Engineering An Interprocedural Optimizing Compiler

Jon Loeliger
Robert Metzger

CONVEX Computer Corporation
3000 Waterview Parkway
Richardson, Texas 75083-3851

April, 1994

Abstract

CONVEX Computer Corp. has developed a language-independent interprocedural optimizer that is now available to users of its C-Series supercomputers. This optimizer is packaged together with FORTRAN and C compilers in a product called the Application Compiler. This product was developed to enable engineers and scientists to develop supercomputer applications that run faster and that take less effort to optimize and debug.

The project goals were increased automatic optimization, language-independent optimization, I/O device independence, application independence, co-existence with current development tools, reasonable compilation speed, and ease of use.

The interprocedural analyzer performs call analysis, alias analysis, pointer tracking, scalar analysis, constant propagation, inline analysis, clone analysis, array analysis, storage optimization, and error analysis. The execution order of the algorithms is basically name binding, side effect analysis, and optimization. The order of the algorithms changed during the implementation based upon user input and insights into the relationships between the algorithms. Significant changes were made to the user interface during the project, based upon input from early users.

Introduction

There were three main reasons the Application Compiler was developed. Each of them were based on the needs of technical computing users.

Scientists and engineers use supercomputers because they enable them to solve real-world problems in less time than other means might provide. Thus the first reason for this product was to make applications execute faster. Some supercomputer users leverage faster application execution into more jobs run, while others increase the size of the data sets they are processing. In either case, they are enabled to do their job more effectively. Interprocedural analysis increases the information the procedure compiler has, thus enabling it to do more optimizations.

The second reason for this product was to enable supercomputer users to develop applications more quickly. Supercomputer users are not generally computer scientists, but professionals who use computers to get their jobs done. Such users want to minimize the time they must spend preparing an application. If the compiler does optimizations automatically that they would otherwise have to do manually, they can spend their time solving problems in their discipline, rather than profiling and modifying their programs. Compilers that do interprocedural analysis decrease development time in several ways. They preserve the modular structure of an application, reducing design and integration time. They make applications run faster, reducing the time needed to run tests. They find errors that procedure compilers don't find, thus reducing debugging time and increasing application quality.

The third reason this product was developed was to provide a bridge to future massively parallel architectures. These systems offer the promise of great performance, but we have found that most engineers and scientists who use high performance computers are unwilling to endure the pain of recoding their application to make use of such parallelism. The Application Compiler provides a base-level technology to support automatic data decomposition for parallel processing. Efficient use of parallel processors requires knowledge of the entire application, not just a single procedure.

Project Goals

The basic purpose of this project was to provide a compiler that performed interprocedural analysis for optimization. The specific project goals were developed by examining the existing research projects [1] [2] [3] [4] in interprocedural analysis. After considering the needs of our customers, we chose seven goals, each of which differed in varying degrees from the approaches taken by the research projects.

These major project goals are explained below.

Automatic Optimization

Most interprocedural research projects have required the user to interact with the compiler and environment. We knew that a system that required hours of interactive use to achieve better performance would be ignored by most of our users. We decided to continue in the direction that we have achieved success with until now – to provide maximum automatic optimization.

Language Independent Optimization

Almost all interprocedural research projects have only handled FORTRAN code. While almost all CONVEX customer systems have licensed the FORTRAN compiler, two thirds of them have licensed the optimizing C compiler, and several dozen have licensed the Ada compiler. All three compilers perform automatic vectorization and parallelization. We decided we must meet the needs of all our customers.

I/O Device Independence

Most interprocedural research projects have required the user to work at relatively expensive bit-mapped graphics workstations. Many organizations that use supercomputers have a large investment in traditional ASCII terminals. They expect our software products to support those devices. We decided that our interprocedural optimizer must work equally well whether the user is logged in at a glass teletype or an X-Windows workstation.

Application Independence

Most interprocedural research projects have focused on finding high-level parallelism. This approach did not make sense to us for two reasons. First, most of our customers have either our first-generation vector processor (C1), or have our second-generation vector-parallel processor (C2) in a minimal configuration – one CPU. Second, we knew that several interprocedural techniques held great promise for improving vector and scalar application performance. We decided that we must not limit our optimizations to those that would help only applications that were inherently parallelizable. Instead we would improve a wide range of applications including those that were inherently vectorizable or inherently scalar-only.

Co-existence with Current Development Tools

Almost all interprocedural research projects have been part of a larger software development environment. Software development environments include proprietary source editors, revision control systems, debuggers, profilers, etc. Many users have an emotional attachment to the editor they currently use. We did not want to have to proselytize for followers of a new editor. We decided to exclude the following features from this product: syntax-directed editor, revision control system, Symbolic debugger, execution profiler.

Reasonable Compilation Speed

Many interprocedural research projects have used algorithms that terminate in a reasonable time only for small programs. No matter how much we improve the compilation speeds of our existing compilers, users always ask us to make them faster. We decided that we would not bring a product to the marketplace unless compile times for large applications were measured in hours, not days.

Easy to Learn

All new compilation systems require some learning by the user. User interfaces to unfamiliar products are easier to learn when they are patterned after a familiar product. Since most UNIX(tm) users are familiar with the *make* utility, we decided to pattern the user interface of our compiler after it.

Architecture of the Compiler

The Application Compiler consists of a number of executables and data files. The compilation process is controlled by the single executable, *build.* It both manages the execution of the other compiler passes and serves as the user interface to the Application Compiler.

The compilation process centers around the "program data base". It is a memory mapped shared disk file that contains intermediate representation of the application and results of the analyses. It also contains information used to coordinate the compilation process. Each phase communicates with other phases via the program data base.

Figure 1 shows the execution control and data flow within the entire Application Compiler. A brief synopsis of each executable is given below in the order in which they are executed.

Application Compiler driver - *build* is the program the user invokes. It performs Source Analysis to determine which files must be recompiled. It initializes the program database.

Language driver - The *build* driver invokes the *fc* and *cc* programs. They are similar to the drivers used by the regular CONVEX FORTRAN and C compilers. They process command line options, resolve library pathnames, and invoke the preprocessors as needed.

Language front end - *ffront* and *cfront* are the FORTRAN and C front ends respectively. They perform lexical, syntactic, and semantic analysis. They also perform procedure-wide scalar optimization, and then write symbol tables and annotated syntax trees to the program data base.

IPO Pass 1 - *synth1* is the first interprocedural optimization (IPO) analysis phase. It performs some interprocedural error checking, call analysis, alias analysis, and pointer tracking. It reads from the program data base, and writes additional information back to it.

Procedure Analysis - *mend* performs procedure-wide scalar optimization a second time, using the results of the first pass of interprocedural analysis. It reads symbol tables, syntax trees, and interprocedural information from the data base, and writes data structures needed by the second pass of interprocedural analysis back to the data base.

IPO Pass 2 - *synth2* is the second interprocedural optimization analysis phase. It performs interprocedural scalar analysis, constant propagation, array analysis, clone analysis, inline analysis and error analysis. It reads from the program data base, and writes additional information back to it.

Back end - *bend* performs procedure-wide scalar optimization a third time, using the results of the second pass of interprocedural analysis. Then it performs vectorization, parallelization, and code generation. It reads from the program data base and writes object files to disk.

Linker - The Application Compiler always produces one object file for each procedure, regardless of the original source file structure. The procedures are put into an order that maximizes locality of reference within memory pages and the instruction cache. The standard CONVEX OS linker, *ld,* creates an executable image from libraries and object files created by the compiler back end.

Interprocedural Analysis and Optimization

Interprocedural analysis and optimization is a series of passes over a database that contains information about all the procedures in the application. Interprocedural analysis is performed to provide precise information in situations where traditional compilers make worst-case assumptions.

Traditional compilers make the following assumptions:

- Any procedure can be the referent of an indirect call.
- No argument of a FORTRAN procedure is aliased with another argument, or with a COMMON variable, if the argument is assigned.
- Any location in memory can be pointed at by a C pointer.
- All global scalars and all by-reference scalar arguments are used and assigned by a called procedure.
- All elements of all global arrays and all elements of by-reference array arguments are used and assigned by a called procedure.

Interprocedural analysis provides a procedure compiler with precise information so that it doesn't have to make unrealistic assumptions. This makes it possible to perform more optimizations.

Interprocedural optimizations go beyond correcting worst-case assumptions to actually changing the body of procedures. The interprocedural optimizations performed by this compiler were chosen for two main reasons:

- to enhance the effectiveness of dependency analysis, and
- to enhance the usage of the memory hierarchy of high performance computers.

Array subscript dependency analysis is essential for effective optimization on high performance computers, whether they be pipelined vector processors, massively parallel systems, or RISC workstations with large caches. All high performance computers use a memory hierarchy to increase performance. Effective use of the hierarchy is essential to good application performance.

The interprocedural passes are described below.

Call Analysis answers the question: Which procedures are invoked by a call? This would seem to be a trivial problem. FORTRAN allows procedure dummy arguments, however, and C provides for passing the addresses of functions that can be invoked by indirection. This means that only interprocedural compilers can have complete knowledge of which procedures invoke which other procedures. This pass also determines which library procedures are called, and generates information about these procedures for the other passes.

Figure 3 shows an example where call analysis is needed. The function to be evaluated is passed by reference as an argument. Call analysis will inspect every call to *eval* and determine the list of procedures that can be called from eval.

Alias Analysis [5] [6] answers the question: Which names refer to the same location? It determines the aliases of all globals and of each formal of each procedure. The results of alias analysis are used by the algorithms that follow to adjust for the effects of aliasing.

Figure 4 shows an example of the usefulness of alias analysis. The code shown is not valid FORTRAN. The ANSI standard explicitly disallows making an assignment to a storage location when there is more than one name for same location in a program unit. Optimizing compilers simply assume that programmers obey this rule. In some cases, they can unwittingly generate incorrect code. For example, this code will be vectorized by some vectorizing FORTRAN compilers, even though they should not. The Application Compiler will find the alias and warn about it.

Pointer Tracking [7] answers the question: Which pointers point to which locations? Pointer Tracking improves the optimization of(CW procedures. Without it, a safe optimizing C compiler must assume that any pointer can point at any location in memory that contains the appropriate pointee type. Such assumptions lead to crippling aliases that decrease automatic vectorization and parallelization of C. Pointer tracking distinguishes pointer targets symbolically and by storage class (static, automatic, heap). Providing an aggressive pointer tracking algorithm was essential to our goal of providing language-independent optimization.

Figure 5 provides an example of how interprocedural pointer tracking helps optimization. A vectorizing C compiler cannot safely vectorize the loop in the subfunction without knowing where in memory the pointers point. An interprocedural compiler which performs pointer tracking knows that the arguments of this function never point at the same location, and thus it is safe to vectorize the loop.

Scalar Analysis [8] [9] answers the question: Which procedures (and subordinates) use and assign which scalars? Scalar Analysis summarizes for every procedure call the usage of every scalar by that procedure and every procedure it invokes directly or indirectly. Such references are classified according to whether the variable may be used (USE), may be assigned (ASG), or is definitely assigned (KILL).

Figure 6 contains an example which shows why scalar analysis is helpful. A procedure compiler must assume that all global variables are modified by a procedure call. In the example, the results of scalar analysis enable an interprocedural optimizer to substitute the constant 5.0 for the variable a in the assignment to c. It knows that the called procedure does not set the value of the variable a, so it still has the constant value after the subroutine completes. The assignment statement can now be evaluated at compile time, and the multiply is eliminated from the code.

Constant Propagation [11] [12] answers the question: Which globals and arguments are always constant on entry to which procedures? This algorithm performs a symbolic interpretation of the program to find constants arising from static initializations, assignments, and argument passing.

Figure 7 is an example of how interprocedural constant propagation can aid other optimizations. In the absence of information about the argument m a vectorizing compiler cannot vectorize the loop contained in the subroutine. In an interprocedural compiler, constant propagation determines that the argument always has the value 200, and substitutes this into the subroutine. With this additional information, the compiler can vectorize this loop.

Inline Analysis answers the question: Which procedures should be inlined at which call sites? Inline substitution serves two purposes: It eliminates call overhead and tailors the called procedure to the particular set of arguments passed at a given call site. Procedure inlining can be performed manually, and some existing compilers will perform inline expansion if the user manually specifies the calls to replace. Providing a fully automatic inlining system helped achieve our goal of automatic optimization.

Inline analysis chooses procedures based on size. The smaller the procedure, the larger the percentage of its execution is call overhead, and the greater the benefit of inlining. Inline analysis chooses call sites based on frequency of execution. Call overhead on CONVEX C-series systems is low enough that it is not worth eliminating unless the call is in a loop (directly or indirectly). By selecting call sites that are executed most frequently, inline analysis removes barriers to parallelization from those loops that will provide the greatest gain if executed concurrently.

Clone Analysis answers the question: Which procedures would benefit by absorbing a constant on entry? Cloning a procedure results in a version of the callee procedure that has been tailored to one or more specific call sites where certain variables are known to be constant on entry.

Figure 8 shows how procedure cloning assists constant propagation, which in turn makes other optimizations possible. In the absence of information about the arguments n and k, a vectorizing compiler cannot vectorize the loop contained in the subroutine. In an interprocedural compiler, procedure cloning determines that if a copy was made of the subroutine, then constant values for both arguments could be propagated. After the copy is made, -1 is substituted for the argument k in the original, and 1 is substituted in the copy. With this additional information, the compiler can vectorize this loop.

Array Analysis [2] [15] answers the question: Which procedures (and subordinates) use and assign which sections of arrays? The primary reason for array analysis is to make parallelization of loops that contain procedure calls possible. If each invocation of a procedure in a loop processes a different section of an array, then that loop may be a candidate for parallel execution.

Array Analysis summarizes for every procedure call the usage of every array used or assigned by that procedure and every procedure it invokes directly or indirectly. Dependency analysis in the vectorizer/parallelizer can use the array summaries to determine whether there are any loop carried dependencies in the loop, that would prevent parallelization. The results of Array Analysis can also be used to partition data on massively parallel distributed memory systems.

Figure 9 gives an example of array section analysis. Compilers which do not perform interprocedural analysis cannot automatically parallelize loops that contain subroutine calls. In this case, array analysis summarizes the side effects of the call as `A(I,1:100)=.` This means that each invocation assigns elements one through one hundred of column I of the array A. Since each iteration of the loop is processing an independent section of the array, the loop in the main program can be run in parallel.

Storage Optimization answers the question: How should application data structures be rearranged to improve the usage of the memory hierarchy? On a system with banks of interleaved memory, it is important to ensure that arrays are structured so that the elements are spread over the banks. This can be accomplished by extending the dimensions of the arrays where necessary, if it is safe to do this. The results of alias and array analysis provide the information to make this decision.

On a system with data cache lines that contain more than one word, it can be quite useful to group related scalar variables together. When one of them is fetched into the cache, the others come along for free, and subsequent loads of these variables come from the cache, not main memory.

Error Analysis answers questions such as: Which procedures use uninitialized variables? Which procedures have array references which may have invalid subscripts?

The cloning and inlining algorithms typify application independent optimizations. Both inline and clone analysis modify a procedure to the use at a particular call site. The resulting code benefits from a full range of optimizations. Procedure cloning is designed to yield better vectorization, but we often see a scalar benefit from dead code removal or new constants available. Procedure inlining can result in both new vectorizable loops as well as the expected scalar benefit of call overhead removal. We cover a wide range of application types and still see improvements without relying on any one specific type of optimization.

Algorithm Execution Order

A combination of user feedback, experience, and technical necessity suggested a different order of execution for the interprocedural algorithms than was envisioned in the design document. The graph in Figure 2 shows a partial ordering between the interprocedural algorithms. There is an arc from one algorithm to another if the first algorithm produces information that can be used by the second.

For simplicity, this graph does not show all the ordering arcs. Each of the interprocedural analysis algorithms relies on an accurate call graph for the entire application. Without the call graph, there is no way to propagate information from one procedure to another. Thus, the call analysis must be performed early in the interprocedural analysis.

The first three interprocedural analyses to be executed – call analysis, alias analysis, and pointer tracking – perform different aspects of name binding. This is the process of determining to which procedure or variable a symbol refers in any given context. For example, call analysis determines indirect call targets and enables the creation of a complete call graph.

Alias analysis and pointer tracking conceptually perform the same name binding functionality. Alias analysis tracks the aliases created when using the pass-by-reference calling semantics of FORTRAN. It determines memory overlaps for all symbols in an application. Pointer tracking was implemented to handle the name binding caused by taking addresses of variables, and allocating memory on the heap in C. It determines refined target sets for indirections.

Clone analysis was originally placed before call analysis, but was later placed after constant propagation. When we designed the Application Compiler, we knew that there would be a classic catch-22 between the clone analysis and constant propagation. For clone analysis to be effective, it must know what constants can be propagated. For constant propagation to be effective, it must know all the routines, including the clones, to which it could propagate a constant.

To break the dependency each algorithm had on the other, we originally defined clone analysis as a heuristic process that would precede call analysis. It would make guesses about which procedures to clone so that the call graph could be completed. In this way, constant propagation would find and propagate constants into the extant clones at the correct place in the call graph.

In practice, it was difficult to know beforehand what were good candidates for cloning. Once constant propagation was coded, it was clear from the available constant sets what the possible clones were. Choosing good clones was then a separate matter of performance tuning. The decision to move clone analysis from an early heuristic to a later more certain algorithm was easy to make.

In a few instances, the algorithms could be executed in several possible orders. In some of these cases, early feedback from users of the compiler enabled us to make informed choices on algorithm ordering decisions.

A good example of this ordering decision is the interaction between inline and clone analysis. When cloning was developed, it was an integral part of constant propagation. When inlining was coded it occurred after constant propagation and thus, after cloning. Both user preference and technical reasons led to the separation of clone analysis from constant propagation.

Empirically, users saw better improvements with inlining than with cloning and thus wanted inlining to override cloning. Technically, since an inline copy of a procedure both eliminated the call overhead and tailored the callee's code more thoroughly to a call site than a simple cloning, it made sense to have inlining override cloning. Thus we chose to do inline analysis before clone analysis, separating clone analysis from constant propagation.

User Interface

Two primary design goals that applied to the user interface of the Application Compiler are to be I/O device independent and easy to learn.

To maintain I/O device independence, we decided that the Application Compiler should not be an interactive program. Instead, all input comes from command line options and a file that describes the composition of the application. All output for the user is directed to the terminal.

The user calls the Application Compiler through the *build* program. The *build* program is analogous to *make.* It specifies from *what* an application is to be built and *how* the application is to be built. Just as *make* reads the specification from the file *makefile* or *Makefile* command line option, *build* reads the specification from the file *buildfile* or *Buildfile.*

The *build* program recognizes numerous command line options. For familiarity and easy of learning, several *make* options are recognized by *build.* In addition, *build* has several options to control the compilation and output that are specific to the Application Compiler.

Unlike *make,* the *buildfile* does not specify dependencies between source files. These are deduced automatically and do not require any user intervention.

The *buildfile* grammar is simple. The buildfile can contain any of the lines in Figure 10 in any order. The link line must occur at least once. All others can occur any number of times.

Specifying a directory name tells the *build* program to compile all the FORTRAN and C source files found in that directory with the compiler options specified.

The first *buildfile* in Figure 11 suffices for many applications. We believe this basic user interface is simple and easy to learn. In this example, all the files in the current directory will be compiled at optimization level -O2 and linked using the FORTRAN libraries.

Although this example represents one of the simplest *buildfiles,* more complex *buildfiles* for more complex cases are also easy to make and understand.

The second buildfile in Figure 11 shows how to make exceptions to a general rule. Here, all the files in /mnt/me/application are compiled at -O2. For some reason file foo.f should be compiled at -O1 instead of -O2. Furthermore a simple macro is used to provide many flags for all source files.

The options used on the command line can be embedded directly in the buildfile. This simplifies the user's *build* command and records how an application was compiled. The third buildfile in Figure 11 uses these features.

The Application Compiler summarizes the compilation process in a final report written to the terminal. This report consists of two tables, both having one row for each procedure compiled. The first table shows the interprocedural optimizations performed. The second table shows the interprocedural errors diagnosed. A line at the bottom of each table shows which *build* option to use to see details about a given column of a table.

The Application Compiler user interface was patterned after already existing tools where possible, in order to make it easier to learn and use. Much of the compilation process was simplified and automated. The output was kept minimal and concise, yet details are made available when necessary.

Mid-Course Corrections

At various points in the development process, we discovered problems that forced us to reconsider aspects of the original design. These problems ranged from design problems to coding problems. Most of these problems were not clear or present until well into the development of the Application Compiler and required us to perform some mid-course corrections.

The largest mid-course correction made during the implementation of the Application Compiler was the realization that one interprocedural pass was not sufficient for effective optimization. Conceptually, only one pass is needed: first look at every routine and gather the necessary procedural data, synthesize all the data in one interprocedural pass, and finally use the synthesized data to compile each routine.

However, to obtain the desired automatic and language independent optimizations, it was necessary to split the synthesis pass into two distinct passes. The first pass would handle the name-binding mentioned above. The second pass would handle the side-effect analyses and optimizations.

The motivation for splitting the name-binding analyses into an early interprocedural pass was better procedural analysis. Having identified more precisely to what a symbol in a procedure refers, the procedure analysis step can be done much more accurately. Dereferenced pointers now have known target sets; almost all aliases in the application have been exposed; and indirect calls have been resolved.

Having performed the initial name binding interprocedural pass, a more optimistic analysis can be made of each procedure as the basis for the interprocedural algorithms. The interprocedural algorithms are significantly more accurate, and better optimizations are made.

By completing the name binding at an early stage in the interprocedural optimization process, we satisfied the goal of language independent optimization. The procedure analysis no longer needed to be sensitive to the source language and each interprocedural algorithm in the second pass operated independently of the original source language. This abstraction also enabled the compilation of a single application that was written in multiple languages.

The original means of storing the syntax tree information in the program data base consumed too much space. Although this was a probable outcome discussed in the design document, a simplistic approach was taken initially for speed and simplicity. The major problem with space in the program data base was quickly traced to the syntax trees and a compression algorithm was implemented. This algorithm was specific to the data set present in the syntax tree, and yielded an approximately 12:1 compression ratio.

During integration testing, we found that the code resolving references to library functions was taking a large part of the time used the Application Compiler. As originally coded, a simple approach was taken: every time a reference is made to a procedure that is not in the user's code, simply search for it in the libraries. This proved to be slow. Instead, one list of undefined procedures was created and resolved by a one pass search through the libraries. The execution time for the library resolution became negligible.

As mentioned earlier, cloning was originally coded as part of constant propagation. The relationship between inlining and these algorithms caused us to separate them so that inlining could occur between them.

The original design document identified the need for libraries to be "annotated". This annotation characterizes the behavior of the procedures that participate in an application but are not available for direct source code analysis. Many aspects of this task were underestimated in the original design. We underestimated the number of libraries that would need annotation to accommodate a reasonable set of user applications. Some of the libraries, like *libc,* needed to be more thoroughly annotated than originally anticipated. Additional information was needed in the annotations to accommodate newer algorithms as they were introduced into the compilation process. To simplify the original hand-generated annotations for libraries, a method of automatic annotation generation using the Application Compiler itself was later developed.

The original design document incorporated a version of pointer tracking. However, it was a very simplistic algorithm that operated solely on knowledge of addressed variables and treated all memory dereferences equally. Mid-way through the implementation of the other interprocedural algorithms, several deficiencies were noticed that were directly attributable an ineffective pointer tracking algorithm. This forced a reconsideration of the pointer tracking algorithm and a subsequent complete rewrite.

The pointer tracking algorithm was then rewritten to be more effective and thorough. Each pointer was considered independently, and a partially flow-sensitive algorithm was developed. We obtained the results necessary to support the other interprocedural algorithms. However, we discovered that on large applications, we missed our goal of reasonable compilation speed.

Large applications took days to compile. Pointer tracking was rewritten again. The new algorithm was completely flow-insensitive, yet yielded results that were acceptable in practice, and took minutes to perform.

Conclusions

When developing the Application Compiler, we found that we were able to keep our original goals and use them to guide our design decisions. We believe that we will be able to use and follow the same goals for further development of the Application Compiler.

User input was critical to building the user interface. Most of the buildfile commands, source directives, and command-line options were implemented as a direct result of user suggestions and requests.

User feedback also made a major impact on how we analyzed, handled, and reported application errors. The first release treated more than twice as many errors in the application code as fatal errors as the final product did. Our users pressed us to remove restrictions on programming practices which were not standard-conforming but that occurred frequently in real applications. They also asked for, and received, a rationale they could understand for those practices which we had to treat as fatal errors.

User reaction to the error checking facilities of the Application Compiler was an unexpected surprise. An interprocedural compiler must do extra checking that standard compilers don't. The side effect of this is that the compiler can be used simply to find bugs. In some cases, customers perceived the value of this checking to be greater than the optimizations performed!

The performance of the system has been quite acceptable. We originally promised our management that it would run no more than 10 times slower than our standard procedure compilers for the same languages. In fact, it runs about 3 times slower, and most of the additional time is spent in I/O, reading and writing the database. In practice, all of the interprocedural algorithms scale linearly.

We compiled a large computational fluid dynamics code that contained 214,500 source lines in 971 source files. There were 1,576 calls inlined, 221 procedures cloned, and 1,774 constants propagated. The following performance statistics (wall clock time) were obtained on a CONVEX C-3220 with 256Mb of memory.

Compile Time by Phase	
Compiler phase	HH:MM:SS
BUILD	00:00:01
FFRONT	00:46:50
type checking	00:00:10
call analysis	00:00:29
alias analysis	00:02:28
MEND	00:38:16
scalar analysis	00:03:55
constant propagation	01:53:14
inline analysis	00:00:06
clone analysis	00:01:18
array analysis	00:43:51
error checking	00:00:51
BEND	01:14:22
LINK	00:00:32
Total	05:26:29

The first release of the product is now in use by dozens of customer sites. The Application Compiler provides the basis for future optimizations for new supercomputer architectures being developed at CONVEX.

Acknowledgements

Randall Mercer developed the initial design for the interprocedural optimizer. The final design and implementation of the interprocedural optimizer were done by the authors, Sean Stroud, Mark Seligman, and Matthew Diaz. Ken Kennedy at Rice University pioneered this field and gave us helpful advice. Our managers at

CONVEX – Presley Smith, Frank Marshall, and Steve Wallach – believed we could make it work and gave us the time and resources to do so.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, *An overview of the PTRAN analysis system for multiprocessing,* in Proceedings of the 1987 International Conference on Supercomputing, Springer-Verlag, 1987.

[2] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon, *ParaScope: A Parallel Programming Environment,* The International Journal of Supercomputer Applications, Vol. 2, No. 4, Winter 1988.

[3] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten, *The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran,* in Languages and Compilers for Parallel Computing, pp. 423–451, MIT Press, 1990.

[4] K. Smith and W. Appelbe, *PAT – an interactive Fortran parallelizing assistant tool,* in Proceedings of 1988 International Conference on Parallel Processing, Vol. 2, pp. 58–62, 1988.

[5] K. Cooper, *Analyzing Aliases of Reference Formal Parameters,* in Proceedings of the 12th ACM Symposium on Principles of Programming Languages, pp. 281–290, 1985.

[6] J. Banning, *An efficient way to find the side effects of procedure calls and the aliases of variables,* in Proceedings of the 6th ACM Symposium on Principles of Programming Languages, pp. 724–736, 1979.

[7] J. Loeliger,R. Metzger, M. Seligman, and S. Stroud, *Interprocedural Pointer Tracking: An Empirical Study,* in Proceedings of Supercomputing '91, pp. 14–21.

[8] K. Cooper and K. Kennedy, *Interprocedural side-effect analysis in linear time,* in Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 57–66, 1988.

[9] M. Burke, *An interval–based approach to exhaustive and incremental interprocedural dataflow analysis,* in ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, pp. 341–95, July 1990.

[10] D. Callahan, K. Cooper, K. Kennedy, and L. Torczon, *Interprocedural Constant Propagation,* in Proceedings of SIGPLAN '86 Symposium on Compiler Construction, pp. 152–161, 1986.

[11] M. Wegman and F. Zadeck, *Constant Propagation with Conditional Branches,* in ACM Transactions on Programming Languages and Systems, vol. 13, no.2, pp. 181–210, April 1991.

[12] R. Metzger and S. Stroud, *Interprocedural Constant Propagation: An Empirical Study,* ACM Letters on Programming Languages and Systems vol 2., no. 1–4, Mar–Dec 1993.

[13] V. Balasundaram and K. Kennedy, *A technique for summarizing data access and its use in parallelism enhancing transformations,* in Proceedings of the ACM SIGPLAN 1989 Conference on Programming Languages Design and Implementation, pp. 41–53, 1989.

[14] D. Callahan and K. Kennedy, *Analysis of Interprocedural Side Effects in a Parallel Programming Environment,* in Journal of Parallel and Distributed Computing, Vol. 5, pp. 517–550, 1988.

[15] P. Havlak and K. Kennedy, *Interprocedural analysis of array side effects: an implementation,* in Proceedings of Supercomputing '90, pp. 952–960, 1990.

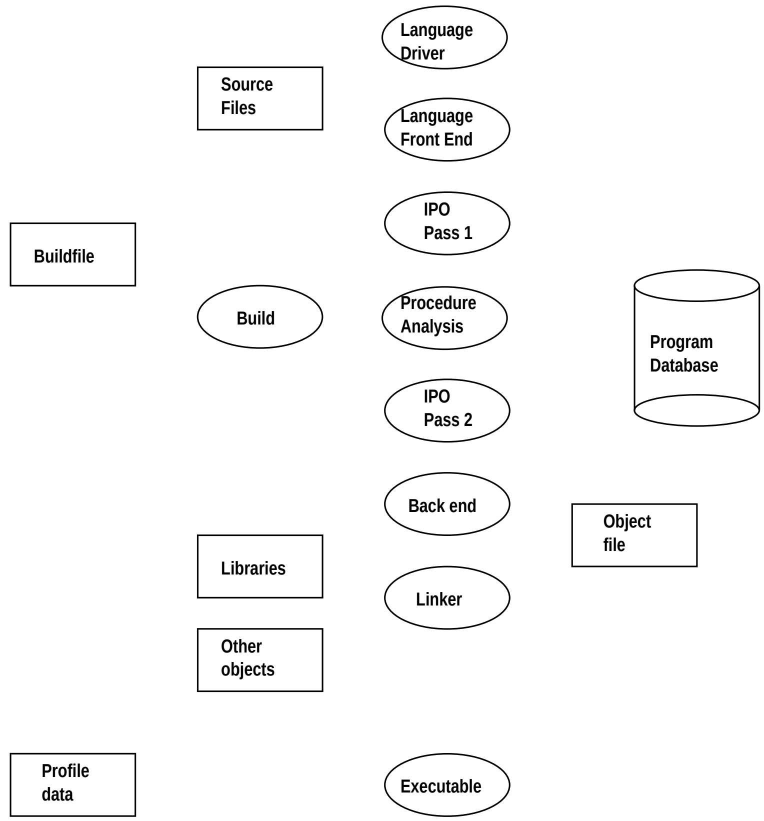

Figure 1: Compiler Architecture

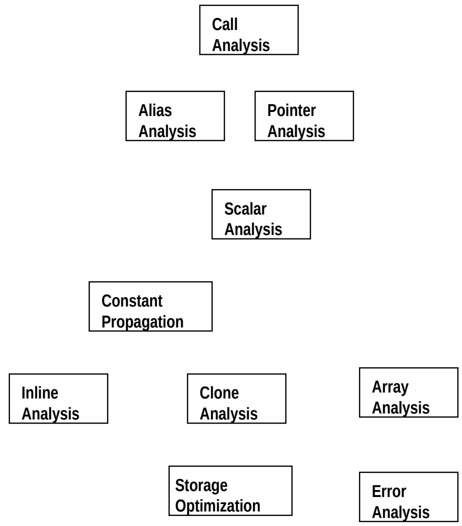

Figure 2: Partial Ordering of Algorithms

```
subroutine eval(f,x,y,n)
external f
real f,x(*),y(*)
integer i,n

do i = 1,n
  y(i) = f(x(i))
enddo
end
```

Figure 3: Call Analysis

```
program main
real x(100)

do i= 1,100
  x(i) = float(i)
enddo
call foo(x(2),x(1),99)
print *,x(1),x(99)
end

subroutine foo(x,y,n)
real x(*),y(*)
integer i,n

do i= 1,n
  x(i) = y(i) + 10
enddo
end
```

Figure 4: Alias Analysis

```
char *malloc();
double static1[200];

main()
{
    double automatic1[200];
    double *p;

    p = (double *)malloc(200*sizeof(double));
    foo(static1,automatic1,200);
}

foo(p,q,r,n)
double *p, *q, *r;
int n;
{
    int i;
    for( i=0; i<n; ++i )
*p++ = *q++ - *r++;
}
```

Figure 5: Pointer Tracking

```
program main
common a,b,c,d,e,f

read *,d,e,f
a = 5.0
call foo(d,e,f)
c  = a * 10.0
print *,c
end

subroutine foo(x,y,z)
x = y + z
end
```

Figure 6: Scalar Analysis

```
program main
parameter(n=100,m=200)
real u(500)
call sub(u,m,n)
end

subroutine sub(a,m,n)
real a(*)
integer m,n
do i = 1,n
  a(i) = a(i+m)
enddo
end
```

Figure 7: Constant Propagation

```
program main
parameter(n=100,m=200)
real u(500)
call sub(u,m,n)
end

subroutine sub(a,m,n)
real a(*)
integer m,n
do i = 1,n
  a(i) = a(i+m)
enddo
end
```

Figure 8: Constant Propagation

```
program main
real a(200)

call foo(a,200,-1)
call foo(a,200,1)
end

subroutine foo(x,n,k)
real x(n)

do i = 1,n
  x(i) = x(i+k)
enddo
end
```

Figure 9: Clone Analysis

```
program main
real a(100,100)

do i = 1,100
  call foo(a,100,i)
enddo
end

subroutine foo(x,m,i)
real x(m,m)

do j = 1,m
  x(i,j) = 0.0
enddo
end
```

Figure 10: Array Section Analysis