APEX:
THE APL PARALLEL EXECUTOR

by

Robert Bernecky

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

APEX:
the APL Parallel Executor

Robert Bernecky
Master of Science
Graduate Department of Computer Science
University of Toronto
1997

APEX is an APL-to-SISAL compiler, generating high-performance, portable, parallel code that executes up to several hundred times faster than interpreted APL, with serial performance of kernels competitive with FORTRAN. Preliminary results indicate that acceptable multi-processor speedup is achievable.

The excellent run-time performance of APEX-generated code arises from attention to all aspects of program execution: run-time syntax analysis is eliminated, setup costs are reduced, algebraic identities and phrase recognition detect special cases, some matrix products exploit a generalization of sparse-matrix algebra, and loop fusion and copy optimizations eliminate many array-valued temporaries. In addition, the compiler exploits Static Single Assignment and *array morphology*, our generalization of data flow analysis to arrays, to generate run-time primitives that use superior algorithms and simpler storage types.

Extensions to APL, including *rank*, *cut*, and a monadic operand for *dyadic reduction*, improve compiled and interpreted code performance.

Dedication

This work is dedicated to my parents, Joe Bernecky and Ann MacIver Bernecky, to whom I owe the greatest debt of gratitude. Their devotion to our family has never waivered – they have always stood behind “us kids,” encouraging us to do what we thought was right and ever urging us to further our education. A child could not ask for better parents.

Acknowledgements

Many friends, relatives, and colleagues are woven into the fabric of this research. It is impossible to thank them all, but certain outstanding individuals must be acknowledged.

I have had the good fortune to have excellent teachers. My encounter with the APL community brought me in contact with even more of them. Ken Iverson, of course, made that happen through his vision of APL as a generalization of mathematical notation. Without the influence of Ian Sharp and Roger Moore, I probably never would have gotten involved with APL. Ian has been a a sounding board for ideas, a mediator for conflict, and an ear for troubles and woes. Roger taught me how to construct very large, robust software systems and to be unafraid to redesign offending hardware or software. Hiroshi Isobe, of the Hitachi Software Works, opened my eyes to new facets of management, software design, and maintenance. A true sensei, he taught me more than he suspects. George Moeckel and Stephen Jaffe of Mobil Research Development Corporation encouraged my early research into compiled APL with the ACORN project. My thesis advisor, Corinna G. Lee, asked the tough questions at the right times and provided me with invaluable assistance in structuring this thesis and making it comprehensible to readers. I am grateful for the criticism and direction she provided to me in the creation of this work.

I owe special thanks to Pat Miller of Lawrence Livermore National Laboratory. Pat dealt patiently with my myriad questions and problems with SISAL and OSC, and solved a number of performance-related problems that I encountered in the course of APEX design and development. I have also been fortunate to have the support of John Feo and his colleagues at the Lawrence Livermore National Laboratory (LLNL) and the National Energy Research Supercomputer Center (NERSC). John has encouraged my line of research and has been helpful in resolving a number of SISAL-related problems. He also provided me with access various supercomputers for benchmarking purposes.

The APEX project has received support from several sources. We gratefully acknowledge financial support from the Information Technology Research Center, a Center of Excellence supported by Technology Ontario. Reuters Information Systems Canada Limited provided benchmark programs from their historical databases systems. Dyadic Systems Limited, Manugistics Inc., and Soliton Associates Limited provided their APL interpreter products for benchmarking purposes. Soliton also provided access to their Sun and IBM workstations for running benchmarks. Silicon Graphics made an SGI Power Challenge available for benchmarks, through the auspices of NERSC and the LLNL Physics and Space Sciences Directorate.

Finally, I thank my family and neighbors, who have been stalwart and patient through it all, for helping me to get through the rough bits. In particular, my dear friend, Marion Barber, through her love and support, gave me the motivation to complete this thesis, get off the computer, and get back on my bicycle.

Contents

List of Figures

Chapter 1

Introduction

Humans are tool builders. The tools we create are, by and large, physical ones that multiply our muscle power. The digital computer stands out as a non-physical exception that instead multiplies our mental power. Yet, the computer is not itself a tool, so much as a meta-tool that helps us to exploit the real tools of thought – formal notations, such as mathematics and logic, and programming languages, such as APL and LISP.

Analysts use programming languages to build new tools – computer programs – for the design, modeling, and analysis of large, complex systems, then execute those programs on powerful computers to explore areas as diverse as banking, chemistry, economic modeling, insurance, and physics. Analysts measure their productivity and, frequently, the success of their organization, by *time-to-solution*; that is, the rate at which they are able to find solutions to the unique and constantly changing set of problems facing them. Since those solutions continually require the creation and the use of new tools, the ability to develop programs faster is as important as improved performance of those programs, because advances in either development speed or execution speed will improve time-to-solution. In areas such as financial trading, economic modeling, or computational chemistry for pharmaceutical design, a late answer can be worthless. Therefore, extremely rapid development is a major factor in analysts' choice of computer languages; time-to-solution is as critical to analysts as time-to-market is critical to manufacturers.

Although analysts are always on the lookout for faster development tools and faster computers, we shall see that choosing a computer language for analysis is not a simple task – languages that facilitate rapid application development often execute slowly, whereas languages that execute quickly may substantially slow the pace of program development. The family of programming languages used as computational analysis tools may be broken into two classes based on their design ethos – those that reflect the computer, and those that reflect human thought. The former class is exemplified by conventional languages, such as FORTRAN and C, which were primarily designed with an eye toward machine efficiency. The latter class of programming languages, such as LISP, APL, PROLOG, and J, include those that arose from logic and mathematics.[1] Each computer language has its own unique advantages and

[1] Indeed, Iverson's Turing Award speech was entitled *Notation as a Tool of Thought* [Ive79].

disadvantages, but several important qualities – development speed, run-time performance, and portable parallelism – tend to be shared by all members of the class to which they belong. We shall now take a brief look at these characteristics.

The thought-oriented class of languages are terse – they hide the details of computation, permitting a programmer to concentrate on *what* a computation should do, rather than *how* it is to be done. APL is a typical example of such a thought-oriented language. Based on linear algebra, APL is an array-oriented extension and rationalization of mathematical notation. APL's abstract data, array-valued primitives, and higher-order functions allow programmers to design and develop systems up to an order of magnitude faster than if they had used computer-oriented languages [Mar82, Yos86, Wea89, RB90]. APL's abstraction improves program reliability and significantly reduces the size of programs. In addition, the terse programs that result from the use of APL may facilitate the comprehension, maintenance, and enhancement of applications. Since these characteristics reduce tool development time, APL is frequently an analyst's language of choice.

By contrast, computer-oriented serial languages, such as C++, ADA, PL/I, and PASCAL, force programmers to pay critical attention to numerous coding details, including memory management, counters, and loop limits, all of which are not germane to the problem being solved. Attention to this irrelevant detail slows the pace of program development by increasing the amount of code that must be written to obtain a functioning program. Furthermore, that detail is likely to increase the program bug rate by an amount proportional to the text size, for typographical and logical errors inevitably creep into any large body of human-authored text. The presence of excessive detail also obscures the algorithm being expressed, increasing the difficulty of enhancing, maintaining, and comprehending programs, and complicating the task of making assertions about program correctness. These factors dramatically increase development and debugging time, and reduce the utility of computer-oriented languages as computational analysis tools.

Turning to run-time performance, a second important characteristic of computer languages, we find that the development-time advantage that abstract languages offer over computer-oriented languages is not obtained without cost. APL programs, for instance, may execute as much as several hundred times slower than the same programs coded in a compiled language, due to interpretive overheads and other factors to be discussed in Chapter 2. This run-time performance penalty may reduce or eliminate any time-to-solution edge offered by an abstract language's superiority in development time.

A third quality of a computer language, now emerging as a critical factor in reducing time-to-solution, is its ability to express parallel computations in a portable and efficient manner. Reduced time-to-solution – getting answers sooner from an application – requires faster computers, improved language translators, and parallel computers. Although we expect the performance of computers and language translators to improve apace, we find that the design of computer languages for exploiting parallel computers lags far behind our capability to design and build such computers. If we are to exploit the power of parallel computation to reduce time-to-solution, we need to provide non-experts with tools for *portable parallelism* – tools that will let them make effective use of a wide range of parallel

computer architectures without requiring that they be familiar with the internals of those architectures, any more than a car driver needs to know whether a car is powered by a V-8 or a straight-6 engine.

1.1 Reducing Time-to-Solution

Time-to-solution, like many other problems in computing, involves a time tradeoff – we can have slow development and fast execution by using FORTRAN, or we can have rapid development and slow computation by using APL. When we seek ways to reduce time-to-solution, we naturally look at methods for improving the slowest parts, since the biggest payoffs will arise from making them faster. This suggests looking at ways to speed up development in FORTRAN and to improve execution performance in APL interpreters, as summarized in Section 2.4. However, to date, such efforts have not been wildly successful, so we turn to the possibility of compiling APL.

Since many languages use compilers to achieve high levels of run-time performance, it is natural to consider compiling APL to obtain the performance levels we desire. Compiling APL is, in fact, the approach we advocate for reducing time-to-solution. If a compiler could eliminate run-time syntax analysis and conformability checking, reduce memory management costs, and perform traditional compiler optimizations, then APL could provide extremely rapid time-to-solution – rapid development would be combined with fast execution and excellent portable parallelism. Chapter 7 shows how such an APL compiler might perform against FORTRAN in both serial and parallel environments to obtain a dramatic advantage time-to-solution over FORTRAN, arising from the longer development times that FORTRAN programs require; it is exacerbated in a parallel environment by the addition time required in FORTRAN to obtain acceptable levels of parallelism.[2]

Given this potential, one expects to see high-performance APL compilers on the market. Yet, APL has proved resistant to high-performance compilation. Although little information about specific performance problems in compiled APL has been published, some known problem areas are inadequate data flow analysis, absence of global and interprocedural optimization, failure to implement state-of-the-art algorithms for APL primitives, poor memory management, and naive code generation. The net result is that the run-time performance of APL compilers have been less than spectacular – much of the work done in interpretive environments was still being done at run-time in the compiled environment, thereby negating much of the potential performance benefit of compilation.

Creation of a high-performance APL compiler requires an integrated approach to design, in which *all* performance-related issues are addressed. Failure to adopt such a design attitude will result in limited performance improvements. To see why this is so, consider the effect of ignoring just one aspect of performance-related design – memory management – in an otherwise superb hypothetical compiler. Note that interpreted APL applications may spend large amounts of time allocating and deallocating array memory: a typical APL interpreter will spend 10–25% of processor time performing memory

[2]Several researchers are building tools to assist the programmer in parallelizing FORTRAN code.

management functions.[3] Now, contemplate the very unlikely case of a hypothetical compiler in which memory management performance was identical to that of an interpreter and *all* operations not related to memory management were executed instantly. Amdahl's Law tells us that a compiled application with the above memory management profile could not run more than four to ten times faster than it would under an interpreter. If we were to pick some other isolated aspect of compiler design, we would observe similar disappointing performance results. Hence, we conclude that, to be competitive, an APL compiler must concurrently address and solve a number of design problems.

Although it is only one of the problems facing an APL compiler writer, run-time memory management is a nightmare for most functional languages. Fortunately, the memory management problem for one functional language, SISAL, has been largely solved [JSA+85]. Researchers on the SISAL Language Project enhanced the Optimizing SISAL Compiler (OSC) so that it was able to remove a majority of memory management operations from typical applications, generating functional language applications whose performance was competitive with FORTRAN [Can92a, SS88]. Intrigued by the possibility of reducing time-to-solution by exploiting these advances in data sharing, copy avoidance, and reference count removal, we were led to write an APL-to-SISAL compiler, dubbed *APEX – the APL Parallel Executor*.[4]

1.2 The APEX Audience

APL has been described as a *boutique language* – it caters to a relatively small group who know what they want and are willing to pay a premium to get it. To see why this group might want an APL compiler, we will now look at the APL user community of today. It is fairly apparent that the community exhibits a bipartite distribution. One major group of APL application writers are those who use mainframes to provide services to a large body of users of shared corporate data. The organizations involved comprise data providers such as Reuters; financial institutions including the Bank of Montreal, Credit Suisse, Deutsche Bank, and Morgan Stanley; insurance companies such as Irish Life, Massachusetts Mutual, and Sun Life; and manufacturing firms including IBM, Novo Pharmaceuticals, and Xerox. There are probably about 2,500–3,000 sites operating such large APL systems. The other large class of APL users are those who operate PC- or workstation-based APL systems serving one or a few users. These people either use APL directly, as an analysis tool, or create software products that happen to use APL as an implementation language. The absence of hard industry data here makes it difficult to estimate the size of this market segment, but it probably lies between 20,000–100,000 sites.

When we consider the set of current users of APL who could readily benefit from the use of APEX, we find that they fall into a few major classes: analysts; developers of large, rapidly evolving applications; and maintainers of legacy systems. Potential new users include computational scientists who have been unable to use APL for their work because of inadequate interpreter performance. The benefits of

[3] This figure is based on the author's analyses of mainframe APL applications, performed with hardware monitors as well as with interrupt-driven statistical monitors. Details of this data are presented in Chapter 2.

[4] Although the term *executor* is usually reserved for interpreters, the acronym was irresistible.

compiled APL for analysis have been discussed already; the other classes of user merit some mention.

The developers of complex, rapidly evolving applications have a major problem with APL performance. They use APL because it is the only tool that permits them to keep pace with changing application requirements. A typical example of such use is the maintenance and delivery of large volumes of financial data. In such data-intensive applications, data may be coming from stock exchanges around the world, in drastically different formats, encoded according to rules that change haphazardly and with little or no warning. The same stock may trade on several exchanges under different names and with different currency and different trading rules. Attempting to receive, filter, and validate data is a major task in itself, as is the job of updating the affected data bases. Developers of such systems have found that only APL is up to the task of being adaptable to such a constantly changing environment.

Since APL facilitates rapid adaptation of applications to changing needs, APL has remained in production use even when management, following the latest in industry fashion trends, has made decisions to cease use of APL and convert to a fourth-generation language, such as C++. Several of these initiatives have clearly demonstrated APL's development time edge to management. In three cases the author is familiar with, large teams of C programmers labored for several years to write code to replace an extant APL application. By the time they were finished, the production APL application had evolved to something radically different from the already-obsolete C code. The APL applications remained in place; the corporations had expended millions of dollars in development funding to no productive end. Wiedmann and Busman cite similar examples [Wie86, BFK95].

The downside of APL's flexibility is a high cost in delivery – APL applications often consume entire mainframes, at a cost of millions of dollars. Some of this cost is mandatory, because the I/O bandwidth available with large mainframes is required to support the demands of such applications. The processor time, however, goes largely to APL interpreter overhead – time that could be eliminated by effective use of an APL compiler, with significant hardware cost reductions for the site.

A related class of users who could benefit from APEX is the maintainers of APL legacy systems. Many of these applications were written in the 1970s, before the advent of the personal computer. Although they would probably not be written in APL today, they often provide functionality that is still unavailable in contemporary products. Hence, these applications remain in use today, providing services such as email, remote printer drivers, and text and program editors. These applications frequently operate on small arrays and are therefore dominated by interpretive overhead. Compilation of these applications could reduce site processor demands and improve response time for their users.

Finally, if APEX parallel performance is high enough, it may attract a new class of users – supercomputer users who absolutely require the performance available with parallel computers to solve their problems. The performance and parallel expression available with APEX, combined with its capability for rapid development, may provide an impetus for supercomputer users to move to APL.

1.3 APEX – the APL Parallel Executor

Our general strategy in the design of APEX was to minimize our implementation effort by using, as much as possible, existing compilers and tools, and to adapt existing compiler theory, such as data flow analysis and static single assignment, to our needs. For example, rather than re-invent the wheel by rewriting the OSC array storage analyzer and optimizer as part of APEX, we chose to exploit the OSC research by generating SISAL as the output of APEX.

We originally decided to generate IF1, the Intermediate Representation language used within OSC. Unfortunately, IF1 proved to be an intractable language for the job. It was extremely difficult to build IF1 code fragments for the code generator. Debugging the fragments was tedious. Trivial errors in code fragments, such as unreferenced definitions, would cause OSC to abort. Generated code was completely opaque – it was impossible to tell what a piece of code was intended to do. After about six months of struggling with the IF1 code generator, we decided to try generating SISAL code instead. This was a breath of fresh air. Within a week, we had a SISAL code generator with more functionality than the then-existing IF1 generator. SISAL code fragments proved to be easy to write and easy to debug. Furthermore, SISAL code generated by APEX is human-readable, facilitating debugging at the generated code level.

Alternatively, the compiler could have produced C code directly, rather than going through the intermediate steps required by generation of SISAL. However, we felt that SISAL provided us with a number of important benefits in exchange for minor restrictions in the APL we could support. Specifically, we obtained the excellent array optimizations of SISAL, inherent parallelism, and support for many target systems, in exchange for the inability to represent certain empty arrays [Can89, Can92b, CWF92].

Similarly, we could have generated machine code directly, but this would have entailed an enormous development effort, limited the range of supported target platforms, and probably resulted in performance levels inferior to that achievable by use of extant C optimizers and machine-dependent code schedulers.

With regard to the application of extant compiler theory, we attempted to minimize the need to invent or reinvent by adapting state-of-the-art algorithms to our needs. Basically, we treated arrays in the same manner that scalar-oriented compilers treat scalars, and counted on OSC to perform the array copy-avoidance analysis that is required for adequate performance of operations such as indexed assignment.

Data flow analysis and static single assignment were both critical in our quest for performance. Data flow analysis let us statically determine array type and rank, both of which are critical factors in generating efficient code to handle scalars and to avoid run-time code dispatching based on type or rank. Static single assignment (SSA) enabled us to perform data flow analysis on semi-globals[5] and to create purely functional SISAL programs from APL programs that were not expressed in functional form.

Figure 1.1 shows the global structure of APEX. The shaded components are ones that we developed

[5]By semi-globals, we mean variables that are local to one function, but visible to its callees, due to APL's dynamic scoping rules.

for this thesis. We see that the SISAL output from APEX is passed to OSC, after which the C code produced by OSC is passed to a C compiler which ultimately generates executable code. This cross-compilation approach offers a sensible division of labor among the three compilers involved in the process of compiling APL, yet does not result in excessive compilation times. With APEX, we obtain the abstract expressiveness of APL, the parallelism and array storage optimizations of SISAL, and the excellent performance of compiled C code.

APEX is written in Extended ISO Standard APL, using the control structure extensions implemented in APL2000's APL+Win interpreter. We chose APL as an implementation language because it offered a rapid development environment in which we could conveniently and quickly explore variants in data structures and algorithms. In addition, compiler debugging was facilitated by APL's interactive nature.

We ran into two problems with the adoption of APL as a compiler implementation language. First, the compiler is relatively slow, with large programs taking several minutes to compile. This is due, in part, to the poor performance of APL interpreters on iterative code, but it also arises from our concentration on the run-time performance of APEX-generated code, rather than of APEX itself. We believe that this problem will be resolved by rewriting a few major code hot spots and by compiling the compiler. The second problem is that our licensed version of APL+Win will only run under MS Windows, but we have not yet ported the SISAL compiler to operate in that non-Unix environment. Hence, we currently have to switch computers or operating systems during the compilation process. We plan to port the SISAL compiler to Windows to resolve this inconvenience.

APEX performs parsing, syntax analysis, static single assignment translation, and data flow analysis at the level of APL primitives, defined functions, and derived functions [Int84]. The data flow analysis phase extracts morphological information such as the type and rank of each array created during the execution of a program. APEX then uses this information to generate its output – a SISAL program. The APEX code generator uses code fragment tables and relies on cpp, the C preprocessor, to simplify the task of code generation and ease the task of generating optimized code. We created C macro definitions for APL primitive scalar operations, adverbs and conjunctions, type coercions, and run-time library functions. Each APL primitive or derived function generates an invocation of some C macro that is expanded by cpp into an appropriately customized piece of SISAL source code.

The Optimizing SISAL Compiler, OSC, performs parsing and syntax analysis on the SISAL code generated by APEX. OSC handles array storage and reference count analysis and does function inlining. It also performs traditional compiler optimizations at the array level, including common subexpression elimination (CSE), code motion, and loop fusion. OSC determines which code can be executed in parallel on the target system and then generates appropriate C code as its output. Target systems that OSC currently supports are the Alliant FX, Apple Macintosh, CRAY T3D, CRAY X-MP, CRAY Y-MP, CRAY C-90, CRAY 2, Encore Multimax, Hewlett-Packard Series300/400/700, IBM RS/6000, IBM-compatible PCs running Linux, Sequent Balance, Sequent Symmetry, Silicon Graphics SGI Power Challenge, SUN3, SUN Sparc, Thinking Machines CM-5, and generic uniprocessor UNIX workstations. Systems to be targeted in the future include IBM-compatible PCs running Windows 95, the

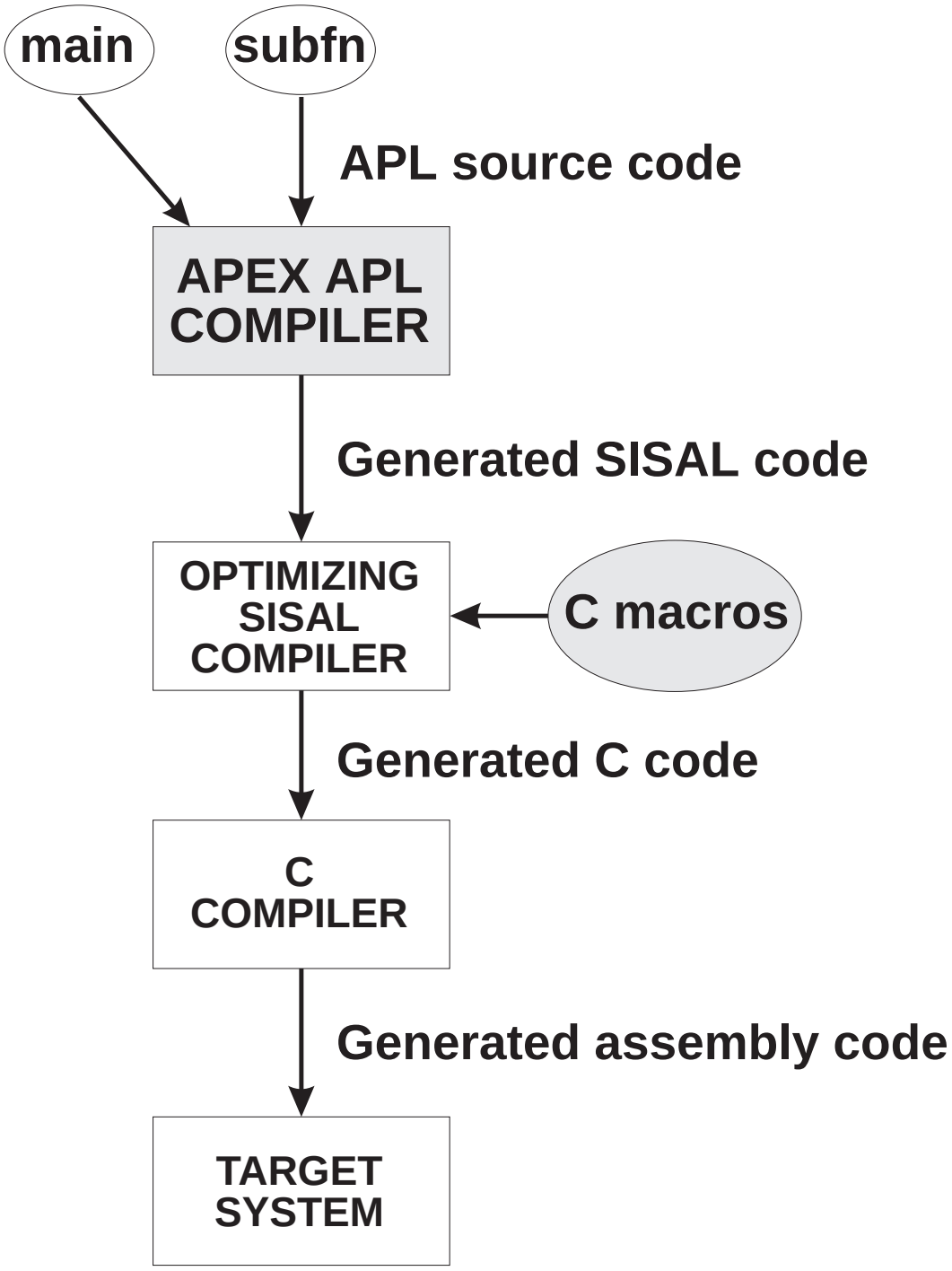

Figure 1.1: The Big Picture

IBM SP2, and a version using explicit message passing (MPI).

By generating C, a portable target language, rather than generating assembler code, OSC allows the target system's C compiler to perform the tasks it is best suited for, such as register allocation and code scheduling on RISC architectures. The C compiler, of course, also performs traditional scalar optimizations such as strength reduction and common subexpression elimination.

Thus, we see that each compiler performs work that is uniquely suited to its nature. There is little or no redundancy among their tasks, and each is able to exploit the work done by its predecessor.

1.4 Contributions of Thesis

We have created APEX – the APL Parallel Executor – a high-performance APL compiler that produces SISAL code for serial or parallel computers. We claim that APEX makes compiled APL an effective tool for array-based computation, offering excellent performance on serial computers. We will demonstrate compiled APL performance that is frequently competitive with hand-coded C and FORTRAN and that can be as much as three orders of magnitude faster than interpreted APL. Preliminary results also suggest that APEX-generated code obtains good speedup on parallel systems, with kernel benchmarks showing nearly linear speedup on a small number of processors. Letting the SISAL back end handle the synchronization and communication details of parallel computation for parallel systems facilitates the porting of applications across both serial and parallel target systems. Programs written in the traditional APL style, when compiled with APEX, provide excellent performance, thereby reducing time-to-solution by a considerable amount.

These achievements have been accomplished by:

- applying static single assignment, data flow analysis, and other static analysis methods to APL applications;
- exploiting advances in memory management and parallelism in functional languages;
- applying local, global, and interprocedural data flow analysis at the array level;
- automatically transforming APL programs containing side effects into structured, purely functional form;[6]
- adopting an integrated approach to compiler design, in which all aspects of compiled code performance are considered;
- designing the code generator to best exploit the optimization capabilities of the Optimizing SISAL Compiler;
- showing why it is important to use applications, rather than kernels, to measure the relative performance of language processors;

[6]By purely functional, we mean that all functions accept explicit arguments and produce explicit results, and that all assignments to variables are static single assignments.

- using state-of-the-art algorithms for the implementation of APL primitives and derived functions; and by
- piggy-backing on the SISAL and C compilers to automatically generate optimized, parallel code.

Details of these contributions are described in the remainder of this thesis. Chapter 2 examines APL performance issues and suggests methods by which an APL compiler might resolve them. It also discusses some ways that the APL language both helps and hinders a compiler in generating high-performance code. Chapter 3 presents the design of APEX in the light of problems that we had to address in compiling APL to SISAL – hindrances arising from the APL language itself, from compilation into a functional language such as SISAL, and from the SISAL compiler, OSC. Chapter 4 gives the experimental framework benchmarks we used to evaluate the performance og APEX. Chapter 5 and Chapter 6 present the results of a series of experiments we conducted on various hardware platforms to quantify the extent to which our implementation realized the design goals of APEX. Those experiments compared the execution time performance of APEX-generated code to that of interpreted APL and FORTRAN 77 for applications and for synthetic kernels. Chapter 7 suggests some ways in which APL can hide the details of parallel expression from the programmer, permitting parallel applications to be written in a highly portable fashion. Finally, Chapter 8 summarizes our research findings and presents topics for future research.

Chapter 2

Issues in Compiling APL

In this chapter, we examine some APL performance issues and suggest methods by which an APL compiler might resolve some of them. We also present some of the ways that the APL language both helps and hinders a compiler in generating high-performance code. Finally, we give a summary of related work. We will start with a look at APL interpreters to see why many APL applications perform slowly in an interpreted environment.

2.1 Why are APL Interpreters Slow?

Most interpreted APL applications execute slowly. That is, they consume significantly more computational resources than they would if written in a scalar-oriented compilable language. Understanding why interpreters are slow may provide enlightenment into how compiled APL can make APL applications execute faster.

As noted earlier, APL's poor run-time performance arises from its abstract nature. Each name in an APL application can theoretically change meaning from one instant to another. A function's argument may be a Boolean vector in one call, but a character matrix in the next; a function may become a variable or disappear entirely. The task of executing an APL program can become quite complex with all these meanings shifting underfoot. In computing, the price paid for complexity is lost performance. This complexity manifests itself largely in the form of interpreter run-time overheads, also known as *setup cost*, and in the form of array-valued intermediate results. We now examine each of these overheads in some detail.

2.1.1 Setup Cost

Much of the run-time overhead encountered in interpreted APL is due to the manner in which an APL primitive is executed. Execution of an APL primitive comprises four phases: syntax analysis, conformability checking, memory management, and execution of primitive-specific operations. During *syntax analysis*, an interpreter examines the currently executing line of the program to determine which

function to execute next and what its arguments are. *Conformability checking* ensures that the function's arguments conform in rank and shape. It also determines, in most cases, the shape of the result. *Memory management* obtains space for the function's result; after the primitive has completed execution, memory management discards the arguments to the function. The *execute phase* performs the actual work of the function, *e.g.*, summing the elements of an array.

Of these four phases, the first three consist of run-time checks. APL interpreters perform these run-time checks for *every* APL primitive executed in order to deal with the potentially fluid situation engendered by APL's abstract and interactive nature. We denote the time spent by the processor in performing these run-time checks as *setup cost* and the time spent in the execute phase as *execution cost*. Whereas compiled languages incur setup cost only once, during compilation, the setup cost within an interpreter is ongoing, incurred during every execution of each primitive in an application.

To get a feel for the impact of setup cost on execution time, consider the performance profile shown in Figure 2.1. This figure shows the breakdown of the execution time of an actual APL application that is a major part of a very large application used to facilitate the borrowing and lending of securities. It was written by an expert team of APL programmers and is highly tuned for maximum performance. Since this part of the application includes a database join on large arrays, setup costs are amortized over a considerable amount of computation on large arrays. Hence, this represents a well-written APL application, operating in the optimum part of the APL performance envelope.

We collected the data in December 1990 at Reuters Information Systems in Toronto as part of a performance study of the application. The data presented in Figure 2.1 was obtained by running the application under control of the IBM 3090 Program Event Recording facility, creating a histogram of each instruction executed. The resulting histogram data was then matched against the APL interpreter link-edit map and then accumulated by subfunction. Each subfunction name was manually classified as belonging to one of four classes: syntax analysis, conformability checking, memory management, and execution. The histogram data was then summarized into those classes to produce the information shown.

Figure 2.1 highlights the excessive amount of time consumed by setup cost: Half of all processor time is devoted to the overheads of syntax analysis, conformability checking, and memory management; half goes to the execute phase which represents useful work. Yet, this application represents an excellent performance profile for a realistic APL application. Applications operating on smaller arrays will exhibit far greater relative setup cost. This is because the execute phase, having less work to do, will execute faster, but setup remains the same, because the same primitives are executed in both cases. Figure 2.2, presenting the results of Wiedmann's measurements of APL expressions on small arrays, shows the execute phase comprising less than 32% of total execution time [Wie86].

Our observations of APL systems, made over more than twenty years with both hardware and software monitoring tools, shows remarkable stability of that distribution for all but a few applications. In fact, deviation from that distribution is cause to examine the application's performance in detail, as it often reflects a serious bug in the application. For example, in the securities application, original mea-

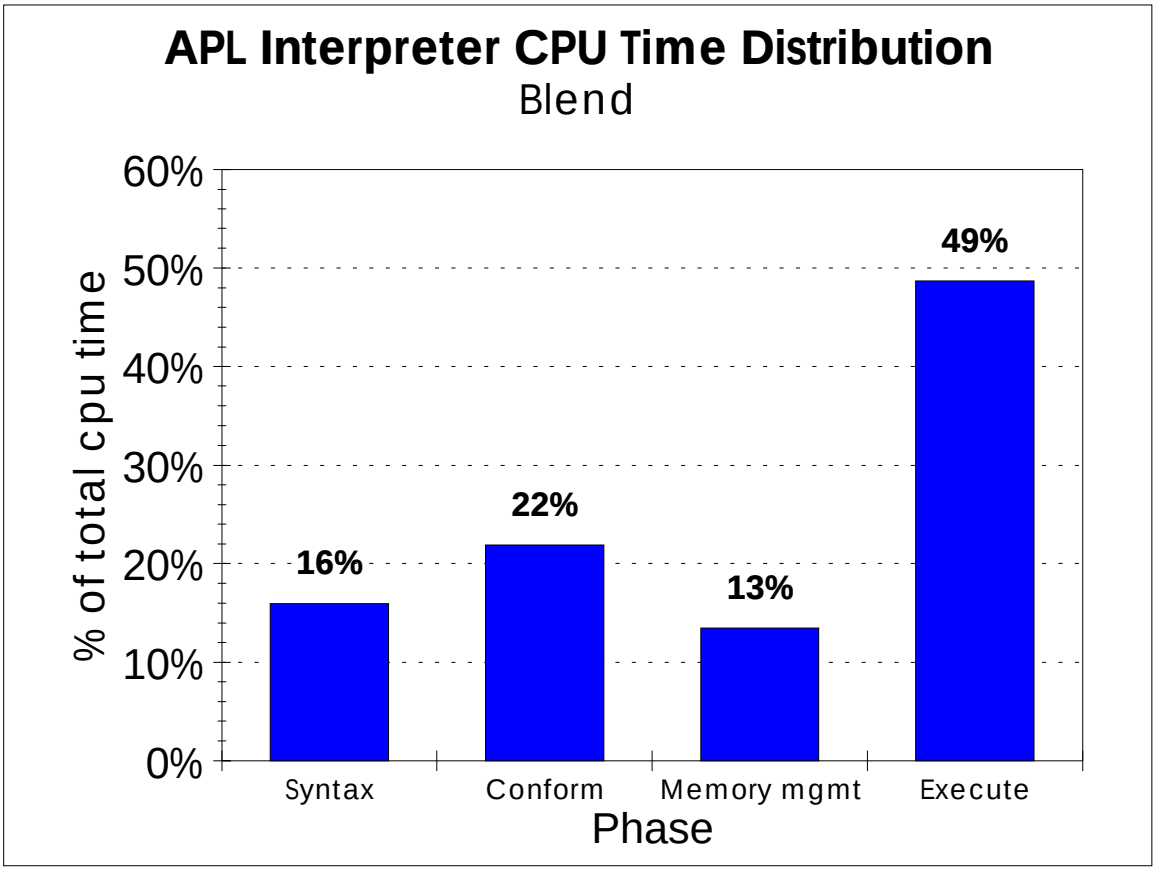

Figure 2.1: Distribution of APL interpreter CPU time

surements showed excessive time being spent in symbol table management. A trivial correction to the application cut its cost in half and restored the processor time distribution to the values shown.

As array sizes are reduced, the relative proportion of time spent in the execute phase goes down and setup cost rises. When array sizes are very small, such as when operating on scalars, the impact of setup cost is considerable. To get an indication of the effect of setup cost on per-element execution time, we performed the following simple experiment on an IBM 3090 mainframe system. We used the system's high-precision processor timer facility to measure the CPU time required to perform a simple APL operation (`a+b`) on integer vectors of varying length. Figure 2.3 presents per-element execution time versus array size. Note that an operation on a scalar executes at least thirty times slower than the same operation on one element of a 25-element vector. Setup cost, therefore, has the potential to cause severe performance degradation on small arrays. These ratios are supported by Wiedmann, who claims that "...interpretive overhead in the form of setup is awesome, and is of the order of 100 times as much as the per-element time" [Wie83].

Excessive setup cost on small arrays naturally leads to the question of whether, in practice, small arrays are used extensively in APL applications. Since APL is an array language, one would naturally

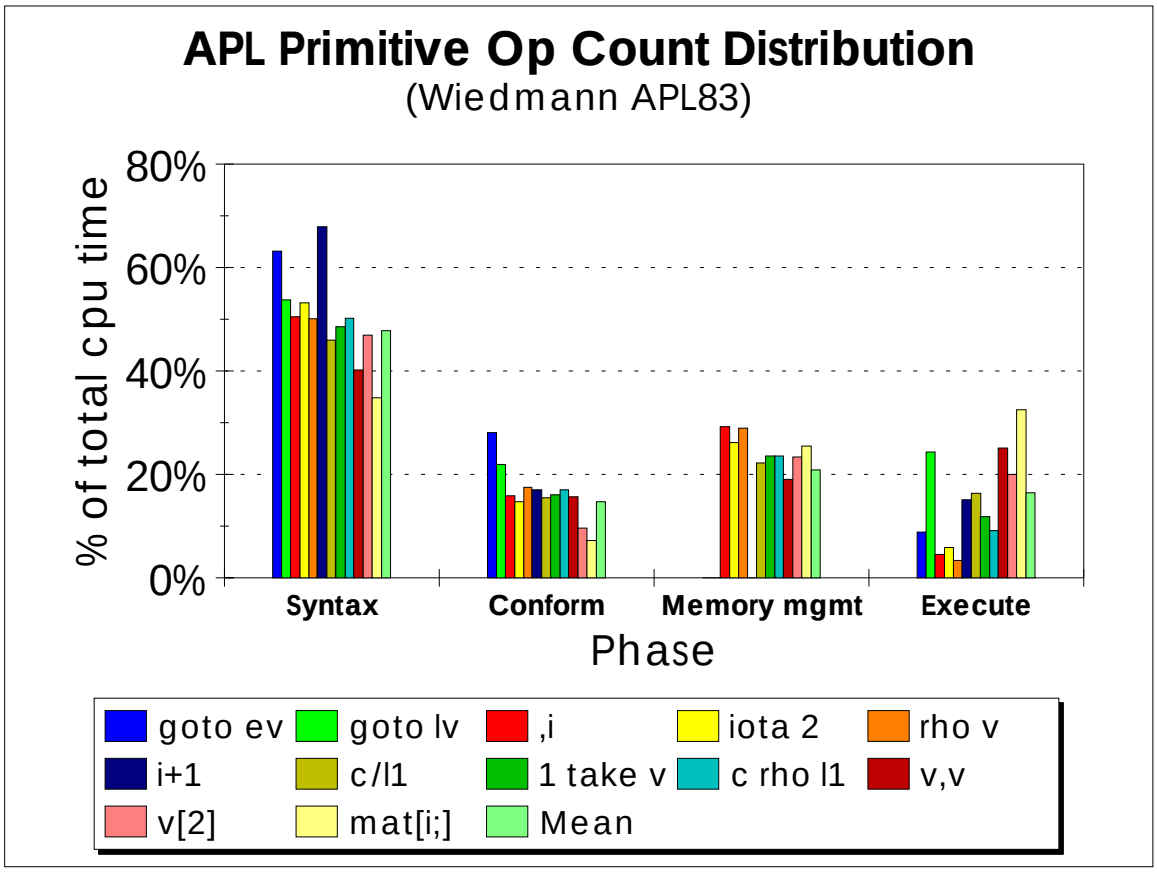

Figure 2.2: Distribution of APL interpreter CPU time

expect that programmers would exploit its array semantics; large array sizes would be very commonly observed during the execution of an application and small arrays would be a rarity. The reality of the situation is quite different – small arrays abound in APL applications.

In order to measure the distribution of array sizes in production APL applications, we instrumented a mainframe time-shared APL interpreter at Soliton Associates Limited to increment a histogram of array size for every operation performed by the interpreter. We then ran this instrumented interpreter in production mode for several weeks at the end of 1993, collecting information for every executing APL task. This system is typical of APL, executing APL-based applications for electronic mail, system scheduling, high-speed print facilities, application development, data base systems, and program editors. The mainframe legacy applications discussed in Chapter 1 have similar execution profiles. Most of these applications were written by highly experienced APL programmers and hence are representative of excellent APL coding style. The distribution of array sizes executed during the experiment is summarized in Figure 2.4. Note that, across a wide variety of APL applications, 76% of *all* operations occur on arrays with fewer than eight elements, and about half of all operations are performed on zero- or one-element arrays. The last line of Figure 2.4 summarizes a similar study of 1,917 APL sessions

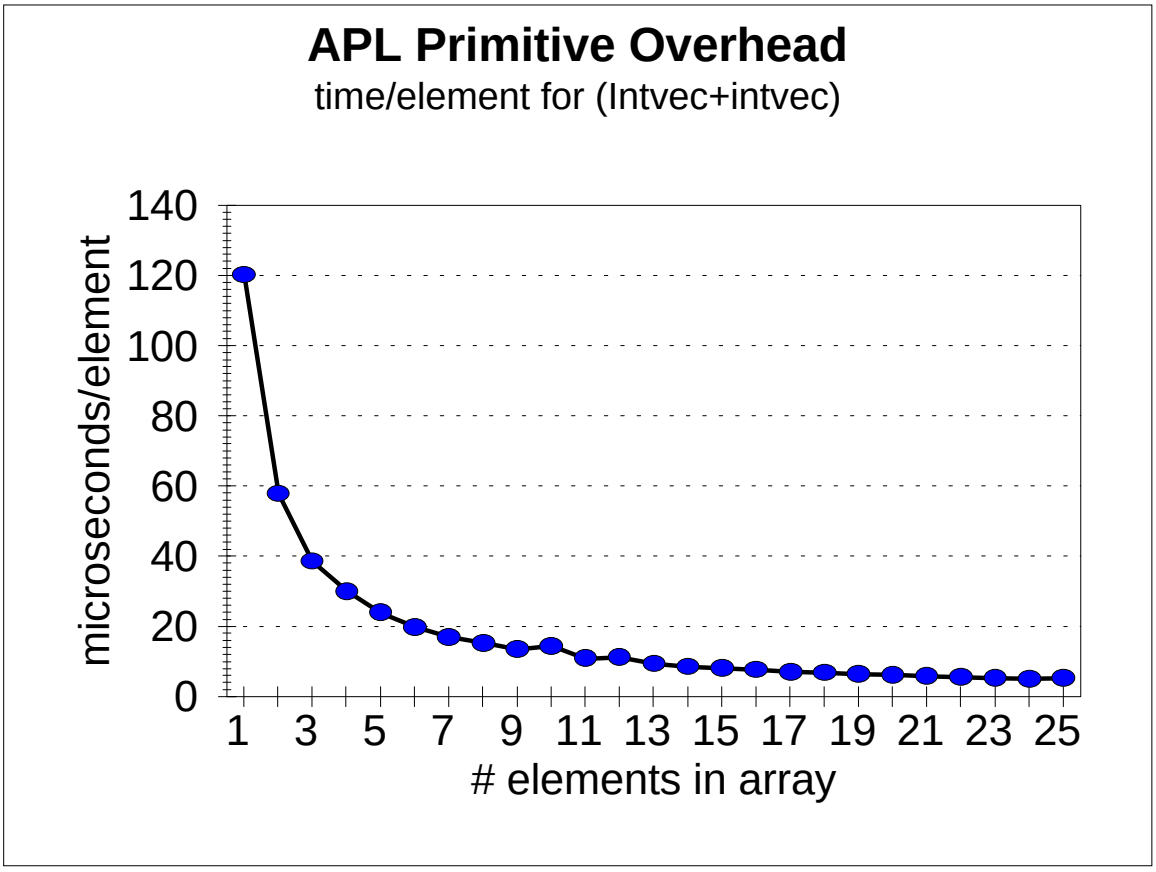

Figure 2.3: Per-element cost of APL array operations

performed by Jordan [Jor79].

Since these figures include measurements made of applications, such as text editors, that are thought of as atypical for APL, an argument might be made that they skewed the data toward small arrays. This argument can be refuted by appeal to a detailed study of APL array size distribution by Willhoft, who collected data for three large APL applications: a data base verification system, an interactive education catalog and enrollment system, and an engineering/business graphics system. His results are highly consistent with ours and those of Jordan: one-element arrays account for 45%, 53%, and 56%, respectively, of all primitive operations [Wil91].

Thus, setup cost plays a large role in even the most well-written APL application. We have seen that half of the cost of a typical interpreted APL application is spent in setup. This suggests that elimination of setup cost would produce at most a factor of two improvement in execution time. However, as we shall see later, removal of setup is only the first step. It is often possible to obtain two or more orders of magnitude improvement in performance via compilation.

% Primitive operation distribution by argument size										
Left size	Right size									
	0	1	2-3	4-7	8-15	16-31	32-63	64-127	128-255	>255
0	1	1								
1	1	35	7	4	3	2	1	1	1	1
2-3		1	2							1
4-7		1		1						
8-15		1			1					
16-31		1								
32-63										
64-127										
128-255		1							1	
>255		1							1	
monadic	3	9	3	3	2	1	1	1	1	3
totals (rounded)	5	51	12	8	6	3	2	2	4	5
cumulative totals	5	56	68	76	82	85	87	89	93	98
Jordan totals	8	55	67	80	88	92	96	97	99	100

Figure 2.4: Percentages of primitive operations by argument sizes

2.1.2 Array-Valued Intermediate Results

Another major source of interpreter run-time overhead is the creation of array-valued intermediate results. Naive APL interpreters execute one primitive or derived function to completion at a time. This mode of execution creates array-valued intermediate results, or *temps*, which are relatively rare in scalar-oriented, non-functional programs. Temps have potentially high memory and processor costs.

To see why this is so, consider the simple task of summing the first `N` integers. In a scalar language such as FORTRAN, this would be done by an iterative construct that would create a succession of integers and add them to a running total. For example:

```
      total = 0
      do 666 i=1,N
666   total = total + i
```

This approach requires memory for only two integers, regardless of the value of `N`; a compiler would probably allocate those integers to registers, resulting in increased performance. Contrast this with an equivalent array-oriented APL expression: `+/⍳N`. First, the *index generator* function, `⍳N`, creates a list of the first `N` integers, then the plus reduction, `+/`, sums the elements of that list. The list created by *index generator* produces a temp which requires `N` integers of memory.[1] Hence, a computation that requires a fixed, small amount of memory when expressed as a loop has turned into one that requires a variable, potentially very large, amount of memory when expressed as array operations.

[1]Some APL interpreters, such as Siemens APL/4004, provide support for *arithmetic progression vectors* (APVs) to circumvent this particular problem [Ive73]. APVs are a primitive data type that represents an arithmetic progression as `start+stride×⍳count`. APVs do not, however, solve the general problem of array-valued temps.

Furthermore, since the list cannot be allocated to registers, it must be placed in memory. This can produce a significant slowdown in today's computers, due to the disparity between processor and memory speeds. The difference in performance can exceed an order of magnitude. For example, a prime number generator (**primes**) we used as an early APEX benchmark exhibited poor performance due to a compiler bug. The bug inhibited loop fusion, thereby causing excessive memory usage due to creation and use of temps. When the bug was fixed, we observed a 67-fold improvement in performance of that benchmark.

Two further examples of array-valued temp generation in typical APL code are taken from the benchmarks we present in Chapter 5. Consider the computation of the pivot row in the matrix inverse benchmark, **mdiv**, and the location of silent T's in text (as in "witch") in the **metaphon** benchmark. The pivot row of the matrix `A` used in the `i`th iteration of **mdiv** is computed by the expression shown on the first line below:

```
i+(|i↓A[;i])⍳⌈/|i↓A[;i]
 ^ ^ ^ ^     ^ ^^ ^ ^
```

Execution of this expression creates nine temps, marked by the carets in the second row. Of these, six are vectors, three are integer scalars. Similarly, in the **metaphon** benchmark, silent T's are marked using the expression:

```
(omega='T')∧(1⌽='C')∧2⌽='H'
       ^     ^   ^^     ^ ^^
```

Naive execution of this expression creates seven Boolean temps of the same size as the argument array. In both of these examples, rewriting the code using scalar-oriented loops would eliminate *all* array-valued temps, albeit at a considerable cost in code clarity and maintainability.

Thus, we see that the temps associated with naive execution of APL can significantly increase the processor time and memory requirements of APL applications. When this factor is combined with setup cost, it becomes very clear why APL interpreters have a severe performance handicap compared to compiled languages. With this understanding of the performance problems, let us now see how compiled APL could solve some of those problems.

2.2 Reducing Interpreter Costs via Compilation

In order to see how compiled APL can out-perform interpreted APL, it is natural to look first at the areas of high interpreter overhead. Figure 2.1 highlighted the considerable setup cost that interpreters encounter in performing syntax analysis, conformability checking, and memory management. First, we shall examine ways in which those overheads could be reduced by compilation. Then, we shall present methods by which an APL compiler can improve on the performance of the execute phase of primitive functions.

2.2.1 Reducing Cost of Syntax Analysis and Conformability Checking

The first, and most obvious, part of setup cost that compilation can deal with is that of syntax analysis. As mentioned earlier, syntax analysis is an ongoing activity in an APL interpreter, brought about by

the possibility of the environment changing underfoot. For example, an identifier may change from a variable to a function; a function argument may be a Boolean scalar for one call, but an integer matrix for the next; a user may interrupt execution in mid-line, erase or create objects, then restart execution. Given this state of affairs, it may seem that an APL language processor has no choice but to repeatedly reexamine its environment. However, the reality of the situation is quite different – most production applications are well-behaved and rarely exhibit the above exotic forms of behavior. If we accept a prohibition on disrupted execution, then we can exploit the above characteristic of good behavior to avoid repetitive syntax analysis. Such a prohibition is considered a natural aspect of compiled code, but it does require that the APL user agree to abandon the flexibility of interaction.

Since we have now established a *terra firma* of names and their meanings, we can then perform syntax analysis on the application once and be assured that the meaning assigned to each token encountered in the analysis will remain unchanged. Therefore, run-time syntax analysis can be eliminated, producing a significant speedup in the performance of APL applications.[2]

Having eliminated syntax analysis from the run-time picture, we turn to conformability checking, the largest single overhead in many APL applications. Conformability checking comprises activities including, but not necessarily limited to:

- Type checking – in `x+y`, are both arguments numeric?
- Rank checking – in `x+y`, do both arguments have the same number of axes? If not, is one argument a scalar or singleton?[3]
- Shape checking – in `x+y`, do both arguments contain the same number of elements along each axis?
- Determination of compute type – depending on the argument types and the primitive function being executed, the data type used to perform the actual computation will vary. For example, in `x+y`, if `x` is integer and `y` is Boolean, the compute type will be integer. In the power function `x*y`, the compute type will be floating unless both arguments are Boolean.
- Determination of result type – the result type is frequently the same as the compute type, but there are exceptions. For example, in `x=y`, the result type will be Boolean, although the compute type could be Boolean, integer, floating, or character, depending on the types of `x` and `y`.
- Determination of left and right fetch type – Depending on the compute type, it may be necessary to coerce the left or right argument from its storage type to the compute type. For example, in `x+y`, if `x` is Boolean and `y` is integer, then the fetch type of `x` will indicate Boolean-to-integer conversion, and the fetch type of `y` will indicate the identity, integer-to-integer conversion.
- Determination of execution function – there may be several specific functions associated with a generic primitive function, depending on the compute type required. For example, `x+y` will use different code for addition if the compute type is integer than if the compute type is floating.

[2] ACORN was a simple research APL compiler that did little more than eliminate run-time syntax analysis, yet scalar loops compiled under ACORN produced a five-fold performance improvement over interpreted APL [BBJM90].

[3] A singleton is an any array that contains only one element, *e.g.*, a scalar, one-element vector, one-by-one matrix. A singleton, `X`, is any array that satisfies the predicate `1=×/⍴X`.

These conformability checking tasks, although fairly straightforward, are responsible for a considerable amount of interpreter overhead. This overhead can be largely eliminated by analyzing the well-behaved nature of production APL applications through the use of data flow analysis, as will be discussed in Chapter 3. Data flow analysis, particularly when combined with the renaming of variables through static single assignment analysis, permits a compiler to make decisions about the properties of each array created during the execution of an APL application. Therefore, the work done in conformability checking can be done once, at compile time, except for shape checking, some of which usually must be deferred until run-time.

The specific form of data flow analysis devoted to determination of array properties is denoted as *array morphology*, to emphasize the fact that we are interested in the structure and properties of arrays rather than scalars [Ber93]. In APEX, array morphology is required to determine the rank and type of arrays, but it is also used to deduce other array properties, including shape, value, and predicates such as *ArrayIsSortedUp*, *ArrayIsSortedDown*, and *ArrayIsPermutationVector*. These properties and predicates often enable a code generator to generate code that exploits superior algorithms for special cases of certain functions. For example, if an array is known to be a permutation vector, then it can be sorted using a linear time pigeon-hole algorithm; binary search can be used with set membership if one argument is known to be sorted; upgrade of an already-sorted array can be done in linear time, because the result can be generated without any examination of the data elements of the argument array.

The only remaining part of conformability checking, shape checking, could be conditionally removed for operations where morphological analysis at compile time was able to show that two arrays must always be conformable. APEX does not yet fully implement this particular optimization. Nonetheless, even the relatively simple analysis performed by APEX permits the vast majority of conformability checks to be eliminated from the run-time picture.

We conjecture that performing morphological analysis in a naive interpreted environment is not worthwhile, as it is a very time-consuming operation. Its use in an interpreted environment is precluded by its detrimental effect on setup time. It would, however, be interesting to explore a hybrid compiler-interpreter that performed such analysis in the background or during a user's think time. Such an approach might be able to combine the interactive flexibility of an interpreter with the high performance of compiled code.

2.2.2 Reducing Cost of Memory Management

Having thus dealt with the overheads of syntax analysis and conformability checking, it is time to turn to the next major performance bottleneck of interpreted APL – memory management. Array memory management is a challenge for APL and other functional languages. In functional languages, arrays are not statically named objects to be allocated in memory and modified *in situ*, as they are in FORTRAN. Rather, arrays are the ephemeral progeny of functions, created and destroyed dynamically as execution proceeds. This process of creation and destruction of array-valued temps hurts performance. If we could but escape this cycle of continual creation and destruction of large intermediate results, considerable

run-time performance improvements would accrue. Luckily, in functional programming, unlike in real life, there are ways to get off the wheel of birth and death. Some of the methods by which a system can reduce the performance impact of array-valued temps are loop fusion, copy avoidance, reduced reference counting, and special treatment of scalars. Let us examine these methods in turn, to see how memory management costs can be reduced.

In programming, the fastest code is the code that is not there: the best way to make a piece of code run faster is to eliminate it from the program. Similarly, the best way to handle the performance loss engendered by large temps is to avoid generating them. In APL, it is quite common for one function to generate its result *in toto* as a temporary array in one function, then to feed that temporary result to another function as one of its arguments. This materialization of an entire array increases memory requirements and slows execution speed, due to the need to access main memory for storing and fetching the temporary array elements. Consider, instead, a *lazy* method of function evaluation, which postpones generation of results until they are needed by another function [Abr70, Bud88]. Lazy evaluation often eliminates the need to generate array-valued temps entirely, thereby saving execution time and memory space. For example, in the `+/⍳N` example, a lazy evaluator would have the reduction (`+/`) generate repeated requests for the *index generator* function (`⍳N`) to produce the next element of its result. As these were generated, the reduction would consume them by adding them to its total. Thus, only one element of the list of integers would exist any point in time. In addition, the processor time penalty imposed by memory access times could also reduced, because the few array elements in existence at any given point in time could often be allocated to registers or would be very likely to reside in cache memory, permitting rapid access.

Lazy evaluation can be implemented by having the compiler perform *loop fusion* to merge the control flow structures associated with similar computations [Abr70]. For example, if `x`, `y`, and `z` are vectors, the expression `x+y×z` might be naively written as:

```
       do 666 i=1,shape(y)
666     temp1(i) = y(i)*z(i)
       do 777 j=1,shape(y)
777     temp2(j) = x(j)+temp1(j)
```

The above code, which reflects the way that most APL interpreters would actually execute the expression, generates an array-valued temp, `temp1`. The code includes two instances of control flow overhead for loop counter increment, limit testing, and conditional branching.

Modern compilers recognize loops such as the above and combine, or fuse, them. Hence, the name *loop fusion*. The result of applying loop fusion to the APL code fragment produces:

```
       do 666 i=1,shape(y)
        temp1(i) = y(i)*z(i)
666     temp2(i) = x(i)+temp1(i)
```

Loop fusion has removed one set of control flow overheads from the execution.

Further optimizations replace the use of `temp1(i)` with a scalar temp, to produce:

```
       do 666 i=1,shape(y)
        temp3 = y(i)*z(i)
666     temp2(i) = x(i)+temp3
```

This optimization has replaced an array-valued temp, `temp1`, by a scalar temp, `temp3`. This change may significantly reduce the memory cost of the computation; both of the changes described above reduce execution time.

As noted earlier in this chapter, the use of loop fusion to remove array-valued temps can provide a performance boost that is in excess of an order of magnitude. Thus, loop fusion is a quite sensible and powerful feature to include in an array language compiler, but it is difficult to introduce into interpreters except in very special circumstances. The problems of environmental shift and the requirement to perform conformability checking across a number of primitives makes the task complicated and, moreover, increases setup cost.

Two other techniques for reducing the cost of memory management are elimination of array copying operations, and elimination or reduction of reference count operations. Computer languages usually provide a way to replace elements of an existing array with new values. In a functional language, the semantics of such a *replace* operation usually specifies generation of a new array – a copy of the original array – with certain elements replaced by new values. Copying arrays during *replace* operations is very time-consuming when arrays are large. Therefore, most APL interpreters attempt to eliminate the copy operation when there are no other active referents to the argument array, so they can efficiently perform the *replace* operation in place. This is typically done by associating a reference count with each array when it is created, and maintaining the reference count as primitives refer to, or cease to refer to, the array. A *replace* operation that intends to modify an array will perform the operation in place if the array's reference count indicates that there are no other referents to the array. Otherwise, a copy must be made. Reference counting has other benefits, such as eliminating the need to copy arguments to functions. These optimizations tend to provide a system-wide improvement in interpreter performance, but they do exact a price in increased setup cost. If the benefits of copy avoidance could be achieved without having to maintain reference counts for all array operations, then overall system performance would improve. The analysis required to perform this is fairly extensive, but is quite tolerable in a compiled environment.

As noted earlier, much of the work in APL applications is performed on scalars. Most APL systems represent scalars as rank-0 arrays, with concomitant overhead to maintain their descriptors. If scalars were special-cased, instead of being treated as merely another array, their maintenance would be simpler since no descriptor maintenance would be required. However, special cases are the hobgoblins of interpreters. Determination and filtering of special cases increases setup cost and drives up interpreter size, even when the probability of being able to exploit the special case is low. The interpreter designer therefore walks a tightrope between increased setup cost and the ability to exploit a highly beneficial, but rarely encountered, special case. A compiler, by contrast, can detect scalars at compile time. This permits the compiler to generate high-performance scalar code without adversely impacting the perfor-

mance of the rest of the system.

2.2.3 Reducing Cost of Execution

Although the above picture of setup cost may make it seem otherwise, every APL function does eventually get around to actually operating on an array to do real work – *e.g.*, adding the elements of arrays. This facet of operation, denoted the *execute phase*, is where APL interpreters can shine – for those classes of applications where a few APL primitives dominate the computational load. Array primitives offer the interpreter designer excellent high-level semantic information about the operation to be performed. There is usually much more information available to the interpreter than a scalar-oriented compiler would ever have available. For example, the interpreter will know that it is performing a specific form of matrix product on a Boolean matrix and a rank-3 integer tensor. This gives the interpreter the ability to exploit a high-performance, special-case algorithm to perform the operation. By way of contrast, a scalar-oriented compiler would only observe several nested loops combined with IF statements. Thus, an array-oriented compiler or interpreter has a good grasp of the situation, whereas the scalar-oriented compiler sees only a fog of detail obscuring the big picture of the operation to be performed. Extensible languages with overloading, such as C++, may be able to obtain a view of the situation similar to that of APL, but it is not clear whether they are able to efficiently handle other aspects of abstraction, such as function composition. In any event, APL, whether interpreted or compiled, has a good view of the computation in the large, and thus has opportunities for exploiting improved run-time algorithms.

Within an interpreter, increased setup cost limits the performance gains that can be obtained by exploiting APL's larger view of the computation. However, an APL compiler can extract most of the potential benefit with no increase in setup cost. To see how this can be done, we will look at determination of special-case functions, phrase recognition, and array coordinate mapping.

Once a compiler or interpreter knows the type and rank of a function's arguments, it has most of the information it needs to efficiently perform the execute phase of the function. In an interpreter, a relatively generic choice of execution function is made during setup. Special cases, however, are function-specific, and their detection is deferred to the interpreter execute phase. For example, the APL table lookup function, *indexof*, is frequently used to search for a single value, rather than a whole array of values. In such a case, a sophisticated search algorithm usually does not pay off; a simple linear search is most effective [Ber73]. To choose the appropriate algorithm, an interpreter must always, therefore, check the array sizes involved in the operation. A compiler, by contrast, knows at compile time that one argument is scalar and will generate optimal code for that case without requiring a run-time check. Thus, a compiler can obtain the benefit of special-case algorithm selection without the burden of increased run-time cost to detect its presence.

Compilers and interpreters can also generate more efficient code by performing *phrase recognition* or idiom recognition [Per79]. Idiomatic expressions often appear in APL programs in place of calls to library routines. For example, the phrase ⍴⍴x gives the rank – the number of axes – of the array

x; the expression ((⍳⍴x)=x⍳x)/x produces the set of unique elements of the vector x ; the phrase (∨\x≠' ')/x removes leading blanks from the text vector x. The ability to recognize phrases offers an opportunity to generate faster or more memory-efficient code to implement them. Phrase recognition is done to a limited extent in most interpreters [Ive73, Bro85], but it is usually restricted to a few simple, common expressions, because of the cost of pattern matching and expression validation within an interpreter. A compiler, by contrast, could detect a large set of phrases with no loss of run-time performance.

Array coordinate mapping is a third method that compilers and interpreters can use to produce more efficient code. Pioneered by Abrams and later extended by Guibas and Wyatt and others [Abr70, GW78, Bud88, Mul88, Mul90], array coordinate mapping offers a way to eliminate the per-element execution time associated with many APL structural functions. Structural functions, such as ravel, reversal, transpose, take, drop, and some cases of reshape, change the array structure or the ordering of array elements without changing the element values. When naively implemented in a form that involves element-by-element copying of the argument, their execution time is proportional to the number of elements in the result array. Array coordinate mapping, by contrast, converts these operations on array elements into manipulations of their array descriptors. This permits functions that would otherwise require time and space proportional to their result size to operate in a small, fixed amount of time and space. Array coordinate mapping has been implemented in a few APL compilers and interpreters [Bud88, Bro85]. Within an interpreter, array coordinate mapping increases setup cost, because all functions must be able to distinguish normal arguments – those given as a basic set of descriptors – from those given as a function of descriptors. Since a compiler would statically deduce whether a function's argument was normal or mapped, no increased setup cost would be incurred for compiled code. Thus, a compiler can obtain all of the benefits of array coordinate mapping with none of the costs.

We now see that compilers are better able than interpreters to exploit APL's larger view of computation, by not having to withstand increased setup cost. Moreover, compiled code enjoys an added benefit – synergy – that is not possible to achieve in an interpreted environment. Synergy results when optimizations work in concert to produce performance levels that exceed the sum of the optimizations working independently. To see how synergy arises in this context, we will look at the impact of one very important aspect of compiled APL: function inlining.

Many scalar language compilers routinely inline functions, replacing their invocations with a suitably modified copy of the text of the function itself. The immediate and obvious benefit of inlining, particularly for leaf-level functions, is to remove the cost of function call and exit from each such invocation and to provide a potential improvement in the cache performance of instruction fetch. In addition, inlining offers additional performance benefits through synergy with local and global optimizations.

In an APL compiler, these optimizations work across the boundaries of APL primitives, offering levels of performance that are difficult or impossible to obtain in a purely interpretive environment in which optimizations are limited to a single primitive. For example, function inlining tends to increase the size of basic blocks, giving register allocators and RISC code schedulers the opportunity to do a

better job than they could in the absence of inlining. Inlining also exposes other possible optimizations, including common subexpression elimination, partial evaluation, and in-place updating of arrays. Most importantly, function inlining enables loop fusion to take place across APL primitives. OSC inlining merges, into a single block, the APEX-generated SISAL code that implements several APL primitives and defined functions. This action enables OSC's loop fusion optimizations to reduce loop control overhead. It also facilitates the complete removal of many array-valued temps. Inlining thereby provides performance improvements that are all out of proportion with what we expect from inlining and loop fusion working independently of one another. This particular synergy is critical in obtaining APL performance that is competitive with computer-oriented languages.

Interpreters, by contrast, have several difficulties in attempting to exploit these optimizations in isolation, let alone in concert. Inlining is contrary to the nature of a pure interpreter, as it involves dynamic code generation. Those interpreters that do achieve some of the effect of inlining by dynamic code generation are either unable to perform loop fusion or perform loop fusion only in restricted circumstances when vector hardware assists are available [MM89]. Without loop fusion, the potential benefit of code scheduling is substantially reduced. We are not aware of any APL interpreter that exploits common subexpression elimination. This situation is probably due to the complexity of analysis and increased setup cost entailed by run-time validation of the subexpression. Interpreted applications, therefore, are not able to take advantage of the synergy available to compiled applications and, therefore, suffer concomitant performance loss.

2.3 Language Characteristics

APL has a number of characteristics that set it apart from most popular computer languages. These characteristics include array operations, higher-order functions, absence of declarations, nameclass changes, dynamic scoping, and little need for explicit loops. Some of these ease the task of compiling APL; others complicate or frustrate it. Let us look at both sets of these characteristics in turn and see how they impact the compilation of APL.

2.3.1 Language Characteristics that Benefit Compilation

Among the characteristics of APL that simplify the task of compiling or interpreting APL are array operations, higher-order functions, and little need for explicit loops. They also often enable interpreters or compilers to execute complex primitive or derived functions, such as *upgrade* or *indexof*, considerably faster than the code that most programmers would tend to write for serial languages.

Array operations are frequently viewed as syntactic sugar, merely offering the user convenience of expression. Array operations, in fact, make the language processor's task of high-performance code generation easier in several ways. First, by subsuming the decision of elemental evaluation order, the language processor is able to maximize the effective use of target system resources such as cache and distributed memory on a multiprocessor. A programmer working at the serial language level, by con-

trast, might ignore or misuse these resources, *e.g.*, permuting loop order and thereby making poor use of cache. Second, APL's array operations eliminate most side effects and loop-carried dependencies. This simplifies the compiler's job and facilitates extraction of parallelism. Third, array operations can easily be performed in parallel on multiprocessors. In addition, several elemental operations can be performed in parallel, even on uniprocessors. For instance, if Boolean data is stored as one element per bit, the Boolean functions can be performed a word at a time, thereby producing a considerable parallel speedup with no changes to application code. Our application of this technique to the class of Boolean inner products, such as $\vee.\wedge$, typically used for computation of transitive closure in Boolean graphs, resulted in a thousand-fold speedup in the speed of these derived functions in SHARP APL. Clearly, such optimizations can be exploited by compilers as well as by interpreters.

The higher-order functions of APL describe families of commonly used operations including scans, reductions, and inner products. As with the array operations, these operations often can be performed with a level of efficiency achievable by few programmers, just as the BLAS (Basic Linear Algebra Subroutines) obtain high performance by paying careful attention to algorithms and to cache and processor characteristics. The higher order functions of APL are also usually implemented using extremely sophisticated algorithms that are similar in spirit to the BLAS, but APL's algorithms are more general because of the nature of APL's higher-order functions. For example, APL performs all matrix products efficiently, not just the $+.\times$ of the BLAS. Few serial language programmers would take the time to directly write algorithms of the complexity of BLAS into applications, for reasons of maintainability or development schedule. Yet, that level of performance is immediately available to the user of interpreted or compiled APL, because of the power of higher-order functions.

For higher-order functions, a compiler has a slight performance edge over an interpreter, because algorithm selection is made statically, rather than at run-time. Given sufficient predicates from array morphology, a compiler would gain a significant edge. For example, if one argument to matrix search (a variant on inner product represented as $\wedge.=$) was known to be sorted, a compiler could generate highly efficient binary search code instead of the more expensive, general case of the code for inner product.

The dearth of explicit loops in most APL programs offers a slight compile-time advantage over other languages – APL programs have a small number of basic blocks. This speeds up data flow analysis and other compilation tasks. Other improvements derive from the minuscule size of APL programs compared to those written in scalar-oriented languages. We have not attempted to measure the magnitude of these differences, as they are probably not significant in most cases.

2.3.2 Language Characteristics that Hinder Compilation

Just as APL has certain characteristics that enhance compilation, it has others that hinder the work of a compiler. These include the absence of variable declarations, imprecise conformability rules, dynamic changes of array type, dynamic scoping and overloading of functions, nameclass changes, and absence of an ISO APL Standard for flow control structures. We will briefly discuss each of these characteristics and their impact on the compilation process.

The APL language does not include declarations for variables; functions are declared by virtue of their presence in the workspace. APL interpreters and some compilers address this problem by attaching type and rank information to each array created during execution and propagating it as execution proceeds. Since this approach increases the work required to execute each primitive and reduces opportunities for inter-primitive optimizations, we shall not discuss it further. A better way to handle the absence of declarations is to let the compiler deduce the type and rank of each array created by the program. This method permits generation of highly efficient code [Ber93, CNS89, Saa78].

The deductive approach works quite well for realistic applications, except for the arguments to the "main" functions, for which we have no way to deduce types and ranks. To see why these arguments present a problem, consider a trivial program consisting of a single function, such as `⍳N`. It is not possible to deduce the type of `N`. It could be Boolean, integer, or floating. It could be *any* rank, as long as it is a one-element array. We know the result is an integer vector, but that is no help to us in tracing backwards in the data flow graph.[4] Thus, there is no way to deduce the type and rank of certain function arguments without assistance from a programmer.

A second problem is that the type and rank of a variable can change from point to point in a program. In the contrived expression `X×(X←0.3 0.4 0.5)+X+X←2`, the variable `X` is first an integer scalar, then a floating vector.

Finally, an array resulting from a single expression may vary in type from call to call, due to type promotion. For example, the power function `N⋆5` may return a Boolean (if `N` is a Boolean), an integer (if `N` is a small positive integer), or floating (if `N` is floating or large enough to cause integer overflow). This presents a problem to a compiler that must determine the type of each array at compile time.

A compiler must, therefore, deal with at least three distinct problems arising from the absence of variable declarations in APL – type and rank deduction, the same name being used for different objects, and type promotion.

The dynamic scoping rules of APL present another challenge to compiler writers. ISO Standard APL is defined to use dynamic scoping, just as early dialects of LISP did. Dynamic scoping allows a function's globals to change type or rank depending on the call site of the function. Similarly, globals that are set by that function may change their type and rank, thereby affecting all later uses of those globals by the application. Dynamic scoping makes it impossible to perform data flow analysis on a single function in isolation. Instead, the entire application must be analyzed as a whole using interprocedural data flow analysis.

A related problem in analyzing how arrays are used in APL arises from APL's ability to dynamically manipulate the set of names – functions and variables – in the workspace. Functions can be created and expunged on the fly; character strings created at run-time can be executed by the interpreter as if they were APL expressions embedded in the program text. A programmer or user can interrupt execution of a running application, create, expunge, or edit any function or variable in the workspace, then continue

[4] This problem arises, in part, to the imprecise conformability rules that were adopted for APL, in the name of *programmer convenience*. The folly of programmer convenience has since been recognized by APL language designers and imprecise conformability rules have been rejected in more recent languages, such as J, that are derivatives of APL.

execution of the application. This leads to a potential for continually shifting sands in a desert of names and meaning. Since compilers depend on a certain stability in names, they will have problems handling these aspects of APL.

A final area of difficulty in compiling APL is that of control flow. APL, unlike most other modern languages, has no standard for flow control structures; a conditional *goto* is its only method of controlling flow. This complicates the job of computing a control flow graph for an application, a necessary task if the application is to be compiled. APL's *goto* presents several unique problems to the compiler writer. The ultimate argument to a *goto* is a line number, not a line label. This permits users to create such perverse, but ISO Standard-conforming, code as that which implements a CASE statement by branching to the sum of a line label and an integer index. Second, the definition of conditional branches complicates matters, as a *goto* is defined as not-taken if the argument to the *goto* is an empty vector. Although several vendors have released APL interpreters that support structured flow control mechanisms, other vendors have not yet taken any action in this area. Thus, application writers who wish to write portable code are constrained to use APL's *goto* construct, thereby sacrificing code maintainability and a substantial amount of performance [Ber93]. Clearly, *goto* represents a considerable challenge to both the application programmer and to the APL compiler writer.

2.4 Related work

Projects to improve the productivity of programmers and computers have been with us since the first computer was invented. These efforts, all ultimately aimed at reducing time-to-solution, involve enhancing programming languages to provide more general and concise methods of expressing algorithms, providing compilers to map those languages into a form directly executable by computers, and using sophisticated compiler optimization techniques to generate more efficient code. We will now summarize some of these research results, where they are germane to this thesis.

2.4.1 Methods for Reducing Time-To-Solution

Time-to-solution can be dealt with by reducing the time it takes to write an application or by reducing the execution time of the application. Not surprisingly, the APL and FORTRAN communities are adopting concepts and methods from each other to achieve this end.

FORTRAN is characterized by poor development time and excellent run-time performance. The Fortran 90 and HPF communities have undertaken to support parallelism and to streamline application development through adoption of APL-like features, including array and structural operations and a few reductions. Although this approach would seem to offer the performance of a compiled language combined with the flexibility and expressiveness of APL, it fails to achieve that end [Ber91]. First of all, many facilities taken from APL were not treated in a general manner. For example, ADJUSTL, akin to APL's rotate primitive, is defined for character vectors only – it will not operate on numeric data nor on matrices, limiting the utility of this added facility. Secondly, the absence of unifying design principles

inhibits the creativity and expressiveness of the programmer. The ability to compose functional expressions and to derive new functions in a simple, concise, and consistent manner is the key to idiomatic expression and rapid application development in APL, yet this capability was not added to Fortran 90. Finally, the fundamental problem with moving a serial language into a parallel world is that it is difficult to make substantial changes to a language once it has an installed user base. HPF and Fortran 90 retain their scalar heritage; they do not lend themselves to the type of expression that is essential to insight and rapid application development. Thus, although these languages do offer shorter development time than their ancestors, their scalar lineage suggests that they will remain unable to compete with thought-oriented languages in the application development phase of time-to-solution.

In contrast to FORTRAN, APL offers excellent development time and poor execution time. APL interpreter designers, seeing the poor performance of APL on many applications relative to compiled code, recognized that, if APL run-time performance could be increased to the point where APL was competitive with FORTRAN and C, time-to-solution would be greatly decreased. However, we are not aware of any projects to improve interpreter performance that have achieved this goal. As an example of one such approach, interpreters that use semi-static parsing methods to reduce syntax analysis overhead provide about a twofold performance improvement, yet they have proved to be fragile and difficult to debug and maintain.[5] Interpretive systems that perform even the simplest array analysis in order to exploit algebraic identities or known properties of arrays do not run substantially faster than existing naive interpreters [Ive73]. These disappointing levels of performance arise from the considerable amount of run-time bookkeeping required by these methods. The increased setup cost incurred by keeping track of possible optimizations usually outweighs any performance benefit gained by their exploitation. Our own research into improving the performance of an existing interpreter, by reducing instruction path lengths for primitive function setup, suggests that production application performance could be improved by 13 – 20%. Although this is quite significant for certain large mainframe applications, it is not in the same ball park as the performance improvement we expect from compiled code.

Interpreter performance is also affected by the inability to exploit powerful compiler optimizations. Only in rare cases do interpreters employ optimization techniques such as loop fusion and common subexpression elimination. Absence of loop fusion introduces additional temporary arrays and radically increases memory subsystem traffic. IBM's mainframe APL2 interpreter does support loop fusion, but only under special coding circumstances and then only with optional vector processing hardware. Even then, the performance gain at the kernel level is usually limited to a factor of two to three [MM89]. The dynamic code dispatch techniques used in most interpreters limit the ability to exploit low-level optimizations such as code scheduling and register assignment on RISC machines. Given the unresolved performance problems of current APL interpreters, we can safely assume that dramatically improved

[5] In the course of running benchmarks for this thesis, we discovered such a performance bug in one such commercial APL interpreter. The interpreter used a semi-static parsing scheme to reduce run-time syntax analysis, rescanning a line when it detected that the environment had changed in a way that might invalidate the previous syntax state. Unfortunately, a bug had been introduced into the interpreter, causing syntax analysis to be performed on every execution of a line containing a comment. This resulted in a factor of 25 performance degradation for iterative benchmarks such as **shuffle**. We informed the vendor of the problem and the bug was fixed. The benchmark results presented here use a corrected version of the interpreter.

interpreter performance is not likely to materialize in the near future, and we must turn to compilation as the path to high performance.

2.4.2 APL Compilers

Most compiled APL projects have gotten off to a good start, but have ended up as rusting hulks on the banks of the river of APL history. To date, no commercially successful APL compiler has appeared on the market, and no APL interpreter has ever been able to compete with compiled scalar languages on iterative applications. Nonetheless, a number of valuable ideas that have emerged from efforts to improve the performance of APL are worthy of mention, as we have adopted some of them in APEX.

Several authors, including Driscoll and Orth, Saal, Weigang, and Wiedmann, have written comprehensive summaries of the problems of compiling APL and possible benefits arising from its compilation [JO86, Saa78, Wei85, Wie85, Wie79]. These outline the history of APL performance problems and attempts to cope with them.

The history of APL compilation consists of two major streams. One stream is that of true compilers whose output is another computer language, assembly or machine code, or source code in FORTRAN, C, or SISAL. The other stream is that of dynamic compilation, in which an interpreter generates executable code in the process of running an APL program. The latter might immediately discard the executable code after use or preserve it for reuse. The second stream, although of considerable interest and coincidentally that which the author has had most experience with, is beyond the scope of this thesis, because of the difficulties of dynamic code generation without the assistance of a compiler. Consequently, we shall not discuss it further. The first stream, that of language translators for APL, is our major interest here.

Several compilers for APL or APL-like languages have been constructed over the past decade, mostly as research projects. These included Wiedmann's APL*PLUS compiler for the S/360, Ching's APL/370 and COMPC compilers, Driscoll and Orth's Yorktown APL Translator (YAT), Budd's APLe compiler, Bernecky and Brenner's ACORN, and Sofremi's AGL compiler. To date, none of those have been commercially successful, in the sense of being widely used in the APL community, although Wiedmann's compiler achieved enough customer use that we might properly call it a product. These set the stage for our work.

Early work in compile-time conformability checking is discussed by Bauer and Weiss [BS74]. Weiss and Saal used similar techniques to make a static determination of the syntax class of APL objects [WS81]. Their approach, which formed the basis for data flow analysis methods used in several APL compilers, uses *def-use* analysis to determine the syntactic class of each named object in an APL application. This differs from our method in that they did not make any attempt to determine the array morphology of the program. That is, their analysis ceased at the point of having found out which names are niladic, monadic, or dyadic functions, and which names are variables. Our approach concentrates on determining the morphological information – primarily type, rank, and shape – associated with each function (primitive, derived, or defined) in the program.

One measure of the success of a research project is its eventual availability as a product. The earliest commercial offering of an APL compiler was that offered by STSC and designed by Wiedmann [Wie79, Wie83, Wie85]. The Wiedmann compiler translated a single APL function into S/360 machine code, which manifested itself as a locked function in a mainframe timesharing user's APL workspace. The compiler's major drawback was its limited applicability and excessive compile times, with compilations of relatively simple functions requiring overnight processing on a large mainframe.

Budd developed a compiler for APLe, an APL-like language. We say APL-like, because his compiler did not strictly reflect the semantics of APL [Bud88, Bud83]. For example, his static name scoping rules were quite different from those of APL and some primitives were implemented with quite different semantics from those of ISO APL. The array copying avoidance techniques proposed by Guibas and Wyatt were adopted by Budd in his APL to C compiler [Bud88]. These techniques were later extended by Treat and Budd [BT82, TB84], although the extensions appear to be of little practical value in today's machines, as they require execution of integer *modulus* operations in the inner loop of operand fetch. The extensions may yet prove useful if the performance of *modulus* can be improved or if current trends in processor speed versus memory access time continue. The compiler also used data flow analysis techniques similar to those used by us in APEX and by Ching in his compiler. Budd's compiler did not perform interprocedural data flow analysis, which may have caused its performance to suffer. Budd presents an algorithm for performing data flow analysis in APLe [Bud85]. He also presents a rationale for data flow analysis in an APL compiler, and discusses some of the issues associated with APL data flow analysis.

Ching developed a compiler for a subset of VS APL that directly generated S/370 machine code [Chi86a, Chi86b, CX87, CNS89]. The back end was later rewritten to emit C code [JC91]. Ching's compiler is also the only one other than APEX and ACORN that have produced code for a parallel computer. His compiler did not perform optimization beyond the level of a basic block in APL.

Driscoll and Orth, of IBM Research, developed YAT (Yorktown APL Translator), an APL to FORTRAN compiler [JO87, JO86]. This compiler was eventually licensed to Interprocess Systems, but its current product status is unknown. From the standpoint of performance, code optimization, and compiler sophistication, their compiler is probably superior to the others discussed in this section. It also provides inter-primitive optimizations which go beyond anything we have encountered and which have the potential to out-perform APEX-generated code for certain idioms. Driscoll and Orth also present a concise and insightful history of APL compilation [JO86]. Like APEX, YAT requires knowledge of the type and rank of all arrays created during the execution of a program.

Bernecky and Brenner designed and implemented ACORN, a research APL to C compiler targeted at large seismic applications on a CRAY X-MP [BBJM90, Ber90a, Ber90b]. ACORN, a joint project between I.P. Sharp Associates Limited and Mobil Research Development Corporation, was intended to study "the use of APL as a delivery vehicle for parallel computation." In a related project, Schwarz ported the compiler to the Thinking Machines Corporation CM-2 [Sch90]. The ACORN compiler performed fairly well for a compiler that was designed and written by two people in two months and did

no optimization whatsoever. Despite the absence of vectorized C versions of many library routines, the **mconvo** convolution benchmark performed within a factor of four of the CRAY production FORTRAN code.[6] Since the only data type supported by ACORN was double-real, the performance of array indexing code adversely impacted several benchmarks. The ACORN tokenizer, like that of APEX, was non-iterative, using APL SIMD operations controlled by a Boolean mask array to identify the tokens of an APL program. The ACORN compilation unit was a single APL function. Compilation of a complete application required repeated invocation of the compiler for each function to be compiled. However, the absence of a data flow analysis phase meant that compilation speed on a mainframe was essentially instantaneous, so this did not represent a serious problem.

Alfonseca describes a translator which compiles a subset of APL, defined to operate on integer vectors only, to System/7 assembler code [Alf73]. Alfonseca's introduction of control structures into his APL dialect established a behavioral pattern which is still with APL compiler writers today. Jenkins attributes part of the performance problem of APL versus other languages to the lack of control structures in APL [Jen73].

Sofremi, a French company, developed an APL to C compiler, but we are unaware of its current status [Gui87]. The author claims that "...compiled code is as efficient as standard FORTRAN code," but does not present any benchmark results. In our experience the Sofremi booth at two APL conferences, their product was unable to compile even relatively simple APL functions.

2.4.3 Compiler Optimizations in Compiled APL

The use of traditional compiler optimizations in APL compilers has often taken a back seat to the more fundamental problems of identifying array characteristics and dealing with control flow. Nonetheless, as an understanding of solutions to these problems was reached, compiler writers started to deal with the compilation issues that are common to all programming languages. Data flow analysis techniques have evolved from none to ad hoc to *def-use*. Now, with APEX, static single assignment methods are being used. Weiss and Saal exploited *def-use* chains as the basis for their analysis of syntax classes. The method we use in APEX, although similar in spirit, begins by transforming the compilation unit into static single assignment (SSA) form. This simplifies and speeds up analysis, in that each name has but a single definition point. It also improves the quality of analysis, in that SSA provides more precise information for optimization, in areas such as reaching definitions.

Loop fusion, often important in scalar languages, is crucial in obtaining excellent performance of compiled APL applications. This fact has been recognized by a number of those who have worked with APL. First suggested by Abrams, who uses the term *dragalong*, the role of loop fusion and other traditional optimizations in compiled APL is also discussed by Johnston, Saal, Weigang, Weiss, and Wiedmann [Abr70, Wei85, Wie85].

Like APEX, the YAT compiler supports a number of traditional compiler optimizations including

[6] We got around the lack of a vectorized square root library routine by writing an APL function implementing a Newton-Raphson square root function and using ACORN to compile that into vectorized form.

common subexpression elimination and constant folding. We presume that YAT, like APEX and unlike traditional language compilers, performs these optimizations on arrays as well as scalars, but have not been able to confirm this presumption.

Chapter 3

The Design of APEX

Now that we understand some of the ways that an APL compiler might alleviate many of the run-time performance problems associated with interpreted APL applications, we are in a position to learn how the design of an actual APL compiler – APEX – deals with the hindrances to compilation that were presented in Chapter 2. Once we know the solutions to these fundamental problems of compiling APL, we can focus our attention on the specific problem of compiling APL into SISAL.

Since APL was originally designed as a notation for teaching mathematics rather than as a compiled language, it presents a number of unique challenges to the compiler writer. Inference of array type and rank are fundamental problems, as is the ability to redefine names dynamically. Our strategy for dealing with these compilation problems began by classifying language features into one of four areas: mandatory, deferrable, rarely used, and prohibited. Our classification based on our familiarity with the behavior of large APL applications, generally corresponds to the static frequencies of APL operations observed by other researchers[Bin75].

Mandatory features are those, such as the scalar functions and array restructuring operations, that are required by nearly all APL applications. Without support for mandatory features, an APL compiler cannot be deemed to be operational.

Deferrable features are those that may be used in large applications, but that can usually be avoided in new code or small benchmarks by adoption of an appropriate coding style or by making small changes to existing applications. Deferrable features included those that we felt were required in a practical compiler, but that presented us with considerable difficulties in design, analysis, or code generation. Among these were explicitly iterative code using *goto*, overloading of defined functions, and support for globals, semi-globals, and side effects.

Rarely used features are those that are defined by the ISO APL Standard, but that are not often used in production application code. Examples of these include the hyperbolic arctangent variant of the *circle* primitive and *dyadic transpose* with a variable length left argument.

Prohibited features of APL are those incompatible with a compiled APL dialect. Typical functions in this class are those, such as `⎕fx`, `⎕nl`, `⍎`, and `⎕cr`, that manipulate the APL namespace or create functions dynamically. The APEX compiler provides no support for prohibited features.

This classification of language features, shown in Figure 3.1, allowed us to concentrate our early design and development effort on the fundamental issue of compiling simple, straightforward APL programs, in order to make decisions about the feasibility of the project as a whole. Mandatory features were implemented immediately, as an APL compiler without them would be of little practical value. This approach let us make an early verification that our generated code would have a reasonable level of performance. Then, as specific benchmarks arose that required use of deferrable facilities, we undertook their design and implementation. If a workaround of some sort was available, we often used it in the interest of expediency. For instance, some of our application benchmarks needlessly used semi-globals when pure functions would have worked just as well. Before we had implemented support in APEX for semi-globals, we were able to run these benchmarks by rewriting them into functional form. Once we had semi-global support implemented properly, we could then run the benchmarks in their original form.

We also deferred, again in the interest of expediency, the implementation of language features that are rarely used or tedious to implement properly. For instance, we did not initially write all the code fragments required to support primitives operating on arrays of rank four or higher. These were written on an as-required basis. In general, when we could easily provide more functionality for a facility, such as support for arbitrary argument rank or type, we would do so. However, if the facility required significant development work for little or no immediate payoff, we deferred its implementation. Since generation and testing of code fragments is a generally straightforward process, this turned out to be an effective way to prioritize our development efforts.

During the course of compiler development, the structure of APEX evolved into that shown in Figure 3.2. The APEX compiler comprises five major phases: tokenization, syntax analysis, static single assignment transformation and semi-global analysis, data flow analysis, and code generation. These phases will be referred to in the discussion that follows. A related technical report presents detailed information on the structure and internals of APEX [Ber97].

We now discuss how we dealt with the challenges to APL compilation. We begin with resolutions to the semantic issues arising from the APL language itself that are common to the design of any APL compiler. We then turn to a presentation of the challenges we encountered in the use of SISAL as an intermediate language for parallel compilation. Finally, we address the semantic differences that are inherent between compiled and interpreted dialects of APL.

3.1 APL Compilation Challenges

The semantics of APL present a number of challenges to the compiler writer. Foremost among these is the absence of declarations for identifiers. Overloading of the domain of defined and primitive functions is a related problem. The scoping rules of APL, loved by neither user nor implementor, combined with APL's limited argument passing capability, often force application writers away from a functional programming style and into an imperative style that uses side effects and semi-global variables. Dynamic

Language feature	Mandatory	Deferrable	Rarely used	Prohibited
Monadic scalar functions	•			
Most dyadic scalar functions	•			
Some circle functions, binomial			•	
Indexed reference & assign	•			
Reduction& scan	•			
Inner product & outer product	•			
Ravel	•			
Shape	•			
Index generator	•			
Table	•			
Reshape	•			
Join	•			
Rank conjunction		•		
Roll & deal		•		
Grade up, grade down		•		
Reverse	•			
Monadic transpose	•			
Dyadic transpose	•			
Matrix inverse		•		
Matrix divide			•	
Execute				•
Join along axis		•		
Index of	•			
Member of	•			
Replicate	•			
Expand	•			
Rotate	•			
Base value		•		
Representation		•		
Take & drop	•			
Left & right	•			
Identical	•			
System functions		•		
System & shared variables		•		
Monadic & dyadic format		•		
Quad input				•
Quad output		•		
Quote quad input & output		•		
Bracket axis notation		•		

Figure 3.1: APEX classification of APL language features

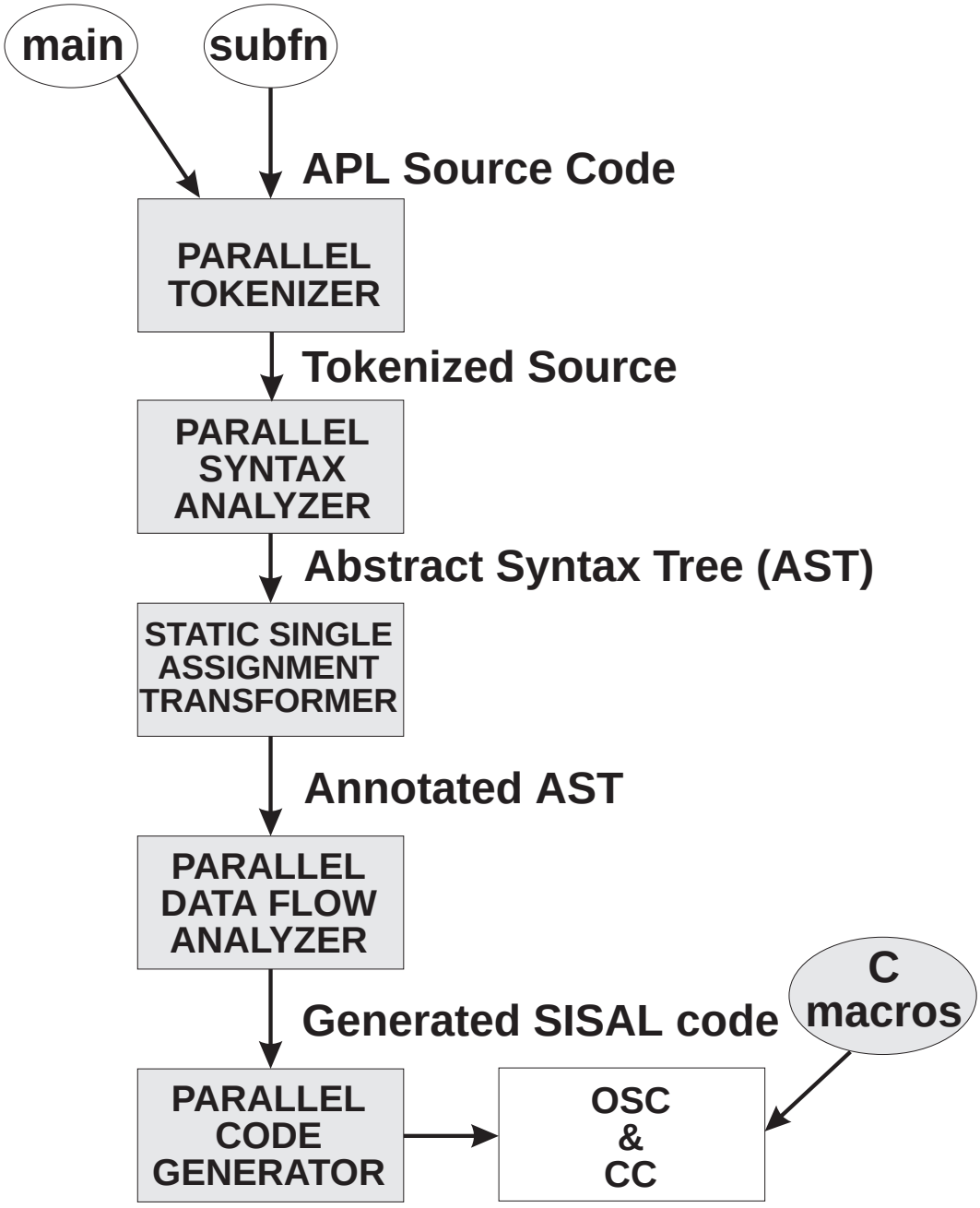

Figure 3.2: Structure of APEX

changes in the nameclass of identifiers is another serious problem. The absence of structured control flow capabilities in most APL dialects also complicates data flow analysis and hinders generation of high-performance code.

These challenges, summarized in Figure 3.3, must be dealt with in some form by any APL compiler, but they are greater if high performance of generated code is a primary goal of the compiler. We will now discuss each of these challenges in the light of the design of APEX.

Challenge	Solution or workaround
Array type and rank determination	Deduce via data flow analysis. Specify via declarations.
Defined function overloading	Clone functions.
Variable type and rank conflict	Use SSA.
Dynamic scoping of names	Clone functions. Transform code into functional form. Use SSA to facilitate transformation.
Semi-globals	Transform code into functional form. Use SSA to facilitate transformation.
Unrestricted *goto*	Defer. Use APL+Win control structures. Eventually support via Erosa & Hendren transforms.
Side effects	Use SSA Transform code into functional form.
Dynamic nameclass changes	Prohibit.

Figure 3.3: APEX solutions to APL language compilation challenges

The first stumbling block one encounters when attempting to write a high-performance APL compiler is the absence of variable declarations in APL. In order to generate efficient code, a compiler must know the type and rank of all result arrays created during execution of the source APL program. Traditional languages obtain this information by requiring the user to declare that information as part of the source program. Since the APL language does not include declarations for variables, APEX takes another approach to obtain most of this type and rank information.

We adapted traditional data flow analysis methods to the analysis of APL programs, in the spirit of others who have trod this path [Mil79, CNS89, Bud88, Ber93]. One difference between some forms of traditional data flow analysis and the methods used to analyze APL programs is that we treat all arrays as atomic values; no attempt is made to disambiguate references to specific array elements. Theoretically, this may inhibit certain optimizations and reduce the effective amount of parallelism in compiled code, but the array-oriented coding style encouraged by APL largely mitigates this effect.

The data flow analysis methods used by APEX to extract type and rank information from APL programs may require an occasional assist from the application programmer in cases where there is inadequate semantic information available to deduce that information. For example, in the trivial function:

```
r←x add y
r←x+y
```

there are no restrictions on the ranks of `x` and `y`; they may be scalars, vectors, or arbitrary arrays. Because addition is defined only on numbers, their types are known to be numeric, but they may be any combination of Boolean, integer, or double-real numbers. There is no way to deduce this information statically, so we have no choice but to adopt the well-known technique of requiring that the programmer supply a declaration to the compiler specifying the function arguments' types and ranks. To do this, we defined a simple pragma-like form of variable declaration, represented in the source APL program as comments, expressed as `⍝ dcl` *TYPE RANK* `VARIABLENAME`. For example, if `x` was a Boolean vector and `y` was an integer scalar, the function with declarations would look like this:

```
r←x add y
⍝ dcl boolean vector x
⍝ dcl integer scalar y
r←x+y
```

The number of declarations required is typically one or two per application for the "main" function arguments, as in the above example. Such declarations proved adequate, with a few exceptions, for the entire suite of benchmarks presented in Chapter 5 and Chapter 6. Almost all other types and ranks can be deduced given that information. This agrees with the data flow analysis findings of Ching [CNS89]. For instance, in the above example, if `add` were called from another function with a Boolean left argument and an integer right argument, no declarations would be required for the `add` function, as interprocedural data flow analysis would determine this information automatically.

The compiler does not, at present, prompt the user with information as to which variables require explicit declarations. However, it does deduce where data flow analysis has failed, so addition of such feedback would be straightforward. Our experience with the compiler suggests that a graphical presentation of this information, with the offending function displayed in a window, would be quite effective. The function's variables could be colored to indicate their properties, highlighting those for which data flow analysis had failed.

Closely tied to data flow analysis and identification of the morphological properties of each function in an application is the issue of conflicting morphologies of defined functions as a result of overloading. Consider `nub`, a typical utility function that returns the unique elements of a list:

```
r←nub y
r←((⍳⍴y)=y⍳y)/y
```

This function might be invoked with an integer argument at one call site and with a double-real argument at another. Each invocation of `nub` will, therefore, require different data flow analysis, since the function's argument, intermediate results, and result may have types that differ for each call. A simple way to handle the type conflict is to *clone* a distinct copy of the function at each calling site. To

clone a function, we replace each invocation of the function with a uniquely named copy of the original function, then make as many copies of the function as there are invocations, each renamed to match the new invocation name. This permits data analysis to proceed for each invocation without morphological conflict. After data flow analysis is complete, cloned functions with identical morphology could be merged to reduce the volume of generated code. The cloning approach increases compilation time, but it is simple and robust.

Data flow analysis is also used to extract other morphological information, including array shape, element count, syntax class, value, and predicates such as *ArrayIsSortedUp* and *ArrayIsPermutationVector* (to permit algebraic simplification of expressions and to allow generation of improved execute-phase algorithms). Another useful property, not currently collected, is the original source of array shape information. This could be exploited to permit improved removal of run-time length conformability checks, as was apparently done in YAT [JO86].

Data flow analysis propagates morphological information throughout the abstract syntax tree as analysis proceeds, with constant propagation occurring as part of data flow analysis. For example, catenation of two arrays whose shapes are known, but whose values are not, will propagate the correct result shape of the catenation, potentially improving the quality of generated code.

Data flow analysis is the key to generation of high-performance code. Nonetheless, data flow analysis, by itself, is not an adequate tool for determining the type and rank of all variables in APL. This is because APL permits the same name to take on different type and rank at any point during a computation. In the unlikely expression `X×(X←0.3 0.5)+X+X←2`, the variable `X` appears first as an integer scalar, then as a double-real vector. Naive data flow analysis will detect a conflict in the properties of `X` and be unable to proceed, leaving the compiler with insufficient information for code generation.

Such reuse of variable names frequently reflects a APL coding style used by application programmers to implicitly free the memory associated with the previous value of the variable. This style arises largely from the naive execution model used by APL interpreters, wherein there is no detection of dead variables. The space a dead variable occupies is freed only when the variable is assigned a new value, or at function exit time, when the variable is delocalized. The inability of interpreted APL to know when a variable is dead is a constant source of annoyance for application programmers who, working in a memory environment of fixed size, the APL workspace, must *explicitly* eliminate large dead variables if their applications are to run successfully.

Interpreted APL programs would produce identical results if this sort of name reuse was avoided, although memory requirements would increase due to the presence of dead variables. If variables in APL programs were renamed to avoid reuse, the problem of conflicting array properties would not arise and code generation could proceed apace. This brought about two questions. First, how could renaming be performed without changing the semantics of the program? Second, what effects on memory usage would arise from such renaming?

We had looked to work done by Cytron, *et al.*, in static single assignment (SSA) for a solution to the mapping of APL programs into a form suitable to the semantics of SISAL [CFR+89]. We found

that if we treated APL arrays as atomic entities, the mapping of APL applications into SSA form was relatively straightforward, with the exception of a minor problem arising from the APL concept of *value error*, which we discuss later in Section 3.2. In the course of our research into this mapping, we realized that SSA was exactly the tool we needed for performing the renaming of APL variables so that data flow analysis would not produce conflicting array properties. SSA would also provide the information needed to deallocate each variable as soon as it becomes dead, thereby avoiding any potential problems with excessive memory usage.[1] We added an SSA phase to APEX, thereby solving both problems – array property conflict and memory management – in one fell swoop.

The use of static single assignment also served as the basis for addressing the problem of side effects in APL applications engendered by use of global and semi-global variables. SISAL and other functional languages prohibit references to global names; they require that all arguments to, and results from, a function be explicit. APL functions, by contrast, cannot accept more than two explicit arguments nor produce more than one explicit result. This restriction forces programmers to write in a non-functional, implicit style, referencing and setting global and semi-global variables by name. This practice causes problems in program comprehension and maintenance, but it is unavoidable for all but the most trivial applications.[2] Because the use of globals and semi-globals remains a common practice in production APL applications, an industrial-strength APL compiler must be able to perform the interprocedural analysis required to support globals and semi-globals.

Support for semi-globals required extensions to the data flow analysis and code generator phases of APEX, as well as requiring the introduction of two new analysis phases into the compiler. We will now discuss APL-related aspects of this support; SISAL-related issues of code generation for semi-globals are discussed in detail elsewhere [Ber97].

The primary task of data flow analysis is to determine the type and rank of all arrays created during the execution of an APL application. Within a pure function, these are precisely determined by its formal arguments. However, if semi-global parameters are involved, this morphological information can only be determined if the type and rank of the incoming semi-globals are known at the invocation point of the function. Since there is no obvious connection between the last setting of the semi-global and the function invocation, it would seem that the semi-global could potentially take on as many different types and ranks as there are sets to the semi-global in the entire application, thwarting any attempt to determine the correct morphology for the function.

This is where SSA, combined with interprocedural analysis of semi-global sets and references, saves the day. Static single assignment resolves the ambiguity of multiple definitions of semi-globals. Once this is done, interprocedural analysis transforms all references to semi-globals into purely functional form.

The potential difficulty of analyzing semi-globals was exacerbated by APL's lack of a standard for

[1] SISAL, being a single assignment language, performs dead variable deallocation automatically for us. All we needed to do was to map APL programs into single assignment form to exploit this capability.

[2] Newer dialects of APL permit the use of recursive data structures (nested arrays) to work around these restrictions, but the code required for such circumlocution is convoluted and difficult to maintain.

structured control flow. The *goto*, implemented via the right arrow, is the only mechanism prescribed by the ISO APL Standard for non-linear control flow. Besides encouraging convoluted paths of control flow, *goto* entails the use of side effects.[3] The anarchic nature of APL's *goto* permits run-time branching to *any* line of a function. This potentially complicates data flow analysis to the extent that compilation of functions with arbitrary branch constructs may be fruitless, as it hinders the generation of efficient code by a compiler.

Ultimately, we chose to sidestep the issue of arbitrary branch targets, as they are almost never used in practice, because of the obvious application maintenance problems engendered by their use. Sidestepping allowed us to attack analysis of control flow independently from analysis of side effects. First, we deferred dealing with the issue of *goto* by forbidding its use for the present, and adopting structured flow controls as defined in the APL+Win APL interpreter. Because the problem of analyzing structured code is considerably simpler than that of analyzing the spaghetti-like code resulting from use of *goto*, we were quickly able to write an analyzer that allowed us to compile APL programs written in that dialect of APL. Our intent is to support *goto* expressions by mapping them into control flow structures. This will be done by use of the structuring algorithm of Erosa and Hendren [EH93].

A solution to our second problem, that of eliminating side effects within loops, was facilitated by static single assignment. Introduction of ϕ-functions by the SSA phase of APEX identified induction variables and loop-carried variables. This permitted us to create functional iterative constructs that were free from side effects: all inputs to, and outputs from, the iterative block were explicit. Use of structured control flow and SSA thus simplified data flow analysis and allowed us to generate efficient, functional code.

Happily, the approach we adopted for dealing with *goto* also provided us with the means to map iterative APL constructs into the purely functional forms supported by SISAL. SISAL differs from other languages in that it does not have a *goto* and its structures are strictly functional in nature, returning explicit results with no side effects. Unlike APL, SISAL is a single assignment language, in which a given name may be assigned a value at only one locus within any given lexical scope. Furthermore, SISAL's iteration constructs are functional, accepting arguments and producing explicit results, whereas APL's iteration constructs resemble those of traditional languages. The mapping of such general iteration constructs into SISAL form would, therefore, have been extremely complicated. Our use of SSA and structured flow controls facilitated this mapping, simplifying the task of code generation. By translating the APL source program into SSA form early in the compilation process, the set of names used or defined within each iteration block was clearly identified, thereby simplifying the analysis and code generation problem.

Another challenge to the APL compiler writer is nameclass changes during execution, as when an executing program changes the definition of a name from a variable to a function. Dynamic nameclass changes are not always easy to handle in APL interpreters, where they remain a rich source

[3] By side effects, we mean using variables to store state information, as opposed to a purely stateless, functional approach: a purely functional program can be written without the use of named variables.

of system errors. However, nameclass changes are rare in production code for the simple reason that they exacerbate code maintenance problems. Hence, we, like other designers of APL compilers [Bud88, CNS89, Wie79], chose to disallow dynamic nameclass changes in our compiler. We feel that the theoretic restriction this places on application designers is a fair price to pay for the gains obtained from a high-performance compiler.

In summarizing how we solved the problems of APL analysis, we see that data flow analysis and static single assignment were key factors. Introduction of structured control flow into APL facilitated use of the above analysis techniques. These techniques are general, in the sense that they are applicable to any APL compiler. We encountered other problems, some arising from differences in semantics between interpreted and compiled APL, and others arising from our choice of SISAL as an intermediate code. We now turn to the approaches we took in dealing with those semantic incompatibilities.

3.2 Semantic Differences in Compiled and Interpreted APL

Just as there are semantic differences in the APL dialects defined by various APL interpreters, the code generated by APEX differs in some respects from that of interpreted APL. One of these differences, *type determination*, arises out of traditional compiler design; the others arise from a design decision to use SISAL as an intermediate language for the compiler. Although the impact of these differences is generally minimal, their presence is visible to the application programmer. This visibility necessitates a discussion of the differences and possible resolutions of those semantic differences. These issues are summarized in Figure 3.4.

Challenge	Solution or workaround
Integer overflow	Use declarations. Prohibit implicit promotion.
Floor and ceiling	Produce double-real result. Use declarations for integer coercion.
Non-scalar singletons in dyadic scalar functions	Restrict to vectors.
Empty arrays	Support subset. Eventually, fix SISAL or redesign code generator.
Semantics of value error	Workaround.

Figure 3.4: APEX solutions to APL-SISAL semantic challenges

To achieve high levels of run-time performance, language compilers fix the storage type of arrays at compile time. APL interpreters, by comparison, make type determinations dynamically, during execution. Normally, this distinction is not important for production applications, as data types tend to remain unchanged over the course of execution. There are, however, two situations in which dynamic

type determination is desirable. These are the handling of integer overflow and treatment of the APL *floor* and *ceiling* primitives.

Although applications tend to work consistently on the same data types, numbers that are considered to be integer may be forced, on occasion, to a storage type of double-real, because of integer overflow. An example of this might be stock trading volumes on Black Monday. Although failure to handle such events properly is undesirable, APEX currently requires the programmer to explicitly declare the type of such potentially double-real quantities. This is safe from the standpoint of getting correct results, but it does require additional analysis by the programmer and has the potential to incur a loss in memory space and performance. More exotic solutions would require considerable development work, such as generation of multiple sets of code and run-time detection and handling of integer overflow. As the C language does not provide any mechanism for detecting overflow at the hardware level, other methods must be used. Two approaches that have been taken with APL interpreters are assembly code, which does support such overflow detection, and coercion to and from floating point for all integer arithmetic. A third method would be to adapt Wortman's legality assertions for signed integer arithmetic operations in Euclid[Wor79] to this task. This would permit run-time integer overflow to be detected explicitly but, like the floating point method, exacts a harsh performance penalty even when overflow does not occur. In the longer term, generation of multiple code sets and run-time integer overflow detection may represent a sensible way to obtain the flexibility of interpretive execution without sacrificing run-time performance. However, since it has the potential for creation of an exponential number of code sets, it may be impractical for real applications.

Another area where compilers force a change in APL's semantics is in the *floor* and *ceiling* primitive functions. Since APL interpreters attempt to represent arrays in the most compact form that does not lose numeric precision, the operations of *floor* and *ceiling* have the dual effects of removing the fractional part of a double-real number and simultaneously attempting to coerce the storage type of the resulting integer-valued double-real to integer. If the magnitude of the resulting value is too great to allow it to be represented as an integer, it remains stored as a double-real number. The APL programmer need not, in most cases, be concerned about the storage type resulting from use of *floor* or *ceiling*, except for its impact on memory usage.

This behavior creates a conundrum for APL compiler designers, because the code generated for a specific invocation of *floor* or *ceiling* must always produce either an integer result or must always produce a double-real result. Either approach is problematic. An integer result risks producing the wrong result when the magnitude of the argument is too large to be stored in an integer; a double-real result may cause performance problems or it may produce wrong answers if, for example, passed to a C function that expects an integer. We chose to separate the coercion function from that of *floor* or *ceiling*, by always returning a double-real result. This preserves correct results at the expense of increased memory space. In the rare case where integer or Boolean storage type is required, APEX requires that the programmer insert, after the invocation of *floor* or *ceiling*, an assignment to a named variable with declared integer or Boolean type. This technique will produce the same result as an interpreter because

the type coercion will have already been performed by the interpreter and the assignment is harmless. Other approaches to this problem, such as introducing a system function for coercion, are less palatable, because they would be incompatible with interpreted dialects of APL.

We see, therefore, that the act of compiling APL into code that has fixed storage types produces run-time behavior that may differ from that of interpreted behavior. Generation of multiple code threads, to be dispatched on the basis of array types at run-time, may be a feasible solution in practice, but runs the risk of exponential code growth and increased setup cost.

One more semantic difference between interpreted APL and compiled APL lies in the treatment of singleton arrays in dyadic scalar functions. The APL Standard definition of scalar function execution makes it impossible to use the argument ranks of dyadic scalar functions as the sole determinant of the result rank. Rather, the result rank is dependent upon argument ranks *and* argument shapes. Since the data flow analysis phase of APEX requires that rank be determinable at compile time, APEX is unable to support this aspect of the definition. However, since singleton extension is rarely exploited in applications, we opted to disallow it, and instead require that the argument ranks match or that one argument be a scalar or one-element vector. We do support extension of one-element vectors, due to the high frequency of their use in applications, particularly when invoked implicitly, such as in the expression `(100 52⍴4)+⍤1(100 1⍴5)`. Adoption of this restriction has not presented a problem for any of our benchmarks.

Other semantic differences in compiling APL arise from the target language generated by the compiler. The decision to compile to SISAL, rather than directly generating C code, was a tradeoff wherein we gained significant benefits in performance and portability in return for minor differences in the behavior of APL applications. In one case, the change in behavior is potentially troublesome; one other represents, in our opinion, an improvement over ISO Standard APL; others are merely a nuisance for the compiler writer.

The most serious limitation that SISAL imposed on APEX is SISAL's inability to represent a subset of the empty arrays. It is serious in the sense that this limitation is the one most likely to cause unexpected run-time failure of production applications. Empty arrays are those containing at least one zero in their shape vectors, *e.g.*, an n-by-0 or 0-by-n matrix. Although empty arrays may seem to be of little interest, the algebra of such edge conditions is often important to the proper functioning of programs. The problem with SISAL in this regard is that it does not support arrays as true arrays, but as vectors of vectors, much as C does. SISAL is, therefore, unable to represent a 0-by-n matrix, although an n-by-0 matrix does not present a problem.

Currently, we are ignoring the empty array problem, as it has not materially affected the behavior of any application we have compiled. In the longer term, it may turn out that relaxing the conformability requirement for catenation of empty arrays may be adequate for most purposes. However, we are not comfortable with this approach, as we have not considered all of its design implications, nor discussed it with other APL language designers. A better long-term solution is to enhance SISAL to include proper support for matrices, rather than vectors of vectors. Besides correcting a semantic problem in APEX,

such an enhancement would provide two performance benefits for SISAL. First, it would eliminate the requirement to dereference array row pointers, thereby producing a substantial performance improvement for certain applications, as will be shown in Chapter 5. Second, it would permit introduction of *array coordinate mapping*, as discussed in Chapter 2, offering fixed-time execution of many structural array operations. We are currently discussing the benefits of such a change in the definition of SISAL with the SISAL language designers. The likelihood of this change being adopted depends largely on its impact on extant SISAL applications and, to a lesser degree, on the difficulty of implementing the change within OSC.

The use of SISAL as a target language also forces a change in the behavior of *value error*. In APL, a reference to a name that has no current referent, *i.e.*, an undefined variable, causes the interpreter to signal *value error* and suspend execution. The programmer can then correct the program or give the offending name a value, then continue execution from the point of suspension. The semantics of SISAL, however, are such that value error cannot occur. Potential value errors are detected at compile time; programs that offend in this manner are rejected by the SISAL compiler. Since APL provides no mechanism for static detection of value error, a programmer is forced to perform careful testing of applications in an attempt to ensure that value error cannot arise. By contrast, when an APEX-generated SISAL program is compiled successfully, it is *guaranteed* to contain no value errors. Since APEX does a better job of APL code validation than does interpreted APL, we declare this semantic difference to be a feature, rather than a bug or implementation restriction.

The guaranteed absence of undefined variables presents a minor problem in mapping APL to SISAL. The loop semantics of SISAL are purely functional – they have no side effects. That is, each SISAL loop accepts arguments and returns results, just as a pure function does. This requires that each variable that is assigned within the scope of a loop be assigned an initial value. This provides support for the case of a zero-iteration loop, whereby that initial value becomes the result of the loop. When we map APL into SISAL, we have to provide such an initial value for each loop. The proper initial value to be used is obviously the value of the variable at the time of entry into the loop, as a zero-iteration loop would then have the effect of propagating that value, unchanged, through the loop code. But what if the variable has not been assigned a value yet? We would like to preserve its undefined value, *value error*, but SISAL has no way to represent such an undefined value. Hence, APEX uses an array of fill elements as the initial value for the variable, *e.g.*, zero for numeric arrays and blank for character arrays. This works perfectly, unless the original APL source program contains a bug that would permit the loop to be executed zero times. In that case, the original APL program would eventually signal *value error* upon a reference to the variable, but the SISAL program would continue execution, potentially producing erroneous results.

One possible resolution to this problem is a SISAL extension to support a distinct *undefined* value; another approach is to generate different SISAL code for loops that have such an undefined variable as an initial value, so that zero-iteration loops cause a run-time abort. Although these cases are trivial to detect at compile time, the compiler presently ignores this aspect of the problem.

3.3 Summary

We have now discussed the hindrances to compiling APL and the methods – function cloning, static single assignment, interprocedural data flow analysis, semi-global analysis – that we used to solve or circumvent them: modifying the semantics of some parts of APL, introduction of declarations, and the use of modern APL flow control structures in lieu of *goto*. We consider the altered behavior of undefined variables in APEX to be a language feature, while the problem of empty arrays in APEX-generated code is clearly the major outstanding semantic problem facing us today. In the remaining chapters, we will examine the performance of code compiled with APEX and look at the impact of the unsolved problems that are still with us.

Chapter 4

Experimental Framework

We conducted a series of experiments on various hardware platforms to quantify the extent to which our implementation realized the design goals of APEX. This chapter presents the experimental framework we established for that purpose. Our experimental results, comparing the execution time performance of APEX-generated code to that of interpreted APL are shown in Chapter 5; a similar comparison against FORTRAN 77 is given in Chapter 6. The results of preliminary experiments, performed to measure the parallel speedup achieved by compiled APL on three multiprocessor systems, are presented in Chapter 7. We give results for both applications and synthetic kernels so that we may illuminate both the salient features and the dark underbelly of system behavior.

We performed uniprocessor benchmarks of compiled APL on two RISC and one CISC processor, using interpreters provided by two APL system vendors. Our aim in using several platforms was to isolate hardware effects, such as concurrent instruction issue, from the effects induced by the use of a particular C compiler or APL interpreter. Henceforth, we will refer to these systems by the platform name only (e.g., RS/6000, SUN, 486), with the understanding that they refer to the testbeds as configured here. The systems and compiler options we used on these platforms are shown in Figure 4.1. The SISAL compiler generates parallel code by default; we used the compiler option *-seq* to force generation of sequential code on uni-processor platforms.

The remainder of this chapter opens with a discussion of the metrics we used as the basis for measurement. We then introduce the benchmarks we chose to evaluate the performance of APEX.

4.1 Metrics

Our desire to obtain equitable performance measurements of interpreted APL, FORTRAN, and APEX-generated code led us to face several problems. In the timing of benchmarks, we had to deal with processor time granularity issues and the differing modes of use of interpreted and compiled code. In scaling of benchmark sizes, we had to address problems of considerable disparities in performance and real memory size. We finally adopted a measurement protocol for benchmark timing and scaling that we believe provides relatively precise and reproducible results.

Platform	Hardware	Operating System	APL Interpreter
ASUS 486DX4-100	256KB L2 cache PCI/ISA bus	Win3.1, DOS 6.22, Linux 1.2.13	APL+Win V1.3.00
SUN 4m	SPARC 10	SunOS R4.1.3-U1	SAX V4.7.0
IBM RS/6000	Model 550	AIX V3.2	SAX V4.7.0
SGI Power Challenge	12 processors	UNIX	
CRAY C90	32 processors	UNIX	
CRAY EL92	2 processors	UNIX	

Platform	Compiler & Version	Compiler invocation
486	gcc V2.6.3	gcc -O3
486	g77 V0.5.15	g77 -O3 -m486
486, SUN, RS/6000	OSC 13.0.3+loop fusion patch	osc -O -externC fmod -seq -cc="-O3" -DCONFORM -inlineall -cpp -I/dos/g/apex/lib
SUN	C Development Set V2.0.1	cc -O
RS/6000	cc V3.2	cc -O
SGI	cc	cc -O
CRAY C90	cc	cc -O
CRAY EL92	cc	cc -O

Figure 4.1: APEX Testbed Configurations

The timing facilities that were available on our testbed platforms forced us to use slightly different measurement techniques to obtain acceptable timer precision. For APEX code running on the 486, we measured *user* time, via the Linux shell *time* command. We measured APL interpreter time on the 486 by use of the □ts system function.[1] On the UNIX RISC platforms, we measured time via the */bin/time* command, as the shell *time* command was only precise to one second. As the RISC machines were multi-user platforms over which we could not exert control over user load, we restricted our use to periods of minimal system usage. For interpreted APL on those platforms, we measured user time with the □ai system function,[2] to minimize the effects of interference due to other usage of the system.

We wished to minimize the effects of non-systematic garbage collection, interference from other tasks, and startup effects. To achieve this, we ran each benchmark a minimum of three times, on lightly loaded or dedicated systems. If all times were not relatively close to each other (within a few percent), we re-ran the offending benchmark until stable results were achieved. In such cases, we recorded the lowest three such measurements. Although one might argue that the discrepancies arising from the above effects should be included in our timings, our interest lay in measuring the relative performance

[1] This system function returns the time of day, typically to the precision of the host system clock.

[2] This system function returns the processor time used by the APL task, typically to the precision of the host system processor timer.

of the applications, rather than system-wide performance. In any case, the times we measured were, by and large, within one percent or less of one another.

The mode of use of compiled code is different from that of interpreters. Compiled codes are typically initiated by a command shell, whereas interpretive use typically involves starting an APL interpreter, loading a workspace, then executing one or more applications under that interpreter. In an attempt to address this difference in usage style, we *did not* measure the time to start the APL interpreter or to load the appropriate application workspace. In contrast, for APEX-generated code and FORTRAN, we *did* include the time required to initiate and terminate the application. We realize that this approach may give interpreted APL a slight edge in performance, but feel that this approach is fair, as it reflects normal modes of use.

Benchmark scaling and tuning presented us with a problem, in that scaling of benchmark problem size in order to make one platform's timings measurably large sometimes introduced a failure on another platform due to real memory exhaustion (**mconvo**) or excessive execution time (**shuffle**). We nonetheless attempted to scale all benchmarks so that they used enough processor time that clock precision did not materially affect relative performance measurements. Furthermore, we attempted to scale all benchmarks so that they operated entirely within memory, with no system paging activity.[3]

4.2 Application Benchmarks

Our choice of benchmarks was influenced by two factors. In the interest of fairness, we wished to have a suite of benchmarks that would explore all regions of the APL performance envelope, so that areas of significant APEX superiority or inferiority would be highlighted. We also desired feedback on the performance of specific aspects of APEX-generated code, to assist us in determining where the deficiencies of our compiler lay.

We view the APL performance envelope as a multi-dimensional space, with the axes of this space representing measures of application characteristics including, but not limited to, the amount of explicit iteration, array sizes, number of array elements manipulated by each APL operation, and the computational complexity of the primitive APL operations performed during execution. We attempted to place benchmarks throughout this space, so that we could map the performance of APEX throughout the envelope. For example, we chose only two highly iterative scalar-dominated benchmarks, **crc** and **shuffle**, because we feel that they adequately characterize the performance of applications of the sort that have earned APL a reputation for poor performance compared to compiled languages.

From the opposite corner of the performance envelope, we chose several benchmarks that represent areas in which APL is traditionally thought to perform quite well in relation to compiled code. This class of benchmark included a variety of inner products, intended to let us perform a comparative evaluation of different inner product algorithms and to explore the non-iterative, computationally complex part of

[3]The effect of paging, when it did occur, was to drive down execution time values, presumably due to coarse timer resolution causing truncation error in processor time measurements made by the operating system.

the envelope. We also included several signal processing applications that are non-iterative, operate on large arrays, and have lower computational complexity than inner product.

Our benchmarks fall into two major categories, the first being applications or computationally intense portions of applications; the second comprises small, problematic fragments of industrial applications and synthetic kernels intended to measure specific aspects of performance. Kernel performance results are discussed in Chapter 6.

We chose 12 application benchmarks, listed alphabetically in Figure 4.2, to measure the relative performance of APEX against FORTRAN and interpreted APL on problems that arise in the real world. These benchmarks, chosen from applications used in finance, manufacturing, and engineering, comprise code written by both professional and non-professional programmers, as well as algorithms that reflect applications in those domains.

We extracted the computationally intensive portions of complete applications, as the complete applications were often too unwieldy to work with or simply not available for our use due to corporate policy. We modified them only to the extent required to make them acceptable to APEX, *e.g.*, replacing *goto* with `:for` loops and writing driver functions to facilitate repeatable benchmarking. We will now discuss each benchmark in turn, giving an overview of its purpose and the rationale for its inclusion in the benchmark suite.

Benchmark	Source	Application	Dominant Computation
crc 50000	Soliton	CRC computation	iterative character and Boolean vector
dtb 30000 150	Reuters	data bases	character matrix
logd 500000	J.K. Bates	signal processing	double-precision vector
100 mconvo 9000	Mobil Research	convolution	double-precision vector
mdiv 200	IBM	matrix inversion	double-precision matrix
metaphon 2000	Reuters	data bases	character vector
nmo2 200	Mobil Research	geophysics	double-precision matrix
primes 2200	Pythagoras	primes computation	double-precision matrix
rle 10000	Reuters	data bases	integer vector
shuffle 1000	Baase	dynamic programming	iterative Boolean matrix
tomcatv 257	SPEC	mesh computation	double-precision matrix
wivver 1200000	J.K. Bates	signal processing	double-precision vector

Figure 4.2: Application benchmark summary

The **crc** benchmark computes the 32-bit cyclic redundancy check for a character vector of length N, using a table lookup method [Sar88]. It was obtained from Leigh Clayton of Soliton Associates Limited, where it was written as part of an inter-platform communications subsystem. It is highly iterative, representing the sort of application that is difficult to implement efficiently in APL.

The **dtb** benchmark represents part of the index-rebuilding code for a large data base (MRD) at Reuters Information Systems Canada Limited. To give some indication of the size of the data base, the

data base index is itself larger than 600 megabytes. The **dtb** function removes trailing blanks from character index matrices, in order to reduce the size of the index file and speed index searching. Although this function is trivial in complexity, it consumes about 10% of the processor time required to rebuild the entire data base index.

The **logd** benchmark is a signal processing kernel. It appeared as the **logderiv** benchmark in Bates' article about COMPC, an APL to C compiler developed at IBM Research [Bat95, Chi86a]. According to Bates, **logd** "...extracts the logarithmic derivative of a waveform by computing the ratio of the derivative of the input waveform to the waveform itself, as a way to normalize the shape features and to extract the real and complex zeros." The code is non-iterative APL, operating on acoustic signals of 500,000 double-precision numbers.[4] Since the benchmark executes only a dozen APL primitives, each operating on these long vectors, setup cost is negligible; APLers would characterize **logd** as being well-written APL code. Our naive expectation is that we should observe interpreted performance that is highly competitive with traditional languages – compiled code should not offer much, if any, performance improvement here.

The **mconvo** benchmark is a signal-processing program that performs one-dimensional convolution on double-precision vectors. The driver function arguments specify the lengths of the filter and trace, respectively. The APL version of this code was provided by George Moeckel of Mobil Research and Development Corporation during the ACORN project [BBJM90]. Convolution is typical of the numerically intensive computations performed in the course of analyzing seismic trace data for the purpose of petroleum exploration.

The **mdiv** benchmark computes the inverse of a square double-precision matrix of order N. The benchmark is M.A. Jenkins' APL model of ⌹, the APL matrix inverse primitive [Jen70]. It is, therefore, identical in functional specification to that APL primitive operating on double-precision data. The compiled model should give some insight into the performance of compiled high-level APL relative to the hand-coded C that is representative of modern implementations of the primitive. Since the model was published in early 1970, written with no thought of being executed on parallel computers, it should also serve as a good example of available parallelism in APL functions. We made minor changes to **mdiv**, including removal of globals and gluing.[5] We also revised some code to improve its visual appearance when formatted for publication. Finally, we changed the final array constructor from `z←z[⍋pp;]` to `z[pp;]←z`, as there did not seem to be any virtue in performing the extra work associated with upgrade. These changes were reflected in both interpreted and compiled timings.

The **metaphon** benchmark is another part of the MRD data base index-rebuilding subsystem. This part of the system maps a matrix of names into phonetic form to facilitate data base searching when the precise spelling of a name is not known. For example, **metaphon** maps both "Reuters" and "Royturz"

[4] Bates used 24,000 element samples, but we were unable to get sufficient precision in processor timings with such short vectors.

[5] *Gluing* is a derogatory term applied to an APL coding style that provides a measure of brevity at the cost of readability and performance. Gluing was popular in the early days of APL, but has lost favor with the advent of statement separators and Better Taste In Programming.

into “RTRS” and both “Bernecky” and “Burnekee” into “BRNK”. The benchmark comprises 189 lines of non-iterative code. In the index-rebuilding application described above, it consumes 37% of all processor time associated with index rebuild for the MRD data base.

The **nmo2** benchmark performs *normal move out*, a geophysics application that analyzes seismic trace data to construct a picture of the underlying rock strata. The benchmark was provided by George Moeckel of Mobil Research and Development Corporation, as a benchmark used during the ACORN project [BBJM90].

The **primes** benchmark finds the first N prime numbers. It is frequently touted as an example of the expressive power of APL, in spite of its brute force approach to the computation of primes. The `prim` subfunction works by generating a list of the first N integers, computing the residue outer product of that list against itself, summing the number of zero residues, and extracting the list elements which have exactly two elements with zero residues – the number divided by itself and by one – as the prime numbers.

The **rle** benchmark performs a run-length encoding of integer data. This function is used at Reuters Information Systems Canada Limited as part of a compressor for time-series historical information. The compressor is used to reduce the space required to store such data before placing it in large data bases. Because these data bases are historical, they grow hour by hour – there is no deletion of old data. Data compression is, therefore, highly desirable from the standpoints of retrieval time and storage media cost. The high rate of data acquisition and heavy demands on the system for access to this data require that it be online at all times, rather than stored on dismountable media.[6] The **rle** function represents the largest consumer of processor time in the compression step of data base maintenance. The argument to the driver is the length of the vector to be be compressed.

Dynamic programming has been used extensively in large APL applications such as airline crew scheduling [Tut95]. Although the inherently iterative nature of dynamic programming makes it quite expensive to use in interpreted APL, alternative algorithms usually have exponential computational complexity, rendering them absolutely unsuitable for solving real-world problems. We chose the *string shuffle* decision problem as being representative of such applications, because it clearly invites a solution using dynamic programming and it is also simple enough to be understood without extensive study [Baa88, Ber95]. Typically, dynamic programming algorithms iterate over a large array, filling in a single element per iteration based on the results of previous iterations. The amount of computation per iteration is generally low, making dynamic programming algorithms ideal examples of a highly iterative solution domain. Other examples of applications falling within this domain are simulation, circuit analysis, finite state automata, real-time data acquisition, computer simulation, ray tracing, and Monte Carlo methods.

The **tomcatv** benchmark is an APL-coded version of the SPEC CFP92 FORTRAN benchmark of the same name. It is a mesh generation program using Thomson’s solver. Since it offers a high level of parallelism, it is often used as a parallel computer benchmark. The benchmark is dominated by double-precision floating point operations on square matrices of order N. It performs 100 iterations of

[6] In 1996, the sustained data transfer rate for access to this data base was measured at about 300 gigabytes per day.

substantial calculations on large arrays.

The **wivver** benchmark is also from Bates' article. He states that it "...derives both unidirectional and bidirectional zeros of the waveform as the edges from which a clipped waveform may be constructed. It has 18 lines of code and no loops." It resembles **logd** in many ways and, since we drive it with data similar to that presented to **logd**, we expect similar levels of performance for both benchmarks.

Complete source and generated code for the application and kernel benchmarks is available from the author. The source code and generated code for two benchmarks, **prd** and **mdivr**, appear in the Appendix. The **prd** benchmark shows the SISAL code generated for a trivial kernel benchmark; the **mdivr** benchmarks is intended to highlight more complex aspects of APEX code generation.

Chapter 5

Performance of Compiled APL and Interpreted APL

This chapter presents the first part of the results of our performance experiments, outlined in Chapter 4. We give comparisons and analysis of the execution time performance of APEX-generated code against that of interpreted APL.

As expected, we found that the performance of APEX-generated code, relative to interpreted APL, is highly dependent on the characteristics of the application. For highly iterative applications, APEX can produce code that executes several hundred times faster than interpreted APL, whereas small, straight-line APL applications may execute at similar speeds in both environments. Given this wide variance in relative performance, it is obviously unwise to characterize it without a broader understanding of the origins of this variance.

Figure 5.1 summarizes the absolute performance of the benchmarks we discuss in this section. The x-axis gives benchmark names; the y-axis shows the execution time, in CPU seconds, for each benchmark. The leftmost three bars of each benchmark are CPU time for interpreted code on the various platforms; the rightmost three bars are CPU time for APEX-generated code. Figure 5.2 presents the same information in a similar fashion, except that execution times are relative, giving interpreted time divided by compiled time. Therefore, values greater than one indicate superior performance for APEX, whereas numbers less than one indicate superior performance for the interpreter. The figures present the application benchmarks in their original, untuned state. At the end of this chapter, Figure 5.13 and Figure 5.14 give performance results for these applications after they have been tuned using methods to be described.

Two salient features of Figure 5.2 are worthy of note. First, the performance of APEX on the RS/6000, relative to interpreted code, is generally higher than it is for the other testbed platforms. We conjecture that this is an artifact of the RS/6000 APL interpreter, based on the observations that all RS/6000 interpreter times are higher than the SUN interpreter timings, whereas the better APEX times are evenly split between the two testbeds. The same phenomenon was observed in the kernel

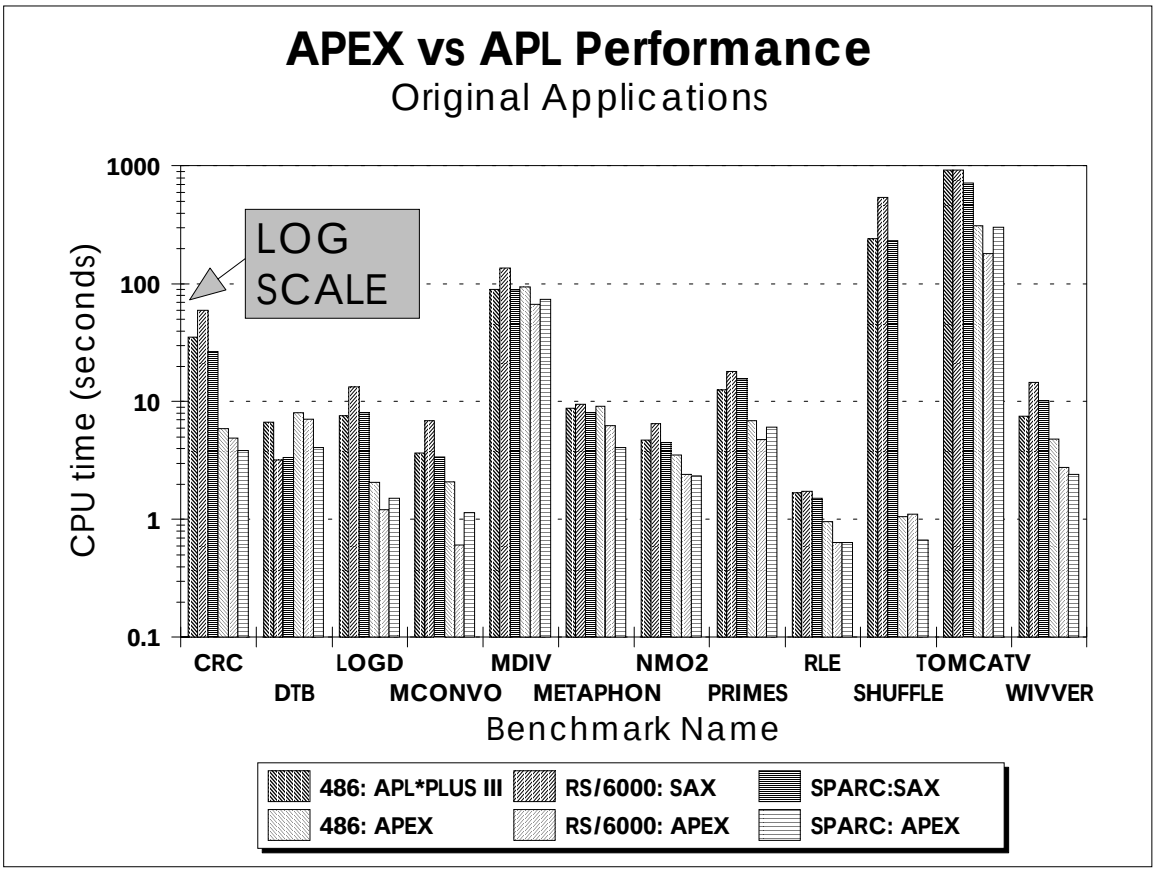

Figure 5.1: Initial absolute application performance of APL vs APEX

benchmarks.

The second feature of note is the dramatic difference in relative performance levels of the various benchmarks. Note that most of the APEX-generated application codes are currently performing 1.5–10 times faster than the interpreted versions, that one is several hundred times faster, and that one is slower. So that we may shed light on this extreme variance, we present our analysis of the benchmark results within the scope of specific performance-related topics. We open with a discussion of the effect of setup cost removal and loop fusion, the two factors that made the largest contribution to high performance in APEX-generated code. We then turn to selection of special-case algorithms and the role played therein by array predicates. Next, we examine the problem of array copying, an area that has resulted in lackluster performance of some APEX benchmarks, but in which future work may lead to substantial performance improvements. Finally, we investigate the performance improvements available by use of Extended APL features, APEX compiler features, and application tuning.

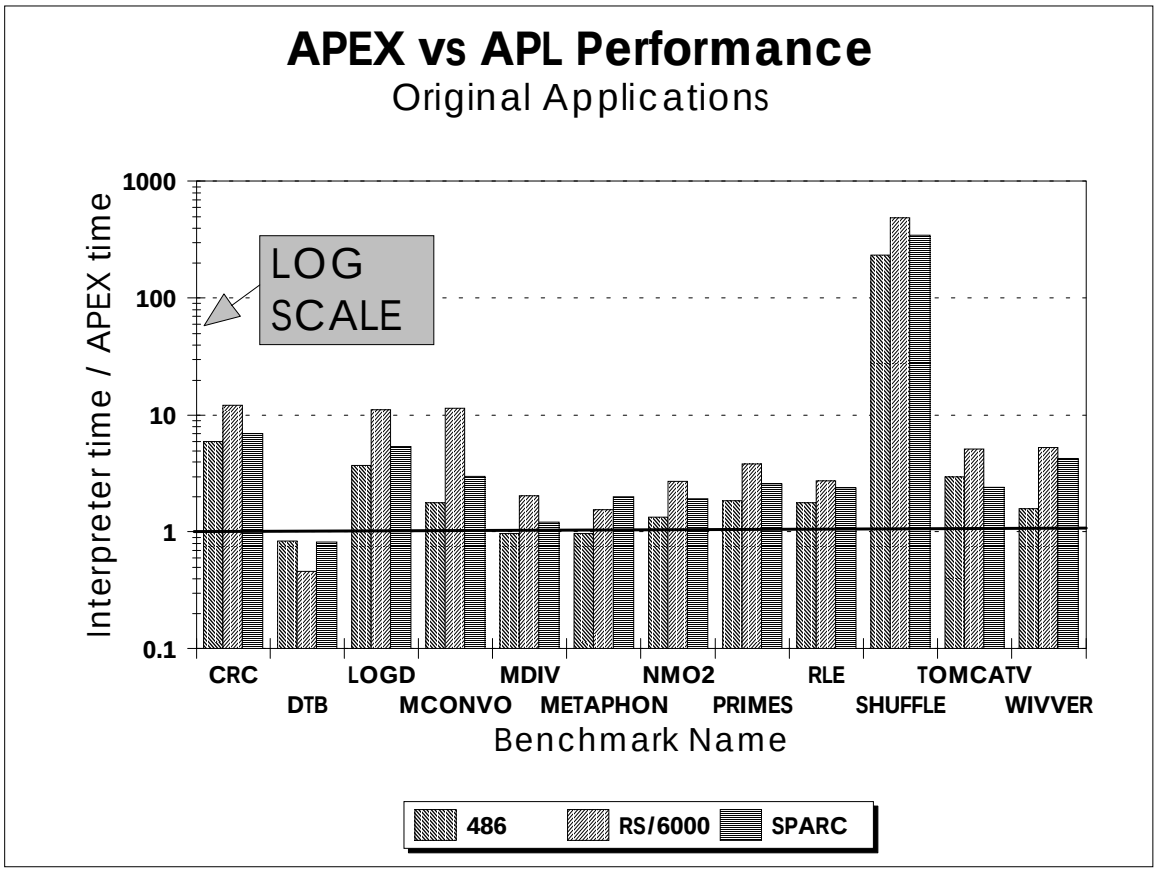

Figure 5.2: Initial relative application performance of APL vs APEX

5.1 Setup Cost Removal

As discussed in Section 2.1.1, setup cost – syntax analysis, conformability checking, and memory management – is the primary barrier to acceptable performance of highly iterative APL applications; our techniques for breaking through that barrier were presented in Chapter 3. In order to quantify the impact of setup cost reduction on the performance of highly iterative APL applications, we compared the interpreted and compiled execution times of the **shuffle** and **crc** benchmarks on our reference platforms. We chose these programs because they are inherently iterative and, therefore, exhibit high setup cost in an interpretive environment.

The results were gratifying, especially considering that the interpreters are mature products, whereas APEX is still in the embryonic stage. Figure 5.3 presents the execution times and the speedup ratio achieved by APEX over the interpreters on these benchmarks. In the case of **shuffle**, the effect of removing setup cost from this highly iterative application is dramatic. Execution time has dropped by more than two orders of magnitude, with speedups ranging from 230–486, moving this application from the realm of the performance nightmare into that of the non-issue.

Performance of **shuffle 1000** (seconds)			
Platform	APL	APEX	Speedup
486	241.78	1.04	232.48
RS/6000	535.60	1.10	486.91
SUN	231.02	0.67	344.81

Performance of **crc 50000** (seconds)			
Platform	APL	APEX	Speedup
486	35.10	5.90	5.95
RS/6000	59.13	4.90	12.07
SUN	26.54	3.83	6.93

Figure 5.3: Performance impact of setup cost removal

The significant decrease in execution time stems partially from elimination of setup overheads and partially from the fact that the inner loop of **shuffle** operates primarily on scalars or single elements of large arrays. Whereas the interpreters treat scalars as arrays, allocating and deallocating heap space for them, the compiler recognizes that the operands are scalars and emits code that uses register- or stack-allocated scalars. Since the latter approach has substantially less run-time overhead than arrays, we observe better performance with compiled code.

The other application that iterates over small arrays is worthy of note because of its poor performance relative to that just discussed. The **crc** benchmark, in computing the CRC of a character vector, examines each character in turn, performing what should be a trivial amount of computation on it. Essentially, the CRC algorithm is a finite state machine operating over the text string. We expect, based on the performance levels we observed for **shuffle**, to see the compiled code executing about two orders of magnitude faster than the interpreted version. Yet, the actual speedup is inordinately small – less than an order of magnitude.

When we performed run-time profiling on **crc**, we determined that major contributors to its relatively poor performance are *base value* and *represent* – radix-changing APL primitives that are executed in the course of processing each character of the argument. This language problem, and a compiler-based solution to it, are discussed within the context of exploiting APEX compiler features in Section 5.6. For now, we note that rethinking the algorithm to avoid radix changes made the compiled code execute about eight times faster on the 486 than when radix changing code was generated.

The performance improvement achieved by compilation of **shuffle** makes it clear that the effect of removal or reduction of setup cost can be significant. In fact, setup cost reduction probably outweighs the effect of any other single factor in APEX performance. Nevertheless, the other factors remain significant – they must be addressed if APL performance is to be boosted to the level of traditional compiled languages. There are many squeaky wheels still to be greased, and the next ones we will silence, or at least, muffle, will use loop fusion as the lubricant.

5.2 Loop Fusion

The curse of array-valued intermediate results, faced by APL programmers on a day-to-day basis, was discussed in Section 2.1.2; loop fusion's potential to alleviate this problem was discussed in Section 2.2.2. Unlike setup cost removal, which has its greatest impact on small array operations, the benefits of loop fusion are quite effective in operations on large arrays, where it can significantly reduce memory traffic associated with a computation. The effect of loop fusion on the three signal processing applications (**logd**, **mconvo**, **wivver**), the prime number finder (**primes**), and the text manipulation application (**metaphon**), is shown in Figure 5.4.

The choice of these benchmarks was based on their excellent use of the power of interpreted APL, as they are non-iterative and operate on large arrays. Our hope was that loop fusion would be fully exploited in the compiled versions of these applications.

The **logd** benchmark is a signal processing application that computes the time derivative of acoustic signals. It is well-written APL – non-iterative and operating on long vectors. Thus, we do not expect to observe much speedup from compilation. Nonetheless, when we compiled **logd**, it executed 3.7–11 times faster than when interpreted, depending on the testbed.

Figure 5.5, which shows a fragment of the code generated for the following line of **logd**, demonstrates that loop fusion is a major contributor to the performance speedup:

```
L←¯50⌈50⌊50×(DIFF 0,WV)÷RR+WV
```

Without loop fusion, the generated code would have contained five non-nested *for* loops, one for each APL primitive (⌈, ⌊, ×, ÷, +), each generating an array-valued temp. Instead, the array-generation code is almost entirely absent. The code invoking the expression `(DIFF 0,WV)`, not shown here, comprises one fused loop, creating one array temp, `tmp15`. Similarly, the loop that computes `RR+WV`, creating the array temp `tmp16`, is also not shown. These two loops are executed before the one shown in Figure 5.5. The code generated for the other four APL primitives used in `LOGDERIV` is the fused loop shown in Figure 5.5. This loop produces one result element, `tmp22`, at a time. The two `OptAElm` macros select the current scalar element from the two array temps, `tmp16` and `tmp15`. The six lines of code starting at `if` perform the divide (including APL's check for zero divided by zero); the next three lines perform the multiply, min, and max operations. The `OptPlus`, showing interprocedural loop fusion in action, implements the sum reduction used in the benchmark driver, **benlogd**, to reduce the benchmark output volume to a manageable size.

The array-valued temps we associate with execution of APL primitives do not exist in the compiled code; they neither require memory to be allocated or freed, nor do they need to have data elements stored into them and loaded from them – the operations are all performed on scalars. In addition, all but one set of the operations associated with loop initialization, induction variable incrementing, termination testing, and loop closure have been removed, adding to the aggregate performance improvement. Thus, we see that loop fusion is a valuable tool for improving the execution time of applications, even when the applications have been written in a style that exploits the best aspects of interpreter performance.

Performance problems arising out of the fragility of loop fusion in OSC 13.0.3 occurred repeatedly

Performance of **logd 500000** (seconds)			
Platform	APL	APEX	Speedup
486	7.56	2.06	3.67
RS/6000	13.18	1.20	10.98
SUN	8.01	1.50	5.34

Performance of **wivver 1200000** (seconds)			
Platform	APL	APEX	Speedup
486	7.47	4.77	1.57
RS/6000	14.45	2.73	5.29
SUN	10.11	2.40	4.21

Performance of **primes 2200** (seconds)			
Platform	APL	APEX	Speedup
486	12.54	6.84	1.83
RS/6000	17.90	4.73	3.78
SUN	15.50	6.03	2.57

Performance of **100 mconvo 9000** (seconds)			
Platform	APL	APEX	Speedup
486	3.65	2.07	1.76
RS/6000	6.84	0.60	11.40
SUN	3.39	1.13	3.00

Performance of **metaphon 2000** (seconds)			
Platform	APL	APEX	Speedup
486	8.76	9.83	0.89
RS/6000	9.46	6.17	1.53
SUN	8.06	4.07	1.98

Figure 5.4: Performance impact of loop fusion

```
  for ( ; tmp2 <= tmp3; tmp2++ ) {  /* Normal Loop */
    Plus( tmp17, tmp2, tmp4 );
    OptAElm( double, tmp21, tmp16, tmp17 );
    OptAElm( double, tmp22, tmp15, tmp2 );
    if ( tmp22 == tmp21 ) {
      tmp22 = (1.0e0);
    } else {
      IncFlopCountA(1);
      OptDiv( tmp22, tmp22, tmp21 );
    }
    OptTimes( tmp22, tmp22, tmp19 );
    COptMin( tmp22, tmp19, tmp22 );
    COptMax( tmp22, tmp11, tmp22 );
/* YankedRed */
    OptPlus( tmp20, tmp20, tmp22 );
    }
```

Figure 5.5: C code generated by APEX for **logd**

in the course of code generator development and continue to cause performance problems. In the **logd** benchmark, for example, the `DIFF` function, invoked by `LOGDERIV`, was not fused into the loop generated for `LOGDERIV`. We were unable to coerce OSC into generating fused code for `DIFF`, despite trying a number of different code generation schemes. At that point, we decided to measure the effects of code tuning and use of extended APL features to perform the first difference operation performed by `DIFF`, in an attempt to obtain complete loop fusion for this part of the application. Our success in achieving this is discussed within the context of extended language features in Section 5.5 and in the context of code tuning in Section 5.7.

Before we leave the **logd** benchmark, we note that Ching's COMPC compiler [Bat95, Chi86a], which does *not* perform loop fusion, obtained a speedup of 1.45 over interpreted code for **logd** on a 486/50 computer. Since APEX obtained a speedup of 3.67 on this same benchmark, it is performing almost 2.5 times faster than COMPC. Thus, we see that loop fusion, when exploited, can produce a substantial difference in execution time, even for well-written APL applications.

The **wivver** benchmark bears a strong resemblance to the **logd** benchmark, yet the APEX-generated code did not perform well on the 486, running only 1.57 times faster than the interpreter. Performance on the RISC platforms was better, running 4–5 times the speed of the interpreter, but this is still less than the speedup we observed for **logd**. The exact cause of this level of performance remains to be determined, but examination of the generated code reveals that loop fusion is not being performed to the same extent that we observed with **logd**.

We now turn to more serious problems involving failure of loop fusion, wherein **primes**, **mconvo**, and **metaphon** exhibited relatively poor performance. We look first at the **primes** benchmark which, when compiled, runs 1.8–3.8 times faster than the interpreted version. We assumed that both **logd** and

primes would obtain substantial benefit from loop fusion, yet the speedup of **primes** was considerably less than that of **logd**. Examination of the code generated for **primes** revealed that the first axis reduction was not being loop fused with the outer product code that generates its argument.

This finding led us to investigate the behavior of OSC loop fusion and of the overheads associated with a vector-of-vectors-based, as opposed to an array-based, memory management system. We found that the current release of OSC has at least three problems with loop fusion, which we shall discuss presently.

The fundamental problem with **primes** arises from OSC's present inability to fuse an outer loop with an inner loop. This inability to fuse, in turn, arises from interactions between the APEX code generator design and SISAL's vector-of-vectors design, wherein our desire to generate high-performance intra-primitive code had a detrimental effect on inter-primitive performance. Currently, APEX generates APL primitives independently. That is, within the code generated for one primitive, there is no consideration given to the way in which its result will be used. This simplifies the code generator considerably, but because it does not concern itself with inter-primitive issues such as order of element generation versus order of element use, it opens the door to creating interpreter-style code. Such code generates entire arrays and then uses them in that form, rather than generating and consuming an element at a time, in the style of scalar languages.

The optimizers of OSC operate well to mask much of this, as most APEX-generated code generates result arrays in row-major order, for reasons including code consistency, minimization of dereferencing of array row descriptors, and maximization of stride-1 memory access. Although this style of coding encourages loop fusion, it is not a panacea, because some APL operations are not amenable to row-major ordering or have other constraints that interfere with loop fusion. In the **primes** benchmark, for instance, its key expression, `+⌿0=b∘.|b`, sums each column of the zero test of the matrix created by the outer product. The outer product code and the reduction code were both written to perform in row-major order, but we were unable to develop an expression for the reduction using a parallel SISAL *product form* loop. Unfortunately, loop fusion did not occur, because OSC did not interchange the loop order to enable such fusion. Hence, OSC creates a very large temp array for the outer product, with concomitant performance loss.

We evaluated several methods of correcting this problem at the level of a single primitive, including transposition of reduction order, explicitly transposing the argument to the reduction, and executing the reduction with non-unit stride along the raveled matrix. None of these techniques improved performance.

We then indirectly investigated the effect of generating code that takes into consideration interactions among primitives, by rewriting **primes** to use a rank expression rather than an outer product, thereby effecting an implicit transpose of the matrix. The good success we had with this technique, discussed in detail in Section 5.7, suggests that a more sophisticated code generator, performing algebraic manipulation of the abstract syntax tree, could generate substantially more efficient code than we can do currently. Driscoll and Orth present examples of such a code-merging technique but, unfortunately,

do not elaborate on the methods by which they achieved their results [JO86].

We believe that some of the fundamental problems we face here are at the SISAL level. Its inability to perform matrix reduction except along a row vector is a serious problem. The absence of non-vector arrays exacerbates the situation by causing performance loss when stepping down an array column. Finally, OSC's current restrictions on loop fusion make alternative approaches worse than the code we presently generate. We make a brief digression here to present an example of the kind of difficulties we face in generating code that performs loop fusion properly.

The loop fusion difficulties we face with the current version of OSC are fairly substantial. First, as we saw with **primes**, OSC only fuses outer loops. Second, it will not interchange the order of nested loops to produce code that can be loop-fused, nor will it make other linear adjustments that would be obvious to a trained programmer by visual inspection. Finally, it will fuse only those loops that have identical loop boundaries *expressed in a certain way*.

As an example of the latter problem, consider two SISAL programs which compute the APL expression `(⍳n+1)+2+666` using two possible methods of code generation within APEX. OSC performs loop fusion on the first version of the program, which is expressed using SISAL iterators with implicit iteration bounds (shown in boldface):

```
%$entry=willfuse
define willfuse

function a(y1: array[integer] returns array[integer])
for y0 in y1 at i
returns array of y1[i]+2 end for
end function

function b(y1: array[integer] returns array[integer])
for y0 in y1 at i
returns array of y1[i]+666 end for
end function

function willfuse(siz: integer; returns array[integer])
b(a(for i in 0,siz returns array of i end for))
end function
```

When compiled, this SISAL code generates a single C loop that also includes constant folding for the expression `2+666`:

```
for ( ; tmp6 <= tmp2; tmp6++ ) {  /* Normal Loop */
  Plus( tmp7, tmp6, (668) );
  GathATUpd( int, tmp5, tmp7 );
  }
```

However, another version of the same program, expressed using explicit iteration bounds (also shown in boldface), is not fused by the OSC compiler:

```
%$entry=wontfuse
define wontfuse

function a(y1: array[integer] returns array[integer])
for i in array_liml(y1),array_limh(y1)
returns array of y1[i]+2 end for
end function

function b(y1: array[integer] returns array[integer])
for i in array_liml(y1),array_limh(y1)
returns array of y1[i]+666 end for
end function

function wontfuse(siz: integer; returns array[integer])
b(a(for i in 0,siz returns array of i end for))
end function
```

When compiled, this SISAL program generates three non-nested C loops and creates two extra intermediate arrays during execution:

```
  for ( ; tmp10 <= tmp2; tmp10++ ) {  /* Normal Loop */
    GathATUpd( int, tmp9, tmp10 );
    }
...
  for ( ; tmp10 <= tmp3; tmp10++ ) {  /* Normal Loop */
    OptAElm( int, tmp7, tmp11, tmp10 );
    OptPlus( tmp7, tmp7, (2) );
    GathATUpd( int, tmp9, tmp7 );
    }
...
  for ( ; tmp7 <= tmp4; tmp7++ ) {  /* Normal Loop */
    OptAElm( int, tmp10, tmp11, tmp7 );
    OptPlus( tmp10, tmp10, (666) );
    GathATUpd( int, tmp9, tmp10 );
    }
```

Although, in both coding examples, the boundaries for all three *for* loops are identical, the manner in which they are *expressed* in the second example prevents OSC from fusing the loops. The reader may be curious as to why we would even concern ourselves with the latter coding style, which appears more verbose and error-prone that the former. Our desire was to leave the door open for support of array coordinate mapping within APEX. In order to do this without modifying OSC, we had to be able to explicitly specify stride information within loops, something that is not possible with implicit iteration bounds. Having to make a choice between support for array coordinate mapping and support for loop fusion, we reluctantly chose the latter, knowing that certain benchmarks would inevitably suffer performance loss regardless of our choice.

This behavior, we note, is contrary to that implied by the OSC manual [Can92a]. Once we un-

derstood this problem, we revised the APEX code generator to produce code using the *willfuse* coding style that encourages loop fusion. As time permits, we have been replacing code generator fragments with new ones that adhere, when appropriate, to this style. Note that this technique does not solve our problem with **primes**, because we are still have the inner loop versus outer loop problem. Nonetheless, it does improve the performance of our other benchmarks.

The SISAL/OSC developers are well aware of these problems; work is in progress to improve the situation when the arrays involved are rectangular [Fit93]. However, the fruits of this labor are not yet available, so we must work with the tools we have at hand. The bad news is that these operations perform poorly today. Part of the good news is that a solution is in the works. The other part of the good news is that we can work around some of these problems. We can reach into our toolbox of APL primitives and tune applications for performance; we can also contemplate the use of phrase recognition to perform such tuning automatically. We discuss **primes** in the light of both topics in Section 5.7.

Returning to benchmark results, we note that **mconvo** suffered from a problem similar to that of **primes**. The **mconvo** convolution benchmark reshapes the trace into a matrix with as many rows as the filter has elements. The matrix is then padded with zeros and skewed with a row-by-row rotate. The convolution is then computed as a reduction of the outer product of the filter with the skewed matrix. Although visually pleasing and elegant when expressed in APL, the interpreted performance of **mconvo** is poor, because of the two large matrix temps that the interpreter creates. APEX did little better with convolution than it did with **primes**, for the same reason – inadequate loop fusion by OSC, for the reasons stated above, caused the compiled code to generate two large matrix temps.

In Section 5.5, we describe a modification to **mconvo** that resolved this performance problem. We recognize that this is, in a sense, conceding defeat in the battle against large intermediate arrays, but the war is not over – that is why Future Work was invented.

The performance of the **metaphon** benchmark disappointed us. Because the benchmark is non-iterative, it looks as if it should be able to exploit loop fusion to full advantage and thereby obtain substantially higher performance than the interpreter. However, **metaphon** compiled performance on the 486 is about 10% worse than that of the interpreter; on the RISC platforms APEX-generated code is only 1.5–2 times faster than the interpreter.

We propose several possible reasons for this poor performance. First of all, the benchmark makes extensive use of the APL *rotate* primitive. As implemented in APEX today, this will always force array copying and thereby defeat loop fusion. Second, substantial use is made of the phrase `/⍳`. This phrase is special-cased in the interpreters, but APEX currently has no special case support for it. Hence, APEX is doing a considerable amount of extra work that the interpreters are able to sidestep. Third, OSC stores one Boolean element per byte, in contrast to commercial APL interpreters, which store 8 Boolean elements in one byte of memory. This inefficient representation has serious implications for the performance of several of our benchmarks, as the OSC Boolean storage scheme requires more machine instructions to perform common APL Boolean array operations than does the dense storage scheme. For example, relational array operations and search operations can be performed a word at a time – 32 or 64

elements in parallel – with a dense storage scheme. Finally, there are still a number of code fragments remaining in APEX that inhibit loop fusion. These may also be contributing to the problem.

Another place we encountered inhibition of loop fusion in many of our benchmarks involved checking for length errors during the execution of scalar functions.[1] APEX generates code to check for length error when it is unable to make a compile-time determination of the conformability of the operands. For example, `scalar+vector` does not generate conformability checking code, nor does `vector+singleton` when the singleton is detected at compile time. However, `vector1+vector2` generates a check for length error if APEX is unable to determine at compile time that the vectors are the same length or that one is a singleton.

The presence of such checks inhibited loop fusion due to a bug in the OSC optimizer.[2] This produced a significant performance loss in some benchmarks, in spite of the fact that there are surprisingly few run-time length error checks present in APEX-generated code.[3] We were unable to locate or correct the OSC fault without substantial analysis of OSC optimizer internals, so we elected to adopt a workaround for the problem. We introduced a preprocessor directive, `CONFORM`, that optionally disables the generation of checks for length error in scalar functions. Inclusion of this directive at compile time permits loop fusion to occur properly, at the price of potentially undetected run-time length errors. Once we have a fix for the OSC bug, we will remove this workaround. All execution times reported in this chapter reflect usage of the workaround.

In spite of the problems we encountered with loop fusion, we remain convinced that it offers substantial performance benefits for many APL applications. The benefits of loop fusion are, moreover, effective regardless of coding style or array size. Loop fusion and setup cost reduction are, nonetheless, inadequate to let us reach our performance goals. We need to be able to replace generic algorithms with high-performance algorithms that let us exploit special cases arising from compile-time knowledge of data and the problem domain.

5.3 Special Case Algorithms and Array Predicates

Knowledge of sub-domain properties of a general algorithm often permits solution of the sub-domain problem in less time or space than the general problem. These algorithms, known as *special case algorithms*, are exploited, often unwittingly, by application programmers.

[1] In APL, a *scalar function* is one that is defined purely by its behavior on scalar arguments. This class includes the common arithmetic and elementary functions such as addition and exponentiation. A scalar function is extended to array arguments in an element-by-element fashion if its arrays are identical in shape. If the arrays are not the same shape, then the so-called *scalar extension* rules come into play: If one argument is a singleton – a single-element array of any rank – the singleton is extended to be the same shape as the other argument. Otherwise, the arrays are non-conformable and a *rank error* or *length error* is signaled, indicating a difference in array ranks or in length along some axis. For example, `1 2 +4 5 6` would signal a *length error* because the vector arguments were of different shapes, but neither is a singleton.

[2] The SISAL project team was notified of this bug, but they have not yet been able to develop a fix or workaround for it.

[3] Of 679 scalar function invocations in our benchmarks, only 138 of them (20%) generated run-time checks. The other 80% were removed by the extant shape-checking facilities in APEX, which had exploited shape information propagated by data flow analysis.

Programmers who write applications in C or other scalar-oriented languages are forced, by the very nature of those languages, to think about data and algorithms in very low-level terms. As a consequence, the resulting programs often naturally exploit currently known characteristics of data. Every line of code written reflects those characteristics, becoming, in effect, a special case. In APL, programmers tend to shun special case code except when circumstances force them to do otherwise, as generality may provide unexpected benefits. That same generality often leads to performance problems, because interpreters are not able to detect and exploit special cases to the same extent as can a programmer writing in a compiled lower-level language. The information required to detect a special case is often evident from a static analysis of the program, but an interpreter does not have the luxury of time to detect, propagate, and exploit such information. Hence, an interpreter is working at a disadvantage and is, therefore, usually forced to execute code that reflects the most general case.

This situation changes when we compile APL. In contrast to an interpreter, a compiler *can* take the time to perform the analysis required to detect, propagate, and exploit known characteristics of data and algorithm, then generate special-case code that offers benefits in execution time, execution space, or type of result. One way that APEX supports this form of analysis is through *array predicates*, properties that are logically associated with each array created by the program being compiled.

We developed the concept of array predicates when we were developing the data flow analyzer for APEX. We realized that detailed knowledge of array properties often permits an expert programmer to improve the run-time CPU or memory performance of an application by using algorithms that exploit those properties. Similarly, an interpreter or compiler with knowledge of array properties can make more aggressive choices of run-time algorithms. For example, knowing that array `PV` is a permutation of the first `⍴PV` integers permits `PV` to be sorted in linear time using a pigeonhole method, rather than having to use a non-linear traditional sorting algorithm. Since these array properties can be described as predicates, or assertions, made about an array, we dubbed them *array predicates*.

Although the above example only affects the run-time performance of a single primitive, other array predicates can also affect the result type of functions. This, as we shall see, may produce an effect on code generation that ripples through the remainder of the program, with concomitant effects on performance and memory requirements. Consider the result type of *represent*, APL's radix-converting function. Given integer arguments, *represent* produces a result of type integer. However, in the special case of a left argument whose value is all twos, as in the expression `(n⍴2)⊤y`, the result is the last `n` digits of the Boolean representation of the integer list `y`. If we could detect this case, we could produce a result of type Boolean, rather than integer. This would reduce the space required to store the result and might also permit references to that result to use faster algorithms, such as word-at-a-time Boolean functions. An interpreter can detect the Boolean special case at run-time at the cost of some overhead, but APEX must establish the result type for each function at compile time. Data flow analysis will provide the APEX code generator with the fact that the left argument to *represent* is an integer vector, but unless `n` has a known value (either a constant or derivable at compile time by partial evaluation), the result generated will be of type integer instead of Boolean. This is undesirable from both performance

and memory usage standpoints. If APEX could deduce, at compile time, the fact that the left argument to represent was a vector of twos, it could produce a Boolean type result and also emit more efficient code using *shift-and* rather than *modulus-subtract*. Clearly, then, if a functional array language is to compete in performance with scalar-oriented languages in which programmers code such algorithms explicitly, it must be able to extract these predicates and make effective use of them.

APEX does just this, and gets substantial performance improvements by doing so. The data flow analysis phase of APEX creates and propagates array predicates, which are then used by the APEX code generator to emit code that exploits special properties of arrays. The predicates currently supported by APEX, along with the rules for generation and propagation of predicates across function boundaries are described in detail elsewhere [Ber97]. In brief, APEX currently generates, propagates, and invalidates, as appropriate, the set of array predicates shown in Figure 5.6.

Predicate Name	Description
PV	Array is a permutation vector
PVSubset	Array is subset of a permutation vector
NoDups	Array elements are all unique
All2	Array elements are all integer 2
SortedUp	Array elements are in upgrade order
SortedDown	Array elements are in downgrade order
KnowValue	Array value is known at compile time
NonNeg	Array elements are all non-negative
Integer	Array elements are all integer-valued

Figure 5.6: Array predicates supported by APEX

We performed an experiment to determine the effect of array predicates on the performance of one application benchmark, **rle**. As part of a data compression step in data base construction, **rle** creates a run-length-encoded Boolean vector representing its time-series integer data argument. APEX originally generated a result of type integer for the application, because the expression `(width⍴2)⊤r` involving *represent*, the radix-converting APL function that turned the integer data into Boolean form, had no way to determine at compile time that the result would be Boolean. If its left argument had been a numeric constant vector consisting entirely of 2s, it could generate code operating in the Boolean domain, generating a Boolean result. This would produce a cascade effect on code generation, in which later uses of that result would also operate in the Boolean domain. Unfortunately, the left argument was not a constant, but a temporary array, whose length varied from call to call depending on the exact values of the argument. Because the information, obvious by inspection, that the left argument must be all 2s, was not available to the code generator, the code generator was forced to emit conservative code for the integer domain. This produced such a loss of performance that the APEX-generated code originally ran slower than the interpreter, which examined the left argument and dynamically made the decision to produce a Boolean result. APEX, having no such luxury, had to decide on result type at

compile time, and was forced to be conservative, producing an integer result.

While contemplating this problem, we realized that it would be possible for data flow analysis to propagate array properties, such as "All elements of this array are the non-negative integer 2," in the same way that it propagates other morphological information. We implemented APEX support for these *array predicates*, and within a day had the code generator producing special-case code that took advantage of them. We compiled **rle** with predicates disabled to generate a SISAL program, `rleinteger`, which produced a result of type integer for the *represent* function. We then recompiled **rle** with array predicates enabled to generate another SISAL program, `rleboolean`, which produced a Boolean result for the *represent* function, by the following compiler actions. The offending expression `(width⍴2)⍉r` appeared in the **rle** benchmark, with the constant array `2` generated with the predicate All2. The *reshape* expression `width⍴2` propagated that predicate to its result. The *represent* function then exploited the All2 predicate of its left argument to produce a Boolean result with predicates of Integer and NonNeg.

Figure 5.7 shows the extent to which array predicates affect the performance of the **rle** application on the 486.

Benchmark	CPU time (seconds)		APEX memory
	APL	APEX	(bytes)
`rleinteger 10000`	1.67	26.04	18325416
`rleboolean 10000`	1.67	1.35	8489096

Figure 5.7: Performance impact of array predicates

Part of the speedup came about because the *represent* primitive was able to emit more efficient code using *shift* instead of *divide*; another part came from loop fusion; the remaining part of the speedup arose from functions that made more efficient use of the now-Boolean result of *represent*. The propagation of Boolean data throughout latter parts of the benchmark offered additional performance benefits.[4] The reduction of processor time by a factor of almost twenty and of memory use by a factor of more than two makes it clear that support for predicates offer a substantial performance improvement for certain applications.

Array predicates offer substantial performance payoffs for minimal effort. Exploiting predicates in the code generator requires a certain amount of custom coding for the special case code fragment and its selection. However, once the predicates are available from data flow analysis, special cases that use them can be written as development resources become available.

The results we have just presented do not typify the magnitude of the speedups available with array predicates. They can, for instance, facilitate the use of special case code that replaces quadratic run-time algorithms with linear ones. Such special case detection permits the APL programmer to continue

[4]This cascading effect is another reason for preferring application benchmarks – kernel benchmarks are unable to observe this effect.

programming in an abstract style, while obtaining performance levels commensurate with hand-coded special cases.

At present, APEX data flow support for array predicates is fairly comprehensive, although some fine tuning still remains to be done. For example, scalar functions currently invalidate all array predicates, although it is clear that some predicates should survive some scalar functions, e.g., NonNeg should survive any scalar function that produces a Boolean result.

Addition of a new predicate to APEX is a simple matter of defining the rules by which it is created, propagated, and invalidated, then amending the data flow analysis phase of the compiler to implement those rules. Exploitation of an existing predicate is merely a matter of introducing appropriate code into the relevant part of the code generator. Although predicate exploitation is currently limited to the case described above as a proof of concept, array predicates offer a rich vein of APEX performance, to be mined as human resources become available.[5] Some of the optimizations that we envision implementing by the exploitation of array predicate information include those shown in Figure 5.8.

Optimizations with array predicates	
Optimization	Relevant Predicates
Quick hit/nohit for `x∈y`, `x⌊y`	PV, PVSubSet, MaxVal, MinVal
Instant upgrade	SortedUp, SortedDown
Upgrade by pigeonhole	MaxVal, MinVal
Instant max reduce or min reduce	MaxVal, MinVal
Fast indexof, membership	SortedUp, SortedDown, PV
Partial evaluation	KnowValue
Domain error check removal	NonNeg, Integer
Index error checking removal	NonNeg, Integer, MaxVal

Figure 5.8: Possible array predicate optimizations

5.4 Copy Avoidance

When we entered into the APEX project, we hoped that the problems of array copying in functional array languages would be addressed completely by our use of SISAL or IF1 as an intermediate language. These hopes were dashed by our realization of the full implications of SISAL's vector-of-vector, as opposed to array, design. This, combined with the scalar language semantics of SISAL, made it difficult to implement certain matrix primitives of APL efficiently, particularly those, such as take and drop, that involve the restructuring of arrays. These operations, which we hoped could be performed purely via descriptor manipulations, turned out to be the root of our most problematic performance problem.

[5] Recently, we implemented two optimizations that exploit the array predicate *PV* (permutation vector) in two benchmarks performed for a European bank. The operations `⍋PV` and `PV⌊y` were enhanced to generate linear-time algorithms. These improved the performance of these benchmarks by 40% and 2500%, respectively.

There are several reasons why these performance problems exist, including the absence of SISAL support for arrays and inadequate loop fusion in OSC. The absence of arrays in SISAL forces arrays to be stored as vectors of vectors. This, in the current version of OSC, may result in matrix rows being placed in non-contiguous memory. With such a storage scheme, it is not possible to compute the address of an array element solely from its index and the base address of the array. Hence, array coordinate mapping, which logically restructures arrays without requiring the copying of array elements, cannot be used to implement the structural functions of APL. As a result, structural operations force the copying of entire arrays, to no useful end. On matrix operations, performance also suffered because SISAL's vector-of-vectors design results in allocation of a separate descriptor for each row of a matrix. This increases memory usage and drives up memory management cost and other overheads associated with the execution of a primitive. Furthermore, even simple structural operations, which should be amenable to removal via loop fusion, were not loop-fused, because of an inadequacy in the OSC loop fusion analyzer, whereby loop bounds must match exactly for fusion to occur. We will look into this performance problem in more detail by examining the performance of **mdiv**, **mdiv2**, and **tomcatv**. Performance numbers for these benchmarks are shown in Figure 5.9.

The Jenkins' model of *domino*, APL's matrix divide primitive, was used as the basis for the **mdiv** benchmark [Jen70]. We hoped the model would provide evidence that straightforward compiled APL can compete favorably with hand-coded C. In fact, the compiled version of **mdiv** executes marginally slower on the 486 than does the interpreted version and almost four times slower than the C-coded *domino* primitive. Speedups on the RISC platforms are slightly better for APEX relative to the interpreter, but are still disappointing.

We conjectured that the poor performance of **mdiv** arose from the requirement to copy subarrays. We sought to confirm that this was indeed the case by making some modifications to **mdiv** in hopes of reducing non-productive copying of arrays. We created **mdiv2** from **mdiv** by adopting a coding style, familiar to performance-conscious programmers, that avoids superfluous data movement and coercions. The changes we made to **mdiv** in creating **mdiv2** were as follows. The type of the identity matrix created in `mmd2` as the left argument to `ls2` is double-real, rather than Boolean, eliminating a run-time coercion. The identity matrix is formed by reshape and overtake rather than by the slower, more complicated outer product. Array rows are swapped in **mdiv** pivot operations by use of the array expression `pp[i,pi]←pp[pi,i]`, necessitating the formation of two small, temporary arrays from the scalars `i` and `pi`. Replacing these by the three statements `t←pp[i]`, `pp[i]←pp[pi]`, and `pp[pi]←t` allowed those operations to be performed without generating the temps. The argument and result matrices were catenated in **mdiv**, reflecting the manner in which matrix inversion is taught in linear algebra, but introducing the need to use complicated index or take expressions to extract portions of the appropriate array. In **mdiv2**, we stored these arrays as distinct objects.

The **mdiv2** benchmark performed significantly better than **mdiv** in both interpreted and compiled environments. More importantly, APEX timings had improved even more than those of the interpreted code. This implicated array copying as a culprit in the performance problem and suggested two ap-

Performance of **mdiv 200** (seconds)			
Platform	APL	APEX	Speedup
486	89.22	92.82	0.96
RS/6000	135.07	66.43	2.03
SUN	88.64	73.50	1.21

Performance of **mdiv2 200** (seconds)			
Platform	APL	APEX	Speedup
486	83.85	67.29	1.25
RS/6000	116.80	28.10	4.16
SUN	81.47	39.53	2.06

Performance of **tomcatv 257** (seconds)			
Platform	APL	APEX	Speedup
486	920.08	310.36	2.96
RS/6000	918.25	180.17	5.10
SUN	716.33	300.70	2.38

Figure 5.9: Performance impact of limited copy avoidance

proaches to correcting the larger problem: make array copying faster or stop copying arrays. We examined the C code generated by OSC for array copying in **mdiv2**. As that code appeared to be near-optimal, faster array copying by array copy improvements was not in the cards. Another way to eliminate many of these data movement operations and to raise performance levels to acceptable values is to introduce array coordinate mapping into OSC, as discussed in Section 2.2.3. Regretably, such an enhancement is beyond the scope of this thesis.

The **tomcatv** benchmark suffered from the same array copying problem as **mdiv**. We observed speedup ratios in the 2.4–5 range. This is good compared to the interpreted code, but inadequate, as we shall see in the next chapter, when compared to compiled scalar languages. Part of the fault lies in the `compmesh` function, which computes step-2 first differences along both axes of two arrays, discarding the edge values. These and related computations introduce a total of 40 multi-axis drop operations and 4 matrix rotate operations, each requiring the copying of large (257x257) arrays. Copying is wasted effort, as it does not contribute to the result of a computation. We shall return to this issue shortly in the discussion of the performance of APEX versus FORTRAN, wherein we will see just how much effort is wasted. If the APL code was rewritten to use a scalar-oriented coding style, no such copying would be required, and we would achieve better performance. However, APL's clarity of expression would be lost.

5.5 Extended Language Features

Most of the benchmarks we have presented here were written in vintage-1985 ISO Standard APL, to ensure cross-platform portability and to present a level playing field for all parties concerned. However, for higher efficiency, we must move beyond the world of Standard APL and look at features in the Extended APL Standard that let us make more effective use of the language [Int93]. In this section, we will revisit some applications, looking at the performance improvements obtainable by the use of two such language features, the *cut* conjunction and the *rank* conjunction [Ber87, BB93].

The adverbs and conjunctions, or operators, of APL are the control elements by which array elements are brought together for computation. The properties of the APL conjunctions for inner and outer product are well-known, as are those of the parallel prefix adverbs, *scan* and *reduce*. These simple, yet powerful, facilities perform the bulk of the computation in APL that cannot be handled by indexing or scalar functions on arrays. Nonetheless, they do not support a variety of commonly required ways of handling data, such as a row at a time, or in a tiled manner across a matrix.

In the late 1970s, a number of researchers in the APL language design community, including K.E. Iverson, A.T. Whitney, and the author, recognizing these shortcomings of APL, designed language facilities that we believed would address them in a general and consistent manner. The author implemented support for what is now known as *function rank* and the *rank* conjunction in SHARP APL in 1983; the *cut* conjunction followed soon thereafter, implemented by Greg Bezoff [Ber87, BB93, BIM+83].[6] These conjunctions simplified the expression of many computations and often improved their performance by large margins.

In the course of examining APEX benchmarks, it became apparent that several of them could take advantage of these language extensions. We now present the results of several experiments we performed to evaluate the utility of the *cut* and *rank* conjunctions in a compiled environment. The results of these experiments are given in Figure 5.10. Several of the APEX benchmarks use language extensions that are not yet available on our testbed APL interpreters. Where that is the case, we show, in italics, the best APL interpreter time available for that benchmark. Specifically, none of our testbed interpreters support *the cut conjunction*, and the 486 interpreter does not support the *rank conjunction*.

The *cut* conjunction, denoted ⍤ in APL, is used to perform moving window operations across arrays, for applications such as moving averages, string search, and convolution. The most commonly used *cut* partitions an array argument into several non-overlapping segments under control of an argument such as a Boolean partition vector, then independently applies a specified operand function to each such segment produced. The final result is formed by lamination of the individual results. The variant of *cut* that interests us here is the one that partitions the array into overlapping segments before applying the operand. To demonstrate the utility of *cut*, consider such a moving-window *cut* that applies an identity function (`Id`) to overlapping windows of width 3 of the text string `'abcdef'`. The following expression, using our extended APL2 *window reduction* notation for *cut*, merely demonstrates the array segments to which an arbitrary operand is applied:

[6] The *rank* conjunction is now part of the ISO Extended APL Standard [Int93].

Performance of **logd2 500000** (seconds)			
Platform	APL	APEX	Speedup
486	6.68	1.79	3.73
RS/6000	13.44	1.10	12.22
SUN	7.66	1.43	5.36

Performance of **logd3 500000** (seconds)			
Platform	APL	APEX	Speedup
486	6.68	1.53	4.37
RS/6000	*13.44*	1.00	13.44
SUN	*7.66*	1.17	6.55

Performance of **100 mconvo 9000** (seconds)			
Platform	APL	APEX	Speedup
486	3.65	2.07	1.76
RS/6000	6.84	0.60	11.40
SUN	3.39	1.13	3.00

Performance of **100 mconvred 9000** (seconds)			
Platform	APL	APEX	Speedup
486	*3.65*	0.49	7.45
RS/6000	*6.84*	0.17	40.24
SUN	*3.39*	0.34	9.97

Performance of **nmo2 200** (seconds)			
Platform	APL	APEX	Speedup
486	4.67	3.50	1.33
RS/6000	6.46	2.40	2.69
SUN	4.46	2.33	1.91

Performance of **nmo2r 200** (seconds)			
Platform	APL	APEX	Speedup
486	*4.67*	2.38	1.96
RS/6000	5.91	1.63	3.63
SUN	3.88	1.30	2.98

Performance of **primes 2200** (seconds)			
Platform	APL	APEX	Speedup
486	12.54	6.84	1.83
RS/6000	17.90	4.73	3.78
SUN	15.50	6.03	2.57

Performance of **primes2 2200** (seconds)			
Platform	APL	APEX	Speedup
486	*12.54*	4.62	2.71
RS/6000	15.92	4.77	3.34
SUN	15.40	5.67	2.72

Performance of **mdiv2 200** (seconds)			
Platform	APL	APEX	Speedup
486	83.85	67.29	1.25
RS/6000	116.80	28.10	4.16
SUN	81.47	39.53	2.06

Performance of **mdiv2r 200** (seconds)			
Platform	APL	APEX	Speedup
486	*83.55*	64.46	1.30
RS/6000	116.80	24.87	4.70
SUN	81.47	37.03	2.20

Figure 5.10: Performance impact of extended language features

```
      3 Id/'abcdef'
abc
bcd
cde
def
```

If we use a more meaningful operand function such as `∧.=&'cde'` to perform an inner product with a fixed argument, we can perform a string search:

```
      3 ∧.=&'cde'/'abcdef'
0 0 1 0
```

Similarly, if we replace the inner product (`∧.=`) in the above example by the linear algebra inner product `+.×` and give it a fixed argument of a numeric filter, we obtain a 1-D convolution function. These examples should give some indication of the power and utility of the *cut* conjunction, and justification for its support in the APEX compiler.

We enhanced APEX to support a variant of the SHARP APL and J *cut* conjunction and the APL2 *dyadic reduce* adverb [BB93, IBM94, Ive96]. The syntax is that of APL2 dyadic reduction; the syntax and semantics are reflected in the Extended ISO APL Standard. We feel, as did the designers of the J *cut*, that the APL2 definition of dyadic reduction is deficient, inasmuch as it only provides the capability of a reduction, whereas the *cut* enhancement permits application to a monadic function, thereby facilitating operations such as convolution. Our implementation is a consistent extension of the ISO definition, offering both capabilities. Specifically, the operand, if monadic, is applied to each window of the argument. If the operand is ambivalent or dyadic, it is applied as a reduction, as it is in APL2. We shall now examine the effect of *cut* on the performance of **logd** and **mconvo**.

We created **logd3** by modifying the `DIFF2` function of **logd2** to perform the first difference operation by use of a moving window reduction of width two.[7] This was implemented using the *cut* conjunction: `RES←2 -/SIG`. It was not possible to time the interpreted version of **logd3**, as our extension is not yet supported on the testbed interpreters. Therefore, we used **logd2** interpreter timings against **logd3** APEX timings. The compiled version of **logd3** ran substantially faster than **logd2** and **logd** because interprocedural loop fusion was now able to fuse the `DIFF3` function's code into the major loop. This eliminated some loop control code and replaced array copy operations with operations on scalars. The net result was a speedup of 4.4–13.2 over **logd2**, the best interpreted code.

Since the *cut* conjunction is eminently suitable to the task of performing convolution, we also rewrote the **mconvo** benchmark to use it, creating the **mconvred** benchmark. This simplified the benchmark code considerably and also provided a substantial improvement in speed, with the compiled code running 7.5–40.2 times faster than the interpreted code. As with **logd3**, utilization of *cut* provided loop fusion and eliminated the large array-valued temps that plague **mconvo**, bringing **mconvred** into the level of reasonable performance.

The *rank* conjunction has long proven to be of value in interpreted APL applications by providing a simple, yet general, way of specifying how subarrays are to be combined in the presence of a function.

[7] In Section 5.7, we discuss the modification we made to **logd** to create **logd2**.

The *rank* conjunction, denoted ⍤ in APL and ” in J, may be thought of as specifying a pair of loop controls for combining two arrays under the control of a dyadic function. For example, the expression `v+⍤0 1 m` specifies that rank-0 elements (scalars) from `v` and rank-1 elements (rows) from `m` are to be combined by addition. This effectively adds the vector `v` to each column of the matrix `m`. Similarly, `v+⍤1 1 m` selects vectors from both sides, adding the vector `v` to each row of the matrix `m`. The rank conjunction permits such operations, and many others, to be performed without recourse to restructuring primitives such as reshape and transpose. This eliminates certain array copying operations that would otherwise be required to achieve array conformability.

In our experiments, we replaced complicated expressions in several APL applications with semantically equivalent *rank* expressions. This presented us with a minor problem because, although the SAX interpreter supports *rank*, the 486 interpreter does not. We were, therefore, unable to time the interpreted *rank* version on the 486, so we used the unmodified 486 code timing for that particular comparison. We performed these experiments on **nmo2**, **primes**, and **mdiv2**. In all cases, the use of the *rank* conjunction reduced the execution time under APEX and under the SAX interpreter.

The compiled performance of **nmo2** was less than spectacular, being just 1.3–2.7 times faster than the interpreted code. This poor performance stems from the `index` subfunction, which performs a large amount of data movement to index elements from an array. We first encountered this bottleneck during the ACORN project, where we replaced the `index` function by the *from with rank* expression `x{⍤1 y` to good effect. We performed the same transformation here, creating the **nmo2r** benchmark. We also replaced several other expressions with simpler ones involving the *rank* conjunction. This boosted performance for the application to between 2–3.6 over that of interpreted code, acceptable for such a numerically intensive application.

We made corresponding changes to **primes**, creating **primes2**, to achieve the effect of a transposed outer product, allowing us to use a last-axis, rather than first-axis, reduction. This resulted in a speedup ratio of 2.7–3.3. Similar changes produced a speedup ratio of 1.3–4.7 for **mdiv2r**, reflecting the computational dominance of inner-product within matrix inverse.

5.6 Inter-Language Calls

Being a compiled language, APEX offers the APL programmer facilities, including inter-language calls and a macro preprocessor, that are not readily available to the user of an interpreter. These can simplify maintenance and development work; they can also improve performance by a substantial margin. As an example of the latter, we will consider the effect of rewriting the **crc** benchmark to utilize the OSC intrinsic functions *shiftl*, *shiftr*, and *xor*.

Many computer languages offer facilities for performing bit-wise logical operations on scalars, thereby providing the programmer with direct access to the processor’s ALU. The APL language, by contrast, offers no such access because of its abstract treatment of numeric data. The APL language definition is silent on the existential issue of computer-oriented data types - word, float, double-precision

– for numeric data. In APL, there are simply numbers; the implementation is responsible for treating them in a sensible manner to effect maximum precision, efficiency, and accuracy in computation.

Typically, implementations of APL support at least four types of numeric data: Boolean, integer, double-precision real, and double-precision complex. Since the mode of storage is not specified, the actions of Boolean operations such as *exclusive or* and *shift* are undefined in APL, except for Boolean data. For Booleans, *exclusive or* can be written as *not equal* and the logical shifts can be written, albeit clumsily, as phrases using *take* and *drop*. There are no corresponding mappings for the integers or other data types. Therefore, processor ALU operations that are naturally expressed as Boolean functions on the internal representation of integers or characters in other languages have no direct cognate in APL. Instead, the APL programmer must invoke a generic, heavy-duty, radix-changing function – the *represent* primitive – to convert an integer scalar to a Boolean vector. The Boolean vector can then be manipulated as desired with reasonable efficiency, after which another radix-changing operation, *base value*, is required to convert the Boolean vector back into an integer scalar.

All of these operations are time-consuming, even when a compiler is able to speed up the radix changing functions through special case detection, because the definition of the operation on non-scalar arguments involves a bit-wise transpose of the result. The performance problem is exacerbated by OSC's storage of Boolean data in byte-wise, rather than bit-wise form. In the context of the **crc** benchmark, these radix-changing operations are the main reason that a highly iterative benchmark executes only 6–12 times faster when compiled than when interpreted.

We realized that the inter-language call facility in OSC [Can92a] could be used to advantage in this benchmark by letting the APL programmer work in the integer domain and letting OSC intrinsics handle the work in the Boolean domain. Since we have not yet defined a formal mechanism to support the inter-language call facility, we will describe the manual approach we used in conducting our experiment. On the assumption that a set of APL library functions with semantics identical to those of the OSC intrinsics would be a necessary first step, we wrote simple APL library functions to perform *xor*, *shiftl*, and *shiftr* on integer scalar arguments. These functions allow an APL programmer working in an interpretive environment to write and debug applications that will eventually use OSC intrinsics. We then rewrote the **crc** benchmark to use these library functions instead of the radix-changing functions already present, thereby creating **crc2**. We then compiled **crc2** to SISAL and manually replaced the APL library routine calls with invocations of the OSC intrinsics.[8]

Figure 5.11 presents the timings for **crc2**. Because the APL versions of the library functions, of necessity, embed the scalar-oriented radix-changers inside them for the interpreter's use, this intermediate step results in slightly worse interpreted performance than the original code. For the sake of fairness, the APL interpreter timings in the figure are, therefore, for **crc** rather than for **crc2**.

The **crc2** code, using the intrinsics, ran substantially faster than the original code. Moreover, of the 0.76 seconds execution time on the 486, 0.68 seconds were consumed by initialization code that

[8]Obviously, a production version of this facility would identify such library routines and perform the replacements automatically .

Performance of **crc2 50000** (seconds)			
Platform	APL	APEX	Speedup
486	35.10	0.76	46.49
RS/6000	59.13	0.60	98.56
SUN	26.54	0.47	56.87

Figure 5.11: Performance impact of inter-language calls

built the CRC table. This initialization time could be completely eliminated with partial evaluation during compilation, as the data involved are all compile-time constants. This would give performance characteristics similar to those we observed in the other highly iterative benchmarks. We have not explored the inter-language call facility in great detail, but it is apparent that it will offer substantial benefits to the APL programmer who uses APEX.

5.7 Tuning

Benchmarking attracts two types of people: those who would sell their grandmother for a 20% performance boost and those who consider their favorite benchmark to be immutably cast in concrete. The former group, strictly results-oriented, feels free to rewrite benchmarks to any extent necessary in order to achieve maximum performance. The latter group feels that benchmarks are inviolate works of art to be displayed in museum cases. Although both viewpoints have their place, we sit, like the mugwump, firmly on the fence between the two. Having presented the museum piece results in earlier sections of this chapter, we will now look into the effect of tuning them to improve their efficiency.

While studying the loop fusion problem in **logd**, we noted that extra work is done when computing the first difference. The catenate, drop, and rotate can be replaced by a catenate and a drop, to create the **logd2** benchmark. Since these structural functions usually induce array copying in both interpreted and compiled environments, we expected that this change would improve performance overall. Indeed, reduction of copy operations had precisely this effect, except on the interpreted code for the RS/6000, which ran marginally slower. Figure 5.12 shows that, on the 486, **logd2** executed 1.13 times faster than **logd** in an interpreted environment, and 1.15 times faster in a compiled environment.[9] The net result is that the compiled version of **logd2** is 4–12 times faster than the interpreted version of **logd**, and 3.7–12 times faster than the interpreted version of **logd2**.

These performance differences often arise because the toolbox of APL primitives is large enough that a given problem may educe several isomorphic solutions. Of these, the choice of algorithm used by the programmer to solve that problem may be decided on the basis of esthetics, maintainability, readability, or naivity. In production environments, the choice is frequently dictated by a programmer's

[9]This magnification of effort, wherein a performance improvement to interpreted APL code resulted in a larger improvement when compiled, was also noted by Bates, although he was not specific about the form of his code modifications.

Performance of **logd 500000** (seconds)			
Platform	APL	APEX	Speedup
486	7.56	2.06	3.67
RS/6000	13.18	1.20	10.98
SUN	8.01	1.50	5.34

Performance of **logd2 500000** (seconds)			
Platform	APL	APEX	Speedup
486	6.68	1.79	3.73
RS/6000	13.44	1.10	12.22
SUN	7.66	1.43	5.36

Performance of **dtb 30000 150** (seconds)			
Platform	APL	APEX	Speedup
486	6.61	8.01	0.83
RS/6000	3.19	7.00	0.46
SUN	3.32	4.07	0.82

Performance of **dtb2 30000 150** (seconds)			
Platform	APL	APEX	Speedup
486	4.01	0.31	12.94
RS/6000	2.83	0.30	9.43
SUN	5.42	0.13	41.69

Figure 5.12: Performance impact of tuning

perception of the relative performance of those various solutions on the system being used at the time. For example, early work by L.M. Breed and others on reshape and dynamic compilation of scalar functions and reductions in SHARP APL made the expression +/m×(⍴m)⍴v several times faster than the equivalent m+.×v [Ive73]. Expert APL programmers, obsessed by performance concerns, began to use the former phrase in lieu of the latter, in spite of its reduced clarity. Several years later, the author and D.B. Allen used similar compilation techniques and the CDC STAR inner-product algorithm to design and implement a dynamic compiler for inner product, implemented in the SHARP APL interpreter. This tilted the balance the other way, so that inner products were now faster than the now-popular reshape phrase; programmers soon switched back to the expression using inner product.[10] This sort of performance flip-flop is an on-going part of the interpreter experience. Such performance imbalances among isomorphic expressions also appear in APEX, as we shall now see.

The compiled form of **primes** executed slowly because the order of data generation differed from the order of use, thereby defeating loop fusion and introducing significant overhead involving maintenance of array row descriptors. A similar problem occurred with the **dtb** benchmark. If we examine **dtb**, we realize that the expression ∨⌿' '≠m is isomorphic to ' '∨.≠m. Therefore, we can trivially rewrite **dtb** to use the latter expression, producing **dtb2**, and letting us sidestep the generate-use dichotomy described in Section 2.2.2. Because the generation and consumption of the array elements both occur within the bounds of the inner-product conjunction, the code generator is able to avoid building large array-valued intermediate results. As Figure 5.12 shows, this simple change has a substantial impact on performance, as the APEX-generated code is now significantly faster than the interpreted code.

But what of the application programmers? Are we to leave them to guess which of several isomorphic expressions offer the best performance on a specific platform today? Do we expect them to retune their applications for each compiler release in order to exploit the hottest code? This is unrealistic, yet, until now, they had no other choice. The use of a compiler offers a way around this problem, at least to some degree. A compiler could recognize idiomatic phrases in a program and replace them according

[10] These flip-flops in relative performance make it possible to carbon-date many APL applications based on the intersection of coding styles that were used to write the application.

to a dictionary of isomorphisms. We are working to create a simple phrase recognizer to perform this function, but it remains incomplete at present.

5.8 Summary

In this chapter, we have discussed several code improvements that result in significant performance gains for our benchmark suite. These improvements include the use of Extended APL language features, exploitation of array predicates and OSC compiler features, and application code tuning. Since those levels of performance are not reflected in the original benchmark figures presented at the beginning of the chapter, we present a summary of them in Figure 5.13 and Figure 5.14. Figure 5.15 summarizes the facilities of APEX that materially affected the performance of our benchmarks. In cases where tuning of a benchmark has also resulted in superior performance for the interpreted APL code, we have used those improved performance numbers. The benchmark figures we presented in this chapter validate most of the claims we made for APEX. Specifically, compared to interpreted APL, APEX achieves excellent performance on scalar-dominated codes and on array codes where setup cost reduction and loop fusion are effective. The employment of special case algorithms, array predicates, compiler features, and language extensions have all been shown to play a significant role in improved performance.

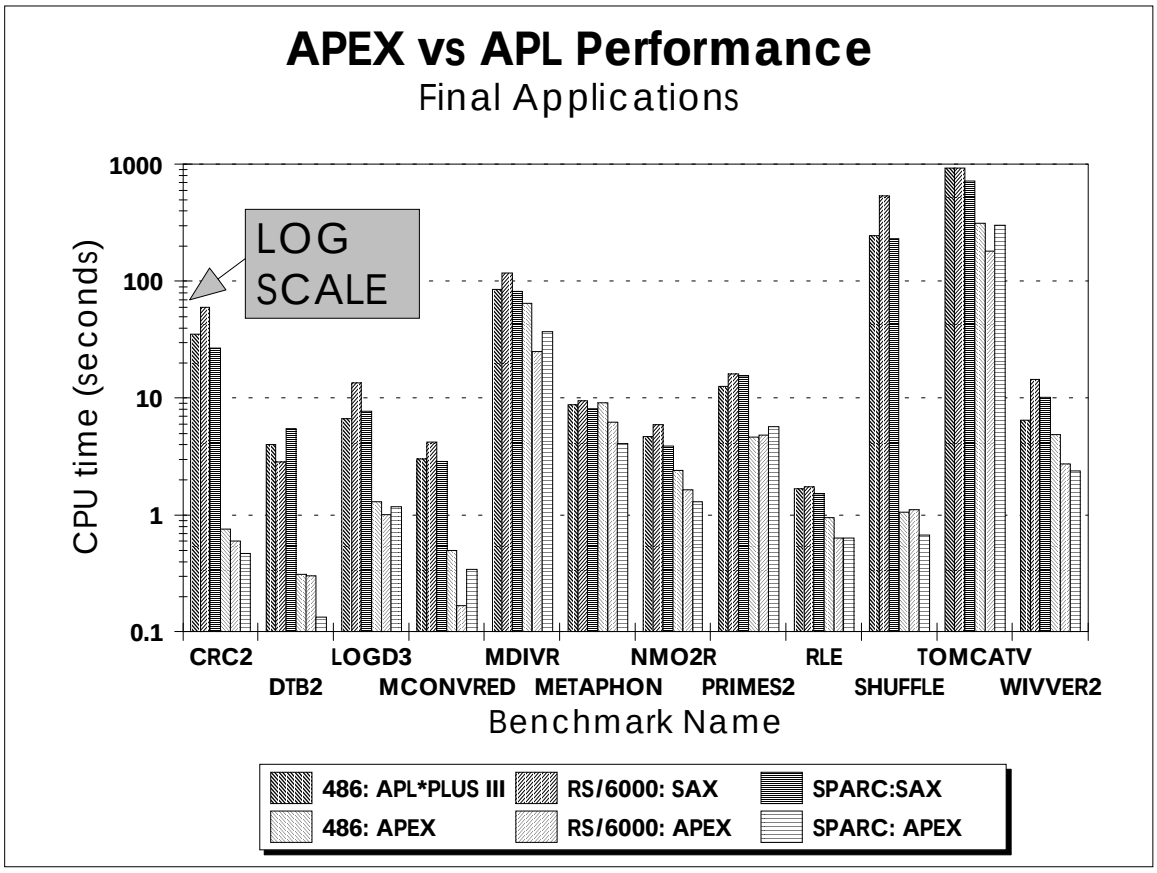

Figure 5.13: Final absolute application performance of APL vs APEX

The challenges remaining will become most apparent in the next chapter, when we look at FORTRAN applications. Some of the challenges arise from deficiencies in OSC; other from deficiencies in APEX. In OSC, support for arrays and reduced array copying are the most glaring problems to be solved, with inadequate loop fusion, excessive descriptor operations, and absence of one-bit Boolean support running close behind. In APEX, the absence of support for universal selectors, the absence of a phrase recognizer, and support for arithmetic progression vectors are significant contributors to the performance problem. Correction of these problems would enhance the performance of most members of the benchmark suite.

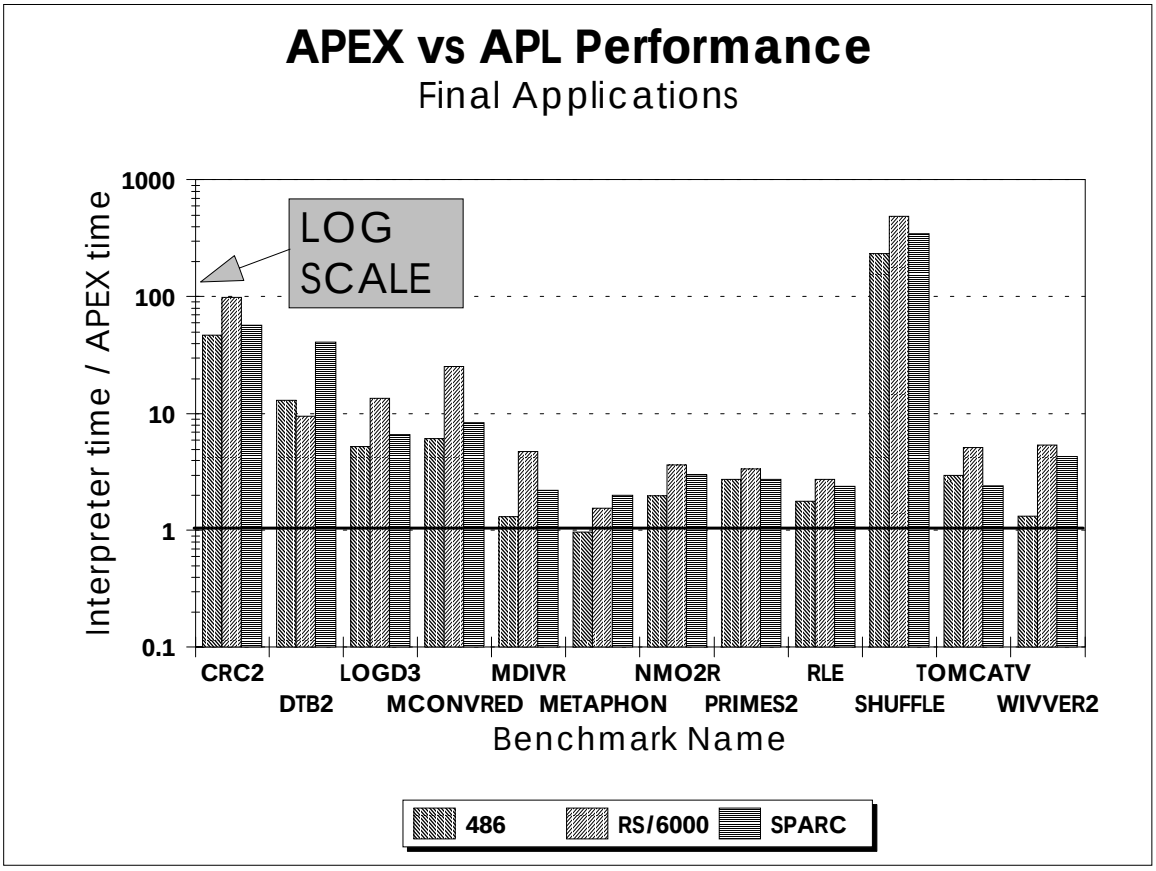

Figure 5.14: Final relative application performance of APL vs APEX

Benchmark	APEX facility							
	setup	loop fusion	array predicates	special-case algorithms	copy avoidance	extended language features	inter-language calls	tuning
crc	●	●						
crc2	●	●	●				●	
dtb								
dtb2		●						●
logd		●			●			
logd2		●			●			●
logd3		●			●	●		●
mconvo								
mconvred		●				●		●
mdiv	●	●						
mdiv2	●	●			●		●	
mdivr	●	●			●		●	
metaphon		●						
nmo		●						
nmo2		●						●
nmo2r		●				●		●
primes								
primes2		●				●		●
rle	●							
shuffle	●				●			
tomcatv	●	●						
wivver		●			●			

Figure 5.15: APEX facilities affecting APL benchmark performance

Chapter 6

Performance of Compiled APL and Compiled FORTRAN

Since one of our design goals was to make the performance of APL competitive with that of FORTRAN, we conducted a number of experiments to determine if we had achieved that goal. We found that compiled APL kernels and applications, particularly if assisted by the use of Extended APL language features, can approach and sometimes exceed the performance of FORTRAN, but that the array copying problem described in Chapter 5 still remains a major hurdle for some matrix-based applications. We will now compare the performance of compiled APL to that of FORTRAN 77 for several application and kernel benchmarks.

6.1 Application Performance

FORTRAN is a traditional tool for signal processing and intensive numerical computation. Given this, it makes sense to use such programs to compare the performance of compiled APL to that of FORTRAN. Thus, we now look into the performance of two signal processing applications, **logd3** and **mconvred**, and a mesh computation program, **tomcatv**. As with the interpreter performance section, the reader may find it useful to refer to Figure 6.1 and Figure 6.2 for summary diagrams of the absolute and relative performance of the benchmarks to be discussed. We do not report RS/6000 performance, as we did not have access to a FORTRAN compiler for this platform.

For **logd3** and **mconvred**, we wrote straightforward FORTRAN, using the best algorithms we could devise to maximize performance. We did *not* utilize application-specific library subroutines, such as the BLAS, as they would not reflect the relative performance of the languages and their compilers. Specifically, the inter-language call capability of OSC would level that part of the playing field – APEX-generated code could invoke them just as easily as FORTRAN could. We used the SPEC CFP92 version of **tomcatv** for the FORTRAN side of this benchmark and wrote an APL version of it for the other side.

The performance of **logd3** against FORTRAN, shown in Figure 6.3, was adequate, with APEX-

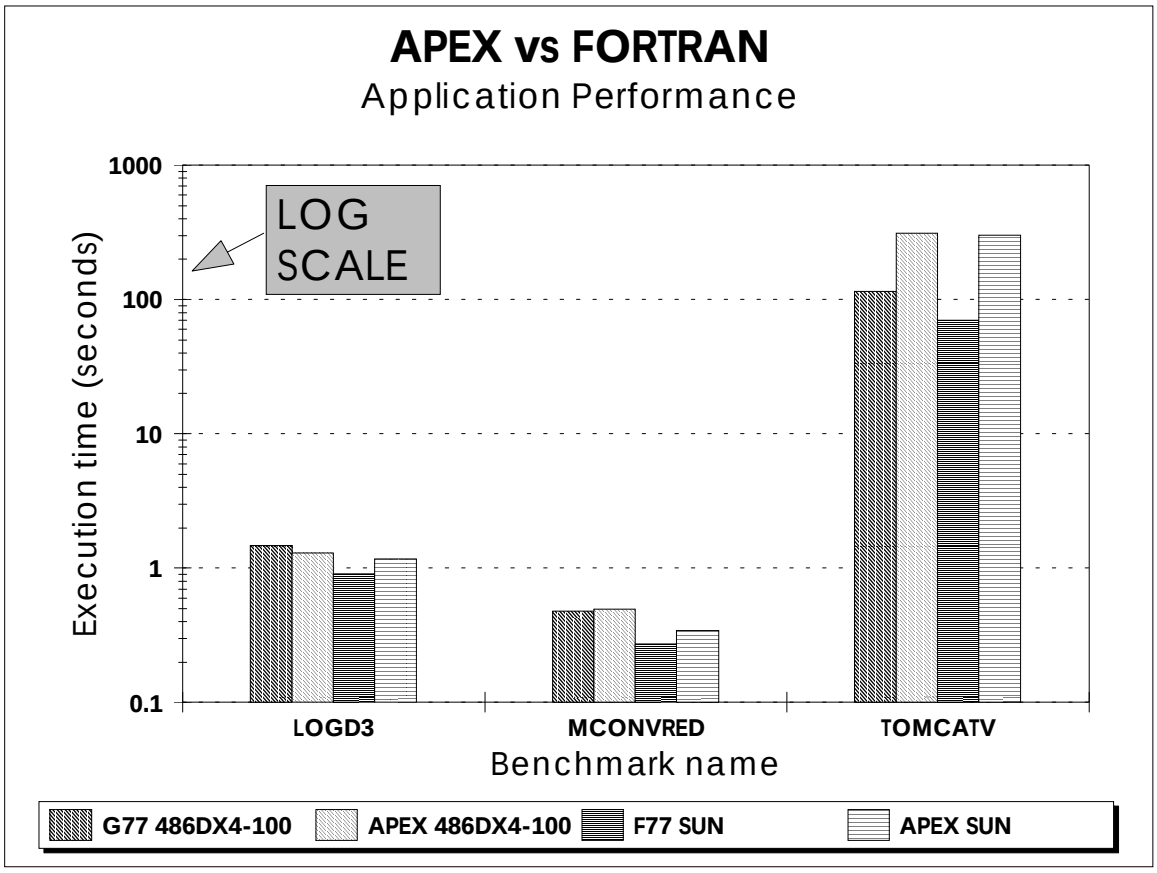

Figure 6.1: Absolute application performance of FORTRAN vs APEX

generated code beating hand-coded FORTRAN on the 486, but still lagging behind on the SUN.

We measured the performance of a convolution written in FORTRAN against the best of the APEX-generated convolutions, **mconvred**. Despite the ability of native SISAL to outperform FORTRAN on convolution [Feo95], we found that we were not quite able to match that speed. Code generated by APEX code executed one-dimensional convolutions slightly slower than FORTRAN, as shown in Figure 6.3.

As discussed earlier, the APEX version of **tomcatv** gave acceptable, but not outstanding performance against interpreted APL, with the APEX-generated code running 2.38–5.10 times faster. Against FORTRAN, however, the performance of APEX-generated code was extremely poor, as Figure 6.3 shows. As noted earlier, array copying operations are dominating the execution scene, bringing the speedup factor down to the 0.23–0.37 level – several times slower than FORTRAN.

We contemplated using *cut* and *rank* to improve the performance of **tomcatv**, particularly in `compmesh`. The expressions `3 ⌽/x` and `3 ⌽[1] x` could be applied within `compmesh` to reduce the amount of array copying performed by that function. However, code profiling showed that, as `compmesh` accounts for about half of the execution time of **tomcatv**, the maximum theoretical performance improvement

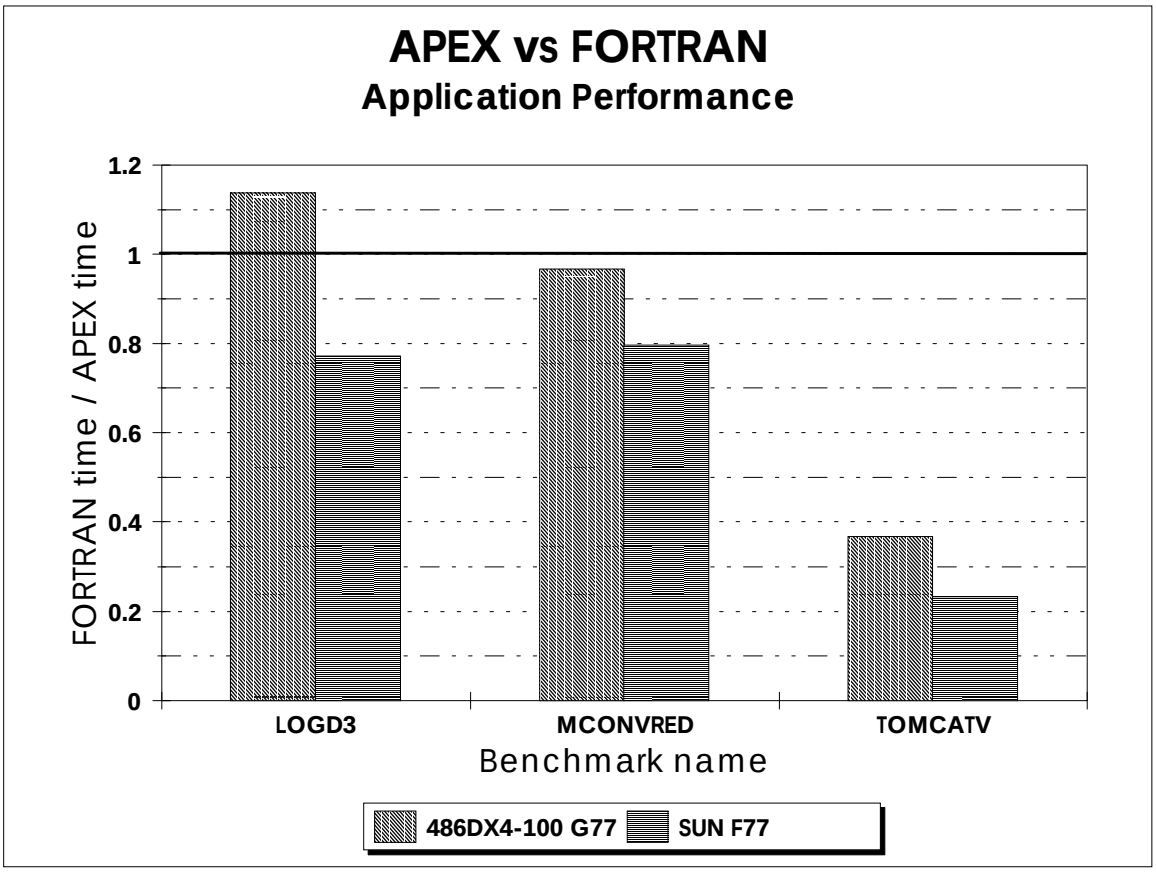

Figure 6.2: Relative application performance of FORTRAN vs APEX

here would be a factor of about two. Since this is still considerably slower than FORTRAN, it is apparent that other performance barriers also stand in our way.

6.2 Kernel Performance

We are of the opinion that kernel benchmarks, although extremely useful for exploratory and diagnostic work, are of little value in predicting the relative performance of compiled versus interpreted applications. We offer two reasons for this claim. First, global and interprocedural optimization may give compiled code additional performance advantages that are not apparent at the level of a kernel. For example, if the **logd** benchmark were decomposed into its constituent primitives and the performance of each such primitive measured as a distinct kernel, we would come away with the impression that compiled APL had little to offer, in terms of performance, against an interpreter. The reality, however, is that the compiled application executes significantly faster than interpreted code, due to cross-primitive performance boosts arising from compiler optimizations such as loop fusion.

A second reason for preferring applications over kernels is that the dominant computational features

Performance of **logd3 500000** (in seconds)			
Platform	FORTRAN	APEX	Speedup
486	1.47	1.29	1.14
SUN	0.90	1.17	0.77

Performance of **100 mconvred 9000** (in seconds)			
Platform	FORTRAN	APEX	Speedup
486	0.48	0.49	0.97
SUN	0.27	0.34	0.80

Performance of **tomcatv 257** (in seconds)			
Platform	FORTRAN	APEX	Speedup
486	114.06	310.36	0.23
SUN	69.83	300.70	0.37

Figure 6.3: Performance of FORTRAN applications against APEX

of applications are usually difficult to determine except empirically. Prediction of the performance of an application based on appeal to the performance characteristics of kernels is unrealistic except for the simplest applications. This suggests that we should focus our attention on applications and known application hot-spots, rather than on primitive and synthetic kernels. Nonetheless, since kernels may highlight salient aspects of performance, we now turn to them in the light of FORTRAN against APEX-generated code.

In the early days of APEX development, we were optimistic about the performance of APEX against FORTRAN, largely because of the results we observed in kernel benchmarks. These numbers, displayed in Figure 6.4 and Figure 6.5, show that APEX does an excellent job of competing with FORTRAN on most of the kernels, often outperforming FORTRAN by a substantial margin. We will now examine these results in some detail in order to shed light on their potential or known causes.

We performed experiments on two sets of kernel codes. One set was highly iterative and scalar-dominated, reflecting a problem domain in which FORTRAN is generally considered to perform excellently and APL considered to perform poorly. The other set was an inner-product suite, in which FORTRAN and APL are considered to be roughly equivalent, as the benchmarks operate on large arrays and are, as expressed in APL, non-iterative.

6.2.1 Scalar-dominated Kernels

We chose several highly iterative, scalar-dominated codes to examine the performance of compiled APL in a domain where FORTRAN traditionally does well and interpreted APL does very poorly. The benchmarks, **loopfs**, **loopis**, and **prd** are ones in which we would expect FORTRAN to be operating at peak efficiency. They are dominated by operations on scalars and long vectors.

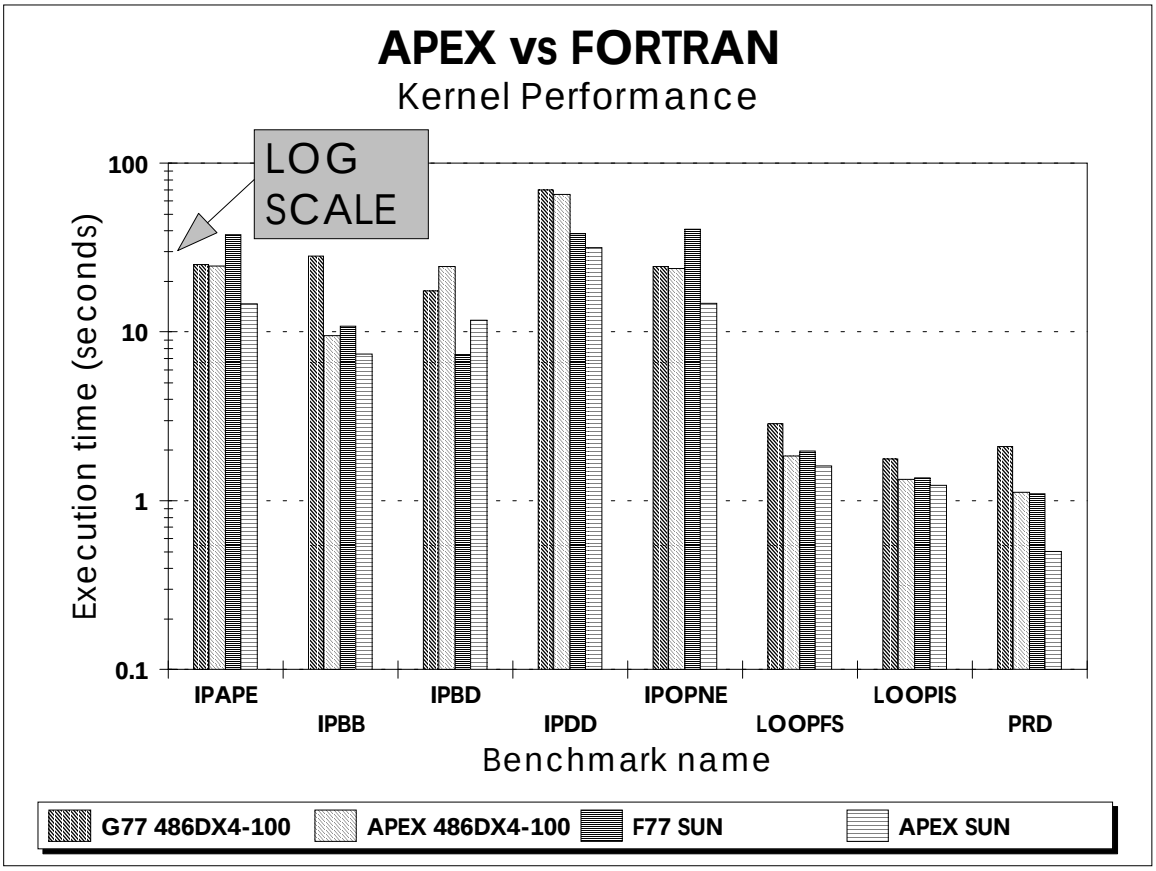

Figure 6.4: Absolute kernel performance of FORTRAN vs APEX

The **loopfs**, **loopis**, and **prd** kernels compute the sum of the first N integers. The former two loop in the manner of FORTRAN or C, on integer and double-precision scalars, respectively; the latter employs the APL expression `+/⍳N`. Apart from their simplicity, we chose these benchmarks because we were interested in the impact of loop fusion on the performance of **prd**. In addition, the former two benchmarks might offer, as does **shuffle**, insight into the performance of highly iterative APL programs. Finally, we wanted to determine the extent by which compilation narrowed the performance gap between iterative and non-iterative algorithms.

Given that these benchmarks reflect FORTRAN at its best, it is somewhat surprising to see that APEX-generated code is uniformly able to outperform FORTRAN on all three benchmarks, as shown in Figure 6.6. Loop fusion is responsible for the excellent performance level we observe in **prd**. The effect of loop fusion here is remarkable, as the benchmark executes with only 24 bytes of space required for data storage, regardless of the value of `N`. This sort of optimization can be achieved in APL interpreters by introduction of arithmetic progression vectors [Ive73], but we have obtained much of their benefit without having to introduce another primitive data type. On the 486, the poor relative performance of FORTRAN in these benchmarks stems from g77-generated register spill code within the inner loop. The

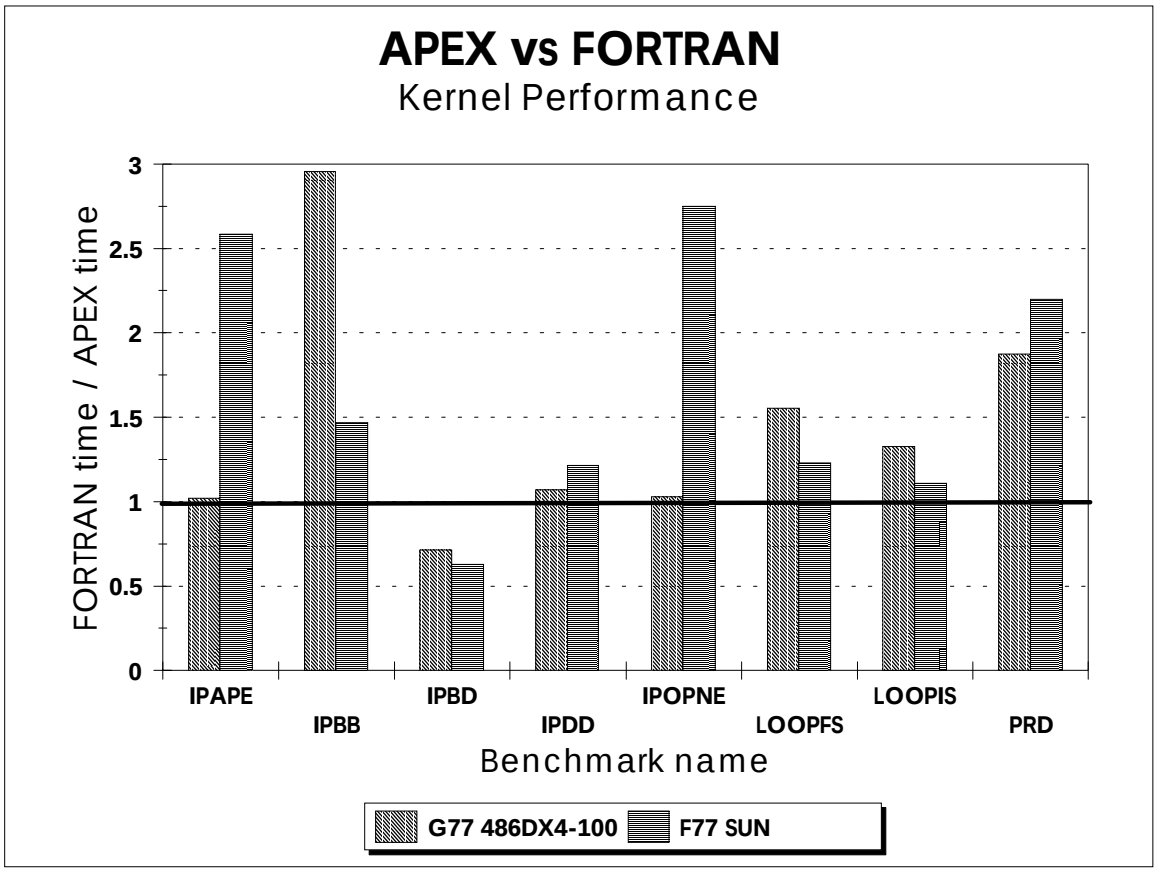

Figure 6.5: Relative kernel performance of FORTRAN vs APEX

486 gcc compiler used by APEX does not generate any spill code for these benchmarks.

6.2.2 Inner-Product Kernels

The other family of kernels we investigated was an inner-product suite, consisting of the benchmarks **ipape**, **ipopne**, **ipbb**, **ipbd**, and **ipdd**. We required an extensive set of benchmarks to properly characterize the performance of APEX-generated inner-product codes. This is because the APEX code generator exploits its knowledge of an inner-product's operands and arguments to emit code for one of three different inner-product algorithms, dubbed *QuickStop*, *STAR*, and *TransposeRight*. An explanation of the performance of these algorithms will be discussed within the context of the relevant inner-product benchmarks. Two of these, **ipape** and **ipopne**, operate on character data and three others, **ipbb**, **ipbd**, and **ipdd**, operate on numeric data.

The **ipape** benchmark (*Inner-Product And Point Equal*) performs the $\wedge.=$ inner product of an order-N character matrix with its transpose. The $\wedge.=$ derived function and its companion $\vee.\neq$, represented by the **ipopne** benchmark (*Inner-Product Or Point Not-Equal*), are frequently used in APL applica-

Performance of **loopfs 5000000** (in seconds)			
Platform	FORTRAN	APEX	Speedup
486	2.87	1.85	1.55
SUN	1.97	1.60	1.23

Performance of **loopis 5000000** (in seconds)			
Platform	FORTRAN	APEX	Speedup
486	1.76	1.33	1.33
SUN	1.37	1.23	1.11

Performance of **prd 5200000** (in seconds)			
Platform	FORTRAN	APEX	Speedup
486	2.09	1.12	1.87
SUN	1.10	0.50	2.20

Figure 6.6: Performance of FORTRAN loops against APEX

tions for tasks such as table lookup and name validation. The heavy use made of these inner-product variants has led almost all APL interpreter vendors to write special-case interpreter code to improve their performance. The presence of these special-case codes in interpreters suggests that the compiled performance of these kernels is unlikely to materially exceed that available under a well-written APL interpreter. Since the **ipopne** benchmark is the logical negation of the **ipape** benchmark, created only as a performance validator for inner product, its performance is essentially identical to that of **ipape** and will not be discussed in detail.

The **ipbb** benchmark (*Inner-Product Boolean-Boolean*) performs the $\vee.\wedge$ inner product of a moderately sparse (33% density) Boolean matrix of order N with its transpose. Expressions involving this particular inner product are used in APL for applications involving connectivity graphs. Common applications include transitive closure and the computation of calling trees and cross-reference tables.

The **ipbd** benchmark (*Inner-Product Boolean-Double*) performs the traditional $+.\times$ inner product of linear algebra on a moderately sparse (20% density) Boolean matrix of order N with a double-precision matrix of the same order. This is a benchmark in which APL interpreters are expected to do well, since the arrays are of a goodly size, interpretive overhead is negligible, and the primitive operation is of sufficient computational complexity to dominate execution time. This form of matrix product is often used for such purposes as computing sums of subsets and determining the number of cases satisfying certain conditions. It is popular in statistics and in financial applications.

The **ipdd** (*Inner-Product Double-Double*) benchmark performs the $+.\times$ inner product of linear algebra on a double-precision square matrix of order N with its transpose. This is the classical matrix product of linear algebra and engineering, appearing everywhere that array computations are performed; application areas include financial modeling, statistics, engineering analysis, computer graphics, fluid dynamics, and thermodynamics. Its ubiquity is such that considerable research and engineering effort

has been expended to improve its performance. Hence, excellent performance on this particular inner product is critical if APL is to be an effective tool for numerically intensive computation.

The performance results we obtained for the inner-product benchmark suite are tabularized in Figure 6.7. Also listed is the algorithm used for each type of inner product. Note that, in the interest of fairness, the FORTRAN algorithm used and the algorithm used by APEX are identical. In all cases, the algorithms used gave the best performance results for both FORTRAN and APEX. Furthermore, we give early uniprocessor results for the Silicon Graphics Power Challenge system for this set of kernels.[1]

The character benchmarks, **ipape** and **ipopne**, exhibit almost identical performance in both FORTRAN and APEX-generated code on the 486, but the SUN shows APEX performing about 2.5 times faster than FORTRAN. We have not investigated the cause of this disparity in performance. Both **ipape** and **ipopne** achieve their levels of performance under APEX by using the *QuickStop* algorithm. This key to understanding this algorithm is the realization that certain reduction functions have the property that particular partial reduction results latch that result value into the reduction; later elements brought into the reduction have no effect on the final result. The most obvious examples of such *latching* reduction functions are *times* (×), *and* (∧), and *or* (∨). In the former two, encountering a zero guarantees a zero result for the reduction; in the latter, encountering a one guarantees a one. APEX generates *QuickStop* code for inner product when, for the inner product `f.g`, the function `g` produces a Boolean result, and the function `f` is one of *times*, *and*, *or*, *max*, or *min*. Thus, we have generalized the `∧.=` special case to handle a fairly large class of matrix products, all of which will enjoy the level of performance observed in **ipape** and **ipopne**.

In **ipape**, we observed the effect of early termination of a reduction based upon a partial result of the reduction. In the **ipbb** benchmark, APEX exploits a similar algebraic property, except that it is based on the value of a left argument element, rather than on the value of a partial reduction.

The first Boolean benchmark, **ipbb**, shows that compiled APL can outperform FORTRAN on some platforms, running nearly three times faster on the 486 and 1.47 times faster on the SUN. Only on the SGI does FORTRAN run faster than APEX-generated code, but this may reflect the immaturity of OSC or the APEX code generator at the time when we ran these benchmarks.[2] Since this level of performance is due entirely to the code generated by APEX for Boolean matrix products, the underlying *STAR* inner-product algorithm is worthy of discussion.

We first encountered the eponymous *STAR* inner-product algorithm in the CDC STAR-100 APL system in the early 1970s, where it had been used to exploit the vector hardware of that computer. The *STAR* algorithm interchanges the inner-product loop order so that each element of the left argument is fetched but once. In cases where the type of the left argument must be coerced (say, from integer to double precision), this can offer some improvement in performance all by itself by avoiding repeated coercions. A larger performance gain, particularly for vector computers, arises from another factor.

[1] The relative performance of APEX-generated code on this system was noticeably poorer than on other platforms. This may be due to the immaturity of OSC on that platform, as the performance of hand-coded SISAL on that platform was quite close to that of APEX-generated code.

[2] The SGI system was a prototype installed at NERSC for a brief period of testing.

Performance of **ipape 1000** (in seconds) Algorithm: *QuickStop*			
Platform	FORTRAN	APEX	Speedup
486	24.99	24.51	1.02
SUN	37.80	14.68	2.58
SGI	6.40	4.40	1.45

Performance of **ipopne 1000** (in seconds) Algorithm: *QuickStop*			
Platform	FORTRAN	APEX	Speedup
486	24.39	23.75	1.03
SUN	40.53	14.73	2.75
SGI	6.30	4.40	1.43

Performance of **ipbb 500** (in seconds) Algorithm: *generalized STAR*			
Platform	FORTRAN	APEX	Speedup
486	28.05	9.49	2.96
SUN	10.80	7.37	1.47
SGI	2.00	3.40	0.59

Performance of **5 ipbd 501** (in seconds) Algorithm: *generalized STAR*			
Platform	FORTRAN	APEX	Speedup
486	17.46	24.46	0.71
SUN	7.33	11.70	0.63
SGI	6.50	10.90	0.60

Performance of **ipdd 500** (in seconds) Algorithm: *TransposeRight*			
Platform	FORTRAN	APEX	Speedup
486	69.46	64.96	1.07
SUN	38.13	31.43	1.21
SGI	1.80	2.60	0.69

Figure 6.7: Performance of FORTRAN inner products against APEX

The left argument element is applied to a row of the right argument in scalar-vector, stride-1 fashion, creating a vector intermediate result. That vector result is then reduced into the final result in vector-vector, stride-1 fashion, making highly effective use of cache memory. For some vector supercomputers, this algorithm can provide excellent performance.

In a non-vector computer architecture, the advantage of the original *STAR* algorithm is often negated by the additional memory subsystem traffic entailed by vector intermediate results. Nonetheless, a generalized version of the *STAR* algorithm pays off handsomely in cases where knowledge of the derived function and the left argument permits the left argument to be treated, in some sense, as a sparse array. In the traditional matrix-multiply algorithm, a zero in the left argument does not contribute to the result and may, therefore, be ignored, but since the time to check for a zero element is roughly the same as the time to do the multiply, there is no performance benefit in checking for a zero. However, with the *STAR* algorithm, the cost of the zero check is amortized over an entire row of computations. If the left argument is sparse, this can make a significant reduction in the amount of computation required and, hence, in the performance of such sparse array computations.

We have generalized the *STAR* algorithm to apply it to a larger class of inner products. The generalization is best understood by examining the theoretical role of a zero left argument element in traditional matrix multiply. Let us denote the inner product as `f.g`, where `f` is the addition function (+) and `g` is the multiply function (×). A left argument element of zero appears in conjunction with `g` to form the function `0×y`. For any argument, `y`, this function produces an array of zeros whose elements are, in turn, the additive right identity for the function `f`. Being the identity, it does not contribute to the result and may, therefore, be ignored. Thus, we see that two pieces of information are required to perform this optimization. First, we need to know the value of the right identity for `f` if it, in fact, has such an identity. That is, what value of `y` will cause `x f y` to produce `x` regardless of the value of `x`? Second, for `x g y`, we need to know what value of `x` will generate that identity element. Since the set of such values for arbitrary `f` and `g` is potentially of infinite size, we currently handle only the cases for which the identity element is zero or one. Even this restricted set, it turns out, handles the common inner products of APL very nicely.

Implementation of the algorithm was simplified by amending the primitive function attribute table to include two columns for the eponymous *zero* and *one* values. The code generator merely examines the appropriate table entries to decide if it is possible to use the *STAR* algorithm for matrix product.

At present, the APEX code generator emits *STAR* algorithm code only for Boolean left arguments, as they are, in a sense, sparse. However, it would be a trivial matter to extend APEX declarations to include a *sparse* modifier which the code generator could then exploit for all data types.[3] Since the performance of this algorithm on dense arrays is usually inferior to that of the *TransposeRight* algorithm, to be discussed shortly, we restrict its use to sparse arrays.

[3]The concept of introducing sparse arrays as a formal attribute of data in APEX (via a user-supplied declaration) is intriguing, as other functions besides inner product could exploit them without requiring the application programmer to write explicit code for sparse arrays. For example, the structural and selection functions would propagate the *sparse* attribute of an array from argument to result. The internal representation of the sparse array need not be evident to the programmer.

Another example of the performance improvement available with the *STAR* algorithm is given by **ipbd**, which uses Boolean left arguments and double-precision right arguments to $+.\times$, the traditional matrix product. This benchmark is intriguing because of the poor performance of APEX-generated code compared to FORTRAN, particularly when its cousin, **ipbb** performs so well. Since both benchmarks employ the *STAR* algorithm to take advantage of the sparse left argument, we expected that their relative performance ratios would be similar. We have not investigated the cause of this performance disparity.

The remaining kernel, **ipdd**, is the traditional inner product of linear algebra and engineering. The performance of **ipdd** was somewhat surprising to us because we expected APEX-generated code to lose some performance due to descriptor maintenance induced by OSC's vectors-of-vectors storage scheme. We were pleased to observe that **ipdd** is marginally faster than FORTRAN on the 486 and the SUN, although the SGI performance is, like **ipbb**, slower.

6.3 Summary

The techniques we have described in this chapter for improving the performance of kernels should be effective in applications, in several ways. The techniques that have materially improved the performance of compiled APL applications are shown in Figure 6.8. First, loop fusion can effectively merge adjacent APL primitives into a single one, as was seen with the **logd** benchmark in Chapter 5. Second, the benefits of the inner-product algorithms described here are available with a wide class of inner products. Since inner products are computationally intensive, they may dominate execution time. Hence, any improvements to their performance may be quite apparent in the execution time of the entire application. Finally, high-performance algorithms that are similar in spirit to those shown here for inner product are applicable to other APL primitives and derived functions. These have been shown to make substantial improvements in the performance of interpreted applications.

Benchmark	APEX facility							
	setup	loop fusion	array predicates	special-case algorithms	copy avoidance	extended language features	inter-language calls	tuning
ipape				•				
ipopne				•				
ipbb				•				
ipbd				•				
ipdd				•				
logd3		•			•	•		•
loopif	•	•						
loopis	•	•						
prd		•						
mconvred		•				•		•
tomcatv	•	•						

Figure 6.8: APEX facilities affecting FORTRAN benchmark performance

The performance of APEX-generated code on kernels makes it evident that APEX has the ability to generate code that is as efficient as, and sometimes more efficient, than FORTRAN. However, the results we observed in the FORTRAN applications make it abundantly clear that our task is not yet completed. As with APEX performing against interpreted APL, array copying, loop fusion, and excessive descriptor operations remain problematic.

We believe that, once we have solved these problems, the performance of compiled APL will approach that of other compiled languages. At that time, the abstract semantics of APL will give it a substantial time-to-solution edge over scalar languages on serial platforms.

Chapter 7

Portable Parallelism

Although parallel computers are now reaching the mainstream computing world, there is a paucity of languages that allow portable and efficient programs to be written for such machines. One of our goals in the design of APEX was to address this problem by compiling APL, a language known for parallel expression, into portable code that is able to execute efficiently on a variety of parallel systems. This chapter presents our preliminary findings in this area.

One reason we chose to explore parallelism in APL was our belief that compiled APL has two advantages over interpreted APL when it comes to parallel execution – coarser *grain size* and *reduced serial fraction*. Although compilers and interpreters both possess full knowledge of fine-grain parallelism at the primitive function level, coarse-grain parallelism, such as at the defined function level, is harder for an interpreter to exploit because of potential semantic violations arising from side effects and race conditions. A compiler, however, can perform static analysis to detect and resolve such hazards and thereby extract and use such coarse-grain parallelism.

The other advantage that compiled APL holds over interpreted APL for parallel execution arises from reduced setup time, which reduces the fraction of time spent executing serial code. By Amdahl's Law, such serial setup time limits the potential parallel speedup available to a program [Amd67]. Since, as noted earlier, about half of all primitives executed by typical APL applications operate on single-element arrays [Jor79, Wil91], setup time can form a significant fraction of execution time in an interpreted environment. By eliminating most setup time, an APL compiler reduces the serial fraction and thereby offers superior fine-grain performance.

The other reason we chose to explore parallelism in APL arose from our desire to write parallel programs in a portable fashion – one that does not reflect any particular parallel computer architecture. The abstract nature of APL, whereby a user specifies what is to be done in a computation, without having to express, or even know, how the computation is actually performed, is the key to this *portable parallelism*. Such *abstract expression* is inherently parallel, as it describes operations on entire collections of data, rather than on individual items. The programmer need not be concerned with the multitude of minutiae related to writing code for a specific parallel computer – the compiler or interpreter maps the constructs of APL onto the target system in a manner that is appropriate to that computer. Thus,

programming in an abstract language permits expression of parallel computation in a way that, making no presumptions about parallel computer architecture, is highly portable. In addition, it permits a programmer who knows nothing about parallel computing to write parallel programs, because the language processor will automatically detect and exploit whatever parallelism exists in the program.

By contrast, languages that are computer-oriented require that the programmer embed information, such as the target system's physical memory layout and cache characteristics, in the application. This approach obscures application code and hinders portability, but it does permit an expert programmer to tailor code for maximum parallel performance. However, that hand-crafted parallelism has three downsides that reduce its perceived performance benefits. First, since the ideal expression of parallelism for one computer architecture may be highly inappropriate for another, the programmer must maintain several versions of the application or to face the loss of parallel performance on differing computing platforms. Second, as noted above, hand-crafted parallelism confounds expression of the application with expression of the parallel system architecture, bringing about comprehension and maintenance problems. Third, hand-crafted parallelism may not always exploit all available parallelism. A tight development timetable may limit the extent of manual tuning for parallelism, and few analysts have sufficient expertise in the field of parallel computation to be able to effectively exploit the power of multiprocessors, as their *forté* may be geophysics or stock market analytics. Since analysts are unlikely to have significant understanding of, or interest in, the performance characteristics and foibles of a particular computer, they tend to write straightforward programs, rather than rewriting and tuning to exploit the state of the art in parallel algorithm design.[1] This suggests that hand-crafted parallelism may not, in practice, deliver maximum parallel performance.

Given that naive parallel programming of either sort – abstract or hand-crafted – is able to achieve maximum parallel performance, does one method have an advantage over the other? We suggest that the non-involvement of the abstract programmer in extraction of parallelism ensures that opportunities for parallelism are not missed – the language processor handles the entire task; it may also customize the algorithms to better exploit each target architecture. By contrast, hand-crafted parallelism places the entire burden of parallelism detection and extraction on the programmer. APL already has a time-to-solution edge over computer-oriented languages; parallelism comes for free. In computer-oriented languages, the extra time required to hand-craft parallelism makes APL's time-to-solution advantage even greater.

To see how this advantage works in concert with compiled APL, assume that interpreted APL executes 10 times slower than FORTRAN.[2] Assume further that APL development time is 10 times faster than developing in FORTRAN, as suggested earlier in Chapter 1. A program that required 15 hours to develop in FORTRAN could be developed in 1.5 hours with APL. If that program then executed for 1

[1] At Supercomputing'88, Christopher Hsiung of Cray Research (at that time) said "We should build machines for average programmers." Similarly, we who are in the language design business would do well to design languages for average programmers.

[2] The wide variance in performance of interpreted APL applications compared to FORTRAN makes any ratio chosen here suspect. This figure is intended to represent well-written APL applications that are not entirely dominated by computations on large arrays.

hour in FORTRAN, it would take 10 hours to execute in interpreted APL. Thus, APL would achieve better time-to-solution, by a factor of about 30%. This is shown graphically in the top third of Figure 7.1. But, if the program required 2 hours to execute in FORTRAN, APL's time-to-solution edge would be lost: the 20 hours of execution time in APL would result in interpreted APL's time-to-solution being 20% worse than FORTRAN. This is shown in the middle third of Figure 7.1. Shorter execution time favors APL; longer execution time favors FORTRAN. Similar arguments may be brought to bear when a program is run, unmodified, many times after creation. In such cases, the benefits of APL's time to solution are lessened, because development time is amortized over many runs of the application.

In a parallel computing environment, APL's parallel semantics should give it a further advantage in development time over traditional scalar-oriented languages. Assume that exploitation of parallelism in a FORTRAN-like language increases development time by a third, whereas APL provides that same amount of parallelism with no extra effort by the developer. Assume further that both methods result in a speedup of four over the same code running on a uniprocessor. This gives us the situation shown in the bottom third of Figure 7.1. APL's advantage in time-to-solution is now even greater than before. Of course, the comment made earlier about repeated runs still applies.

7.1 Multiprocessor Performance of Compiled APL

Early in the APEX project, we conducted several experiments on multiprocessor systems to confirm our assertion that APEX provides portable parallelism. We have not yet been able to conduct detailed multiprocessor performance testing of the complete APEX test suite. We therefore present the early figures as proof of concept, with the hope that future timing tests on parallel systems will substantiate our performance claims.

We conducted timing tests on two parallel systems at the National Energy Research Supercomputer Center (NERSC) – a CRAY C-90 and a Silicon Graphics Power Challenge – as well as on a CRAY EL-92 located at the University of Toronto. At the time these tests were conducted, APEX was just beginning to turn out working code. Since that time, we have made considerable improvements in performance which are not reflected in these timings. We caution the reader that our tests are the inner-product kernels of Chapter 6 – **ipape**, **ipopne**, **ipbb**, **ipbd**, **ipdd** – which possess a considerable amount of potential parallelism. Hence, they may not be a fair measure of the parallel potential of arbitrary APL applications.

On the SGI Power Challenge, we executed the inner-product kernel suite of five benchmarks, varying the number of processors from 1–8. We then plotted the speedup obtained on N processors as the ratio of CPU time for one processor to the *maximum* CPU time used by any of the N processors. We used the maximum value as we felt it was a sensible measure of processor workload imbalance. This choice did not substantially alter the levels of performance presented here.

Figure 7.2 shows the speedups we obtained on the inner-product suite. As can be seen, speedup increases nearly linearly until about 5 processors, when it begins to tail off for the more memory-intensive

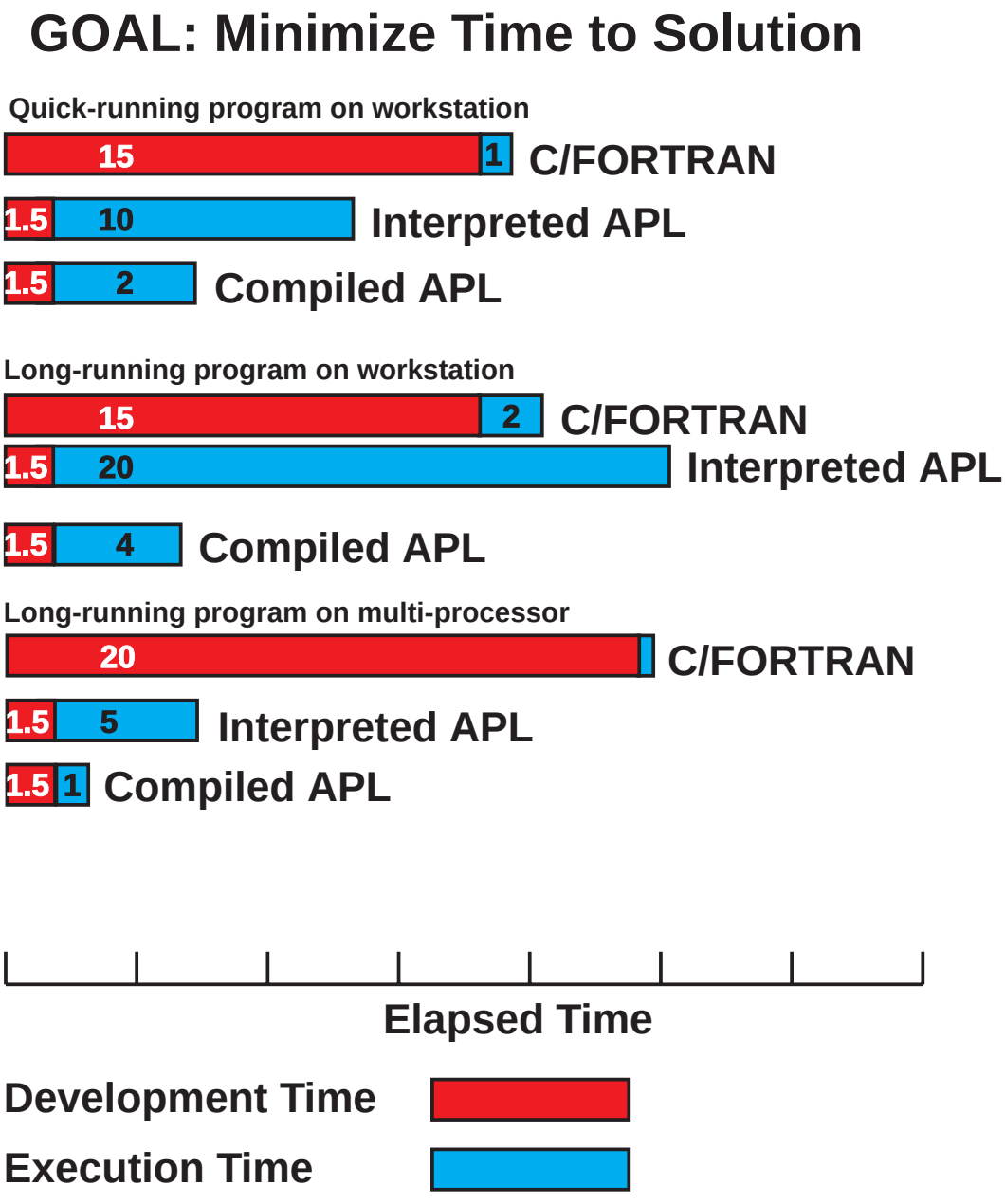

Figure 7.1: Time To Solution

benchmarks. We did not investigate this effect, but note two possible factors that may be relevant. First, the system was extremely heavily loaded, with several hundred other users. This could cause cache and memory subsystem interference, as well as increased processor time due to task switching. Second, we made no attempt to tune the benchmarks to take advantage of OSC run-time options such as strip mining. Considering that neither the kernels nor the APEX compiler makes any attempt to explicitly deal with parallelism, these speedups are quite acceptable.

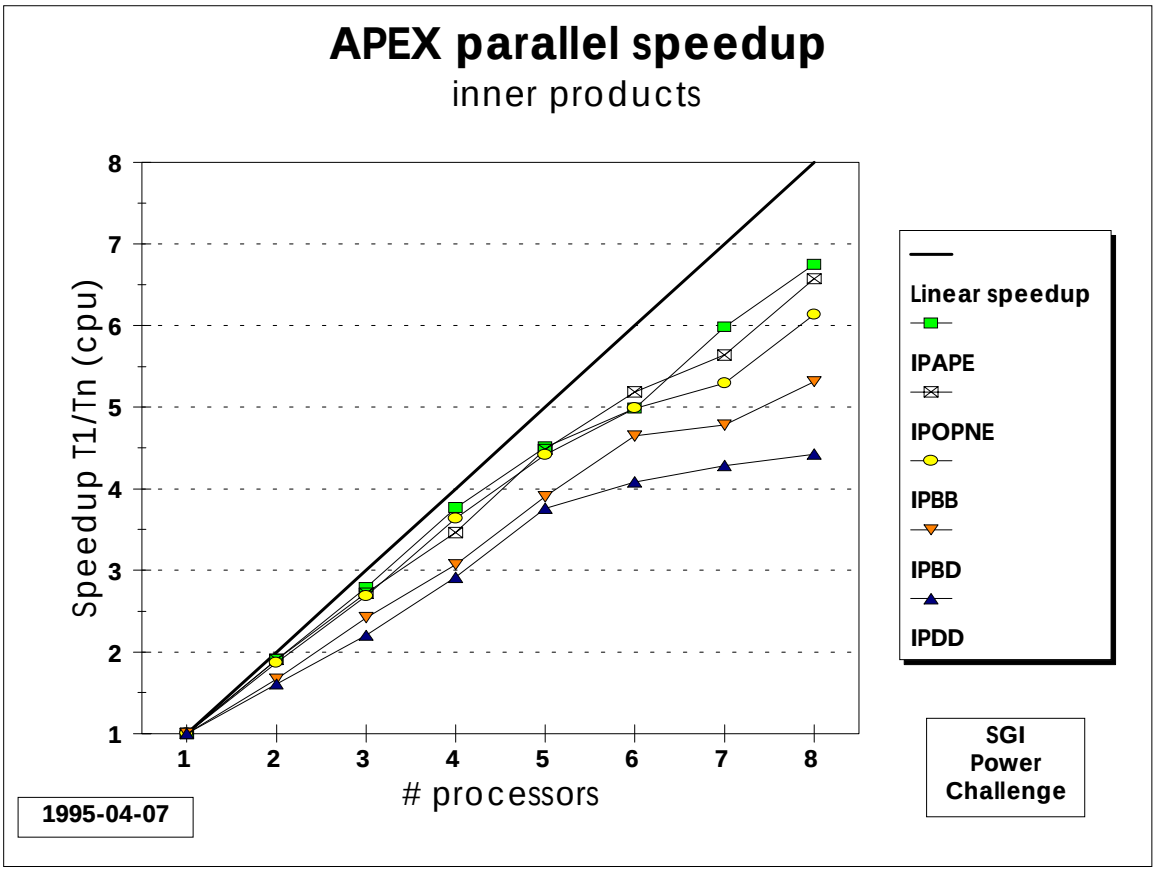

Figure 7.2: APEX parallel speedup on SGI Power Challenge

Our second platform for measuring speedup was a CRAY C90 at NERSC. This system was also heavily loaded, and we only performed a single inner-product benchmark on it, shown in Figure 7.3. The **ipdd** kernel shows very good linearity up to 6 or 7 processors, at which point no additional performance is obtained by adding more processors. We did not investigate the cause of this, but conjecture that it could be the same as the Power Challenge performance problems or perhaps an operating system scheduler characteristic such as placing a ceiling on the number of physical processors allocated to the task. As with the Power Challenge benchmarks, we are pleased at obtaining these levels of speedup for the minimal effort we expended.

Our third platform was a CRAY EL-92 that was installed for a short period of time at the Univer-

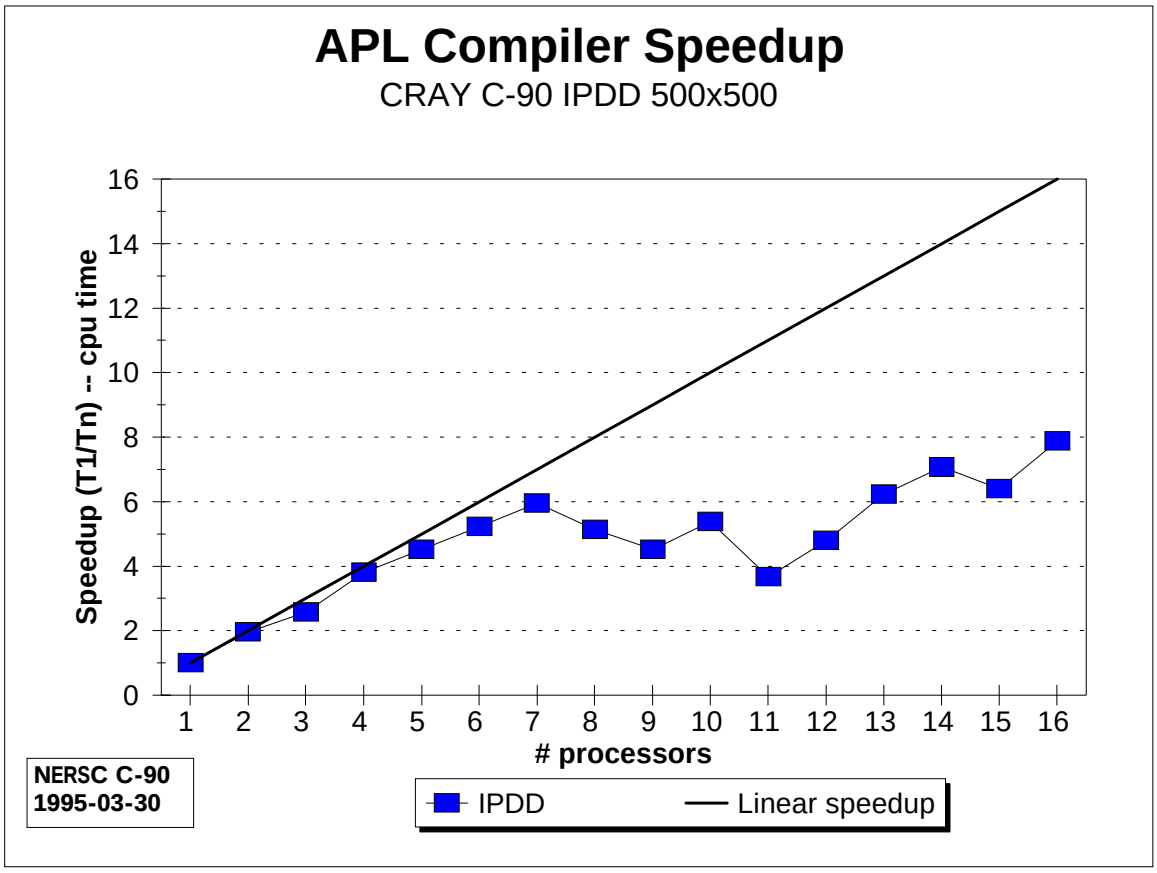

Figure 7.3: APEX parallel speedup on CRAY C90

sity of Toronto. We ran the convolution benchmark **mconvo**, on one and two processors with various argument sizes, and found speedup to be reliably in the 1.93–1.98 range, as shown in Figure 7.4.

7.2 Related Work

If we ignore vector machines, we find that relatively few concrete results have been published on the performance of APL for SIMD and MIMD computer architectures. Ching's APL/370 compiler was ported to two parallel platforms, the IBM 3090 and the IBM Research RP3. A paper on the former experiment discusses their research direction in general terms, but no quantitative results are presented [Chi90]. A second experiment involved the IBM RP3, a prototype shared-memory, 64-processor MIMD computer. On this host, they claim best speedups of about 12 on up to 64 processors for several small APL programs [JC91]. Ching's experiments, using data parallelism within APL primitives only, parallelized the outer loops of such primitives, while leaving the inner loops in serial form.

The only implementation of APL we know of for a commercially available massively parallel processor is that performed by Schwarz, while working in the author's research department at I.P. Sharp

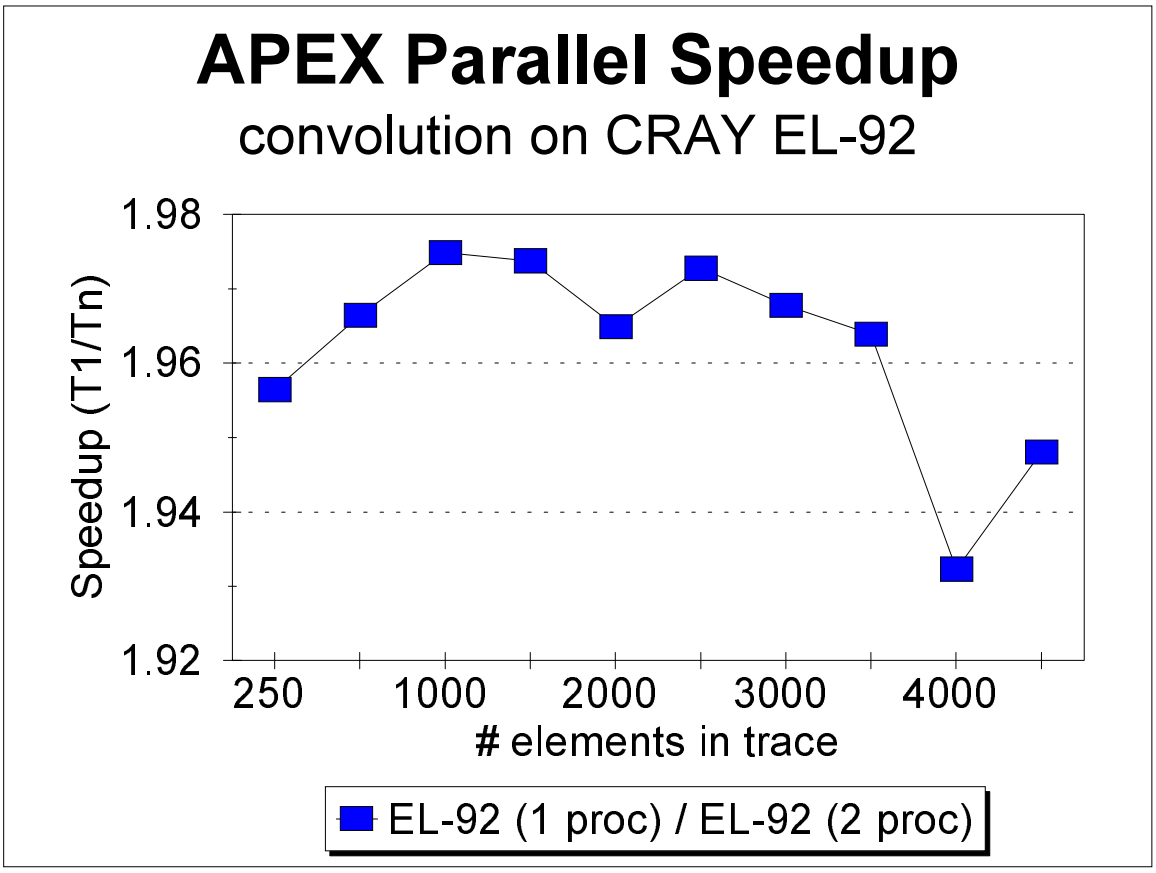

Figure 7.4: APEX parallel speedup on CRAY EL-92

Associates Limited. Schwarz conducted a research port of the ACORN compiler run-time library to the Thinking Machines Corporation CM-2 in 1988–1989 [Sch90]. The CM-2 project, our first attempt to exploit the power of APL on such a machine, explored the impact of linear versus grid geometries on numerically intensive computations. The project was cut short by a major corporate reorganization.

7.3 Summary

The APL language is unique as being the first computer language to support array-based computation, yet its history of being an interpreter-only language has hobbled it as a tool for parallel computation. Our preliminary results suggest that, given suitable compilers, APL can be an effective tool for programming parallel computers and an efficient tool for executing programs on parallel computer systems.

Chapter 8

Summary and Future Work

The APEX project tackled the problem of compiling APL programs for efficient execution in both serial and parallel run-time environments. Our justification for compiling APL stemmed from the language's parallel semantics which, in our opinion, make APL an excellent tool for expressing parallel algorithms in a concise and portable manner, and from our frustration with the poor performance of APL in interpreted environments.

The methodology we adopted was to create APEX, an APL-to-SISAL compiler. The APEX compiler translates applications written in an extended subset of ISO Standard APL into SISAL 1.2. SISAL is a single-assignment, applicative, vector-of-vectors, parallel language. We felt that the use of SISAL as an intermediate code would solve a number of problems for us, including generation of parallel code, portability to a number of serial and parallel platforms, and optimization of array-oriented code.

APEX is a multi-phase compiler, comprising a syntax analyzer, static single assignment transformer, semi-global analyzer, data flow analyzer and code generator. The actions performed by APEX are as follows: syntax analysis of the source program using a parallel reduction parser, followed by translation of the source program into static single assignment form. Next, data flow analysis is used to deduce morphological information about the application: facts including, but not limited to, the type and rank of each array created during the execution of the program. Knowledge of this morphological information permits the code generator phase of the compiler to emit SISAL code. The SISAL code is then compiled to C by OSC, the Optimizing SISAL Compiler. Based on a compile-time option, OSC will generate either sequential C that can execute on single-processor computers or parallel C that executes in SPMD mode on parallel computers. As the final step in the compilation process, the generated C code is compiled by the native C compiler of the target computer.

8.1 Contributions

This thesis makes several contributions to the computing community: creating APEX, an APL compiler that generates high-performance code through a variety of techniques; providing a capability for mapping APL programs into single-assignment, functional form, thereby simplifying the APL compilation

process; generalizing sparse-matrix inner-product algorithms to permit efficient execution for a large family of matrix products; demonstrating that SISAL can be a highly effective intermediate language for compiling functional array languages; clarifying the importance of using applications, not kernels, to measure the relative performance of language processors, and proposing enhancements to the SISAL and APL languages to improve performance and solve semantic problems encountered in the course of compiler development. We also give preliminary results that promise good speedup on parallel computers without requiring changes to APL application programs.

Our primary contribution is the creation of an APL-to-SISAL compiler that generates high-performance, portable, parallel code. That code executes efficiently on a variety of parallel and serial computers without requiring changes to applications. The performance of APEX-generated code, relative to APL executing under state-of-the-art interpreters, is highly variable. For simple, non-iterative applications, APEX currently produces code whose performance is up to an order of magnitude faster than interpreted code. For heavily iterative applications, APEX-generated code executes up to several hundred times faster than interpreted code. Serial performance of kernels is often competitive with FORTRAN, although application performance currently suffers due to excessive array copying operations. Preliminary measurements suggest that speedup on parallel systems will be acceptable.

The performance levels we observed with APEX-generated code arise from several factors working in conjunction. The use of SISAL as an intermediate code lets us exploit OSC's capabilities for loop fusion and copy elimination, as well as letting OSC handle all details of parallel code generation. The use of static single assignment permits APEX to make better decisions about array morphology than would be possible if *def-use* chains were used instead. Our integrated approach to design emphasizes a balanced view of all aspects of run-time code, with concomitant effect on performance. In particular, the cost of syntax analysis is eliminated by the very act of compilation; the cost of per-function setup and memory management is reduced or eliminated by data flow analysis and array morphology; OSC copy elimination analysis also contributes to reduction of memory management costs; the performance of run-time algorithms is improved by use of array predicates and a pattern-matching code generator that exploits case-specific knowledge to select special-case algorithms.

By emitting SISAL, we generate efficient, highly portable, parallel code for a variety of platforms, ranging from the PC through workstations to parallel supercomputers. This simplifies and speeds the creation of parallel applications. Furthermore, it makes the power of APL available on a wide variety of architectures. Heretofore, APL systems were only available for a limited range of computers and not available at all on contemporary supercomputers.[1]

The translation of APL applications into functional, static single assignment form facilitates generation of efficient code. As a side effect of semi-global analysis, APEX provides the programmer with information that can potentially serve as a useful diagnostic tool for the comprehension of large applications.

The generation of SISAL as an intermediate code proved, by and large, to be an excellent choice.

[1]The only APL implementation for a supercomputer that we are aware of was created for the CDC STAR-100, circa 1971.

We obtained all the benefits of the Optimizing SISAL Compiler with little or no cost to us, except for the problem we encountered with empty arrays. The generation of SISAL let us ignore, for the most part, issues such as parallelism, multiple target systems, and traditional compiler optimizations. This freed us to concentrate on the larger issues, such as the algorithmic aspects of code generation for APL primitives.

Loop fusion and other optimizations whose effects transcend that of a single APL primitive make it clear that relative performance claims among language processors should be based on measurements of application, rather than kernel, performance. Kernel benchmarks are unfairly biased against systems that are able to exploit optimizations at a larger scale than those of kernels. Furthermore, they do not reflect differences in coding styles, nor the relative mix of array sizes and primitives found in actual applications.

We also found that a number of enhancements to APL improve application performance in interpreted and compiled code. These include the *rank* and *cut* conjunctions, and extension of the *dyadic reduction* derived function to include monadic operands.

Our adoption of SISAL as an intermediate code revealed several semantic and performance problems. The most serious problem affecting performance on all platforms arises from the absence of SISAL and OSC support for arrays other than vectors. OSC represents arrays as vectors of vectors, in which each row of an array is pointed to by a descriptor vector containing one element per array row, plane, and hyperplane. The presence of these descriptors, together with the absence of any guarantee of contiguous memory allocation for all elements of an array, results in loss of performance in at least four areas: column operations, depth of loop nesting, memory manager overhead, and array copying.

The absence of array support becomes readily apparent in column operations, such as extracting a column from an array. Extracting a column from an array cannot be performed as a vector fetch with stride equal to the number of columns in the array. The extraction involves, instead, dereferencing a row pointer for *each* element fetched. This has a highly detrimental effect on the performance of certain benchmarks.

In addition, the absence of arrays introduces the possibility of non-contiguous array storage, at least in the current version of OSC. This affects code generation and performance for such fundamental stride-1 array operations as adding two matrices element by element. Rather than produce a single loop that runs across all elements of an array, it is necessary to generate a nest of loops, nested to the depth of the array's rank. This introduces additional run-time overhead and increases the volume of generated code.

Moreover, vector-of-vectors storage entails substantial run-time overhead that is not present in array storage schemes. The requisite vector descriptors, attached to each array row in OSC-generated code, introduce an excessive amount of memory manager overhead, another detriment to performance.

Furthermore, non-contiguous array storage prevents the implementation of array coordinate mapping, whereby most APL structural operations and some selection operations could be performed in fixed time by descriptor manipulation, regardless of the size of the arrays involved. Without array coor-

dinate mapping, considerable time is spent moving array elements to perform array operations such as reversal, transpose, and drop. The absence of array coordinate mapping is probably the most significant single performance problem in APEX.

A final area where OSC causes loss of performance and increased memory utilization is in support of Boolean arrays. Boolean run-time performance and memory usage is of great importance in APL, because Booleans are heavily used within APL applications as replacements for what would be control flow operations in other languages. Because OSC stores Booleans as one per byte instead of one per bit, memory costs increase by a factor of eight. This data representation slows operations such as the de Morgan and relational functions. Such operations could be performed a word at a time if stored as one bit per element; their representation within OSC causes these operations to take eight times longer; in the case of operations such as matrix product, 64 times longer.

Returning to the thesis, we believe it offers several contributions to researchers in the array languages community. The most significant of these is the validation of SISAL as an intermediate language for array language compilers. Our use of SISAL let us avoid having to reinvent the wheels of reference count elimination, loop fusion, and traditional compiler optimizations. Portability and implicit parallelism fell out of our compiled code with minimal effort on our part and none on the part of the application writer. Our generalizations of the STAR inner-product algorithm to permit efficient execution of a large class of matrix products should be of value to the implementors of both compilers and interpreters.

On the downside, we identified several problems in the use of SISAL as an array intermediate language. The absence of true arrays is a major problem, evident in terms of semantics as well as run-time performance. The absence of functions that extend uniformly to higher rank arguments, as do the scalar functions in APL, increases the amount of effort required to generate SISAL code, although this problem is largely alleviated by our recursive macro-based code generator.

The ease with which we can generate SPMD code from APL by use of APEX suggests that it may be a useful tool for exploring non-uniform memory access (NUMA) computers and processor-in-memory (PIM) architectures, parallel algorithms, run-time data distribution, and scheduling algorithms.

8.2 Performance Results

Despite the performance-inhibiting features of SISAL and the OSC compiler, the code generated by APEX generally outperforms interpreted APL by a considerable margin. Specifically, the performance of inherently iterative applications, such as those using dynamic programming, is excellent, performing several hundred times faster than their interpreted APL equivalents. The performance of non-iterative APL applications relative to interpreted APL is highly variable. In cases where interpreted APL performance is already considered to be excellent, such as in non-iterative signal processing applications operating on long vectors, APEX-generated code performs quite well if OSC is able to perform loop fusion effectively, often running an order of magnitude faster than interpreted APL. However, when OSC is unable to perform loop fusion or to eliminate array copy operations, the performance of APEX-

generated code suffers to the point of sometimes being slightly slower than interpreted code.

The code generated by APEX is competitive with FORTRAN 77 for kernels, with some benchmarks executing up to 3 times faster than FORTRAN. Application performance varies from poor to competitive, depending on the application. The major hurdles impeding APEX performance in this arena are extraneous array copying operations and excessive memory management overhead.

Preliminary results show that APEX-compiled kernels obtain reasonable levels of speedup on parallel systems without requiring any source code changes by the application programmer. More experiments are required to validate that we retain this speedup on large applications. Because we obtained acceptable levels of performance on a variety of platforms, both serial and parallel, with no changes to application source code, we believe that we are on the road to our goal of obtaining perfect portability of APL applications.

8.3 Future Work

The APEX project has revealed that, although compiled APL can often compete in performance with imperative scalar languages, several challenges remain. It is clear that APEX-generated code performance on Boolean data and on vectors of rank greater than one is inadequate. The former could be improved by implementing one-bit Boolean support within OSC, to permit parallel execution on a word-at-a-time basis. The latter is best addressed by enhancing OSC to represent rectangular arrays directly, rather than as vectors-of-vectors [Fit93]. This approach, as opposed to one in which we mapped all APL arrays into SISAL vectors, would benefit all users of SISAL, rather than merely the APEX subset of the SISAL user community.

From the standpoint of the APL language implemented by APEX, several syntactic and semantic improvements to APEX are desirable. From a syntactic standpoint, the most critical missing feature is support for *goto*. This, as noted earlier, is best achieved by mapping *goto* into structured control flow expressions [EH93]. Support for the bracket axis expression, niladic functions, and functions with no explicit result are desirable from the standpoint of compiling legacy applications. From a semantic standpoint, correcting the behavior of empty arrays is the most evident shortcoming of APEX. As noted later in this section, the proper way to correct this is to enhance SISAL and OSC to support empty arrays correctly. Another semantic shortcoming of APEX is its inability to compile applications, such as APEX itself, that use features of the ISO Extended APL Standard, including nested arrays. The analysis problem here is daunting, but could be eased for a great many applications by introduction of a structural declaration similar to that of the C *struct*, to permit structured data to be addressed by name, rather than by non-obvious index expressions. Such an enhancement would give us a quite substantial benchmark for compiled APL, let us measure the extent to which parallelism can be easily exploited within an APL compiler and, most importantly, improve the maintainability of large applications.

We have contemplated several enhancements for improving the run-time performance of APEX-generated code. The enhancements fall into two categories: improvements to the data flow analysis and

code generator phases of APEX itself, and improvements to OSC and SISAL.

Improvements in the data flow analysis phase of APEX include partial evaluation of array expressions at compile time, phrase recognition, detection of cases where arithmetic progression vectors can be generated, and shape vector algebra. Partial evaluation, an extension of constant folding to arrays and a rich set of primitives, executes APL functions whose arguments are known at compile time, thereby eliminating the need to generate code or execute it at run time for such functions. In the case of benchmarks such as **crc**, this could provide a significant performance benefit. Phrase recognition, or idiom recognition, is intended to detect strings of APL functions of the form `f g. . . h y` and `x f g. . . h y`, and replace them, when appropriate, by special-purpose code. Some examples of where phrase recognition would be beneficial include `x/⍳y`, `x⍳⌈/x`, and `⍴⍴y`. A more general pattern matcher could, of course, be used to recognize a larger class of phrases. The power of arithmetic progression vectors in APL is well-known [Abr70]; their utility in compiled code is substantially greater than that in interpreted code, because they can be exploited in a compiler with no increase in execution time setup cost. Shape vector algebra could be exploited, as it was in YAT [JO86], to remove many of the run-time conformability checks remaining in APEX-generated code.

Possible enhancements to the APEX code generator include greater exploitation of array predicates, result demotion, and code merging. Array predicates can be exploited by generating code for such special cases as can be detected by array predicate analysis. These include, but are not limited to, bucket sorts of permutation vectors, removal of index bounds checks, and fast searching and sorting of already-sorted arrays. Result demotion is a way of generating a result of simpler type or rank for a function when the consumer of that result does not need the entire result usually generated. For example, in the expression `⍳⍴vector`, the index generator code is simplified if its argument is a scalar, rather than the one-element vector that would be generated by the shape primitive in executing `⍴vector`. Result demotion could be used by the *shape* primitive to generate a scalar result instead of the vector. This would simplify generated code for both functions and would also eliminate the memory management overhead associated with creation and destruction of the vector. The code merging technique of Driscoll and Orth could be used to reduce, in many circumstances, the run time of certain expressions, such as `(x∧.=y)⍳1`[JO86].

Several improvements to SISAL and OSC are desirable, including support for array coordinate mapping, real arrays, array operations, improved loop fusion, improved parallelism in generated code, and one-bit Booleans. Although these changes could be made at the APEX level, the impact of making them in OSC would be beneficial to a larger audience, including the SISAL community. Support for array coordinate mapping would improve the performance of a number of our benchmarks, particularly those that currently exhibit the poorest relative performance compared to FORTRAN and interpreted APL. We are of the opinion that this enhancement stands to be the most beneficial of any that we describe in this section. From the standpoint of language semantics, the most glaring problem with SISAL at present is its inability to represent certain empty arrays. This problem should be solved by enhancing SISAL to include arrays. Although we could sidestep this problem by rewriting the APEX

code generator to emit C or C++ code directly, that would entail substantial work to recreate the excellent optimizations that OSC performs already. The addition of array operations to SISAL would simplify the language and improve the performance of generated code for scalar functions and other primitives. The capability to perform more sophisticated loop fusion in OSC would benefit several applications, such as **metaphon**. The semantics of SISAL 1.2 limit the amount of parallelism in some of our generated code. At present, we are forced to use iterative loop forms in certain cases where the expression is clearly of a parallel nature, but SISAL lacks a way to express it in a parallel form. Finally, addition of one-bit Boolean support to OSC would reduce memory usage and enable improved performance for a number of Boolean primitives.

A number of operational improvements are desirable to improve the usability of APEX in a production environment. Among these improvements are graphical presentation of array characteristics and of compile-time errors, such as syntax errors, use of unsupported features, and missing declarations. Separate compilation would speed rebuilding large systems. Improved performance of the SSA mapping phase and of the data flow analysis phase of the compiler would remove the two largest performance bottlenecks in the current APEX compiler. The former is fairly straightforward; the latter is more difficult; both would benefit from having a compiled version of the compiler.

Our success with compiling APL suggests other projects that may be of value. Compilers for other array languages, such as NIAL and J, would give those languages performance levels comparable to those we obtained with APEX [Nia85]. Another possibility is the creation of a hybrid interpreter/compiler that would perform compilation of applications in the background, during user "think time." Such an approach could offer the benefits of interactive use combined with the performance of compiled code.

Finally, we believe that the APEX compiler itself possesses an abundant amount of inherent parallelism, at both coarse- and fine-grain levels, in SPMD and SIMD modes. To date, we have not been able to quantify the exact amount of available parallelism, due to the compiler's present inability to compile itself. Nonetheless, the extensive use of SPMD parallelism in all compiler phases except the latter part of code generation places a lower bound on parallelism of one thread per defined function in the compilation unit. In addition, SIMD parallelism exists in substantial quantities, as most parts of the compiler operate on entire defined functions using APL array primitives.

Compilation schemes that exploit parallelism appear to be relatively rare, although the advent of inexpensive parallel systems is likely to change that situation. Wortman and Junkin developed a parallel Modula-2+ compiler that created an execution thread per source program scope unit [WJ92]. Each thread used standard LEX and YACC technology, operating serially by character within each line of the source program scope unit. This is similar in spirit to the SPMD parallelism that APEX uses across the functions of the compilation unit. We believe the amount of available parallelism in APEX to be significantly higher than that of the Modula-2+ compiler, as APEX offers SPMD parallelism at the level of the function line, SIMD parallelism within the tokenizer, and considerable SPMD and SIMD parallelism within later phases of the compiler. Parallelism within the APEX compiler is discussed in

detail elsewhere [Ber97].

Bibliography

[Abr70] Philip Abrams. *An APL Machine*. PhD thesis, Stanford University, 1970. SLAC Report No. 114.

[Alf73] M. Alfonseca. An APL-written APL-subset to System/7-msp translator. In *APL Congress 73*, pages 17–23. North-Holland Publishing Company, 1973.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485. AFIPS, April 1967.

[Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Publishing Company, 1988.

[Bat95] John K. Bates. Some observations on using Ching's APL-to-C translator. *ACM SIGAPL Quote Quad*, 25(3), March 1995.

[BB93] Robert Bernecky and Paul Berry. *SHARP APL Reference Manual*. Iverson Software Inc., 33 Major St., Toronto, Canada, 2nd edition, 1993.

[BBJM90] Robert Bernecky, Charles Brenner, Stephen B. Jaffe, and George P. Moeckel. ACORN: APL to C on real numbers. In *ACM SIGAPL Quote Quad*, volume 20, pages 40–49, July 1990.

[Ber73] Robert Bernecky. Speeding up dyadic iota and dyadic epsilon. In *APL Congress 73*, pages 479–482. North-Holland Publishing Company, 1973. Second printing.

[Ber87] Robert Bernecky. An introduction to function rank. In *APL88 Conference Proceedings*, volume 18, pages 39–43. ACM SIGAPL Quote Quad, December 1987.

[Ber90a] Robert Bernecky. *Arrays, Functional Languages, and Parallel Systems*, chapter 2. Kluwer Academic Publishers, 1990. Compiling APL.

[Ber90b] Robert Bernecky. Portability and performance. In *Cray User Group Meeting*, April 1990.

[Ber91] Robert Bernecky. Fortran 90 arrays. *ACM SIGPLAN Notices*, 26(2), February 1991.

[Ber93] Robert Bernecky. Array morphology. In Elena M. Anzalone, editor, *APL93 Conference Proceedings*, volume 24, pages 6–16. ACM SIGAPL Quote Quad, August 1993.

[Ber95] Robert Bernecky. The role of dynamic programming and control structures in performance. In Marc Griffiths and Diane Whitehouse, editors, *APL95 Conference Proceedings*, volume 26, pages 1–5. ACM SIGAPL Quote Quad, June 1995.

[Ber97] Robert Bernecky. An overview of the APEX compiler. Technical Report 305/97, Department of Computer Science, University of Toronto, 1997.

[BFK95] Richard J. Busman, Walter A. Fil, and Andrei V. Kondrashev. Recycling APL code into client/server applications. In Marc Griffiths and Diane Whitehouse, editors, *APL95 Conference Proceedings*, volume 26, pages 1 – 5. ACM SIGAPL Quote Quad, June 1995.

[BIM+83] Robert Bernecky, Kenneth E. Iverson, Eugene E. McDonnell, Robert C. Metzger, and J. Henri Schueler. Language extensions of may 1983. Technical Report SHARP APL Technical Note 45, I.P. Sharp Associates Limited, May 1983.

[Bin75] Harvey W. Bingham. Content analysis of APL defined functions. In *APL75 Conference Proceedings*, pages 60–66. ACM SIGAPL Quote Quad, June 1975.

[Bro85] James A. Brown. Inside the APL2 workspace. In *APL85 Conference Proceedings*, volume 15, pages 277–282. ACM SIGAPL Quote Quad, May 1985.

[BS74] A.M. Bauer and H.J. Saal. Does APL really need run-time checking? *Software Practice and Experience*, 4:129–138, 1974.

[BT82] Timothy A. Budd and Joseph Treat. Extensions to grid selector composition. Technical Report 82-7, University of Arizona, July 1982.

[Bud83] Timothy A. Budd. An APL compiler for the UNIX timesharing system. *ACM SIGAPL Quote Quad*, 13(3), March 1983.

[Bud85] Timothy A. Budd. Dataflow analysis in APL. *ACM SIGAPL Quote Quad*, 15(4):22–28, May 1985.

[Bud88] Timothy Budd. *An APL Compiler*. Springer-Verlag, 1988.

[Can89] David C. Cann. *Compilation Techniques for High Performance Applicative Computation*. PhD thesis, Computer Science Department, Colorado State University, 1989.

[Can92a] Davic C. Cann. The optmizing sisal compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, 1992.

[Can92b] David C. Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8), August 1992.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1989.

[Chi86a] Wai-Mee Ching. An APL/370 compiler and some performance comparisons with APL interpreter and FORTRAN. In *APL86 Conference Proceedings*, volume 16, pages 143–147. ACM SIGAPL Quote Quad, July 1986.

[Chi86b] Wai-Mee Ching. Program analysis and code generation in an APL/370 compiler. *IBM Journal of Research and Development*, 30:594 – 602, 1986.

[Chi90] Wai-Mee Ching. Automatic parallelization of APL-style programs. In *ACM SIGAPL Quote Quad*, volume 20, pages 76–80, August 1990.

[CNS89] Wai-Mee Ching, Rick Nelson, and Nungjane Shi. An empirical study of the performance of the APL370 compiler. In *APL89 Conference Proceedings*, volume 19, pages 87–93. ACM SIGAPL Quote Quad, August 1989.

[CWF92] David C. Cann, R. Wolski, and John T. Feo. SISAL: Toward resolving the parallel programming crisis. In *IPPS 1992 Parallel Systems Fair*, March 1992.

[CX87] Wai-Mee Ching and Andrew Xu. A vector code back end of the APL370 compiler on IBM 3090 and some performance comparisons. In *APL88 Conference Proceedings*, volume 18, pages 69–76. ACM SIGAPL Quote Quad, December 1987.

[EH93] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. ACAPS Technical Memo 76, McGill University School of Computer Science, 3480 University Street, Montreal, Canada H3A 2A7, September 1993.

[Feo95] John Feo. Private communication, 1995.

[Fit93] Steven Fitzgerald. Copy elimination for true multidimensional arrays in SISAL 2.0. In John T. Feo, editor, *Proceedings SISAL '93*. Lawrence Livermore National Laboratory, October 1993.

[Gui87] Alain Guillon. An APL compiler: The Sofremi-AGL compiler–a tool to produce low-cost efficient software. In *APL87 Conference Proceedings*, volume 17, pages 151–156. ACM SIGAPL Quote Quad, May 1987.

[GW78] Leo J. Guibas and Douglas K. Wyatt. Compilation and delayed evaluation in APL. *Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978.

[IBM94] IBM. *APL2 Programming: Language Reference*. IBM Corporation, second edition, February 1994. SH20-9227.

[Int84] International Standards Organization. *International Standard for Programming Language APL*, ISO N8485 edition, 1984.

[Int93] International Standards Organization. *Programming Language APL, Extended*, ISO N93.03 committee draft 1 edition, 1993.

[Ive73] Eric B. Iverson. APL/4004 implementation. In *APL Congress 73*, pages 231–236. North-Holland Publishing Company, 1973.

[Ive79] Kenneth E. Iverson. Notation as a tool of thought. *Communications of the ACM*, 23(8), August 1979.

[Ive96] Kenneth E. Iverson. *J Introduction and Dictionary*, J release 3 edition, 1996.

[JC91] Dz-Ching Ju and Wai-Mee Ching. Exploitation of APL data parallelism on a shared-memory MIMD machine. In *Proceedings of the Third ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming (PPOPP)*, April 1991.

[Jen70] M. A. Jenkins. The solution of linear least squares problems in APL. Technical Report Technical Report No. 320-2998, IBM New York Scientific Center, IBM Corporation, June 1970.

[Jen73] M.A. Jenkins. Numerical analysis and APL, can they coexist? In *APL Congress 73*, pages 237–243. North-Holland Publishing Company, 1973.

[JO86] Graham C. Driscoll Jr. and D.L. Orth. Compiling APL: The Yorktown APL translator. *IBM Journal of Research and Development*, 30(6):583 – 593, 1986.

[JO87] Graham C. Driscoll Jr. and D.L. Orth. APL compilation: Where does the time come from? *ACM SIGAPL Quote Quad*, 17(4), 1987.

[Jor79] Kevin E. Jordan. Is APL really processing arrays? *ACM SIGAPL Quote Quad*, 10(1), September 1979.

[JSA+85] McGraw J.R, S.K. Skedzielewski, S.J. Allen, R.R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. Sisal: Streams and iterations in a single-assignment language, language reference manual. Technical Report M-146, Revision 1, Lawrence Livermore National Laboratory, March 1985.

[Mar82] James Martin. *Application Development Without Programmers*. Prentice-Hall, Inc., 1982. With research by Richard Murch.

[Mil79] Terrence C. Miller. Tentative compilation: A design for an APL compiler. In *APL79 Conference Proceedings*, volume 9, pages 88–95. ACM SIGAPL Quote Quad, June 1979.

[MM89] M. V. Morreale and M. Van Der Meulen. Primitive function performance of APL2 Version 1 Release 3 (with SPE PL34409) on the IBM 3090/S Vector Facility. Technical Report Technical Bulletin No. GG66-3130-00, IBM Washington Systems Center, IBM Corporation, May 1989.

[Mul88] Lenore M. Restifo Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, 1988.

[Mul90] Lenore M. Restifo Mullin. *Arrays, Functional Languages, and Parallel Systems*, chapter 12. Kluwer Academic Publishers, 1990. Psi, the Index Function: a basis for FFP with arrays.

[Nia85] Nial Systems Ltd. *The Q'Nial Reference Manual*, 1985.

[Per79] Alan J. Perlis. Programming with idioms in APL. In *APL79 Conference Proceedings*, volume 9, pages 232–235. ACM SIGAPL Quote Quad, June 1979.

[RB90] Jack G. Rudd and James A. Brown. Toward a common prototyping language. In *ACM SIGAPL Quote Quad*, volume 20, pages 322–330, August 1990.

[Saa78] Harry J. Saal. Considerations in the design of a compiler for APL. *ACM SIGAPL Quote Quad*, 8(4), 1978.

[Sar88] Dilip V. Sarwate. Computation of cyclic redundancy checks via table lookup. *Communications of the ACM*, 31(8):1008–1013, August 1988.

[Sch90] Walter Schwarz. *Arrays, Functional Languages, and Parallel Systems*, chapter 2. Kluwer Academic Publishers, 1990. Acorn Run-time System for the CM–2.

[SS88] Stephen Skedzielewski and Rea J. Simpson. A simple method to remove reference counting in applicative programs. Technical Report UCRL-100156, Lawrence Livermore National Laboratory, 1988. This report was prepared for submittal to the ACM SIGPLAN 89 Symposium on Programming Language Design and Implementation.

[TB84] Joseph M. Treat and Timothy A. Budd. Extensions to grid selector composition and compilation in APL. *Information Processing Letters*, 19(3):117–123, October 1984.

[Tut95] J.K. Tuttle. Private communication, 1995.

[Wea89] Kevin R. Weaver. Function points – a productivity measure benefits APL. In *APL89 Conference Proceedings*, volume 19, pages 377–380. ACM SIGAPL Quote Quad, August 1989.

[Wei85] Jim Weigang. An introduction to STSC's APL compiler. In *APL85 Conference Proceedings*, volume 15, pages 231–238. ACM SIGAPL Quote Quad, May 1985.

[Wie79] Clark Wiedmann. Steps toward an APL compiler. In *APL79 Conference Proceedings*, volume 9, pages 321–328. ACM SIGAPL Quote Quad, June 1979.

[Wie83] Clark Wiedmann. A performance comparison between an APL interpreter and compiler. *ACM SIGAPL Quote Quad*, 13(3), March 1983.

[Wie85] Clark Wiedmann. Efficiency in the APL environment, a full arsenal for attacking CPU hogs. In *APL85 Conference Proceedings*, volume 15, pages 77–85. ACM SIGAPL Quote Quad, May 1985.

[Wie86] Clark Wiedmann. Field results with the APL compiler. In *APL86 Conference Proceedings*, volume 16, pages 187–196. ACM SIGAPL Quote Quad, July 1986.

[Wil91] R.G. Willhoft. Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4), 1991.

[WJ92] David B. Wortman and Michael D. Junkin. A concurrent compiler for modula-2+. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, volume 27, pages 68–81, June 1992.

[Wor79] David B. Wortman. On legality assertions in Euclid. *IEEE Transactions on Software Engineering*, SE-5(4), July 1979.

[WS81] Zvi Weiss and Harry J. Saal. Compile time syntax analysis of APL programs. In *APL81 Conference Proceedings*, volume 12, pages 313–320. ACM SIGAPL Quote Quad, October 1981.

[Yos86] Matuski Yoshino. APL as a prototyping language: Case study of a compiler development project. In *APL86 Conference Proceedings*, volume 16, pages 235–242. ACM SIGAPL Quote Quad, July 1986.

Appendix A

Selected Benchmark APL Source Code

A.1 APL Benchmark prd

```
r←prd y;t
⍝ Simple function for compiler
⍝ dcl integer scalar y
r←+/0.0E0+⍳y
```

A.2 APL Benchmark mdivr

```
r←mdivr n;x
⍝ dcl integer scalar n
⍝ Mdivr goes beyond mdiv2 in these ways:
⍝   Better id matrix gen in mmd3.
⍝   Floating point id matrix in mmd3.
⍝   Use of fn rank
x←((2⍴n)⍴(n+1)↑1.5-.5) ⍝ Id n
r←mmdr x
r←+/,|r

z←Singular y;j
⍝ We expect a Boolean scalar y
j←1⌷y
z←j=1

z←b dmdr a;v;m;n
⍝ Dyadic matrix divide using Householder
⍝ triangularization.
b←(2↑(⍴b),1)⍴b
z←b lsr a

z←b lsr a;i;j;sigma;alfa;aii;pp;pi;np;factor;eps;mv;
    m;n;u;ta;tb;tc;td;p;sink;t1;tc1;tc2;td1;td2;newta;newtb;tmp
⍝ M.A. Jenkins.
⍝ IBM Tech Report 320-2989
⍝ Modified for formatting and
⍝ to avoid use of globals, glueing.
⍝ Modified further to run a wee bit faster 1996-03-20 /rbe
⍝ Householder triangularization
⍝ of m×n matrix for linear
⍝ systems and least squares.
```

```
⍝ Does row, column interchanges,
⍝ scales columns near unity.
⍝ assumes (⍴⍴a)=(⍴⍴b)=2
⍝ Modified 1995-05-22 to be origin 0/rbe
p←1↓⍴b
m←1↑⍴a
n←¯1↑⍴a
pp←0.0+⍳n ⍝ KLUDGE to avoid typeproblems in dfa
factor←÷⌈⌿|a divrk10 mv←⌈/|a ⍝ FUNCTION RANK!
a←a mpyrk11 factor ⍝ Function rank!
np←n+p
eps←(16*¯13)×⌈/mv
:for i :in ⍳n
mv←⌈⌿|(i,i)↓a
sink←Singular eps≤⌈/mv
pi←i+mv⍳⌈/mv
tmp←pp[i]
pp[i]←pp[pi]
pp[pi]←tmp
tmp←a[;i]
a[;i]←a[;pi]
a[;pi]←tmp
pi←i+(|i↓a[;i])⍳⌈/|i↓a[;i]
tmp←a[i;]
a[i;]←a[pi;]
a[pi;]←tmp
tmp←b[i;]
b[i;]←b[pi;]
b[pi;]←tmp
sigma←+/(i↓a[;i])*2
aii←a[i;i]
alfa←(1-2×0≤aii)×sigma*0.5
tmp←i↓a[;i]
tmp[0]←tmp[0]-alfa
newta←tmp+.×(i+0 1)↓a
newtb←tmp+.×(i,0)↓b
tb←÷sigma-aii×alfa
tc1←tmp∘.×tb×newta
tc2←tmp∘.×tb×newtb
td1←((i+0 1)↓a)-tc1
td2←((i,0)↓b)-tc2
a[i↓⍳m;(i+1)↓⍳n]←td1
b[i↓⍳m;]←td2
a[i;i]←alfa
:endfor
⍝
z←(n,p)⍴0.0
⍝
:for i :in ⌽⍳n
t1←b[i;]-a[i;]+.×z
z[i;]←t1÷a[i;i]
:endfor
⍝
z[pp;]←z
```

```
z←z mpyrk10 factor ⍝ Function rank!

z←mmdr a;sz;id
⍝Computes inverse by calling dmd with an
⍝ identity matrix as left argument.
sz←(⍴a)[0]
id←(2⍴sz)⍴(1+sz)↑1.0E0
z←id dmdr a

r←x divrk10 y
⍝r←x÷⍉(⌽⍴x)⍴y ⍝ x÷rank 1 0 y
r←x div ⍤1 0 y

r←x mpyrk10 y
⍝r←x×⍉(⌽⍴x)⍴y
r←x times ⍤1 0 y

r←x mpyrk11 y
⍝r←x×(⍴x)⍴y
r←x times2 ⍤ 1 1 y

r←x times y
r←x × y

r←x times2 y
r←x × y

r←x div y
r←x ÷ y
```

Appendix B

APEX-Generated Code for Selected Benchmarks

B.0.1 Generated SISAL Code for Benchmark prd

```
%$entry=prd
define prd
% Compiled by APEX at 1996-06-08 18:17:31.490
#include <stdlib.sis>
#include <dsfctl.sis>
#include <dscalar.sis>
#include <takedrop.sis>
#include <mmisc.sis>
#include <rank.sis>

function plusslxddx10(y1: array[double_real]
 returns double_real)
for y in y1
 returns value of sum DtoD(y)
end for
end function

%
% Start of function prd
%
function prd(y: integer ;
returns double_real)

let
TMP_14:= iotax01sy((y ),I);
TMP_15:= dsfctl011sx(dplus,D,D,( 0.0d0 ),D,double_real,(TMP_14 ),
        I,integer);
 r_0 := plusslxddx10( TMP_15 );
 r:= r_0 ;
in
r_0
end let
end function % prd
```

B.0.2 Generated SISAL Code for Benchmark mdivr

```
%$entry=mdivr
define mdivr
% Compiled by APEX at 1996-06-18 16:08:33.750
#include <stdlib.sis>
#include <dsfctl.sis>
#include <dscalar.sis>
#include <take.sis>
#include <drop.sis>
#include <mmisc.sis>
#include <rank.sis>
#include <from.sis>
function modxddx22(y2: array[array[double_real]]
```

```
 returns array[array[double_real]])
for y1 in y2
 s1 := for y0 in y1
 returns array of abs(DtoD(y0))
 end for
returns array of s1
end for
end function

function modxddx11(y1: array[double_real];
 returns array[double_real])
 for y0 in y1
 returns array of abs(DtoD(y0))
 end for
end function

function divxddx11(y1: array[double_real];
 returns array[double_real])
 for y0 in y1
 returns array of 1.0d0/DtoD(y0)
 end for
end function

function divxddx00(y: double_real
 returns double_real)
 1.0d0/DtoD(y)
end function

function comaxddx21(y2: array[array[double_real]]
 returns array[double_real])
 array_setl(for i in array_liml(y2),array_limh(y2) cross
 j in array_liml(y2[SISALIO]),array_limh(y2[SISALIO]) returns
 value of catenate array[SISALIO:y2[i,j]] end for,SISALIO)
end function

function rotrxiix11(y1: array[integer ]
 returns array[integer])
let
 n := array_size(y1);
in
 for i in SISALIO,SISALIO+n-1
 returns array of y1[(n-1)-i]
 end for
end let
end function

function rhoidd122(x1: array[integer]; y2: array[array[double_real]]
 returns array[array[double_real]])
 % error, identity, normal
VVRESHAPE(double_real)
let
 rows := ConformNonNegativeInt(ItoI(x1[0]));
 cols := ConformNonNegativeInt(ItoI(x1[1]));
 rowsy := array_size(y2);
 colsy := array_size(y2[SISALIO]);
in
 % Check for identity case
 if (rows =rowsy) & (cols = colsy)
 then y2 % Identity
 else
 let
 y1 := RAVEL2(y2);
 y1cols := array_size(y1);
 in
 for i in 0,rows-1
 returns array of vvreshape(cols,mod(i*cols,y1cols),y1)
 end for
 end let
 end if
end let
end function

function rhoiii001(x: integer;
```

```
 y: integer
 returns array[integer])
 array_fill(SISALIO,ConformNonNegativeInt(ItoI(x))+SISALIO-1,y)
end function

function rhoidd102(x1: array[integer];
 y0: double_real
 returns array[array[double_real]])
let
 rows := ConformNonNegativeInt(ItoI(x1[SISALIO]));
 cols := ConformNonNegativeInt(ItoI(x1[SISALIO+1]));
in
 for i in SISALIO,rows+SISALIO-1
 returns array of array_fill(SISALIO,SISALIO+cols-1,y0)
 end for
end let
end function

function rhoidd112(x1: array[integer];
 y1: array[double_real]
 returns array[array[double_real]])
 VVRESHAPE(double_real)
let
 rows := ConformNonNegativeInt(ItoI(x1[SISALIO]));
 cols := ConformNonNegativeInt(ItoI(x1[SISALIO+1]));
 colsy := array_size(y1);
in
 for i in 0,rows-1
 returns array of vvreshape(cols,mod(i*cols,colsy),y1)
 end for
end let
end function

function iotaddi100(x1: array[double_real];
 y0: double_real
 returns integer)
% General case:0
for initial
 i := array_liml(x1);
 lim := array_limh(x1);
 y := DtoD(y0);
 z := array_size(x1);
 topz := z;
 while (z=topz) & (i <= lim) repeat
 i := old i+1;
 z:= if (DtoD(x1[old i]))~=y then old z else old i end if;
 returns value of z+QUADIO
end for % End of cases
end function

function comaiii001(x: integer;
 y: integer;
 returns array[integer])
array[SISALIO: ItoI(x), ItoI(y)]
end function

function comaibi001(x: integer;
 y: boolean;
 returns array[integer])
array[SISALIO: ItoI(x), BtoI(y)]
end function

function comaiii111(x1: array[integer];
 y1: array[integer];
 returns array[integer])
ItoI1(x1) || ItoI1(y1)
end function

function comaibi101(x1: array[integer];
 y: boolean;
 returns array[integer])
array_addh(ItoI1(x1),BtoI(y))
end function
```

```
function indrdxdi1x00(lhs: array[double_real];
 ia0: integer;
 returns double_real)
 lhs[ItoI(ia0)-QUADIO]
end function


function indrdxdb1x00(lhs: array[double_real];
 ia0: boolean;
 returns double_real)
 lhs[BtoI(ia0)-QUADIO]
end function


function indrixib1x00(lhs: array[integer];
 ia0: boolean;
 returns integer)
 lhs[BtoI(ia0)-QUADIO]
end function

function indrdxdii2x000(lhs: array[array[double_real]];
 ia0: integer;
 ia1: integer
 returns double_real)
 indrdxdi1x00(lhs[ItoI(ia0)-QUADIO],ia1)
end function

function indrdxdix2x10x(lhs: array[array[double_real]];
 ia0: integer;
 returns array[double_real])
 lhs[ItoI(ia0)-QUADIO]
end function

function indrdxdxi2x1x0(lhs: array[array[double_real]];
 ia1: integer
 returns array[double_real])
let
 tmp:=for i in 0,array_limh(lhs)
 returns array of
 indrdxdi1x00(lhs[i],ia1)
 end for
in
array_setl(tmp,0)
end let
end function


function indsdddi1010(lhs: array[double_real];
 ia0: integer;
 rhs: double_real;
 returns array[double_real])
lhs[ItoI(ia0)-QUADIO: DtoD(rhs)]
end function

function indsdddb1010(lhs: array[double_real];
 ia0: boolean;
 rhs: double_real;
 returns array[double_real])
lhs[BtoI(ia0)-QUADIO: DtoD(rhs)]
end function

function indsdddi1020(lhs: array[double_real];
 ia0: integer;
 rhs: double_real;
 returns array[double_real])
lhs[ItoI(ia0)-QUADIO: DtoD(rhs)]
end function

function indsdddx102x(lhs: array[double_real];
 rhs: double_real
 returns array[double_real])
array_fill(0,array_limh(lhs), DtoD(rhs))
```

```
end function

function indsdddi1111(lhs: array[double_real];
 ia0: array[integer];
 rhs: array[double_real];
 returns array[double_real])
 for initial
 i:= 0;
 z:= lhs;
 lim:= array_limh(ia0);
 while i<=lim repeat
 indx:= ItoI(ia0[old i]);
 z:= old z[indx-QUADIO: DtoD(rhs[old i])];
 i:= old i +1;
 returns value of z
 end for
end function

function indsdddx111x(lhs: array[double_real];
 rhs: array[double_real]
 returns array[double_real])
for i in 0,array_limh(lhs)
returns array of DtoD(rhs[i])
end for
end function

function indsdddii20200(lhs: array[array[double_real]];
 ia0: integer;
 ia1: integer;
 rhs: double_real;
 returns array[array[double_real]])
let
 i:= ItoI(ia0)-QUADIO;
in
 lhs[i: indsdddi1020(lhs[i],ia1,rhs)]
end let
end function

function indsdddix2120x(lhs: array[array[double_real]];
 ia0: integer;
 rhs: array[double_real];
 returns array[array[double_real]])
let
 i:= ItoI(ia0)-QUADIO;
in
 lhs[i: rhs]
end let
end function

function indsdddxi212x0(lhs: array[array[double_real]];
 ia1: integer;
 rhs: array[double_real];
 returns array[array[double_real]])
 for i in 0,array_limh(lhs)
 returns array of
 indsdddi1010(lhs[i],ia1,rhs[i])
 end for
end function

function indsdddii22211(lhs: array[array[double_real]];
 ia0: array[integer];
 ia1: array[integer];
 rhs: array[array[double_real]];
 returns array[array[double_real]])
 for initial
 i:= 0;
 z:= lhs;
 lim:= array_limh(ia0);
 while i<=lim repeat
 indx:= ItoI(ia0[old i])-QUADIO;
 z:= old z[indx: indsdddi1111(old z[indx],ia1,rhs[old i])];
 i:= old i +1;
 returns value of z
```

```
 end for
end function

function indsdddix2221x(lhs: array[array[double_real]];
 ia0: array[integer];

 rhs: array[array[double_real]];
 returns array[array[double_real]])
 for initial
 i:= 0;
 z:= lhs;
 lim:= array_limh(ia0);
 while i<=lim repeat
 indx:= ItoI(ia0[old i])-QUADIO;
 z:= old z[indx: rhs[old i]];
 i:= old i +1;
 returns value of z
 end for
end function

function indsddddx2221x(lhs: array[array[double_real]];
 ia0: array[double_real];
 rhs: array[array[double_real]];
 returns array[array[double_real]])
 for initial
 i:= 0;
 z:= lhs;
 lim:= array_limh(ia0);
 while i<=lim repeat
 indx:= DtoI(ia0[old i])-QUADIO;
 z:= old z[indx: rhs[old i]];
 i:= old i +1;
 returns value of z
 end for
end function

function maxsl1xddx21(y2: array[array[double_real]]
 returns array[double_real])
 function vvr(x1: array[double_real];
 y1: array[double_real]
 returns array[double_real])
 % vector-vector reduction step for matrix
 array_setl(for x0 in x1 dot y0 in y1
 returns array of max(x0,DtoD(y0))
 end for,SISALIO)
 end function % vvr

 % Following is wrong. Want (_1 take rho y)rho 0
 if IsEmpty(y2) % case of 0 5 rho 0
 then array_fill(0,-1,MINFINITYD) % Gives empty vector !!
 else % Not empty on first axis
 % Reduce vector with vector. We would like to figure a way to
 % to this in parallel, but not today... 1996-02-11
 for initial
 i := 0;
 z1 := array_fill(0,array_limh(y2[0]),MINFINITYD);
 while i <= array_limh(y2) repeat
 i:= old i +1;
 z1 := vvr(old z1,y2[old i])
 returns value of z1
 end for
 end if
end function

function maxslxddx21(y2: array[array[double_real]]
 returns array[double_real])
 for y1 in y2
 t1 := for y in y1
 returns value of greatest DtoD(y)
 end for
 returns array of t1
 end for
end function
```

```
function maxslxddx10(y1: array[double_real]
 returns double_real)
for y in y1
 returns value of greatest DtoD(y)
end for
end function

function plusslxddx10(y1: array[double_real]
 returns double_real)
for y in y1
 returns value of sum DtoD(y)
end for
end function

%
% Start of function Singular
%
function Singular(y: boolean;
returns boolean)
let
j_0:= dsfctl000sx(ddiv,D,D,(true ),B,boolean,(y ),B,boolean);
 z_0:= dsfctl000sx(deq,D,B,(j_0 ),D,double_real,(true ),B,boolean);
 z:= z_0 ;
in
z_0
end let
end function % Singular

%
% Start of function times
%
function times(x: array[double_real];
y: double_real ;
returns array[double_real])
let
r_0:= dsfctl011sx(dmpy,D,D,(y ),D,double_real,(x ),D,double_real);
 r:= r_0 ;
in
r_0
end let
end function % times

%
% Start of function times2
%
function times2(x: array[double_real];
y: array[double_real];
returns array[double_real])
let
r_0:= dsfctl111(dmpy,D,D,(x ),D,double_real,(y ),D,double_real);
 r:= r_0 ;
in
r_0
end let
end function % times2

%
% Start of function div
%
function div(x: array[double_real];
y: double_real ;
returns array[double_real])
let
r_0:= dsfctl101sy(ddiv,D,D,(x ),D,double_real,(y ),D,double_real);
 r:= r_0 ;
in
r_0
end let
end function % div

%
% Start of function divrk10
```

```
%
function divrk10(x: array[array[double_real]];
y: array[double_real] ;
returns array[array[double_real]])
let
r_0:= drank2110(div,x ,y );
 r:= r_0 ;
in
r_0
end let
end function % divrk10

%
% Start of function mpyrk10
%
function mpyrk10(x: array[array[double_real]];
y: array[double_real] ;
returns array[array[double_real]])
let
r_0:= drank2110(times,x ,y );
 r:= r_0 ;
in
r_0
end let
end function % mpyrk10

%
% Start of function mpyrk11
%
function mpyrk11(x: array[array[double_real]];
y: array[double_real] ;
returns array[array[double_real]])
let
r_0:= drank2111(times2,x ,y );
 r:= r_0 ;
in
r_0
end let
end function % mpyrk11

%
% Start of function lsr
%
function lsr(b: array[array[double_real]];
a: array[array[double_real]];
returns array[array[double_real]])

function plusdotmpyddd121(x1: array[double_real];
 y2: array[array[double_real]]
 returns array[double_real])

 function vvr(x1: array[double_real]; y1: array[double_real]
 returns array[double_real])
 % vector-vector reduction step for matrix
 array_setl(for x0 in x1 dot y0 in y1 returns array of x0+y0
 end for,SISALIO)
 end function % vvr

 function svprod(x0: double_real; y1: array[double_real]
 returns array [double_real])
 for y0 in y1 returns array of x0*y0 end for
 end function

for initial
 z:= svprod(x1[array_liml(x1)],y2[array_liml(y2)]);
 i:= array_liml(x1)+1;
 lim:= array_limh(x1);
 limdif := array_liml(y2)-array_liml(x1); % SISALIO differences
while i <= lim repeat
 i := old i +1;
 z:=vvr(old z, svprod(x1[old i],y2[limdif+ old i]));
 returns value of z
end for
```

```
end function

function jotdotmpyddd112(x1: array[double_real];
 y1: array[double_real]
 returns array[array[double_real]])
 for x0 in x1 cross y0 in y1
 returns array of (x0)*DtoD(y0)
 end for
end function

let
TMP_135:= rhox21 ((b ),D);
 p_0:= ddrop011p1((true ),B,(TMP_135 ),I,0);
 TMP_137:= rhox21 ((a ),D);
 m_0:= dtake011p1((true ),B,(TMP_137 ),I,0);
 TMP_139:= rhox21 ((a ),D);
 n_0:= dtake011n1((-1 ),I,(TMP_139 ),I,0);
 TMP_141:= iotax11sy((n_0 ),I);
 pp_0:= dsfctl011sx(dplus,D,D,( 0.0d0 ),D,double_real,(TMP_141 ),
        I,integer);
 TMP_143 := modxddx22 ( a );
 mv_0 := maxslxddx21 ( TMP_143 );
 TMP_145 := divrk10 (a ,mv_0 );
 TMP_146 := modxddx22 ( TMP_145 );
 TMP_147 := maxsl1xddx21 ( TMP_146 );
 factor_0 := divxddx11 ( TMP_147 );
 a_0 := mpyrk11 (a ,factor_0 );
 np_0:= dsfctl111sx(dplus,I,I,(n_0 ),I,integer,(p_0 ),I,integer);
 TMP_151 := maxslxddx10 ( mv_0 );
 TMP_152:= dsfctl000sx(dstar,D,D,(16 ),I,integer,(-13 ),I,integer);
 eps_0:= dsfctl000sx(dmpy,D,D,(TMP_152 ),D,double_real,(TMP_151 ),
         D,double_real);
 TMP_154:= iotax11sy((n_0 ),I);
 pp_1,a_1,b_0,i_0:=for initial
CTR__155:= 0;
CTR__155z := array_limh(TMP_154 );
pp_3 := (pp_0);
a_7 := (a_0);
b_3 := (b);
i_0 := (0);
while (CTR__155<=CTR__155z)repeat
i_0 := TMP_154 [old CTR__155];
CTR__155 := 1+old CTR__155;
 TMP_160 := comaiii001 (i_0 ,i_0 );
 TMP_161:= ddrop122((TMP_160 ),I,(old a_7 ),D,0.0d0);
 TMP_162 := modxddx22 ( TMP_161 );
 mv_2 := maxsl1xddx21 ( TMP_162 );
 TMP_164 := maxslxddx10 ( mv_2 );
 TMP_165:= dsfctl000sx(dle,D,B,(eps_0 ),D,double_real,(TMP_164 ),
           D,double_real);
 sink_0 := Singular ( TMP_165 );
 TMP_167 := maxslxddx10 ( mv_2 );
 TMP_168 := iotaddi100 (mv_2 ,TMP_167 );
 pi_1:= dsfctl000sx(dplus,I,I,(i_0 ),I,integer,(TMP_168 ),I,integer);
 tmp_1 := indrdxdi1x00 (old pp_3 ,i_0);
 TMP_173 := indrdxdi1x00 (old pp_3 ,pi_1);
 TMP_175 := indsdddi1010 (old pp_3 ,i_0 ,TMP_173);
 pp_2 := ( TMP_175 );
 TMP_178 := indsdddi1010 (pp_2 ,pi_1 ,tmp_1);
 pp_3 := ( TMP_178 );
 tmp_2 := indrdxdxi2x1x0 (old a_7 ,i_0);
 TMP_185 := indrdxdxi2x1x0 (old a_7 ,pi_1);
 TMP_188 := indsdddxi212x0 (old a_7 ,i_0,TMP_185);
 a_2 := ( TMP_188 );
 TMP_192 := indsdddxi212x0 (a_2 ,pi_1,tmp_2);
 a_3 := ( TMP_192 );
 TMP_196 := indrdxdxi2x1x0 (a_3 ,i_0);
 TMP_197:= ddrop011((i_0 ),I,(TMP_196 ),D,0.0d0);
 TMP_198 := modxddx11 ( TMP_197 );
 TMP_199 := maxslxddx10 ( TMP_198 );
 TMP_202 := indrdxdxi2x1x0 (a_3 ,i_0);
 TMP_203:= ddrop011((i_0 ),I,(TMP_202 ),D,0.0d0);
 TMP_204 := modxddx11 ( TMP_203 );
```

```
TMP_205 := iotaddi100 (TMP_204 ,TMP_199 );
pi_2:= dsfctl000sx(dplus,I,I,(i_0 ),I,integer,(TMP_205 ),I,integer);
tmp_3 := indrdxdix2x10x (a_3 ,i_0);
TMP_212 := indrdxdix2x10x (a_3 ,pi_2);
TMP_215 := indsdddix2120x (a_3 ,i_0 ,TMP_212);
a_4 := ( TMP_215 );
TMP_219 := indsdddix2120x (a_4 ,pi_2 ,tmp_3);
a_5 := ( TMP_219 );
tmp_4 := indrdxdix2x10x (old b_3 ,i_0);
TMP_226 := indrdxdix2x10x (old b_3 ,pi_2);
TMP_229 := indsdddix2120x (old b_3 ,i_0 ,TMP_226);
b_1 := ( TMP_229 );
TMP_233 := indsdddix2120x (b_1 ,pi_2 ,tmp_4);
b_2 := ( TMP_233 );
TMP_237 := indrdxdxi2x1x0 (a_5 ,i_0);
TMP_238:= ddrop011((i_0 ),I,(TMP_237 ),D,0.0d0);
TMP_239:= dsfctl101sy(dstar,D,D,(TMP_238 ),D,double_real,(2 ),I,integer);
sigma_0 := plusslxddx10 ( TMP_239 );
aii_0 := indrdxdii2x000 (a_5 ,i_0,i_0);
TMP_244:= dsfctl000sx(dstar,D,D,(sigma_0 ),D,double_real,( 0.5d0 ),
          D,double_real);
TMP_245:= dsfctl000sx(dle,D,B,(false ),B,boolean,(aii_0 ),D,double_real);
TMP_246:= dsfctl000sx(dmpy,I,I,(2 ),I,integer,(TMP_245 ),B,boolean);
TMP_247:= dsfctl000sx(dbar,I,I,(true ),B,boolean,(TMP_246 ),I,integer);
alfa_0:= dsfctl000sx(dmpy,D,D,(TMP_247 ),I,integer,(TMP_244 ),
         D,double_real);
TMP_251 := indrdxdxi2x1x0 (a_5 ,i_0);
tmp_5:= ddrop011((i_0 ),I,(TMP_251 ),D,0.0d0);
TMP_254 := indrdxdb1x00 (tmp_5 ,false);
TMP_255:= dsfctl000sx(dbar,D,D,(TMP_254 ),D,double_real,(alfa_0 ),
          D,double_real);
TMP_257 := indsdddb1010 (tmp_5 ,false ,TMP_255);
tmp_6 := ( TMP_257 );
TMP_259:= dsfctl011sx(dplus,I,I,(i_0 ),I,integer,
          (array[0: false,true]),B,boolean);
TMP_260:= ddrop122((TMP_259 ),I,(a_5 ),D,0.0d0);
newta_0 := plusdotmpyddd121(tmp_6 ,TMP_260 );
TMP_262 := comaibi001 (i_0 ,false );
TMP_263:= ddrop122((TMP_262 ),I,(b_2 ),D,0.0d0);
newtb_0 := plusdotmpyddd121(tmp_6 ,TMP_263 );
TMP_265:= dsfctl000sx(dmpy,D,D,(aii_0 ),D,double_real,(alfa_0 ),
          D,double_real);
TMP_266:= dsfctl000sx(dbar,D,D,(sigma_0 ),D,double_real,(TMP_265 ),
          D,double_real);
tb_0 := divxddx00 ( TMP_266 );
TMP_268:= dsfctl011sx(dmpy,D,D,(tb_0 ),D,double_real,(newta_0 ),
          D,double_real);
tc1_0 := jotdotmpyddd112 (tmp_6 ,TMP_268 );
TMP_270:= dsfctl011sx(dmpy,D,D,(tb_0 ),D,double_real,(newtb_0 ),
          D,double_real);
tc2_0 := jotdotmpyddd112 (tmp_6 ,TMP_270 );
TMP_272:= dsfctl011sx(dplus,I,I,(i_0 ),I,integer,
          (array[0: false,true]),B,boolean);
TMP_273:= ddrop122((TMP_272 ),I,(a_5 ),D,0.0d0);
td1_0:= dsfctl222(dbar,D,D,(TMP_273 ),D,double_real,(tc1_0 ),
        D,double_real);
TMP_275 := comaibi001 (i_0 ,false );
TMP_276:= ddrop122((TMP_275 ),I,(b_2 ),D,0.0d0);
td2_0:= dsfctl222(dbar,D,D,(TMP_276 ),D,double_real,(tc2_0 ),
        D,double_real);
TMP_278:= iotax11sy((n_0 ),I);
TMP_279:= dsfctl000sx(dplus,I,I,(i_0 ),I,integer,(true ),B,boolean);
TMP_280:= ddrop011((TMP_279 ),I,(TMP_278 ),I,0);
TMP_281:= iotax11sy((m_0 ),I);
TMP_282:= ddrop011((i_0 ),I,(TMP_281 ),I,0);
TMP_285 := indsdddii22211 (a_5 ,TMP_282 ,TMP_280 ,td1_0);
a_6 := ( TMP_285 );
TMP_287:= iotax11sy((m_0 ),I);
TMP_288:= ddrop011((i_0 ),I,(TMP_287 ),I,0);
TMP_291 := indsdddix2221x (b_2 ,TMP_288 ,td2_0);
b_3 := ( TMP_291 );
TMP_295 := indsdddii20200 (a_6 ,i_0,i_0,alfa_0);
a_7 := ( TMP_295 );
```

```
 returns
 value of pp_3
 value of a_7
 value of b_3
 value of i_0
end for;
 TMP_298 := comaiii111 (n_0 ,p_0 );
 z_0 := rhoidd102 (TMP_298 , 0.0d0 );
 TMP_300:= iotax11sy((n_0 ),I);
 TMP_301 := rotrxiix11 ( TMP_300 );
 i_1,z_1:=for initial
CTR__302:= 0;
CTR__302z := array_limh(TMP_301 );
i_1 := (i_0);
z_2 := (z_0);
while (CTR__302<=CTR__302z)repeat
i_1 := TMP_301 [old CTR__302];
CTR__302 := 1+old CTR__302;
 TMP_307 := indrdxdix2x10x (a_1 ,i_1);
 TMP_308 := plusdotmpyddd121(TMP_307 ,old z_2 );
 TMP_311 := indrdxdix2x10x (b_0 ,i_1);
 t1_0:= dsfctl111(dbar,D,D,(TMP_311 ),D,double_real,(TMP_308 ),
        D,double_real);
 TMP_315 := indrdxdii2x000 (a_1 ,i_1,i_1);
 TMP_316:= dsfctl101sy(ddiv,D,D,(t1_0 ),D,double_real,(TMP_315 ),
           D,double_real);
 TMP_319 := indsdddix2120x (old z_2 ,i_1 ,TMP_316);
 z_2 := ( TMP_319 );
 returns
 value of i_1
 value of z_2
end for;
 TMP_324 := indsddddx2221x (z_1 ,pp_1 ,z_1);
 z_3 := ( TMP_324 );
 z_4 := mpyrk10 (z_3 ,factor_0 );
 z:= z_4 ;
in
z_4
end let
end function % lsr

%
% Start of function dmdr
%
function dmdr(b: array[array[double_real]];
a: array[array[double_real]];
returns array[array[double_real]])
let
TMP_19:= rhox21((b ),D);
 TMP_20 := comaibi101(TMP_19 ,true );
 TMP_21:= dtake011pi((2 ),I,(TMP_20 ),I,0);
 b_0 := rhoidd122 (TMP_21 ,b );
 z_0 := lsr (b_0 ,a );
 z:= z_0 ;
in
z_0
end let
end function % dmdr

%
% Start of function mmdr
%
function mmdr(a: array[array[double_real]];
returns array[array[double_real]])
let
TMP_24:= rhox21((a ),D);
 sz_0 := indrixib1x00(TMP_24 ,false);
 TMP_26:= dsfctl000sx(dplus,I,I,(true ),B,boolean,(sz_0 ),I,integer);
 TMP_27:= dtake001((TMP_26 ),I,( 1.0d0 ),D,0.0d0);
 TMP_28 := rhoiii001 (2 ,sz_0 );
 id_0 := rhoidd112 (TMP_28 ,TMP_27 );
 z_0 := dmdr (id_0 ,a );
 z:= z_0 ;
```

```
in
z_0
end let
end function % mmdr

%
% Start of function mdivr
%
function mdivr(n: integer ;
returns double_real)
let
TMP_21:= dsfctl000sx(dbar,D,D,( 1.5d0 ),D,double_real,(0.5d0 ),
         D,double_real);
 TMP_22:= dsfctl000sx(dplus,I,I,(n ),I,integer,(true ),B,boolean);
 TMP_23:= dtake001((TMP_22 ),I,(TMP_21 ),D,0.0d0);
 TMP_24 := rhoiii001 (2 ,n );
 x_0 := rhoidd112 (TMP_24 ,TMP_23 );
 r_0 := mmdr ( x_0 );
 TMP_27 := modxddx22 ( r_0 );
 TMP_28 := comaxddx21 ( TMP_27 );
 r_1 := plusslxddx10( TMP_28 );
 r:= r_1 ;
in
r_1
end let
end function % mdivr
```