure 9: Speedup graph of Stormer integrations. Ample speedups are available to keep the 8-processor *Supercomputer Toolkit* busy, However, the incremental improvement of using more than 10 processors is relatively small.

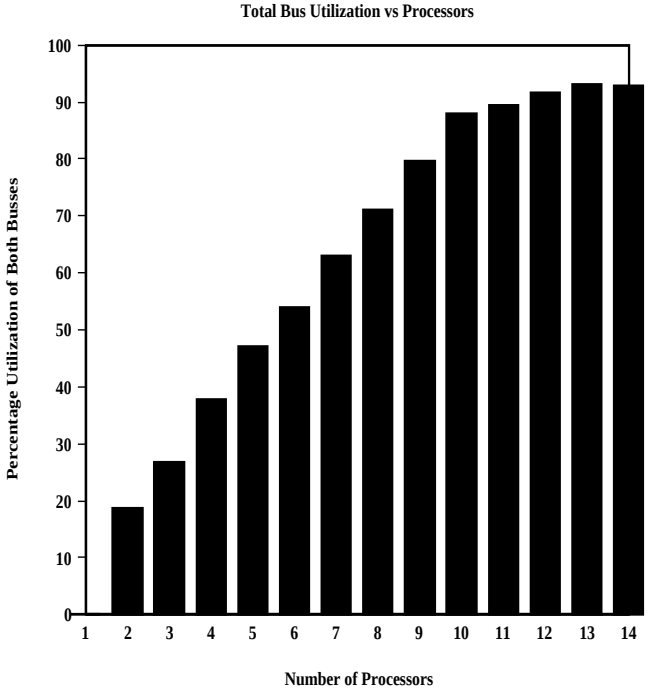

Figure 10: Utilization of the inter-processor communication pathways. The communication system becomes saturated at around 10 processors. This accounts for the lack of incremental improvement available from using more than 10 processors that was seen in Figure 9.

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.

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

9.1 Trace Scheduling

Compilers that exploit fine-grain parallelism often employ trace-scheduling techniques [9] 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 10-30 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[8] 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 the 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.

9.2 Software Pipelining

Software Pipelining [11] 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.

9.3 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 matrix multiplication). 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, the programs do 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.

9.4 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 [7]. 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.

9.5 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.[12]. 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, this 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. It appears that the compiler for *Oscar* could benefit a great deal from the use of partial evaluation.

10 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 partial evaluation with static scheduling techniques in the *Supercomputer Toolkit* compiler has allowed scientists to write programs that reflect their way of thinking about a problem, eliminating the need to write programs in an obscure style that makes parallelism more apparent.

11 Acknowledgements

This work is a part of the *Supercomputer Toolkit* project, a joint effort between M.I.T. and Hewlett-Packard corporation. The Toolkit project would not have been possible without continual support and encouragement from Joel Birnbaum, Gerald Jay Sussman, and Harold Abelson. Willy McAllister of Hewlett-Packard supervised the hardware design and construction of the *Supercomputer Toolkit* parallel processor.

Guillermo Rozas developed the host interface software and microcode assembler that made it possible to execute programs on the Supercomputer Toolkit hardware. He also contributed to the design of the instruction-scheduling techniques we describe in this paper.

Special thanks are also due to everyone who helped to construct the *Supercomputer Toolkit* hardware and software environment, especially Willy McAllister, Gerald Jay Sussman, Harold Abelson, Jacob Katzenelson, Guillermo Rozas, Henry Wu, Jack Wisdom, Dan Zuras, Carl Heinzl, and John McGrory. Karl Hassur and Dick Vlach did the clever mechanical design of the board and cables. Albert Chun, David Fotland, Marlin Jones, and John Shelton reviewed the hardware design. [4 Sam-Sam Cox, Robert Grimes, and Bruce Weyler designed the PC board layout. Darlene Harrell and Rosemary Kingsley provided cheerful project coordination. Sarah Ferguson implemented a package for adaptive-stepsize Runge-Kutta integration that is built on top of the structure provided by our compiler.

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] 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.

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

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

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

[6] S. Borkar, R. Cohen, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B.

Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An Integrated Solution to High-speed Parallel Computing," *Supercomputing '88*, Kissimmee, Florida, Nov., 1988.

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

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

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

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

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

[12] 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

[13] 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.

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

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

[16] 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.

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

[18] D. Padua and M. Wolfe, *Advanced Compiler Optimizations for Supercomputers*, Communications of the ACM, Volume 29, Number 12, December 1986.

[19] 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.

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

[21] 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.

[22] R. Surati, "A Parallelizing Compiler Based on Partial Evaluation", S.B. Thesis, Massachusetts Institute of Technology, June 1992.

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

A Appendix: Architecture of the Supercomputer Toolkit

The *Supercomputer Toolkit* is a MIMD computer. It consists of eight separate VLIW(Very Long Instruction Word) processors and a configurable interconnection network. A detailed review of the *Supercomputer Toolkit* architecture may be found in [1]. Each toolkit processor has two bi-directional communication ports that may be connected to form various communication topologies. The parallelizing compiler is targeted for a configuration in which all of the processors are interconnected by two independent shared communication buses. The processors operate in lock-step, synchronized by a master clock that ensures they begin each cycle at the same moment. Each processor has its own program-counter, allowing independent tasks to be performed by each processor. A single "global" condition flag that spans the 8-processors provides the option of having the individual processors act together so as to emulate a ULIW (ultra-long instruction word) computer.

B The Toolkit Processor

Figure 11 shows the architecture of each processor. The design is symmetric and is intended to provide the memory bandwidth needed to take full advantage of instruction-level parallelism. Each processor has a 64-bit-floating-point chip set, a five-port 32x64-bit register file, two separately addressable data memories, two in-

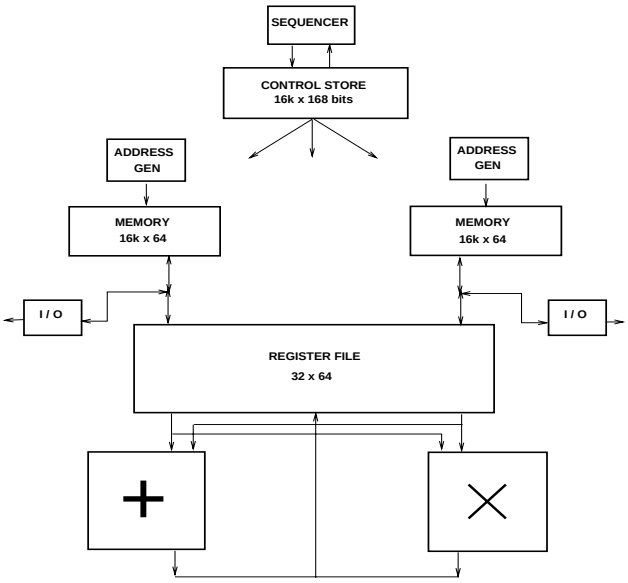

Figure 11: This is the overall architecture of a Supercomputer Toolkit processor node, consisting of a fast floating-point chip set, a 5-port register file, two memories, two integer alu address generators, and a sequencer.

teger processors[16] for memory address generation, two I/O ports, a sequencer, and a separate instruction memory. The processor is pipelined and is thus capable of initiating the following instructions in parallel during each clock cycle: a left memory-I/O operation, a right memory-I/O operation, an FALU operation,[17] an FMUL operation[18], and a sequencer operation.[19] The compiler takes full advantage of the architecture, scheduling computation instructions in parallel with memory operations or communication. The Toolkit is completely synchronous and clocked at 12.5 Mhz. When both the FALU and FMUL are utilized, the Toolkit is capable of a peak rate of 200 Megaflops, 25 on each board. The compiler typically achieves approximately 1/2 of this capability because it does not attempt to simultaneously utilize both the FMUL and the FALU.[20]

The compiler allocates two of the 32 registers for communication purposes (data buffering), while 3 registers are reserved for use by the hardware itself. Thus 26 registers are available for use by scheduled computations. The floating-point chips have a three stage pipeline whereby the result of an operation initiated on cycle N will be available in the output latch on cycle $1 + N + L$, where L is the latency of the computation. The result can then be moved in the register-file during any of the following cycles, until the result is moved into the output latch. There are feedback (pipeline bypass) paths in the floating-point pipeline that allow computed results to be fed back for use as operands in the next cycle. The compiler takes advantage of these feedback mechanisms to reduce register utilization,

The bus that connects the memory, I/O port, and register-file is a resource bottleneck, allowing either a memory load, a memory store, an I/O transmission, or an I/O reception to be scheduled during each cycle. This bus appears twice in the architecture, in each of the two independent memory/I-O subsystems.

C Interconnection Network and Communication

The toolkit allows for flexible interconnection among the boards through its two I/O ports. The interconnection scheme is not fixed and many configurations are possible, although changing the configuration requires manual insertion of connectors. The compiler currently views this network as two separate buses: a left and a right bus. Each processor is connected to both buses through its left and right I/O ports. This configuration was chosen as the one that would place the fewest locality restrictions on the types of programs that could be compiled efficiently. However, targeting the compiler for other configurations, such as a single shared bus on the left side, with pairwise connections between processors on the right side, may prove advantageous for certain applications. Each transmission requires two cycles to complete. Thus in the two shared-bus 8-processor configuration, only one out of every eight results may be transmitted. Pipeline latencies introduce a six cycle delay between the time that a value produced on one processor is available for use by the floating-point unit of another processor.

The hardware permits any processor to transmit a value at any time, relying on software to allocate the communication channels to a particular processor for any given cycle. Once a value is transmitted, each receiving processor must explicitly issue a "receive" instruction one cycle after the transmission occurred. The compiler allocates the communication pathways on a cycle by cycle basis, automatically generating the appropriate send and receive instructions.

[16] Each memory address generator processor consists of an integer processor tied closely to a local register file.

[17] The FALU is capable of doing integer operations, most floating-point operations, and many other one-cycle operations It is tagged + in figure 11

[18] The FMUL is capable of doing floating-point multiplies(1 cycle latency), floating-point division(5 cycle latency), and floating-point square roots(9 cycle latency) as well as many other operations. It is tagged * in figure 11

[19] The sequencer contains a small local memory for handling stack operations.

[20] Simultaneous utilization of the FMUL and FALU is only occasionally worthwhile for long multiply-accumulate operations. Since the FMUL and FALU share their register-file ports, opportunities for making simultaneous use of both units are rare.