44

Feedback-Directed Specialization of C

by

Jeremy H. Brown

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering
and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author .... [illegible] ....
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by .... [illegible] ....
Thomas Knight, Jr.
Principal Research Scientist, MIT Artificial Intelligence Lab
Thesis Supervisor

Accepted by .... [illegible] ....
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

Feedback-Directed Specialization of C

by

Jeremy H. Brown

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Abstraction is a powerful tool for managing program complexity; well-abstracted source code is far more comprehensible, maintainable, and reusable than less abstract, specialized source code. Unfortunately, with traditional compiler technology, executables produced from hand-specialized source code generally run faster than executables produced from abstract source code; as a result, programmers frequently sacrifice abstraction's maintainability in favor of execution-speed. In this thesis I introduce a potential solution to this problem called feedback-directed specialization, a process of profile-directed specialization. Throughout its lifetime, a program is adapted to changing usage patterns through occasional recompilations. The primary contribution of this thesis is fedsocc, a strong foundation for future implementation of fully automatic feedback-directed specialization for the C programming language.

Thesis Supervisor: Thomas Knight, Jr.
Title: Principal Research Scientist, MIT Artificial Intelligence Lab

Acknowledgments

This research and the Reinventing Computing group of the MIT Artificial Intelligence Laboratory are supported in part by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations are those of the author and do not necessarily reflect the views of the National Science Foundation.

Thanks are owed to a number of people and organizations. This thesis has been developed and written under the auspices of the Reinventing Computing group at the MIT Artificial Intelligence Lab; thanks go to all the members of the RC group. Particular thanks are owed to fellow RC group members André DeHon, Ping Huang and Ian Eslick, for many discussions which generated ideas for improvements. André, the RC group's spiritual leader, deserves particular thanks for developing the notion of feedback-directed specialization in a larger framework, and thereby leading me to this particular thesis. And particular thanks also go to Tom Knight, my thesis advisor, who had the faith to give me complete freedom in choosing and developing my thesis topic.

In a less academic vein, thanks go to Michelle Goldberg for food and cookies, and for continuous moral support. My cooking group, Spork Death, has continued to feed me even though I have been failing of late to reciprocate; thanks go to all of the Spork membership for their tolerance.

For providing random distractions from classwork over the years, thanks go to the residents of my undergraduate dormitory, Senior House; Sport Death Forever! For similar reasons, thanks go also to the members of the MIT Assassins' Guild.

For helping me recover from repetitive strain injuries of the arms and wrists, thanks go to Dr. David Diamond at the MIT Medical Center, and to the occupational therapy department at Mt. Auburn Hospital. Without them, I would have been physically unable to write this thesis.

And finally, for their faith in me, and for making it possible for me to attend MIT, thanks go to my parents.

Contents

List of Figures

Chapter 1

Overview

3. ab.stract or ab.stract.er \ab-'strakt, 'ab-., in sense 3 usu 'ab-.\ vt 1: REMOVE, SEPARATE 2: to consider apart from application to a particular instance
– *Webster's Online Dictionary*

1.1 Motivation

Abstraction in programming is highly regarded. Methods of abstraction help programmers manage code complexity; abstract source code is comprehensible and maintainable, and lends itself to development and reuse. To the programmer, reusable abstract routines represent a great improvement over multiple, specialized routines, each of which must be independently written, debugged, and maintained. Unfortunately, with conventional compiler technology, abstraction comes at the price of a great deal of run-time overhead; the cumulative effect of function calls, parameter passing, and using overly general routines is substantial.

Heavily specialized source code is everything that abstract code is not: it makes minimal use of abstractions such as functions, its routines are as specific as possible, it isn't general or reusable, it's hard to comprehend, and it's harder to maintain. But with conventional compiler technology, an executable produced from specialized source code is usually faster than one produced from abstract, general, reusable, maintainable source code.

The result is that programmers are often forced to choose between the maintainability of abstraction and the runtime performance of specialization. Unfortunately, this often means that the source-code of very large, complex, long-running applications gets specialized into near-incoherency.

One means of resolving the dichotomy between abstraction and specialization is to automate specialization – that is, to automatically produce faster but more limited (specialized) code from slower but more general (abstract) code. Sufficiently powerful automated specialization would enable programmers to write abstract code without sacrificing speed.

It must be noted that traditional compilers already make use of simple automated specializations such as constant propagation. These commonly-used specializations are attractive because they don't generally increase program size or runtime; indeed, they very nearly guarantee improvements in code size and speed. However, the code speedups they are able to achieve are modest at best; thus, more aggressive specializations must be employed in order to recover the speed lost to abstraction.

However, more aggressive specializations (inlining, for example) frequently involve duplicating code, and therefore cannot make guarantees about improving code speed – as code-size increases, hardware caches or main memory may overflow, and thus a specialization which static compile-time analysis indicates should improve execution time may actually worsen it. Therefore, code-expanding specializations must be judiciously employed if they are to be fully effective.

1.2 Research Contribution

It is with the goal of recovering speed lost to abstraction that I present a model for compilation which I call feedback-directed specialization.

Feedback-directed specialization, or FDS, automatically employs aggressive specializations based on continual profiling and recompilation of a program throughout its lifetime. In an FDS system, compile-time, profiles of the program from previous runs is provided to the compiler. The compiler uses the profiles to control the specialization process: the program is specialized to run most rapidly under usual usage patterns. If the conditions of a program's use change, whether due to changing user behavior, new hardware, etc., recompilations will re-adapt the program to perform optimally under the changed average-case usage patterns.

Feedback-directed specialization is a specific idea arising from a number of concepts being developed by the Reinventing Computing group at the MIT AI lab. Foundational material may be found in technical reports from that group, particularly [12], [8], [2], [10], [3], and [11].

My contribution in this thesis is twofold: first, I have defined a complete model for feedback-directed specialization; and second, I have implemented a foundation for developing fully automated feedback-directed specialization of C programs. (It is beyond the scope of this thesis to prove or disprove the utility of feedback-directed specialization with a fully operative implementation.)

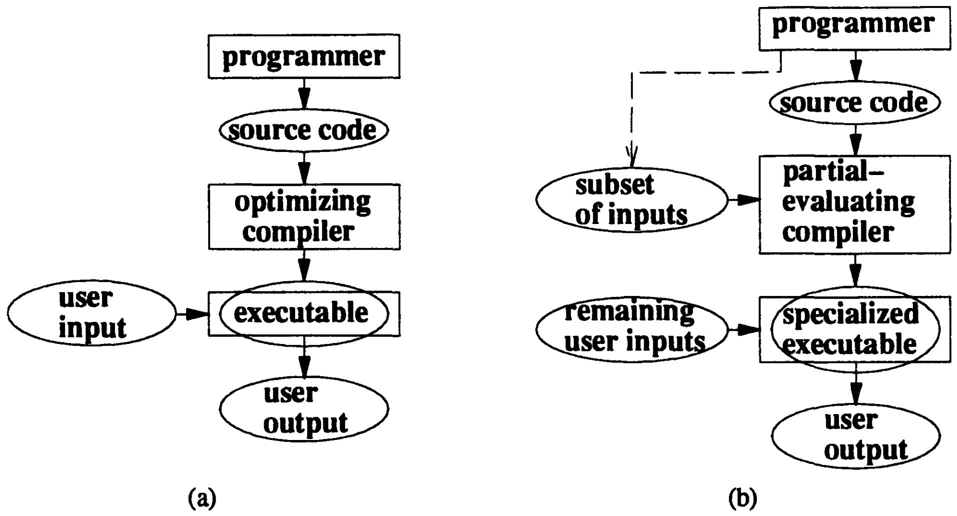
The remainder of this presentation is structured as follows:

To conclude this chapter, in Section 1.3, I will review related work on automated and profile-directed optimization techniques.

In Chapter 2, I will develop the concept of feedback-directed specialization in some detail. I will discuss the anticipated benefits and drawbacks of FDS. Finally, I will present several as yet unproven hypotheses upon which the utility of FDS depends.

In Chapter 3, I will present fedsocc, the FEedback-Directed Specializing C Compiler. Fedsocc does not, by itself, perform feedback-directed specialization; instead, it is intended to be a foundation upon which to build a fully automated FDS system for C. Also in this chapter, I will propose a staged approach to implementing an FDS system based on fedsocc; stages progress from being heavily dependent on programmer annotation of source-code to being fully automatic.

Finally, in Chapter 4, I will relate the hypotheses underlying FDS specified in Chapter 2 to the three-stage development plan proposed in Chapter 3. I will conclude by proposing several avenues for further research.



Ellipses indicate data; rectangles indicate data-consumer/producers.

Figure 1-1: Approaches to automated specialization using only immediately available data. a) standard compilation with an optimizing compiler; b) compilation with partial evaluation.

1.3 Related Work

Many approaches to alleviating the overhead of abstraction have been presented in the past. Some use statically available information to direct automated specialization, while others, more ambitious, use profiles of programs run on test data to direct the optimization process.

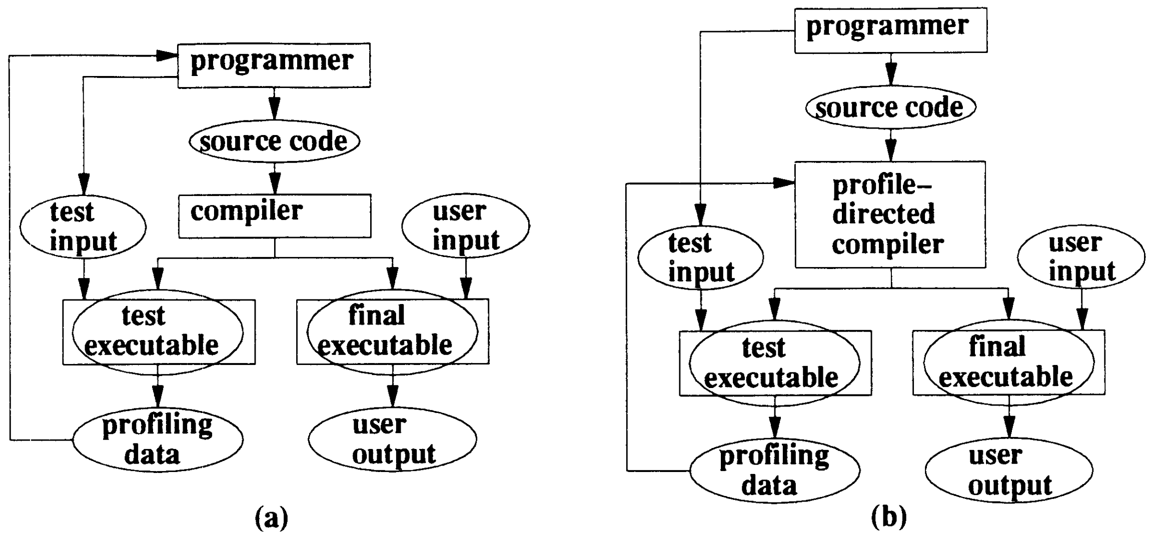
1.3.1 Automated Specialization

Most conventional compilers attempt to improve executable performance and reduce executable size by using compile-time optimizations based on static program analyses. Figure 1-1.a shows the process of compilation associated with such standard compilers. [1] presents a thorough discussion of most conventional optimizations and supportive static program analysis techniques.

Where traditional compile-time specialization makes use only of information derived directly from a program's structure, partial evaluation [16] specializes a program (or smaller functional unit) with respect to some amount of known input data. Figurel-1.b illustrates the process of compile-time partial evaluation. In this process, a subset of a program's inputs are provided statically (i.e. at compile time), and the program evaluated as much as possible with respect to those known inputs – hence the name partial evaluation. The resulting, specialized executable requires correspondingly fewer inputs than the original program. Since substantial compile-time is spent evaluating the program with respect to the fixed inputs, the executable should have relatively little work to perform when provided with values for its remaining inputs; it should therefore take relatively little time to run. Although most work on partial evaluation has focused on functional programming languages, some recent work [17] has focused on partial evaluation of imperative programming languages.

A major disadvantage of compiling with partial evaluation is that a specialized version of a program is useless for any set of inputs which includes elements differing from those against which the program was specialized. However, this problem seldom has a chance to manifest, being overshadowed by the problem that the inputs to most programs are usually completely unknown at compile-time.

These problems are avoided by forms of run-time partial evaluation known variously as deferred compilation, dynamic code generation, and runtime code generation, which delay generation of critical portions of a program until run-time. In these methods, critical pieces of code are not generated at compile time, but are instead preserved as templates; when an executable version of one of these pieces of code is required, it is generated by evaluating its template with respect to data values discovered exactly at the moment of need. The resulting executable code is precisely tailored to the operating conditions of the program. A number of examples of run-time code generation are available. [19] presents a system which completely automates deferred compilation on an imperative language. [13] presents an interface for programmers to explicitly employ dynamic code generation in their programs. The SELF compiler ([4]) employs run-time compilation to double the performance of programs written in SELF, a dynamically typed, object-oriented programming language. The Synthesis operating system kernel ([20]) employs dynamic code generation to provide an extremely fast, high-level operating system interface.



Ellipses indicate data; rectangles indicate data-consumer/producers.

Figure 1-2: Approaches to specialization using profiling data from test inputs a) hand-specialization; b) automated specialization.

1.3.2 Profile-Directed Specialization

In a 1971 paper [18], Donald E. Knuth asserts that "[t]he 'ideal system of the future' will keep profiles associated with source programs, using the frequency counts in virtually all phases of a program's life."

In a general sense, profiling a program consists of monitoring its behavior as it is executed, and producing a description, or profile, of that behavior. Profiling has evolved since Knuth's 1971 statement; modern profiling tools commonly provide function execution times, function or basic-block execution counts, or even call-graph arc-counts. One very popular profiling tool, gprof [15], provides function execution times and counts, as well as call-graph arc-counts, for C programs.

Unfortunately, profiling a program tends to substantially slow its execution due to the overhead of conventional profiling mechanisms. Progress has been made toward reducing this overhead, however. [21] presents an efficient algorithm for instrumenting code to collect execution and call-graph information; the algorithm utilizes any previous profiling information to further optimize the placement of profiling instrumentation. On a different tack, [7] presents a method for using hardware to generate profiling information at relatively low overhead.

Traditionally, profiling tools are used to direct hand-optimization of programs under development; this process is illustrated in Figure1-2.a. In this traditional model, a programmer compiles a program, then runs it on test data to produce a profile; the profile indicates "hotspots", that is, portions of code that are executed with great frequently or are otherwise responsible for a significant fraction of the program's running-time. Guided by the profile, the programmer can focus hand-optimization efforts on hotspots, thus altering a minimum of code for a maximum of effect. The process may be iterated several times, but eventually the program source is declared "frozen", and the corresponding, end-user executable permanently installed.

The great advantage of profile-directed hand-optimization is that a human programmer can change a program in arbitrarily complex and intelligent ways; optimizations may be as severe as a complete change of algorithm. The disadvantage, of course, is that overly optimized source code often becomes so messy that it becomes very hard to comprehend, maintain, or reuse. The more hand-optimized the code, the more likely it is to be unmaintainably complex. Once again, automated specialization appears highly desirable.

It has already been mentioned that many potentially beneficial specializations increase code-size, and must therefore be judiciously employed. A number of approaches have chosen to use profiling information to direct the judicious application of specializations toward program hotspots. The common outline of these approaches is shown in Figure 1-2.b. Trace scheduling [14] uses execution frequencies to sequentially arrange the blocks of assembly code that are most likely to be executed. Superblocking [5] uses execution frequencies to arrange blocks of C code much as trace scheduling arranges assembly code blocks. Superblocking also arranges code such that a conglomerate block is only entered at a single point; this arrangement frequently requires code duplication. [6] provides an interesting twist by demonstrating that code-expanding optimizations employed on the basis of basic-block profiling sometimes actually improve cache performance by improving code locality and/or sequentiality.

In contrast to this array of profile-directed specializations, Typesetter, described in [21], is a system which selects implementations for abstract data types on the basis of execution counts and call-graph arc-counts. However, since Typesetter requires a programmer to provide the multiple data type implementations, it is not as attractive as potentially fully automatic techniques involving no overhead for the programmer.

Chapter 2

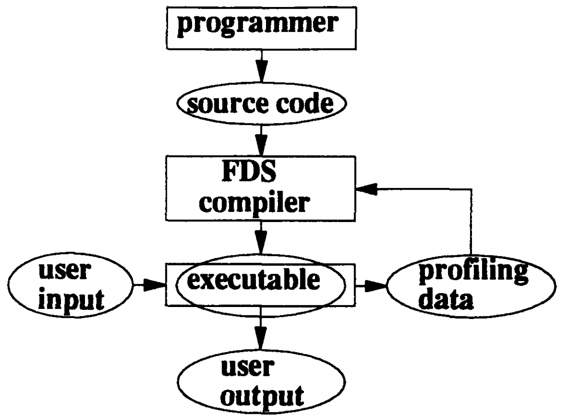
The Feedback-Directed Specialization Model

In this chapter, I will explore the concept of feedback-directed specialization (FDS). In Section 2.1, I will describe the FDS model of program development, and compare it against other methods of automated and profile-directed specialization. In Section 2.2, I will present several hypotheses, as yet unproven, which must be true if the FDS model is to prove advantageous.

2.1 Description

The goal of feedback-directed specialization is to eliminate abstraction-barrier overhead. The tools FDS employs are speculative specializations – specializations which might speed up a piece of code at the cost of increasing its code size, or might speed up some portions of a piece of code at the cost of slowing down others. To determine when such speculation is profitable, FDS uses profiling information resulting from end-user program usage. The feedback-directed specialization process is illustrated in Figure 2-1.

In the FDS process, every program is repeatedly recompiled over its lifetime. When a program is recompiled, the FDS compiler receives feedback consisting of profiling data from all the program's runs since it was last recompiled. The compiler uses the feedback to direct the use of speculative specializations on the program, and to direct the placement of profiling instrumentation in the resulting executable.



Ellipses indicate data; rectangles indicate data-consumer/producers.

Figure 2-1: The feedback-directed specialization process.

Over several recompilations, the FDS compiler is able to experimentally optimize the program. In a single recompilation, the compiler uses its feedback as the basis for theories as to which areas of the program might benefit from various speculative specializations. Where possible, the resulting executable is specialized according to these theories; the executable includes custom profiling instrumentation to determine the success of each speculative specialization. In cases where profiling feedback from previous runs provides inadequate information for theorizing, the compiler instruments the new executable for more thorough profiling.

2.1.1 Comparisons and Further Details

The feedback-directed specialization process offers several advantages over other approaches to automated and profile-directed specialization.

All profile-directed optimization techniques are aimed at improving usual-case program performance. Where previous profile-directed techniques have used profiles from test data to direct program optimization, FDS uses profiles from end user program runs. Also, where previous techniques eventually produce a final, static executable, in an FDS process programs are repeatedly recompiled in order to respond dynamically to changing usage patterns; thus, programs maintained in an FDS process adapt to real end usage conditions, rather than being statically optimized to conditions imposed by possibly unrepresentative test data.

In contrast to the restricted program resulting from compile-time partial evaluation, programs specialized with FDS retain their generality; although optimized to empirically discovered usual-case program behavior and data values, they will still run (albeit possibly slower than an unspecialized program) in atypical cases.

As in previous methods of profile-directed compilation, with FDS, a program's structure is specialized in response to control-flow profiling; thus, an FDS compiler is able to employ a variety of control-flow-optimizing specializations. However, while previous methods of profile-directed compilation have performed only control-flow based optimizations, in feedback-directed specialization, optimization is also performed according to program data behavior. In particular, an FDS compiler is able to duplicate critical branches of code, then partial evaluate each duplicate with respect to a frequently discovered data pattern; the compiler maintains general version of the code being maintained to deal with unusual cases. The program produced by an FDS compiler is therefore specialized both according to control flow and data behavior.

Traditional profiling models control-flow with execution counters and call-graph arc-counting. In order to support data-based specialization, FDS profiling must gather not only control-flow information such as execution frequencies and times, but also information about the values taken on by program variables.

Traditional profiling methods are either on or off, covering an entire program when on. An FDS compiler has direct control over profiling instrumentation so that it can perform aggressive profiling of items of interest (i.e. program hotspots, critical variables), and ignore items of little interest in order to keep profiling overhead to a minimum.

Profiling methods available to an FDS compiler should be as varied as possible, with particular focus on being able to profile real-time program behavior, and on being able to profile variable values. The focus on real-time is important because code-expanding optimizations employed only on the basis of program-time profiling (i.e. all profiling methods using only on execution-counters) may increase code-size so much that additional paging slows real-time execution.

FDS has two advantages and one disadvantage compared to run-time code generation. In run-time code generation, particular portions of code may be generated with great frequency because of characteristic user behavior; exacerbating this inefficiency, aggressive optimization takes too much time to perform at runtime on dynamically generated code.

The result is that critical code may be generated in nearly every run of a program, but will never be fully optimized. By contrast, FDS specializes at compile-time on frequently-occuring conditions; it therefore does not repeat effort on frequently occuring conditions, and because it operates at compile-time it can spend time optimizing aggressively. The relative disadvantage of FDS is that an FDS-compiled program must fall back on general routines when unusual runtime conditions occur, where a program using run-time code generation could produce a routine specialized to the unusual event.

2.2 Foundational Hypotheses

The utility of the feedback-directed specialization model depends upon several hypotheses. FDS's use of profiling information collected from end-user program runs to guide optimizations leads to the first two hypotheses:

Hypothesis 1 *Programs frequently have characteristic behaviors which can be exploited at compile-time to improve expected running time.*

Hypothesis 2 *Characteristic program behaviors are frequently attributable to typical user inputs (and other environmental factors), rather than to intrinsic program characteristics.*

Hypothesis 1 must prove true if FDS's use of profiling information to direct specialization is to be meaningful; in particular, it must be true in regard to program data behavior as well as control-flow behavior if data-based specializations are to prove advantageous.

Hypothesis 1 is partially verified by [14], [5], and [6], all of which describe success in using block and function execution counting to guide code-reorganizing optimizations (trace-scheduling, superblocking, inlining, etc.) Similarly, [21] describes success using basic-block execution counting to select alternative implementations of abstract types. However, the scope of these works is limited by their reliance solely on counter-based (i.e. control flow based) profiling; none consider any data behavior based optimizations.

On the other hand, [19], [13], [4], and [20], all indicate that data behavior can successfully be exploited to improve program performance at runtime, but none indicate whether or not program data tends to have characterizable behavior.

If Hypothesis 2 is true, programs optimized against test input by non-FDS techniques may run relatively poorly on user input that differs substantially from the test data. On the other hand, under FDS, programs are profiled while running actual user input, rather than on programmer-provided test input; also, programs are recompiled throughout their lifetimes, rather than being "orphaned" from the profiling/optimization loop. If Hypothesis 2 is true, then, programs maintained under the FDS model will adapt to changing user needs, changes in hardware, and other environmental changes. If it is not true, however, the overhead of constantly profiling end-user executables will not pay for itself since no information will be gained that could not have been gained with test input, and no additional optimization gained from the continuing feedback loop.

The duties of the FDS compiler are to instrument a program for lightweight profiling, and to specialize it in response to profiling feedback from previous runs. The first duty leads to the following hypothesis:

Hypothesis 3 *Runtime profiling instrumentation as tailored by a compiler can be very low overhead.*

Hypothesis 4 *Automated compile-time specialization based on profiling from end-user runs is sufficiently profitable to overcome profiling overhead and still be competitive with runtime code generation, hand specialization, etc.*

Hypothesis 3 is necessary if FDS is to be usable; since end-user executables will always be instrumented for profiling, the impact on program performance must be small – even if FDS is quickly able to improve program performance on the basis of the gathered profiling, initial program runs will move intolerably slowly if instrumentation overhead is too great.

Some hint as to the veracity of this hypothesis may be derived from two datapoints. First, informal experiments performed by members of the Reinventing Computing group indicate that program-counter sampling, a method used by the profiling tool gprof for determining where program-time is spent, incurs an overhead of less than three percent of a program's running time. However, [15] states that the runtime overhead of profiling instrumentation for gprof is anywhere from five to thirty percent. The logical conclusion is that nearly all of gprof's overhead is due to the runtime value sampling used to determine the calling-arcs on a program's dynamic call-graph. Inspection of the code used to perform this sampling under the SunOS 4.1.3 operating system has convinced me that it is not terribly efficient. Thus, PC sampling is definitively cheap, and the cost of rather indiscriminate value profiling (i.e. profiling every function) is at the very least significantly less than a factor of two.

More support for Hypothesis 3 is given by [21], which describes a method for lightweight call-graph arc-counting called greedy-sewing. Greedy-sewing involves instrumentation of a subset of the arcs in the call-graph. Which arcs are instrumented is flexible; thus, the runtime overhead of the profiling can be minimized by placing instrumentation on arcs that profiling information from previous runs have shown to be least-frequently traversed.

Hypothesis 4 is the final requirement for FDS to be more than an academic curiousity. It specifies that specialization, in particular, when driven by feedback is profitable.

Chapter 3

Fedsocc: a Basis for Feedback Directed Specialization of C

3.1 Overview

In this chapter I describe fedsocc, a system I have implemented to serve as a basis for gradual implementation of feedback-directed specialization of C programs. Fedsocc stands for the FEedback-Directed Specialization Of C Compiler.

Fedsocc provides three elements required to implement FDS:

1. fedsocc provides the feedback-loop that is the heart of FDS; a drop-in FDS transform engine performs the actual specialization and profiling. Details of the feedback loop are presented in Section 3.2.

2. Fedsocc provides a variety of profiling primitives. Primary primitives include PC sampling, execution counters, and variable sampling; derived primitives include block and function execution counting, and dynamic call-graph determination and arc-counting. These primitives are described in are described in Section 3.3.

3. Fedsocc also provides a range of semantics-preserving specializations: inlining functions; producing specialized versions of a function based on calling-site parameters determined to be constant at compile-time; and producing multiple versions of critical code-blocks, each specialized for different constraint. Section 3.4 describes these primitives.

In addition to these required elements, fedsocc allows programmers to annotate source code with suggestions for FDS transform engines that are not fully automatic. In Section 3.5, I present these annotations in the context of a proposed approach to implementing FDS in stages, with each stage making progressively less use of annotations than its predecessor.

Finally, in Section 3.6, I discuss the current state of implementation of fedsocc's features.

3.1.1 The Choice of C

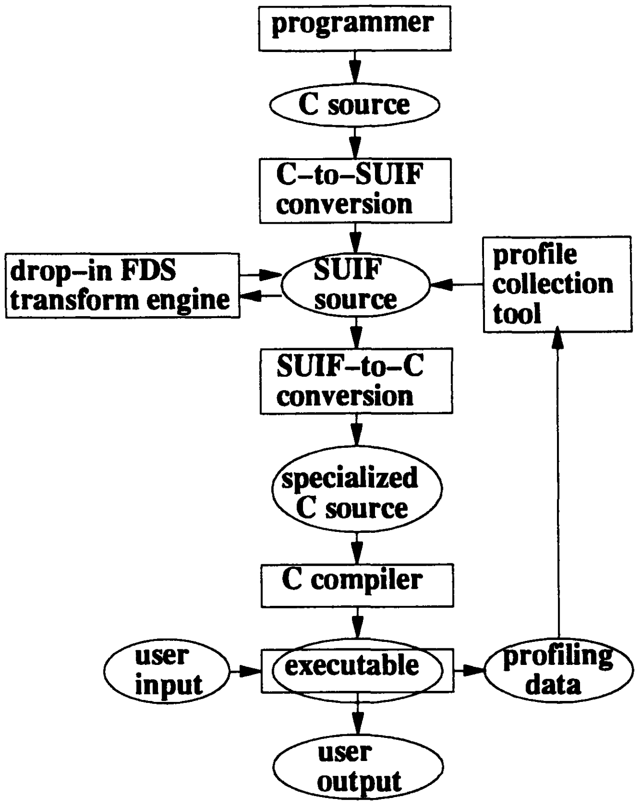
I chose to target an implementation of FDS at the C programming language for several reasons. Primary among them is that C has a large user base; thus, improved optimization of C programs is extremely useful. Also, C is a very low-overhead programming language; as a result, FDS can focus on recovering speed lost to basic abstractions and generalities, rather than first having to compensate for the overhead associated with higher-level languages. Finally, a variety of powerful, freely available tools for C were available to provide a foundation for my development efforts.

Unfortunately, C turns out to have a problem which, in hindsight, could probably have been predicted. Since C is a language of choice for programmers who want fast programs, the source code for many programs with long running times has already been heavily hand-specialized, leaving relatively little room for automated improvement or experimentation with alternative specializations.

However, an equally hindsight-predictable benefit has also emerged: many high-level languages (scheme and CLU, for instance) can be compiled to C. The resulting C code tends to be convoluted by a combination of the abstract form of the original source code, and the additional complexities needed to represent features of the source language in C. I suspect that code of this nature provides a great deal of room for automated specialization.

3.1.2 Software Dependencies

Fedsocc is built primarily on top of the SUIF Compiler System[24], a freely available research compiler suite created by the Stanford Compiler Group. The SUIF Compiler System operates as a series of passes, each of which is an individual program which reads from and writes to SUIF (Stanford University Intermediate Format) files. The SUIF distribution includes a library of routines for reading and writing SUIF files, and for manipulating SUIF data structures while they are in memory; this library greatly eases the process of develop-



Ellipses represent data (files);
rectangles represent consumers/producers of data (readers/writers of files).

Figure 3-1: Outline of the fedsocc system.

ing additional compilation passes. Also, SUIF comes with a linker which supports multi-file optimization, making interprocedural analysis and optimization much more tractable.

Fedsocc also uses gcc[22], the GNU C Compiler; the GNU version of gprof[15], a PC-sampling-based profiling tool; nm, a standard UNIX tool for examining program symbol tables; and the Perl [23] scripting language. For details of how these tools are employed, see AppendixA.

3.2 The Feedback Loop

An outline of the fedsocc system is shown in Figure 3-1. The form, of course, is similar to the generic model of feedback-directed specialization shown in Figure 2-1, in that a feedback loop insures that any program compiled with fedsocc is continually recompiled to take into account empirical measurements of program behavior. However, Figure 3-1 is considerably more detail-oriented than Figure 2-1; in this section, I will discuss those details at some length.

Initial Compilation

In initially compiling a program, fedsocc uses the SUIF-supplied front-end to convert the source from C to SUIF representation. The SUIF resulting SUIF files are maintained for the life of the program; they serves as a repository both for the program structure itself, and for the profiling feedback that is gathered during program runs.

I should perhaps mention here that I have modified the SUIF front-end to understand the annotations described in Section 3.5, and represent them in the resulting SUIF files.

In the Feedback Loop

Once a program enters the feedback loop, it goes through the following cycle:

1. First, an FDS transform engine operates on the SUIF version of the code, using profiling feedback to direct use of specializations and instrumentation of the program for future profiling. fedsocc is designed to support drop-in FDS transform engines; since the bulk of the work of implementing fully automatic FDS lies in developing heuristics to drive an FDS transform engine, it is advantageous to be able to easily select between transform engines using differing heuristics.

2. Next, fedsocc uses a SUIF-supplied back-end to convert the SUIF program representation back into C.

   One disadvantage to this approach is that C is not capable of expressing some information which might be useful at compile time. For instance, it is impossible in C to represent the fact that two variables are guaranteed not to be aliased.

   On the other hand, these source-to-source (C-to-C) transforms have the distinct advantage that benchmarking is very simple; an original C program and a transformed C program can both be compiled to executable form by the same compiler, making it easy to measure differences in each program's performance. Additionally, by using a conventional C compiler, fedsocc gains the benefit of that compiler's conventional optimizations being applied to the transformed C output.

3. gcc compiles the specialized C to executable form.

4. Each time the executable runs, it produces profiling information; the profiling is collected in the SUIF program representation for use as feedback by the transform engine in future passes through Step 1.

5. At the moment, a fedsocc user must explicitly initiate recompilation (i.e. a return to step 1). See the discussion of possible alternatives in Chapter 4.

Feedback Type	Code-size increase	Run-time overhead
Function-size calculation	none	none
PC Sampling	small constant	$< 3\%$
Execution Counter	4 machine-instructions	1 memory read/write
Variable Sampling	33 machine-instructions	for N counters: K memory reads, $4 \leq K \leq 3N + 1$ and 1 or 2 memory writes

Figure 3-2: Approximate costs of primary profiling mechanisms on a Sun SPARC

3.3 Feedback Mechanisms

fedsocc makes a number of different profiling mechanisms available to FDS transform engines. When run, a fedsocc-compiled program produces two files of profiling data. The usual model for running such a program is to call it via a wrapper which, upon termination of the program, automatically processes the profiling data files and insert the results into the program's source SUIF file; the wrapper can be omitted and the data-processing invoked by hand if desired. See Appendix A for more details on the feedback process.

In choosing what sorts of feedback to implement, I was motivated first by the desire to at least match, if not surpass, the control-flow analyses provided by the popular profiling tool gprof [15], and second, by the desire to be able to sample data values.

3.3.1 Primary Methods

Primary feedback mechanisms are very low-level mechanisms which can either be used in their own right, or used indirectly through derived feedback mechanisms3.3.2. There are four primary feedback mechanisms in fedsocc's profiling repertoire: function-size calculation, PC sampling, counters, and variable value sampling. A fifth, stopwatching, has not been added due to an inability to implement it both economically and portably. The code-size and run-time costs of each are listed in Table 3-2.

Function-Size Calculation. Function-size calculation is performed once per compilation, and involves no run-time overhead. After final compilation with gcc, the size of each function is determined from the executable's symbol table. Although the information gathered does not reflect actual run-time program behavior, it is still information which is not

```
void hello()
{
 printf("hello %d\n");
}
```

Figure 3-3: A simple function, `hello`

```
extern void hello()
 {

   {
    unsigned long long *suif_tmp0;

    suif_tmp0 = &_fds_counters[2u];
    *suif_tmp0 = *suif_tmp0 + (unsigned long long)1;
   }

  printf("hello\n")
 }
```

Figure 3-4: `hello`, with an execution-counter

traditionally available to a compiler. On its own, function-size calculation is useful in estimating the code-size increase due to certain specializations such as inlining. In conjunction with other forms of profiling, function-size calculation can help determine when a piece of code becomes too large for the hardware's instruction cache (i.e. when theoretically beneficial but code-size-increasing specializations bloat the code too much.)

Program Counter (PC) Sampling. PC sampling is always performed on fedsocc-compiled programs. When a program is run, an interrupt occurs at operating-system-specified intervals (10 microseconds under most UNIX variants); with PC sampling enabled, the program counter is sampled at each occurrence of the interrupt. The result is a statistical profile indicating how much program time is spent in each function of the program. Some informal experiments (previously mentioned in Chapter 1) by Reinventing Computing group members indicate that the run-time cost of PC sampling is generally less than three percent – the cost of the sampling process is presumably dwarfed by the cost of the interrupts, which would happen in any event. Program counter sampling returns only program-time – its numbers reflect do not reflect paging, swapping, I/O, or other elements of machine load, although they do reflect cache performance.

```
sethi %hi(__fds_counters+16),%o2
ldd [%o2+%lo(__fds_counters+16)],%o0
addcc %o1,1,%o1
addx %o0,0,%o0
std %o0,[%o2+%lo(__fds_counters+16)]
```

SPARC assembly for the execution-counter in **hello**, produced with gcc -O2.

Figure 3-5: SPARC assembly for an execution-counter

Execution Counters. An execution counter does exactly what one might expect – counts the number of times control-flow passes through it in a given program run. Execution counters may be placed anywhere in a program. Execution counters are implemented as 64-bit unsigned integers; since a 64-bit counter incrementing at 1 gigaHertz[1] would take well over five hundred years to overflow, 64 bits is deemed quite sufficient for counting purposes.

Figures 3-3 and 3-4 show a simple function unaltered and instrumented with an execution-counter, respectively. SPARC assembly for the counter is shown in Figure 3-5.

Variable Sampling. A variable sampling directive specifies a variable to sample, and a number, N, of unique counters.

The variable specified may be any form of integer, floating-point, or pointer variable. Variable-sampling directives may be placed anywhere in a program; the chosen variable is sampled each time control-flow passes through a sampling-site.

When the program runs, the first N unique values taken on by variable at the sampling point each are assigned one of the counters; subsequent values that do not fall into that set are counted collectively on a miss-counter. In general, if it is determined that the number of misses at a site is too high, the number of counters should be increased in a subsequent recompilation.

The counters associated with a variable-sampling site may be dedicated to specific values of the target variable (i.e. they may be seeded) at compile-time. Seeding guarantees that values of interest get counted individually, even if they do not appear in the first N samples.

It should be noted that a variable-sampling site also serves as an execution counter; the sum of all the counters, including the miss-counter, is the number of times execution has

[1]1 GHz is four or five times faster than the most aggressive microprocessor's clock speed at the time of this writing.

```
extern int dump(char b)
 {

   {
    struct _BLDR_struct_001 *fds_tmp0;
    int *tmpcntptr1;
    int tmpcnt2;
    unsigned long long *tdllvptr3;

    fds_tmp0 = _fds_samples;
   lup11:
    tmpcntptr1 = &fds_tmp0->cnt;
    tmpcnt2 = *tmpcntptr1;
    if (!tmpcnt2)
      goto postlup22;
    if (!((unsigned long long)(unsigned char)b == fds_tmp0->data.ll))
      goto tmpplusplus33;
    *tmpcntptr1 = tmpcnt2 + 1u;
    goto done44;
   tmpplusplus33:
    fds_tmp0 = fds_tmp0 + 1;
    goto lup11;
   postlup22:
    if (fds_tmp0 < &_fds_samples[11u])
      {
       fds_tmp0->data.ll = b;
       *tmpcntptr1 = tmpcnt2 + 1u;
      }
    else
      {
       tdllvptr3 = &fds_tmp0->data.ll;
       *tdllvptr3 = 1 + *tdllvptr3;
      }
   done44:;
   }

   [...]

 }
```

Figure 3-6: Variable profiling instrumentation.
The argument b to function dump is sampled; 10 unique counters are assigned. No seeding is performed. The body of dump is omitted in an attempt at clarity; however, the sampling code is automatically generated and remains very hard to read.

```
        sethi %hi(__fds_samples),%o0
        or %o0,%lo(__fds_samples),%o3
L3:
        ld [%o3],%o1
        cmp %o1,0
        be L5
        and %i1,0xff,%o5
        ld [%o3+8],%o0
        mov 0,%o4
        cmp %o4,%o0
        bne,a L3
        add %o3,16,%o3
        ld [%o3+12],%o0
        cmp %o5,%o0
        bne,a L3
        add %o3,16,%o3
        add %o1,1,%o0
        b L9
        st %o0,[%o3]
L5:
        sethi %hi(__fds_samples+176),%o0
        or %o0,%lo(__fds_samples+176),%o0
        cmp %o3,%o0
        bgeu L10
        sll %i1,24,%o2
        sra %o2,24,%o1
        sra %o2,31,%o0
        std %o0,[%o3+8]
        mov 1,%o0
        b L9
        st %o0,[%o3]
L10:
        ldd [%o3+8],%o0
        addcc %o1,1,%o1
        addx %o0,0,%o0
        std %o0,[%o3+8]
L9:
```

Figure 3-7: SPARC assembly for a variable sampling site.

passed through that point. However, variable sampling is considerably more expensive than using an execution counter, and thus should never be used when an execution counter will do.

Figure 3-6 shows a function instrumented for variable-sampling of its only argument. SPARC assembly code for the variable sampling is shown in Figure 3-7.

Stopwatching is a method for measuring elapsed real-time between two points of execution. Although stopwatching is attractive, fedsocc does not provide a stopwatching mechanism; I was unable to find a portable implementation which did not have unacceptably high runtime overhead costs.

3.3.2 Derived Methods

Derived methods of feedback are compositions of the primary methods aimed at performing more complex profiling tasks; while each FDS transform engine could derive its own complex profiling mechanisms, the ones here are sufficiently useful to merit implementation in the fedsocc library of feedback methods. Block and function execution-counts derive from counters; dynamic call-graph arc-counting derives from counters and variable sampling.

Block Execution-Counting. Block execution-counting places an execution counter in each block of code. For this purpose, blocks are not the same as basic blocks; instead, they contain the same bodies of code as would normally be delineated by {} pairs in C. In addition, the then/else clauses of if-statements, and the bodies of while or do-while loops, are considered blocks even if they are single statements.

Function Execution-Counting. Function execution-counting places an execution counter at the beginning of every function in a program. Function execution-counting, combined with the omnipresent PC sampling, provides information identical to that provided by standard prof-style profiling, at what I believe should be a somewhat lower runtime overhead. Unfortunately, I have not yet been able to perform comparison tests to verify this belief.

Dynamic Call-Graph Arc-Counting. Dynamic call-graph arc-counting operates on a per-function basis. For each target function, a counter is placed immediately preceding any call directly to that function. These counters serve to determine how frequently arcs on the static call-graph are taken. For each anonymous function call in the program, the variable pointing to the called function is sampled immediately before the call; a temporary variable is created if needed. Each sampling site has a counter seeded for each function whose call-graph is being observed.

This method provides gprof-style functionality, available on a per-function, rather than per-program, basis. While I believe that the overhead of this method should be substantially less than that incurred by gprof, I have not yet been able to test this belief.

```
int absqr(int y) {
  if (y > 0) return (sqrt(y));
  else return (sqrt(-y));
}

main ()
{
  int x;
  int z;

  [...]

  z = absqr(x);

  [...]
}
```

Figure 3-8: main calls absqr

3.4 Specialization Mechanisms

The specializations fedsocc provides fall into two categories: specializations made practical by knowledge of program control-flow, and specializations made practical by knowledge of data behavior.

All of the specialization mechanisms provided can be undone so that FDS transform engines may back out of unprofitable experimental specializations.

Control-Flow-Behavior Enabled Specializations

Knowledge of how a program's control-flow actually behaves enables judicious use of specializations which are static but expensive. By static, I mean that the mechanical specialization can be performed with information present statically in the program itself; by expensive, I mean that the specialization increases code-size, possibly substantially. The two expensive, static specializations available to FDS transform engines are inline function expansion (inlining), and generating specialized versions of general functions.

```
int absqr(int y) {
  if (y > 0) return (sqrt(y));
  else return (sqrt(-y));
}

main ()
{
  int x;
  int z;

  [...]

      {
        int y;
        int suif_tmp0;
        int suif_tmp1;

        y = x;
        if (0 < y)
          {
            suif_tmp0 = sqrt(y);
            z = suif_tmp0;
            goto retpoint12;
          }
        else
          {
            suif_tmp1 = sqrt(-y);
            z = suif_tmp1;
            goto retpoint12;
          }
      retpoint12:;
      }

  [...]

}
```

Figure 3-9: absqr inlined into main

Inlining. Inlining, or inline function expansion, is the process of replacing a call to a function with the body of the function being called. The immediate benefit of inlining is the elimination of function-call overhead – i.e. parameter-passing and non-linear control-flow branching to and from the called function. The followup benefit is that conventional local optimizations are able to operate on the straight-line code created by inlining the call; for example, constant propagation may push constant parameters through the inlined function-body, replacing variable references; dead-code elimination may be able to remove branches within the inlined function-body that are never taken; and register allocation can take the inlined function-body into account. If every possible call to a function is inlined, the "source" function is no longer necessary and can be discarded.[2] Figures 3-8 and 3-9 demonstrate a simple inlining.

Many traditional compilers allow for limited forms of inlining. gcc, for instance, allows the programmer to mark functions with an "inline" directive; such a marked function be inlined everywhere it is called. Most traditional compilers follow a similar model if they permit inlining at all.

In contrast, fedsocc provides inlining by call site. A transform engine can choose whether or not to inline a function of arbitrary size at any given call-site; thus, very large functions can be inlined at one or two points within the program if the guiding heuristic determines that a substantial speedup is likely as a result.

The program-size-increase due to an inline function expansion is always at least slightly less than the size of the function being inlined, since the calling-instructions, if nothing else, may be omitted. If enough conventional optimization is enabled by the inlining, the program-size-increase due to the operation may be much less than the original size of the function.

Generating Specialized Versions Of General Functions. Generating a specialized version of a general functions can be performed when constraints on the function's calling-parameters are known. The function is copied, and the copy is specialized with respect to the constrained parameters. The program-size-increase due to generating a specialized version of a function is equal to or less than the size of the original function.

[2]This does not currently happen automatically.

Although in concept, parameter-constraints could include ranges or sets of values, in fedsocc only constants can be used as constraints. Usually, this sort of specialization happens on the basis of constant arguments at a call-site; the constants replace the parameters in a copy of the function, then conventional optimizations (constant propagation, constant folding, dead code elimination) operate to actually derive benefits from the presence of the constants.

The disadvantage of generating a specialized version of a function instead of inlining it is, of course, that the function-call overhead is not eliminated – it must still be called. The advantage, however, is that the specialized version can be called in place of the general function from every call-site in the program that shares the constraints under which it was specialized; thus, the cost in terms of program-size due to the new version is amortized across all the sites that end up sharing that version.

Data-Behavior Enabled Specializations

Fedsocc provides only one form of data-based specialization, **code-block duplication and specialization.** This form of specialization operates as follows:

1. A block of code is selected for specialization.
2. Sets of constraints on variables used in the block are generated. In fedsocc, each constraint assigns a variable a constant value, since while more complex constraints might prove beneficial, using constant values allows fedsocc to leverage the conventional optimizations provided by using gcc as a back-end.
3. The block of code selected is duplicated once for each constraint-set.
4. Each duplicate is partial-evaluated with respect to the constant values in its associated constraint-set; the original block is left general. In the current implementation of fedsocc, this step consists only of setting the variables to their corresponding constant values at the beginning of each duplicate block; gcc's conventional optimizations (constant propagation, constant folding, dead code elimination) are assumed to take care of the partial evaluation process.
5. Finally, a decision-tree is composed to select the appropriate, specialized block at run-time based on the actual variable values.

3.5 Staged Implementation

Of all the components involved in the fedsocc feedback cycle, the FDS transform engine is the pass that does the bulk of the work. To implement full FDS, this pass must contain heuristics able to use profiling information from previous runs of a program to guide further program specialization and de-specialization, and also to guide placement of new profiling directives into the program. In particular, the FDS transform engine must be able to analyze profiling data, hypothesize as to where specialization is called for, and then experimentally verify the generated hypotheses.

With this goal in mind, in this section I will present a staged approach to implementing fully automated feedback-directed specialization in the fedsocc framework. The first two stages rely on programmer annotation of source-code; the annotations serve to assist a less-than-fully-automatic FDS transform engine in specializing and profiling the program. I have modified the SUIF system's C-to-SUIF front-end to understand the suggested keyword-syntax for each annotation, installing each suggestion on the SUIF representation of the program. It is entirely up to a given FDS transform engine whether or not to obey any or all of these directives.

Note that at the time of this writing I have not performed any of the implementation stages described below; they represent a plan for further work. Thus, the FDS transform engines referenced within are, for the moment, hypothetical.

3.5.1 Manual Hypothesizing and Experimentation

The first four annotations correspond to a first-stage implementation FDS, providing direct orders to an otherwise completely passive FDS transform engine. The first three directly invoke the three specialization mechanisms of fedsocc; the fourth invokes the variable profiling mechanism of fedsocc.

Using these annotations in conjunction with a control-flow profiling tool such as gprof, a programmer should be able to simulate arbitrary heuristics for feedback-directed specialization by hand. This should be useful in developing heuristics, but is clearly impractical for general application to large programs. Note that the model here is that the programmer is performing all of the profile-based hypothesizing and experimenting with specialization.

`specin(`*function(args)*`,` *int*`)`

The specin keyword is used as if it were a function of two arguments: the first is a function call, and the second is a non-negative integer. A programmer may "wrap" any function call in a specin; the value of evaluating a specin expression is the value of evaluating the wrapped function call.

Specin stands for "specialize-inline"; an FDS transform engine should interpret it as a directive to inline the wrapped function call. The integer is a depth count; in the event that the function is recursive, it specifies the number of times the recursion should be unrolled.

`specnoin(`*function(args)*`)`

As with specin, a programmer may wrap any function call with specnoin. Specnoin stands for "specialize-no-inline"; an FDS transform engine should interpret it as a directive to generate a specialized version of the wrapped function, optimized based on any constraints on the arguments to the call (i.e. any constant arguments). As with specin, the value of evaluating a specnoin expression is the value of evaluating the function call it contains.

`blockspec(`*varname1*`,` *constantvalue1*`,` . . . `,`*varnameN*`,` *constantvalueN*`);`

The blockspec ("block-specialize") keyword is employed as a C-style statement. Its arguments are an arbitrary number of variable/value pairs. Multiple blockspecs may appear in a block of code; the order in which they appear is important. Blocks for this purpose are not basic blocks, but program blocks as described in the section on fedsocc's data-based specialization mechanism.

A blockspec statement represents a constraint-list mapping each variable in its arguments to the immediately following constant value. Given a sequence of blockspec statements in a block, an FDS transform engine should use the fedsocc code-block duplication and specialization mechanism to generate one specialized version of the containing block for each blockspec. The lexically first blockspec is assumed to specify the most likely case, and so forth. The specialization mechanism automatically constructs a light-weight version-selection mechanism to pick the right version of the block at run-time.

`sample(`*varname*`,` *int*`);`

The sample keyword is employed as a C-style statement. It instructs an FDS transform engine to sample the named variable, using *int* unique counters.

3.5.2 Automating Experimentation

The next two annotations correspond to a second-stage implementation of FDS; they are used to suggest regions of interest to an FDS transform engine which then has the responsibility for deciding how to instrument marked region for profiling, and how to specialize the regions in response to profiling feedback.

The assumption is that now the programmer is only responsible for providing hypotheses about program behavior; the FDS transform engine has become responsible for verifying the accuracy of these hypotheses and specializing accordingly.

`specauto(`*function(args)*`)`

Specauto, for "specialize-automatically", is employed identically to specnoin. However, rather than requiring a particular specialization from the transform engine, it merely indicates that the call is of sufficient importance to be considered for optimization.

The transform engine is thus responsible for determining the tradeoffs between inlining the call, generating a specialized version of the called function, or leaving the call alone. Issues that it may want to consider include:

- How many sites is the called function called from, and how often?
- What are the common characteristics of those calling sites? (I.e. how many could share a version of the function specialized to the constraints of the call under consideration?)
- What is the contribution of this function as called from this call-site on overall program running-time? Impact of the calling function?
- How large is the function being called? How large is the function that is calling it?
- How much has the code grown due to specializations so far?

This is, of course, a var from comprehensive list.

`autoblockspec(`*varname1, constantvalue1, . . . ,varnameN, constantvalueN*`);`

Autoblockspec is employed identically to blockspec; however, the ordering of autoblockspec statements within a block does not matter – the FDS transform engine is responsible for deciding which of the suggested constraint sets are worth using to generate specialized versions of the containing block, and in what order. Strategies for making these decisions may include generating all suggested specialized blocks and profiling their performance, and/or profiling variables listed in the constraint lists to see how frequently they take on the suggested values.

3.5.3 Automating Hypothesizing

The third stage is to remove the burden of hypothesizing about program behavior from the programmer, and place it on the FDS transform engine. If this can be accomplished, the programmer will no longer need to perform any annotation of source-code; instead, the FDS transform engine will be fully responsible for completely automated feedback-directed specialization of a given program.

Taking this final step with respect to control-flow based specialization should not be terribly difficult. In the previous stage, function calls marked by specauto annotations were considered for each type of control-flow optimization available from fedsocc. In this stage, all call sites will receive the same consideration. Many call-sites should quickly be eliminated from serious consideration by virtue of occuring in relatively infrequently executed or otherwise non-time-consuming code; calls in more critical segments of code will, however, be examined in greater depth.

Eliminating human hypothesization with respect to data-value based specialization will present more of a challenge. First, blocks of interest must be identified; in general, these will be blocks which are frequently executed and/or particularly time-consuming, and which rely on variable values sufficiently that block specialization is potentially profitable.

Once these blocks are discovered, critical variables must be profiled in order to determine frequent-case values to use as constraints. Since fedsocc provides no provisions for profiling the relationships between variable values, either: one, the transform engine will have to try randomly collecting constraints together and testing the utility of the generated constraint sets; or two, the transform engine will have to specialize based on a single variable, then profile other variables within the specialized versions of the original block, thus gradually generating more complex constraint-sets. The second option appeals to my intuition, but both options merit investigation.

3.6 Fedsocc Implementation Status

Most of the basic modules composing the fedsocc system are functional. The major exception to this is the profiling-collection module, which at this point can only store PC sampling information into the SUIF representation of a program. Also, the modules must currently be invoked sequentially by hand. Eventually three scripts will automate the process: one will automate the process of generating initial SUIF files from C files; one will automate the process of running a program and then collecting the resulting profiling information into the SUIF program representation; and one will invoke program recompilation.

Of fedsocc's library of feedback mechanisms, all the primary methods (PC sampling, execution counters, and variable sampling) are implemented, although variable sampling does not yet support seeding. The derived methods, on the other hand, are not yet implemented.

All of fedsocc's library of specialization mechanisms are implemented; the mechanisms for "backing out" each of the specializations, however, are not yet implemented.

Finally, the front end has been modified to support most of the annotations for staged implementation; the keywords "sample" and "autoblockspec" are not yet recognized, however.

Chapter 4

Conclusions

4.1 Analysis

In Chapter 2, I presented the concept of feedback-directed specialization. I explained its advantages and disadvantages, and in particular, I described the four unproven hypotheses upon which its utility depends.

In Chapter 3, I described fedsocc, a platform providing a framework within which to develop fully automated FDS for C. I described a three-stage process for developing FDS-driving heuristics.

At this point I will explain how each of the four hypotheses underlying FDS is demonstrated in the proposed stages of implementation.

The first unproven hypothesis states that programs frequently have characteristic behaviors which can be exploited at compile-time to improve expected running time. Previous work illustrates that program control-flow does indeed frequently have characteristic behavior which can be exploited at compile time. However, there is no evidence as to whether or not program data tends to have exploitable characteristic behavior.

In the first of the proposed stages of implementation, programmer source-code annotations completely control profiling and specialization; these annotations include control over variable value sampling, and variable-value-based specialization. Thus, a programmer using these annotations should be able to gather evidence as to the truth of the first hypothesis: if they prove useful, then the hypothesis is almost certainly true.

The second unproven hypothesis states that program behaviors are frequently attributable to typical user inputs and other environmental factors, rather than to intrinsic program characteristics.

In the second proposed stage of implementation, programmer source code annotations mark regions of interest; the FDS transform engine is responsible for profiling these regions, and specializing them appropriately. Thus, in this stage, programs will be specialized differently for different characteristic behaviors; therefore, if changing the usage pattern of a program results in its being respecialized differently on the next compilation, the truth of the second hypothesis will be confirmed.

The third unproven hypothesis is that run-time profiling instrumentation as tailored by a compiler can be very low overhead. The truth of this hypothesis may be partly determined by the second stage of implementation, since at that stage the compiler is responsible for profiling regions of interest. Full determination of the truth of this hypothesis, however, must be delayed until the third stage of implementation, in which programmer source-code annotations are no longer used, and the compiler is responsible for profiling to discover regions of interest. Only at that point will the compiler be making full use of profiling, and thus only then can definitive judgements be made about the resulting overhead.

The fourth and final unproven hypothesis is that automated compile-time specialization based on profiling from end-user runs is sufficiently profitable to overcome profiling overhead and still be competitive with runtime code generation, hand specialization, etc. The determination of the truth of this hypothesis must clearly be delayed until the third and final stage of implementation; only when fully automated FDS has been implemented can such comparative measurements be made.

4.2 Future Work

4.2.1 Complete FDS Implementation

The concept of feedback-directed specialization has a number of very attractive features. FDS seems to address the overhead of abstraction very successfully, certainly more successfully than hand-specialization of program source code. In particular, FDS maintains a dynamic balance between abstraction and specialization by automatically re-specializing programs in response to changing usage patterns; thus, with FDS specialization is applied as needed, where needed, rather than at fixed places within a program as with hand-specialization. In fact, the flexibility available to the FDS process actually increases with the abstraction level of the code.

Given these attractions, the obvious first task is to complete full implementation of FDS to verify its utility. Fedsocc is designed to be a foundation for this task; it seems likely that the proposed, staged development will lead to a successful implementation of FDS for C.

The benefits of a successful implementation have already been stated. However, should an implementation of FDS built on the foundation of fedsocc fail to prove profitable, fedsocc should be considered the limiting factor before FDS as a concept. Indeed, even if an implementation of FDS based on fedsocc is successful, it might be further improved by the addition of new features to fedsocc.

Improvements to fedsocc could take a number of forms. New types of specializations could be developed. Better constraint models could be employed; for instance, variables could be constrained to ranges or sets, rather than just constants. New methods of variable profiling could be added; for instance, variables could be modeled as gaussian distributions. The translation of specialized code back to C could be replaced with direct compilation, enabling back-end code-ordering, variable aliasing detection, etc. Hardware assistance could be very useful in gathering profiling information at very low overhead.

Finally, if FDS proves viable, there are a number of issues that will need to be addressed before it can become generally used.

The process of recompilation will need to be automated. One approach is to simply recompile a program after a set period of time, or a set number of runs. A more sophisticated approach might allow the FDS compiler to specify conditions under which a program should be recompiled; the feedback collection mechanisms then would be responsible for detecting these conditions, and initiating recompilation. More sophisticated models can be imagined at an operating-system level.

Methods for dealing with programs used under widely varying circumstances will need to be developed; if different users employ a program very differently, or if a single user employs a program from multiple hardware platforms, profiling information may take on a schizophrenic appearance.

The mechanics of profiling must be transparent to end-users; files of information should not appear each time they run a program, nor should they have to explicitly invoke collection. In addition, if a large number of programs come to rely on FDS, the amount of profiling information collected may become unwieldy.

4.2.2 Collaboration

If FDS proves profitable in its own right, combining it with other forms of optimizations should prove additionally profitable. For instance, FDS with runtime code generation techniques seems profitable; compile-time specialized routines generated with FDS could handle frequent case program behavior with heavily optimized routines, and run-time generated routines could handle unusual case program behavior rather than requiring control-flow to fall back on some general piece of code as in standalone FDS.

Finally, other work in the Reinventing Computing group may work well with FDS. In particular, Global Cooperative Computing [9] proposes several ways of computing over networks; collecting feedback information from a single program being used at multiple sites could provide valuable insight into how it is actually being employed, guiding both automated and human optimization.

Appendix A

Fedsocc Implementation Details

fedsocc has been developed entirely on Sun SPARC workstations running SunOS 4.1.3; although it should eventually be portable, at the time of this writing it probably contains several platform-dependent artifacts.

A.1 SUIF

My original intent was to build fedsocc on top of the GNU C Compiler [22]; however, gcc's internals are hideously contorted and poorly documented, so I sought an alternative, and found it in the SUIF compiler suite [24] from the Stanford University Compiler Group.

The SUIF compiler suite has a number of advantages over other compilers, the foremost of which is that it is designed to be easily extended. The SUIF compiler operates as a sequence of individual programs which take as input files of, operate on, and write out files of, programs represented in the Stanford University Intermediate Format (SUIF); a well-documented library of C++ routines make it easy to develop additional programs, or passes, to add to the compilation sequence. The SUIF format supports annotations that let developers add nearly arbitrary data structures to SUIF structures; fedsocc state is maintained between recompilations of a program entirely within these annotations.

Another major advantage of the SUIF compiler system is that it can perform compile-time linking[1]; this makes interprocedural optimizations such as inlining much, much easier

[1]This is not entirely true. The SUIF compiler system is advertised as being able to perform compile-time linking, but at the time of this writing, the available linker is very preliminary, entirely unsupported, and so buggy as to be unusable.

```
/*  "suif_copyright.h" */

/*

    Copyright (c) 1994 Stanford University

    All rights reserved.

    Permission is given to use, copy, and modify this software for any
    non-commercial purpose as long as this copyright notice is not
    removed.  All other uses, including redistribution in whole or in
    part, are forbidden without prior written permission.

    This software is provided with absolutely no warranty and no
    support.

*/
```

Figure A-1: The SUIF copyright file.

to implement for programs with multiple source-files. And finally, all of SUIF's source-code is available, so existing passes can be used as models for writing new ones, and even as sources for needed routines.

My work has been with SUIF version 1.0. The SUIF copyright is reproduced in Figure A-1.

A.2 Fedsocc Processes

A.2.1 Initially Generating SUIF from C

Five passes are actually performed to convert a set of C files into SUIF files suitable for the fedsocc specialization and feedback routines to operate on. A simple wrapper script will eventually be written to run all the necessary passes when invoked with the names of the C source files for a program as command-line arguments; until that time, the passes must be run by hand.

The first three passes can actually be invoked in one command using scc, the default SUIF driver. scc runs the first three passes: cpp, the system C preprocessor; snoot, the

C-to-SUIF front-end from the SUIF distribution; and porky, a package of various code transforms, also from the SUIF distribution. scc is invoked as follows:

```
scc -option SNOOT -Tsparc -option PORKY_DEFAULTS -no-call-expr -.spd\
    foo1.c ... fooN.c
```

snoot translates each preprocessed .c C file to a .snt SUIF file. The -Tsparc option causes snoot to define the sizes of types according to a template for SPARC computers; I have altered the template locally to support 64-bit long long integers, which I use as counters in profiling. (The C++ compiler I am using, g++, can support integers of that length.)

Ordinarily, scc just uses porky to correct some improper SUIF forms emitted by snoot; the addition of the -no-call-expr option causes porky to separate function calls from expression trees. This separation makes inlining much easier. The -.spd option tells scc to stop after porky has generated correct SUIF files; without it, scc would go on to transform the SUIF files back to C and then invoke /bin/cc on them.

The fourth pass is invoked to link the SUIF files' global symbol tables so that they can be used jointly by future passes:

```
linksuif foo1.spd foo1.lnk ... fooN.spd fooN.lnk
```

The .spd files are the the input SUIF files; the corresponding .lnk files are the output SUIF files. Linksuif is a program from the SUIF distribution which unifies the global symbol tables stored in each SUIF file; it is a prerequisite to running passes which involve processing multiple SUIF files in the same run (i.e. anything involving interprocedural analysis or optimization.) Unfortunately, the version of linksuif I am using is an unsupported prerelease, and many legal C constructs seem to cause it to crash.

The fifth and final pass must be invoked to prepare the program for general use with fedsocc:

```
feedinit foo1.lnk foo1.glb ... fooN.lnk fooN.glb
```

feedinit is a program I wrote to prepare a program for FDS with fedsocc: first, it converts all static functions into global functions, alpha-renaming where necessary; and second, it installs some initial data structures (in the form of annotations) needed by the profiling primitives fedsocc provides. The .lnk files files are the input SUIF files; the corresponding .glb files are the output SUIF files, and are suitable for processing by fedsocc profiling and specialization primitives.

A.2.2 The Feedback Loop

The fedsocc feedback loop involves several programs, and can generally be broken into two sections: producing an executable using gathered feedback; and running the resulting executable, collecting new profiling feedback.

Producing an Executable

There are four programs involved in producing a specialized, profiling-instrumented executable. The first is the FDS transform engine, which must be provided by the programmer. The engine is responsible for performing the actual feedback-directed specialization of the code, including specializing and despecializing as well as inserting and removing profiling directives. See Chapter 3 for an overview of the primitive operations available to authors of transform engines. The engine is invoked as:

```
<engine> foo1.glb foo1.out ... fooN.glb
```

where engine is its name. The .glb files are the SUIF files representing the program; the .out files receive the specialized, profiling-instrumented output of the transform engine. The .out files should immediately be moved to replace the .glb files; they are the new representation of the program until the next compilation cycle.

I have considered adding another pass which would produce temporary SUIF files from the .glb files, eliminating any functions which are never called; such functions should not be eliminated from the .glb files, as they could be needed to back out an inlining specialization in some future compilation. For the moment, no such pass exists.

The SUIF compiler system ships with two back-ends; one produces MIPS assembly code files from SUIF files, the other produces C code files from SUIF files. Since fedsocc was developed on Sun SPARCs, the third program that needs to be invoked is s2c, the SUIF to C back-end:

```
s2c foo1.glb foo1.out.c ... fooN.glb fooN.out.c
```

In the next pass, the GNU C Compiler [22], gcc, should be used to generate object (.o) files from the C (.out.c) files:

```
gcc -c -O foo1.out.c ... fooN.out.c
```

The -c option to gcc causes it to generate .o files without trying to perform final linking of the program.

Finally, in the fifth and final pass, the .o files must be linked together:

ld /lib/gcrt0.o fds.o foo1.out.o ... fooN.out.o -lgcc -lc -lgcc execname

Execname should be the name of the program being generated. Linking against /lib/gcrt0.o provides execname with a startup routine that activate PC sampling, and a shutdown that writes the results to a file called gmon.out. fds.o is part of fedsocc, and includes a routine, automatically called at program termination, to write the results of fedsocc profiling directives to a file called fdsmon.out. -lgcc and -lc are standard libraries. A number of other standard ld options probably also need to be included in the ld invocation, but are omitted here for the sake of clarity of presentation.

Collecting Profiling

After running a program compiled with fedsocc profiling directives, the data in gmon.out and fdsmon.out need to be collected into the SUIF representation for the program. Unfortunately, the process of collecting data from fdsmon.out has not yet been implemented; collecting the data from gmon.out is invoked:

```
gprof execname gmon.out | pcback foo1.glb foo1.tmp ... fooN.glb fooN.tmp

gcc -c -O foo1.out.c ... fooN.out.c
```

Appendix B

Source Code

B.1 fedsocc library routines

B.1.1 Interface

interface.h

```
/* All operate on call-isolated expr-form suif trees */


/* Inlining ---------------------------------------------------------------*/

tree_block *startinline(in_cal *thecall);
/* Returns the block which represents the to-be-inlined piece. */

void endinline(in_cal *thecall, tree_block *theblock);
/* Finishes the inlining job, replacing thecall with theblock and doing
   relevant cleanup. */

/* the above two functions are separate in order to give the
   programmer the option of recursing on the inline-block before it's
   inlined (necessary when recursive situations are present to avoid
   exponential code growth!)*/


/* Specialization ----------------------------------------------------------*/

int fullspec(in_cal *thecall, sym_node_list *newlist);
/* Replaces thecall with a call to a fully specialized version of the function.
   Deals with unification issues: generates new spec'ed version if needed,
   uses pre-created one if possible. Int = 0 if no spec possible;
   1 if old one used; 2 if new one made */

int tryspec(in_cal *thecall);
/* Attempts to replace thecall with a call to a more specialized version --
   draws from already-created spec'ed versions, but does NOT create new ones.
   Picks strongest form -- alg needed here, probably most args. */


/* block specialization ------------------------------------------------*/

void dumbblockspec (tree_block *theblock, constraint_list *conlist);
/* replaces theblock with a suite of specialized blocks --
   order of blocks is direct from conlist */
```

```
/* a constraint is a list of conditions, and a frequency. */
/* (the frequency is ignored by dumbblockspec, and may not be needed
   at all, but I figure what the heck) */
/* a condition is a class; for purposes of this thesis, it will
  support a variable equaling a constant, but should be exensible
  later without too much pain */

#include "condition.h"
#include "constraints.h"
#include "funcspec.h"
#include "blockspec.h"
#include "myutils.h"
#include "feedback.h"
```

condition.h

```
/* What's a condition, anyhow? */

extern char *k_funcspec;

enum cond_type {CONST_VAL};

template<class T> class condition {

private:
  T parmid;
  immed constval;
  int useless;

public:
  condition(T p, immed c) {parmid = p; constval = c; useless = 0;}
  condition() {useless = 1;}
  cond_type mytype() {return CONST_VAL;}
  immed constv() {return constval;}
  T parm() {return parmid;}

  boolean operator==(condition<T> &r) {return (parmid == r.parm()) &&
                                               (constval == r.constv());}
  boolean operator!=(condition<T> &r)    { return !(*this == r); }
};
```

constraints.h

```
enum relenum {
  r_strict_subset,
  r_strict_superset,
  r_equal,
  r_disjoint};

#if 0 /* Interface -- actual code is under this */

template<class type, class list, class iter>
class constraints {
public:
  /* create */
  constraints() {conds = new list();}

  /* destroy */
  ~constraints() {delete conds;}

  /* is there a definition for a particular parameter number? */
  int isdef(type parmnm);

  /* return def. for particular parameter.*/
  condition<type> deflookup(int parmnm, int usecache = 0);

  /* determine the setwise relationship between this and that */
  relenum relationto(constraints* that);

  /* obvious */
  int subsetof(constraints *foo) {relenum r = relationto(foo);
                                  return ((r == r_strict_subset) || (r == r_equal));}
  int supersetof(constraints *foo) {relenum r = relationto(foo);
                                  return ((r == r_strict_superset) || (r == r_equal));}
  int equalto (constraints *foo) {return (relationto(foo) == r_equal);}
  int disjoint (constraints *foo) {return (relationto(foo) == r_disjoint);}
  list *condlist() {return conds;}
};
#endif /* interface */
```

```
template<class type, class list, class iter>
class constraints {

private:
  condition<type> cache;

protected:
  list *conds;

public:
  /* create */
  constraints() {conds = new list();}

  /* destroy */
  ~constraints() {delete conds;}

  /* is there a definition for a particular parameter number? */
  int isdef(type parmnm) {
    iter ilit(conds);
    while (!ilit.is_empty()) {
      if (ilit.peek().parm() == parmnm) {
            cache = ilit.peek(); return 1;}
      ilit.step();}
    return 0;
  }

  /* return def. for particular parameter. */
  condition<type> deflookup(type parmnm, int usecache = 0) {
    if (usecache && (cache.parm() == parmnm)) return cache;
    iter ilit(conds);
    while (!ilit.is_empty()) {
      if (ilit.peek().parm() == parmnm) return ilit.peek();
      ilit.step();}
    return condition<type>();
  }

  /* determine the setwise relationship between this and that */
  relenum relationto(constraints* that) {
    int subset = 1, superset = 1, equal = 1;
    iter thisiter(conds);

    while (!thisiter.is_empty())
      if (!that->condlist()->lookup(thisiter.step())) {
          equal = 0; subset = 0;}

    iter thatiter(that->condlist());
    while (!thatiter.is_empty()) {
      if (!conds->lookup(thatiter.step())) {
          equal = 0; superset = 0;}
    }

    if (equal) return r_equal;
    if (subset) return r_strict_subset;
    if (superset) return r_strict_superset;
    return r_disjoint;'
  }

  /* obvious */
  int subsetof(constraints *foo) {relenum r = relationto(foo);
                                  return ((r == r_strict_subset) || (r == r_equal));}
  int supersetof(constraints *foo) {relenum r = relationto(foo);
                                  return ((r == r_strict_superset) || (r == r_equal));}
  int equalto (constraints *foo) {return (relationto(foo) == r_equal);}
  int disjoint (constraints *foo) {return (relationto(foo) == r_disjoint);}
  list *condlist() {return conds;}
};
```

funcspec.h

```
/* conditions for function specialization */
typedef condition<int> funccond;

/* List of function-specialization conditions */
DECLARE_LIST_CLASS(funccond_list, funccond);

int foo;

/* function-specific constraints */
typedef constraints<int,funccond_list, funccond_list_iter> funconstraints;

class funcspec : public funconstraints {

 public:
  proc_sym *root, *next, *prev;
  proc_sym *self;

  /* create from a list of immediates */
  funcspec(immed_list *srcmat, proc_sym *slf);

  /* create more normally */
  funcspec(proc_sym *rot = NULL, proc_sym *slf = NULL,
           proc_sym *nxt = NULL , proc_sym *prv = NULL)
    : funconstraints() {
      root = rot; self = slf; next = nxt; prev = prv;}

  /* create based on a call */
  funcspec(in_cal *thecall);

  /* destroy */
  ~funcspec() {}

  /* translate to list of immeds */
  immed_list *unparse();

  /* complex operations */
  /* if there's a version of the root function that matches _this_,
  return the funcspec associated with it. */
  funcspec *equivproc();

  /* return the closest existing match to this that's been instantiated */
  funcspec *closest();

  /* generate a version of the root function tailored to these specifications;
  give it its own copy of this funcspec, and return that copy */
  funcspec *instantiate(sym_node_list *addlist);

  /* changes the call such that it calls sfunc. NOTE: thecall probably
    doesn't exist when this comes back!!! It'll have been replaced
    in its tree_instr.
    tailor returns 1 if the tailoring is successful; 0 if not.
    (most likely reason for 0 is that the call is already tailored to that
     function) */
  int tailor(in_cal *thecall);
};
```

blockspec.h

```
/* conditions for blocktion specialization */
typedef condition<var_sym *> blockcond;

/* List of block-specialization conditions */
DECLARE_LIST_CLASS(blockcond_list, blockcond);

/* block-specific constraints */
typedef constraints<var_sym *,blockcond_list, blockcond_list_iter> blockconstraints;


class blockform : public blockconstraints {
 private:
  tree_block *orig;

 public:
  /* creators */
  blockform(tree_block *src) {orig = src;}
  blockform(immed_list_iter &iliter, tree_block *src);

  void unparseto(immed_list *iml);

  /* create a version of the original block as constrained by this */
  tree_block *instantiate();

  /* create a conditional expression representing the constraints */
  tree_node_list *preconds(label_sym *jumplab);
};


DECLARE_LIST_CLASS(blockform_list, blockform*);

class blockspec {

 private:
  tree_block *orig;
  tree_node *rootpoint;
  blockform_list *possible, *actual;
public:
 /* creators */
 blockspec(tree_block *src) {rootpoint = orig = src;
                              possible = new blockform_list();
                              actual = new blockform_list();}
 blockspec(immed_list *il, tree_block *src);

 blockform_list *poslist() {return possible;}

 /* turn into immed_list */
 immed_list *unparse();

 /* add a possible form */
 void add_possible(blockform *poss) {possible->push(poss);}

 /* realize a possible form on top of the currently realized forms */
 int add_actual(blockform *act);

 /* remove the possibility of a form -- resets the actual form, as
    well, if done to a currently-realized form */
 void remove_possible(blockform *poss);

 /* resets the set of forms to just the original tree_block --
    no specializations */
 void reset_actual();
};
```

myutils.h

```
void proc_readall(int argc, char * argv[],
                  boolean writeback=FALSE,
                  boolean exp_trees=TRUE,
                  boolean use_fortran_form=TRUE);

void proc_processall (prociter_f fun);

typedef void (*procsymiter_f)(proc_sym *);

void procsym_processall (procsymiter_f fun);

void proc_saveall();
```

feedback.h

```
/* routines for manipulating various feedback things */

tree_instr *newcounter();


class feedstate {
 private:

  /* for counters */
  var_sym *counters;
  var_sym *numcounters;
  int freecounters;
  array_type *arrtype;
  type_node *cnttype;
  ptr_type *cntptrtype;
  ptr_type *arrptrtype;

  int newcountnum();

  /* for samples */
  var_sym *samples;
  var_sym *numsamples;

  array_type *sarrtype;
  struct_type *sampletype;
  ptr_type *sampleptrtype;
  ptr_type *sarrptrtype;

  int samplesize();
  int addsamples(int numbuckets);

 public:
  feedstate(immed_list *il);
  immed_list *unparse();

  tree_block *newcounter(block_symtab *bs);
  tree_block *varsample(block_symtab *bs, var_sym *var, int numbuckets,
                              tree_proc *procin);
};

void *feedstate_parse(char *name, immed_list *il, suif_object *obj);

immed_list *feedstate_unparse(char *name, void *data);


extern feedstate *fbstate;
```

B.1.2 Profiling Mechanisms

feedback.cc

```
#include "suif.h"
#include "feedback.h"
#include "builder.h"

extern char *k_fds_counter;

feedstate *fbstate;

/* hacks to deal with struct_annote requirements */
void *feedstate_parse(char *name, immed_list *il, suif_object *obj) {
  return (void *)new feedstate(il);}

immed_list *feedstate_unparse(char *name, void *data) {
  return ((feedstate *)data)->unparse();}



/* create from immed_list */
feedstate::feedstate(immed_list *il) {
  counters = (var_sym *) il->pop().symbol();
  numcounters = (var_sym *) il->pop().symbol();
  freecounters = il->pop().integer();
  samples = (var_sym *) il->pop().symbol();
  numsamples = (var_sym *) il->pop().symbol();

  fbstate = this;

  arrtype = (array_type *) counters->type();
  cnttype = arrtype->elem_type();
  cntptrtype = (ptr_type *) cnttype->parent()->install_type(new ptr_type(cnttype));
  arrptrtype = (ptr_type *) cnttype->parent()->install_type(new ptr_type(arrtype));

  sarrtype = (array_type *) samples->type();
  sampletype = (struct_type *) sarrtype->elem_type();
  sampleptrtype = (ptr_type *) sampletype->parent()->install_type(new ptr_type(sampletype));
  sarrptrtype =(ptr_type *) sampletype->parent()->install_type(new ptr_type(sarrtype));
}
```

```
/* TODO: gc unused numbers */
int feedstate::newcountnum() {
  /* fix the type of counters */
  immed_list_e *ncvale = ((immed_list *) numcounters->definition()->peek_annote(k_repeat_init))->tail();
  int newsize = ncvale->contents.integer()+1;
  ncvale->contents = immed(newsize);

  arrtype->set_upper_bound(array_bound(newsize));

  return newsize-1;
}



tree_block *feedstate::newcounter(block_symtab *bs) {
  tree_node_list *tnl = new tree_node_list();
  tree_block *tb = new tree_block(tnl, bs->new_unique_child("fds_counterblock"));

  /* create the new counter, store it in new var */
  var_sym *cv = tb->symtab()->new_unique_var(cntptrtype);

  in_array *ia = new in_array(cntptrtype,
                              operand(cv),
                              operand(new in_ldc(arrptrtype, operand(),
                                                 immed(counters))),
                              cnttype->size(),
                              1);
  int nc = newcountnum();
  ia->set_index(0,operand(new in_ldc(type_unsigned_long, operand(), immed(nc))));

  /* Create the actual increment instruction */
  in_rrr *incr = new in_rrr(io_str, type_void, operand(),
                            operand(cv),
                            operand(new in_rrr(io_add, cnttype, operand(),
                                               operand (new in_rrr(io_lod,
                                                                   cnttype,
                                                                   operand(),
                                                                   operand(cv),
                                                                   operand())),
                                          operand (new in_ldc(cnttype,
                                                        operand(),
                                                        immed(1)))
                                          )));

  /* Glom them into a tree_block and return */
  tnl->append(new tree_instr(ia));
  tnl->append(new tree_instr(incr));

  /* annotate the block before releasing it */
  immed_list *inl = new immed_list();
  inl->push(nc);
  tb->set_annote(k_fds_counter,(void *)inl);
  return(tb);
}

immed_list *feedstate::unparse() {
  immed_list *il = new immed_list();
  il->append(immed(counters));
  il->append(immed(numcounters));
  il->append(immed(freecounters));
  il->append(immed(samples));
  il->append(immed(numsamples));

  return il;
}


tree_block *feedstate::varsample(block_symtab *bs, var_sym *var,
                                  int numbuckets, tree_proc *procin) {
  /* here's a model for what we're trying to produce */
#if 0
  {
    sampletype *tmp = &(samples[15]);
    while(tmp->cnt) {
      if (v == tmp->data.i) {
        (tmp->cnt)++;
        goto done;
      }
      tmp++;
    }
    if (tmp < &(samples[19])) {
      tmp->data.i = v;
      (tmp->cnt)++;
    }
    else
      tmp->data.ll++;
  done:;
  }
#endif

  /* containing block */
  tree_node_list *tnl = new tree_node_list;
  block_symtab *tbs = bs->new_unique_child("fds_sampleblock");
  tree_block *tb = new tree_block(tnl, tbs);


  /* tmp->data.vartype, convert var to match vartype */
  char *vname;
  type_node *dvt;
  in_rrr *varcast;

  switch (var->type()->op()) {
  case TYPE_INT:
    dvt = type_unsigned_longlong;
    vname = "ll";
    if (((base_type *) var->type())->is_signed()) {
      base_type *tn = (base_type *) var->type()->copy();
      tn->set_signed(FALSE);
      tn = (base_type *) var->type()->parent()->install_type(tn);
      varcast = new in_rrr(io_cvt, type_unsigned_longlong,
                           operand(),
                           operand(new in_rrr
                                   (io_cvt, tn, operand(), operand(var),
                                    operand())),
                           operand());
    } else
      varcast = new in_rrr(io_cvt, type_unsigned_longlong,
                                operand(),
                                operand(var),
                                operand());
    break;
  case TYPE_FLOAT:
    dvt = type_longdouble;
    vname = "d";
    varcast = new in_rrr(io_cvt, type_longdouble,
                              operand(),
                              operand(var),
                              operand());
    break;
  case TYPE_PTR:
    dvt = type_ptr;
    vname = "v";
    varcast = new in_rrr(io_cvt, type_ptr,
                              operand(),
                              operand(var),
                              operand());
    break;
  default:
    assert(FALSE);
  }



  /* Okay, let's take it line-by-line */
  /* first, a mark just to keep things from crashing */
  in_rrr *marker = new in_rrr(io_mrk);
  tnl->append(new tree_instr(marker));
  immed_list *anlist = new immed_list;
  marker->set_annote(k_line, (void *)anlist);
  anlist->append(immed(0));
  anlist->append(immed("fds_sample"));

  /* line 1 */
  var_sym *tmp = tb->symtab()->new_unique_var(sampleptrtype, "fds_tmp");
  in_array *tmpasgn = new in_array(sampleptrtype,
                                   operand(tmp),
                                    operand(new in_ldc(sarrptrtype,
                                                       operand(),
                                                       immed(samples))),
                                    sampletype->size(),
                                    1);
tnl->append(new tree_instr(tmpasgn));
tmpasgn->set_index(0,operand(new in_ldc(type_unsigned_long,
                                         operand(),
                                         immed(samplesize()))));

/* lup:*/
label_sym *lup = tbs->new_unique_label("lup");
tnl->append(new tree_instr(new in_lab(lup)));

/* tmpcntptr = &(tmp->cnt); */
int cntnum = sampletype->find_field_by_name("cnt");
var_sym *tmpcntptr = tbs->new_unique_var
  (tbs->install_type
   (new ptr_type(sampletype->field_type(cntnum))),
   "tmpcntptr");

in_rrr *ir = new in_rrr
  (io_add,
   tmpcntptr->type(),
   operand(tmpcntptr),
   operand(tmp),
   operand(new in_ldc
             (type_unsigned_long, operand(),
              immed(sampletype->offset(cntnum)/8))));
immed_list *iltmp = new immed_list;
iltmp->push(immed("cnt"));
ir->set_annote(k_fields,(void *)iltmp);
tnl->append(new tree_instr(ir));

/* tmpcntval = *tmpcntptr */
var_sym *tmpcntval = tbs->new_unique_var
  (((ptr_type *)tmpcntptr->type())->ref_type(),"tmpcnt");
tnl->append(new tree_instr
              (new in_rrr
            (io_lod,
                ((ptr_type *)tmpcntptr->type())->ref_type(),
                operand(tmpcntval),
                (tmpcntptr))));

/* if (!tmpcntval) goto postlup */
label_sym *postlup = tbs->new_unique_label("postlup");
tnl->append(new tree_instr
            (new in_bj
             (io_bfalse, postlup,
                operand(tmpcntval))));

/*    if (!(v == tmp->data.vartype)) { */
/*           tmp->data.vartype */
cntnum = sampletype->find_field_by_name("data");
struct_type *un = (struct_type *)sampletype->field_type(cntnum);
int uncntnum = un->find_field_by_name(vname);
in_rrr *tmpdatavaraddr = new in_rrr(io_add,
                                    tb->symtab()->install_type
                                    (new ptr_type
                                     (un->field_type(uncntnum))),
                                    operand(),
                                    operand(tmp),
                                    operand(new in_ldc
                                            (type_unsigned_long, operand(),
                                             immed(sampletype->offset(cntnum)/8))));
iltmp = new immed_list;
iltmp->append(immed("data"));
iltmp->append(immed(vname));
tmpdatavaraddr->set_annote(k_fields, (void *)iltmp);
in_rrr *tmpdatavar = new in_rrr(io_lod, un->field_type(uncntnum),
                                operand(),
                                operand(tmpdatavaraddr->clone(tbs)));
label_sym *jt = tbs->new_unique_label("tmpplusplus");
/* if .... goto tmpplusplus; */
tnl->append(new tree_instr
            (new in_bj
             (io_bfalse, jt,
                operand(new in_rrr(io_seq,
                                        type_signed,
                                        operand(),
                                        operand(varcast->clone(tbs)),
                                        operand(tmpdatavar->clone(tbs)))))));

/* tmp->cnt++; */
in_rrr *tmpcntplusplus = new in_rrr
  (io_str,
   type_void,
   operand(),
   operand(tmpcntptr),
   operand(new in_rrr
          (io_add, tmpcntval->type(),
           operand(),
           operand(tmpcntval),
           operand(new in_ldc
                  (type_unsigned, operand(), immed(1))))));
tnl->append(new tree_instr(tmpcntplusplus->clone(tbs)));

/* goto done; */
label_sym *donept = tbs->new_unique_label("done");
tnl->append(new tree_instr
           (new in_bj(io_jmp, donept)));

/* tmpplusplus: */
tnl->append(new tree_instr(new in_lab(jt)));
/* tmp++; */
tnl->append(new tree_instr
           (new in_rrr
            (io_add,
              tmp->type(),
              operand(tmp),
              operand(new in_ldc(type_signed, operand(),
                                 immed(sampletype->size()/8))),
              operand(tmp))));

/* goto lup; */
tnl->append(new tree_instr
           (new in_bj(io_jmp, lup)));
```

```
/* postlup: */
tnl->append(new tree_instr(new in_lab(postlup)));

/* if (tmp < &samples[highend]) { */
label_sym *jumpto = tbs->new_unique_label("jumpto");
tree_node_list *header = new tree_node_list;
tree_node_list *thenpart = new tree_node_list;
tree_node_list *elsepart = new tree_node_list;
tnl->append(new tree_if(jumpto, header, thenpart, elsepart));
in_array *tmpcmp = new in_array(sampleptrtype,
                                operand(),
                                operand(new in_ldc(sarrptrtype,
                                                   operand(),
                                                   immed(samples))),
                                sampletype->size(),
                                1);
tmpcmp->set_index(0,operand(new in_ldc(type_unsigned_long,
                                       operand(),
                                       immed(addsamples(numbuckets)))));
header->append(new tree_instr
               (new in_bj
                (io_bfalse,
                 jumpto,
                 operand(new in_rrr
                         (io_sl,
                          type_signed,
                          operand(),
                          operand(tmp),
                          operand(tmpcmp))))));

/* then:  tmp->data.vartype = v; */
/* note: we use up tmpdatavaraddr and varcast here */
thenpart->append(new tree_instr
                 (new in_rrr(io_str,
                             tmpdatavaraddr->result_type(),
                             operand(),
                             operand(tmpdatavaraddr),
                             operand(varcast))));
```

```
/* then: tmp->cnt++; */
/* uses up tmpcntplusplus */
thenpart->append(new tree_instr(tmpcntplusplus));

/* else tmp->data.ll++ */
uncntnum = un->find_field_by_name("ll");
var_sym *tdllvptr = tbs->new_unique_var(tb->symtab()->install_type
                                        (new ptr_type
                                            (un->field_type(uncntnum))),
                                        "tdllvptr");
in_rrr *tmpdatallad = new in_rrr(io_add,
                                 tdllvptr->type(),
                                 operand(tdllvptr),
                                 operand(tmp),
                                 operand(new in_ldc
                                          (type_unsigned_long, operand(),
                                           immed(sampletype->
                                                 offset(cntnum)/8))));
iltmp = new immed_list;
iltmp->append(immed("data"));
iltmp->append(immed("ll"));
tmpdatallad->set_annote(k_fields, (void *)iltmp);
elsepart->append(new tree_instr(tmpdatallad));
elsepart->append(new tree_instr
                 (new in_rrr(io_str,
                             type_unsigned_longlong,
                             operand(),
                             operand(tdllvptr),
                             operand(new in_rrr
                                      (io_add,
                                       type_unsigned_longlong,
                                       operand(),
                                       operand(new in_ldc
                                                (type_signed,
                                                 operand(), immed(1))),
                                       operand(new in_rrr
                                                (io_lod,
                                                 type_unsigned_longlong,
                                                        operand(),
                                                        operand(tdllvptr))))))));

  /* done: ;*/
  tnl->append(new tree_instr(new in_lab(donept)));

  return(tb);
}

int feedstate::samplesize() {
  immed_list_e *ncvale = ((immed_list *) numsamples->definition()->peek_annote(k_repeat_init))->tail();
  return(ncvale->contents.integer());
}

int feedstate::addsamples(int numbuckets) {
  /* fix the type of samples */
  immed_list_e *ncvale = ((immed_list *) numsamples->definition()->peek_annote(k_repeat_init))->tail();
  int newsize = ncvale->contents.integer()+numbuckets+1;
  ncvale->contents = immed(newsize);

  sarrtype->set_upper_bound(array_bound(newsize));
  return(newsize);
}
```

B.1.3 Specialization Mechanisms

inline.cc

```
#include <stdlib.h>
#include "suif.h"

/* grunge support */

/* holds the label to which to jump and the variable into which to store
   a return value when converting returns into store-and-jumps in inlined
   procedures */
struct sjpack {
  label_sym *jlab;
  var_sym *rlab;
  sjpack(label_sym *jl, var_sym *rl) {jlab = jl; rlab = rl;}
};

void storjmp (tree_node *tn, void *data);
void justjmp (tree_node *tn, void *data);

/* Inlining ------------------------------------------------------------*/

tree_block *startinline(in_cal *ic) {
  tree_instr *t = ic->parent();
  /* make sure it's not an anon. func. call */
  operand adop = ic->addr_op();
  if (adop.instr()->opcode() != io_ldc) return NULL;
  in_ldc *adopldc = (in_ldc *)adop.instr();

  /* I think this has to be a symbol, so we won't check it. */
  proc_sym *calledptr = (proc_sym *) adopldc->value().symbol();

  /* function is in fileset? */
  if (calledptr->is_extern()) return NULL;

  /* Okay, let's do it... */
  /* Step 1: pretend it's just a block of code */
  /* Step 2: create a clone of the proc blocks */
  /* because we're converting from proc to block, we have
     to do a lot by hand here. */
```

```
tree_proc *src = calledptr->block();

replacements r;
src->find_exposed_refs(t->scope(), &r);
t->scope()->resolve_exposed_refs(&r);
block_symtab *newsymtab = src->symtab()->block_symtab::clone_helper(&r, FALSE);
tree_node_list *newbody = src->body()->clone_helper(&r);
tree_block *cln_blk = new tree_block(newbody, newsymtab);
src->clone_annotes(cln_blk, &r, FALSE);

/* Step 4: assign formals from call... */
if (!getenv("INLINE_INHIBIT_1")) {
  sym_node_list *parmlist = ((proc_symtab *)src->symtab())->params();
  sym_node_list_e *curr, *next = parmlist->head();
  int parmcnt = 0;

  while (next) {
    curr = next;
    next = curr->next();

    /* formal receiving a value */
    var_sym *parm = (var_sym *)
                    ((var_sym *) curr->contents)->clone_helper(&r);

    /* while we've got it: tell it it's not a parameter anymore. */
    parm->reset_param();

    /* The argument that would've been passed... */
    operand rval = ic->argument(parmcnt++);

    /* ...gets freed from the clutches of the call statement... */
    rval.remove();

    /* ...and subsequently is bound to the variable that used to be
          the formal parameter. */
    switch (rval.kind()) {
    case OPER_SYM:
          if (!getenv("INLINE_SYM_INHIBIT")) {
            /* type conversions look like instructions, not syms,
        so if we made it here, typing already works okay */
        cln_blk->body()->push
          (new tree_instr
           (new in_rrr(io_cpy, parm->type(), operand(parm),
                       rval)));}
       break;
    case OPER_INSTR:
       if (!getenv("INLINE_INSTR_INHIBIT")) {
        /* typing is dealt with already, again... */
        instruction *expr = rval.instr();
        /* dst should be blank by now */
        expr->set_dst(operand(parm));
        /* overkill checking -- shouldn't be a parent problem */
        if (expr->parent())
          if (expr->parent()->instr() == expr) {
           tree_instr *dead = expr->parent();
           dead->remove_instr(expr);
           dead->parent()->remove(dead->list_e());
           delete dead;
          }
        cln_blk->body()->push
        (new tree_node_list_e
         (new tree_instr(expr)));
       }
       break;
    default:
       assert_msg(FALSE,("This expression is incapable of having a dest?!?"));
    }
   }
  }

  return cln_blk;
 }


/* Finishes the inlining job, replacing thecall with theblock and doing
   relevant cleanup. */

void endinline(in_cal *ic, tree_block *cln_blk) {
```

```
tree_instr *t = ic->parent();

/* Step 3: insert the copy... */
t->parent()->insert_before(new tree_node_list_e(cln_blk), t->list_e());

if (!getenv("INLINE_INHIBIT_2")) {
  /* Now it's time to turn returns into assigns and jumps */
  /* step 1:  put label at end of block to jump to */
  /* First, put it into the symbol table */
  label_sym *retpoint = cln_blk->symtab()->new_unique_label("retpoint");

  /* Now put it at the tail of the body */
  cln_blk->body()->append(new tree_instr(new in_lab(retpoint)));

  /* step 2:  if the return value is used: */
  if (!ic->dst_op().is_null()) {
    assert(ic->dst_op().is_symbol());
    /* step 2.t1:  retval is the variable to which things were assigned */
    var_sym *retval = ic->dst_op().symbol();

    /* step 2.t2:  now, convert each return into store/jump */
    sjpack *sj = new sjpack(retpoint,retval);
    cln_blk->map(storjmp, (void *) sj, FALSE);
    delete sj;
  }
  else
    /* step 2.f1:  otherwise convert each return into jump */
    cln_blk->map(justjmp, (void *)retpoint, FALSE);
}

/* Step 5: nuke the call */
/* a) remove from list */
tree_node_list_e *dying = t->parent()->remove(t->list_e());


/* b) destroy call node itself... -- this takes out the ldc also
  since we're assuming exprs */
delete t;
/* c) destroy the list_e that held it */
```

```
  delete dying;

}

/* more grunge support */

void storjmp (tree_node *tn, void *data) {
  sjpack *labs = (sjpack *)data;

  if (!tn->is_instr()) return;
  tree_instr *ti = (tree_instr *)tn;
  instruction *in = ti->instr();

  /* Specifically, a return? */
  if (!(in->opcode() == io_ret)) return;
  in_rrr *ret = (in_rrr *) in;
  operand srcop = ret->src1_op();
  srcop.remove();
  type_node *rtyp;

  switch(srcop.kind()) {
  case OPER_NULL: rtyp = NULL;  break;
                    case OPER_SYM:  rtyp = srcop.symbol()->type(); break;
                    case OPER_INSTR: rtyp = srcop.instr()->result_type(); break;
                    }

  in_rrr *asgn = new in_rrr(io_cpy,rtyp,operand(labs->rlab),
                                   srcop);
  tn->parent()->insert_before(new tree_node_list_e(new tree_instr(asgn)),
                                    tn->list_e());
  tn->parent()->insert_after(new tree_node_list_e(new tree_instr
                                                              (new in_bj(io_jmp,
                                                                          labs->jlab))),
                                   tn->list_e());

  /* a) remove from list */
  tree_node_list_e *dying = tn->parent()->remove(tn->list_e());

  /* b) destroy call node itself... */
  delete tn;
  /* c) destroy the list_e that held it */
  delete dying;

}

void justjmp (tree_node *tn, void *data) {
  label_sym *jumppoint = (label_sym *)data;

  if (!tn->is_instr()) return;
  tree_instr *ti = (tree_instr *)tn;
  instruction *in = ti->instr();

  /* Specifically, a return? */
  if (!(in->opcode() == io_ret)) return;
  in_rrr *ret = (in_rrr *) in;
  operand srcop = ret->src1_op();
  srcop.remove();
  tn->parent()->insert_after(new tree_node_list_e(new tree_instr
                                                  (new in_bj(io_jmp,
                                                             jumppoint))),
                             tn->list_e());

  /* a) remove from list */
  tree_node_list_e *dying = tn->parent()->remove(tn->list_e());

  /* b) destroy call node itself... */
  delete tn;
  /* c) destroy the list_e that held it */
  delete dying;
}
```

specnoin.cc

```
#include <stdlib.h>
#include "suif.h"
#include "interface.h"


int fullspec(in_cal *thecall, sym_node_list *newlist) {
/* Replaces thecall with a call to a fully specialized version of
   the function.  Deals with unification issues: generates new spec'ed
   version if needed, uses pre-created one if possible.  returns 0 if no
   (further) spec possible, 1 if used old one, 2 if made new one. */

  funcspec *sfunc;
  /* describe the call's constraints, etc. */
  funcspec callform(thecall);

  /* if a version already fits the requirements, just tailor the call to it.
     Tailor returns 1 if it actually changes things, 0 if the call was
     already tailored to the named site.  Not efficient, but a little
     cleaner than other options. */
  if (sfunc = callform.equivproc())
    return (sfunc->tailor(thecall));

  /* otherwise, time to make a new function */
  sfunc = callform.instantiate(newlist);
  assert(sfunc);
  sfunc->tailor(thecall);
  return 2;
}

int tryspec(in_cal *thecall) {
  /* Attempts to replace thecall with a call to a more specialized version --
     draws from already-created spec'ed versions, but does NOT create new ones.
     Picks strongest form -- alg needed here, probably most args. */

  /* profile the call */
  funcspec callform(thecall);
  return callform.closest()->tailor(thecall);
}
```

funcspec.cc

```
#include<stdio.h>
#include"suif.h"
#include"interface.h"

#define callee(a) ((proc_sym *)((in_ldc *)((a)->addr_op().instr()))->value().symbol())



proc_sym *unique_proc_sym_copy(proc_sym *src) {
  proc_sym *retval = (proc_sym *) src->copy();

  char bfr[1024];

  /* provide a default base for the variable name */
  char *base = src->name();
  int var_counter = 1;

  /* add a number to the base name and make sure it's unique */
  while (TRUE) {
    sprintf(bfr, "%s_%d", base, var_counter++);
    if (!((file_symtab *)src->parent())->lookup_proc(bfr, FALSE)) break;
  }
  retval->set_name(bfr);
  return retval;
}

funcspec *formof(proc_sym *variant) {
  if (!variant) return NULL;
  funcspec *compform = (funcspec *) ((variant)->peek_annote(k_funcspec));
  if (compform) return compform;

  compform = new funcspec(variant, variant);
  variant->set_annote(k_funcspec,(void *)compform);
  return compform;
}


int is_constr(operand foo) {
  return (foo.is_instr() && (foo.instr().opcode() == io_ldc));}

immed constraint(operand foo) {return((in_ldc *)foo.instr()).value();}

proc_sym *rootof(proc_sym *variant) {
  funcspec *compform = formof(variant);
  return compform->root;
}



/* create based on a call */
funcspec::funcspec(in_cal *thecall) {
  self = NULL;
  proc_sym *called = callee(thecall);
  root = rootof(called);
  next = prev = NULL;
  conds = new funccond_list();

  funcspec *form = formof(called);

  int rootsize = root->block()->proc_syms()->params()->count();
  int callarg = 0;

  /* figure out constraints -- this may have to change when
    constraints become more complicated.  Inherit from called
    function, and then include any new constraints in the call */
  for (int parmcount = 0; parmcount < rootsize; parmcount++) {
    if (form->isdef(parmcount))
      conds->push(form->deflookup(parmcount));
    else if (is_constr(thecall->argument(callarg++)))
      conds->push(funccond(parmcount,
                                     constraint(thecall->argument(callarg-1))));
  }
}

/* create from a list of immediates */
funcspec::funcspec(immed_list *srcmatpt, proc_sym *slf)
: funconstraints(){
  immed_list &srcmat = *srcmatpt;
  int numims = srcmat.count();
  assert(numims >= 3);
  conds = new funccond_list();
  self = slf;
  root = (proc_sym *) (srcmat[0]).symbol();
  next = (proc_sym *) (srcmat[1]).symbol();
  if (next == root) next = NULL;
  prev = (proc_sym *) (srcmat[2]).symbol();
  if ((prev == root) && (self == root))
    prev = NULL;
  int cnt;
  for (cnt = 3; cnt < numims; cnt += 2) {
    funccond c(srcmat[cnt].integer(),srcmat[cnt+1]);
    conds->push(c);}
}


/* translate to list of immeds */
immed_list *funcspec::unparse() {
  immed_list *il = new immed_list();
  il->append(immed(root));
  if (next)
    il->append(immed(next));
  else
    il->append(immed(root));
  if (prev)
    il->append(immed(prev));
  else
    il->append(immed(root));
  funccond_list_iter ilit(conds);

  while (!ilit.is_empty()) {
    il->append(immed(ilit.peek().parm()));
    il->append(ilit.step().constv());
  }
  return il;
}
```

```
/* complex operations */
/* if there's a version of the root function that matches _this_,
return the funcspec associated with it. */
funcspec *funcspec::equivproc() {
  funcspec *cform = formof(root);

  while (cform) {
    if (equalto(cform))
      return cform;
    cform = formof(cform->next);
  }
  return NULL;
}


/* return the closest existing match to this that's been instantiated */
funcspec *funcspec::closest() {
  funcspec *cform = formof(root);
  funcspec *winform = cform;
  while (cform) {
    if (supersetof(cform)) {
      relenum r = cform->relationto(winform);
      if ((r == r_strict_superset) ||
          ((r == r_disjoint) &&
             (cform->condlist()->count() > winform->condlist()->count())))
        winform = cform;
    }
    cform = formof(cform->next);
  }
  return winform;
}

/* generate a version of the root function tailored to these specifications;
give it its own copy of this funcspec, and return that copy */
funcspec *funcspec::instantiate(sym_node_list *addlist) {
  proc_sym *calledptr = root;

  /* function is in fileset? */
```

```
if (calledptr->is_extern()) return NULL;

/* Okay, let's do it... */
/* originating file */
file_symtab *myfile = (file_symtab *) calledptr->parent();

/* Step 1: create a clone of the proc */
proc_sym *newfunc = unique_proc_sym_copy(calledptr);

/* Here's where we add it to the global symbol table -- is
   this what we wanted to do?!? */
myfile->add_sym(newfunc);

/* TODO: should probably change the name here to something
  unique */
newfunc->set_block(calledptr->block()->clone(newfunc->parent()));

/* Step 1.1: create a new type for newfunc. */
func_type *newtype = (func_type *) newfunc->type()->copy();
myfile->add_type(newtype); /* it may be a duplicate now, but we're
                              mutating it, so keep it distinct... */
ptr_type *newtypeptr = new ptr_type(newtype);
myfile->add_type(newtypeptr);

newtype->set_args_unknown();
newfunc->set_type(newtype);

/* Step 2: assign constant formals from this... */
/* scan over this, putting constants in. */

/* iterate over the function's formals */
sym_node_list *parmlist = newfunc->block()->proc_syms()->params();
sym_node_list_e *curr, *next = parmlist->head();

int parmcnt = -1;

while (next) {
  curr = next;
  next = curr->next();
```

```
    parmcnt++;

    if (!isdef(parmcnt)) continue;

    /* Get the formal receiving a value... */
    var_sym *parm = (var_sym *) curr->contents;

    /* assign the constant to the formal from the proc-block */
    in_ldc *nl = new in_ldc(parm->type(),
                            operand(parm),
                            deflookup(parmcnt).constv());

    /* Now insert it at the proc-block list */
    newfunc->block()->body()->push(new tree_instr(nl));

    /* de-formalize the formal, and pull it from the param list */
    parm->reset_param();
    parmlist->remove(curr);
    delete curr;
  }

  /* finally, copy this for the new function, and put it in the variant
    list */
  funcspec *rootform = formof(root);
  funcspec *newbie = new funcspec(root, newfunc, rootform->next, root);
  if (rootform->next)
    formof(rootform->next)->prev = newfunc;
  rootform->next = newfunc;
  funccond_list *lst = newbie->condlist();
  funccond_list_iter thisiter(conds);
  while (!thisiter.is_empty())
    lst->append(thisiter.step());
  addlist->push(newfunc);
  newfunc->set_annote(k_funcspec,(void *)newbie);
  return newbie;
}


/* changes the call such that it calls sfunc.  NOTE: thecall probably
```

```
doesn't exist when this comes back!!! It'll have been replaced
in its tree_instr. tailor returns 1 if the tailoring is successful; 0 if not.
(most likely reason for 0 is that the call is already tailored to that
 function) */

int funcspec::tailor(in_cal *oldcall) {
  /* sanity checks */
  if (!self) return 0;
  proc_sym *oldfunc = callee(oldcall);
  if (self == oldfunc) return 0;

  /* useful info to have around */
  funcspec *oldform = formof(oldfunc);
  sym_node_list *rootparms = root->block()->proc_syms()->params();
  int rootsize = rootparms->count();

  /* counters */
  int oldcnt = 0;
  int newcnt = 0;
  func_type *newtype = self->type();
  ptr_type *newtypeptr = new ptr_type(newtype);
  newtypeptr = (ptr_type *) ((file_symtab *) self->parent())->install_type(newtypeptr);

  /* construct the outline of the new call */
  operand calladdr(new in_ldc(newtypeptr,operand(),immed(self)));
  in_cal *newcall = new in_cal(self->type()->return_type(),oldcall->dst_op(),
                                         calladdr,rootsize);

  /* Iterate through the number of root params.
    for each one, here's what we do --
    if it's defined by the new function
    (if it's defined by the old function  ignore it
     else advance the counter past that arg in the old call)
    else if it's defined by the old function  put that def. into the new call
            else copy arg from old call to new call */

  for (int parmcount = 0; parmcount < rootsize; parmcount++)
    /* defined in new but not old means that we can skip this
    arg in the old call */
    if (isdef(parmcount)) {
      if (!oldform->isdef(parmcount))
            oldcnt++;}
    else
      if (oldform->isdef(parmcount)) {
            operand parm(new in_ldc(((var_sym *)(*rootparms)[parmcount])->type(),
                                  operand(),
                                  oldform->deflookup(parmcount).constv()));
            newcall->set_argument(newcnt++,parm);}
      else {
            operand rval = oldcall->argument(oldcnt++);
            rval.remove();
            newcall->set_argument(newcnt++,rval);}
  newcall->set_num_args(newcnt);

  /* replace old call with new call */
  tree_instr *ti = oldcall->parent();
  ti->remove_instr(oldcall);
  ti->set_instr(newcall);
  delete oldcall;
  return 1;
}
```

dumbblock.cc

```
#include <stdlib.h>
#include "suif.h"
#include "interface.h"

/* replaces theblock with a suite of specialized blocks --
   order of blocks is direct from conlist.  theblock ceases to exist. */
/* returned 0 means no specialization occurred */
int dumbblockspec (tree_block *theblock, blockspec_list *conlist) {
  /* sanity check */
  if (!conlist->count()) return 0;

  /* iterate over the options */
  blockspec_list_iter bliter(conlist);
  tree_node *baseblock = theblock->clone(theblock->scope());
  tree_node *growth = baseblock;

  while (!bliter.is_empty())
    growth = bliter.step()->buildselfon(baseblock, growth);

  tree_node_list *tnl = theblock->parent();
  tnl->insert_after(growth, theblock->list_e());
  tnl->remove(theblock->list_e());
  delete(theblock->list_e());
  delete(theblock);
}
```

blockspec.cc

```
#include<stdio.h>
#include"suif.h"
#include"interface.h"

extern char *k_blockspec;

/* reconstruct from a list of immeds */
blockform::blockform(immed_list_iter &iliter, tree_block *src) {
  orig = src;
  int numconds = iliter.step().integer();
  while (numconds--) {
    var_sym *sym = (var_sym *) iliter.step().symbol();
    conds->push(blockcond(sym, iliter.step()));
  }
}

/* deconstruct to a list of immeds */
void blockform::unparseto(immed_list *iml) {
  /* number of conditions */
  iml->append(immed(conds->count()));

  /* now deconstruct each cond and send it... */
  blockcond_list_iter bliter(conds);
  while (!bliter.is_empty()) {
    iml->append(immed(bliter.peek().parm()));
    iml->append(bliter.step().constv());
  }
}

/* create a version of the original block as constrained by this */
tree_block *blockform::instantiate() {
  tree_block *newblock = orig->clone(orig->scope());
  /* get rid of the annote-clone */
  newblock->get_annote(k_blockspec);
  tree_node_list *bod = newblock->body();

  blockcond_list_iter bliter(conds);
  while (!bliter.is_empty()) {
    var_sym *var = bliter.peek().parm();
    immed val = bliter.step().constv();
    tree_instr *asgn = new tree_instr(new in_ldc(var->type(),
                                                  operand(var),
                                                  val));
    bod->push(asgn);
  }
  return newblock;
}


/* create an expression calculating the truth of the conditions of this
   form */
tree_node_list *blockform::preconds(label_sym *jumplab) {

  /* iterator over conditions */
  blockcond_list_iter bliter(conds);

  /* shouldn't do this */
  assert (!bliter.is_empty());

  /* tree to return */
  tree_node_list *condjmp = new tree_node_list();

  while (!bliter.is_empty()) {
    /* organize content */
    var_sym *var = bliter.peek().parm();
    immed val = bliter.step().constv();

    /* comparison */
    in_rrr *cmp = new in_rrr(io_seq, type_s32, operand(), operand(var),
                             operand(new in_ldc(var->type(),operand(),val)));

    /* branch */
    in_bj *bf = new in_bj(io_bfalse, jumplab, operand(cmp));

    /* add to list */
    condjmp->push(new tree_instr(bf));
  }
  return condjmp;
}


/* ------------ blockspec -------------- */


blockspec::blockspec( immed_list *il, tree_block *src) {
  possible = new blockform_list();
  actual = new blockform_list();

  orig = src;

  immed_list_iter bliter(il);
  /* int rootnum = */ bliter.step().integer();
  /* TODO: make this work for real */
  rootpoint = orig;

  int actcount = bliter.step().integer();
  while (actcount--) {
    blockform *newbie = new blockform(bliter, orig);
    actual->append(newbie);
    possible->append(newbie);
  }

  while (!bliter.is_empty()) {
    blockform *newbie = new blockform( bliter, orig);
    possible->append(newbie);
  }
}

immed_list *blockspec::unparse() {
  immed_list *iml = new immed_list;
  /*  the identity of the rootpoint */
  iml->append(immed(rootpoint->number()));

  /* now each blockform -- actuals first */
  blockform_list_iter abliter(actual);
```

```
iml->append(immed(actual->count()));
while (!abliter.is_empty())
  abliter.step().unparseto(iml);

/* then possibles */
/* no count needed -- last one is last one */
blockform_list_iter bbliter(possible);
while (!bbliter.is_empty()) {
  blockform *cur = bbliter.step();
  /* whether or not it's actualized... */
  if (!actual->lookup(cur))
    cur.unparseto(iml);
}
return iml;
}

/* realize a possible form on top of the currently realized forms */
int blockspec::add_actual(blockform *act) {
  if (actual->lookup(act)) return 0;

  if (!possible->lookup(act))
    possible->push(act);

  actual->push(act);

  /* prepare the if's args */
  label_sym *jumplab = rootpoint->proc()->block()->proc_syms()->new_unique_label();
  tree_node_list *header = act->preconds(jumplab);
  tree_node_list *thenpart = new tree_node_list();
  thenpart->push(act->instantiate());
  tree_node_list *elsepart = new tree_node_list();

  /* make the if */
  tree_if *newif = new tree_if(jumplab, header, thenpart, elsepart);

  rootpoint->parent()->insert_after(newif, rootpoint->list_e());
  rootpoint->parent()->remove(rootpoint->list_e());

  /* now that the old blockspec-tree is free, we can add it to the
```

```
    thenpart of the new if */
  elsepart->push(rootpoint->list_e());
  rootpoint = newif;
  /* TODO: annotate the new if with something indicating its origin */
  return 1;
}


/* remove the possibility of a form -- resets the actual form, as
well, if done to a currently-realized form */
void remove_possible(blockform *poss) {assert(0);}

/* resets the set of forms to just the original tree_block --
no specializations */
void reset_actual() {assert(0);}
```

B.1.4 Miscellaneous

myutils.cc

```
#include "suif.h"
#include "myutils.h"

void
proc_readall (int argc, char * argv[],
              boolean writeback,
              boolean exp_trees,
              boolean use_fortran_form)
{
 extern int optind;

  if (writeback) {
    if (argc-optind == 0)
      error_line(-1, NULL, "No files given");
    if (argc-optind == 1)
      error_line(-1, NULL, "No file to write back");
    if ((argc-optind)%2 != 0)
      error_line(-1, NULL, "File mismatch, file missing to write back");
    for (int i = optind; i < argc; i += 2) {
      fileset->add_file(argv[i], argv[i+1]);
    }
  } else {
    if (argc-optind == 0)
      error_line(-1, NULL, "No files given");
    for (int i = optind; i < argc; i++) {
      fileset->add_file(argv[i], NULL);
    }
  }

  fileset->reset_iter();
  file_set_entry *fse;

  while (fse = fileset->next_file()) {
    fse->reset_proc_iter();

    proc_sym *ps;
    while (ps = fse->next_proc()) {
        if (!ps->is_in_memory()) {
            ps->read_proc(exp_trees,
                          (ps->src_lang() == src_fortran) ?
                          use_fortran_form : FALSE);
        }
      }}}


void proc_processall (prociter_f fun)
{
  fileset->reset_iter();
  file_set_entry *fse;

  while (fse = fileset->next_file()) {
    fse->reset_proc_iter();

    proc_sym *ps;
    while (ps = fse->next_proc())
      (fun)(ps->block());
  }
}

void procsym_processall (procsymiter_f fun)
{
  fileset->reset_iter();
  file_set_entry *fse;

  while (fse = fileset->next_file()) {
    fse->reset_proc_iter();

    proc_sym *ps;
    while (ps = fse->next_proc())
      (fun)(ps);
  }
}


void proc_saveall ()
{
  fileset->reset_iter();
  file_set_entry *fse;

  while (fse = fileset->next_file()) {
    fse->reset_proc_iter();

    proc_sym *ps;
    while (ps = fse->next_proc()) {
      ps->block()->body()->cvt_to_trees();
      ps->write_proc(fse);
      ps->flush_proc();
    }
  }
}
```

B.2 Initialization

B.2.1 feedinit

feedinit.cc

```
#include<stdio.h>
#include<stdlib.h>
#include "suif.h"
#include "builder.h"
#include "interface.h"

char *k_feedstate;
char *k_fds_cntarr;

/* globalize all static procedures, alpha−renaming as needed */
void globalize(proc_sym *tp) {
  global_symtab *gs = fileset.globals();
  if (tp−>parent() == gs) return;

  /* gotta globalize local function's types first */
  if (tp−>type()−>parent() != gs) {
    tp−>type()−>parent()−>remove_type(tp−>type());
    gs−>add_type(tp−>type());
  }


  /* make sure its name is unique in the global scope*/
  if (gs−>lookup_proc(tp−>name())) {
    char bfr[1024];

    /* provide a default base for the variable name */
    char *base = tp−>name();
    int var_counter = 1;

    /* add a number to the base name and make sure it's unique */
    while (TRUE) {
      sprintf(bfr, "%s_%d", base, var_counter++);
      if (!gs−>lookup_proc(bfr)) break;
    }
    tp−>set_name(bfr);
  }
  /* now globalize the uniquely-named function. */
  tp->parent()->remove_sym(tp);
  gs->add_sym(tp);
}

proc_sym *make_feedin() {
  global_symtab *gs = fileset.globals();
  func_type *oetype = (func_type *) gs.install_type(new func_type(type_void));
  return gs->new_proc(oetype,src_c,"_fds_init");
}

/* TODO: should do something more clever than this --
   we're silently relying on canonical names at the moment. */

proc_sym *make_feedout() {
  /* find stuff */
  global_symtab *gs = fileset.globals();
  proc_sym *mn =  gs->lookup_proc("main");

  /* set up for counters ------------------------------ */
  /* array of longlongs for the counter -- annote for uniqueness */
  array_type *newtype = new array_type(type_unsigned_longlong,
                                       array_bound(0),
                                       array_bound(0));
  newtype->set_annote(k_fds_cntarr,NULL);
  gs->add_type(newtype);

  /* create the new counters... */
  var_sym *counters = gs->new_var(newtype, "_fds_counters");

  /* and define them in main's file */
  mn->file()->symtab()->add_def(new var_def(counters, newtype->size()));

  /* same for a variable counting the counters... */
  var_sym *numcounters = gs->new_var(type_unsigned,
                                           "_fds_numcounters");
  var_def *nd =new var_def(numcounters, numcounters->type()->size());
  mn->file()->symtab()->add_def(nd);
  immed_list *ild = new immed_list();
```

```
ild->append(immed(1));
ild->append(immed(numcounters->type()->size()));
ild->append(immed(0));
nd->set_annote(k_repeat_init,(void *)ild);

/* set up for variable sampling -------------------------- */
block::set_proc(mn->block()); /* we'll try the builder, what the hell */
type_node *sdata = block::parse_type(gs,"union{%% ll; %% d; void *v;}",
                                     type_unsigned_longlong,
                                     type_longdouble);

type_node *samps = block::parse_type(gs,"struct{%% cnt; %% data;}",
                                     type_unsigned_longlong, sdata);

array_type *sarrtype = new array_type(samps,
                                      array_bound(0),
                                      array_bound(0));
sarrtype->set_annote(k_fds_cntarr,NULL);
gs->add_type(sarrtype);

/* create the new samples... */
var_sym *samparr = gs->new_var(sarrtype, "_fds_samples");

/* and define them in main's file */
mn->file()->symtab()->add_def(new var_def(samparr, sarrtype->size()));

/*  a variable counting the samples... */
var_sym *numsamples = gs->new_var(type_unsigned,
                                        "_fds_numsamples");
var_def *nsd =new var_def(numsamples, numsamples->type()->size());
mn->file()->symtab()->add_def(nsd);
immed_list *nild = new immed_list();
nild->append(immed(1));
nild->append(immed(numcounters->type()->size()));
nild->append(immed(0));
nsd->set_annote(k_repeat_init,(void *)nild);


/* Finally, document this stuff in an annote */
```

```
  immed_list *il = new immed_list();
  /* final format of il:  counters, numcounters, freecounters */
  /* samples, numsamples */
  il->append(immed(counters));
  il->append(immed(numcounters));
  il->append(immed(0));
  il->append(immed(samparr));
  il->append(immed(numsamples));

  /* hang it on main's proc_sym */
  mn->set_annote(k_feedstate, (void *)il);

  func_type *oetype = (func_type *) gs.install_type(new func_type(type_void));
  return gs->new_proc(oetype,src_c,"_fds_feedout");
}


/* actually put the feedback calls in */
void install_feed(proc_sym *fbi, proc_sym *fbo) {

/* first, put the on_exit call in (not very portable) */
  global_symtab *gs = fileset.globals();
  func_type *oetype = (func_type *) gs.install_type(new func_type(type_signed));
  proc_sym *oe = gs->new_proc(oetype,src_c,"on_exit");

  proc_sym *mps = gs->lookup_proc("main");

  /* make the call instr */
  ptr_type *oeptr = (ptr_type *) gs.install_type(new ptr_type(oetype));
  in_cal *tc = new in_cal(type_signed,operand(),
                                   operand(new in_ldc(oeptr, operand(), immed(oe))),
                                   2);
  ptr_type *fboptr = (ptr_type *) gs.install_type(new ptr_type(fbo->type()));
  tc->set_argument(0,operand(new in_ldc(fboptr, operand(), immed(fbo))));
  tc->set_argument(1,operand(new in_ldc(type_signed, operand(), immed((int) 0))));

  /* stick it in */
  mps->block()->body()->push(new tree_instr(tc));
```

```
/* now, put in the call to init */
 mps->block()->body()->push(new tree_instr
                              (new in_cal
                               (type_void,operand(),
                                operand(new in_ldc
                                        (gs.install_type
                                           (new ptr_type(fbi->type())),
                                           operand(),
                                           immed(fbi))),
                                0)));

};

main (int argc, char *argv[]) {

 start_suif(argc, argv);
 ANNOTE(k_feedstate, "feedback state", TRUE);
 ANNOTE(k_fds_cntarr, "k_fds_cntarr", TRUE);
 proc_readall(argc,argv,TRUE);

 procsym_processall(globalize);
 proc_sym *fbi = make_feedin();
 proc_sym *fbo = make_feedout();
 install_feed(fbi, fbo);

 proc_saveall();
 delete fileset;
}
```

B.2.2 fds.c

fds.c

```
#include<stdio.h>
extern int _fds_numcounters;
extern int _fds_numsamples;

extern unsigned long long _fds_counters[];

typedef struct _fds_sampletype {
  long cnt;
  union {unsigned long long ll;
          long double d;
          void *v;} data;}
_fds_sampletype;

extern _fds_sampletype _fds_samples[];

void _fds_init() {
}

/* We actually write out accumulated feedback info here */
void _fds_feedout(int stat, void *arg) {
  FILE *outfile = fopen("fdsmon.out","w");

  /* Header: write out counts of types of data */
  fwrite((char *) &_fds_numcounters, sizeof(int), 1, outfile);
  fwrite((char *) &_fds_numsamples, sizeof(int), 1, outfile);

  /* First, write out all counters */
  fwrite((char *) _fds_counters,sizeof(unsigned long long), _fds_numcounters,outfile);
  /* printf("%d\n",(int)_fds_numcounters);*/

  /* Now, write out all samples */
  fwrite((char *) _fds_samples,sizeof(_fds_sampletype), _fds_numsamples, outfile);

  fclose(outfile);
}
```

B.3 feedback collection

B.3.1 PC Sampling

pcback.cc

```
#include<stdio.h>
#include<stdlib.h>
#include "suif.h"

char *k_pcdata;

main (int argc, char *argv[]) {
  char buf[1024];
  char name[255];
  double pertime, cumsecs, selfsecs;

  start_suif(argc, argv);
  ANNOTE(k_pcdata, "pcdata", TRUE);

  extern int optind;
  for (int i = optind; i < argc; i += 2)
    fileset->add_file(argv[i], argv[i+1]);

  fileset->reset_iter();
  file_set_entry *fse;

  while (fse = fileset->next_file()) {
    fse->reset_proc_iter();

    proc_sym *ps;
    while (ps = fse->next_proc()) {
      ps->read_proc();
      ps->write_proc(fse);
      ps->flush_proc();
    }
  }
  // we assume all functions are global, so only look here
  global_symtab *gs = fileset->globals();

  // nuke header garbage
  gets(buf); gets(buf); gets(buf); gets(buf); gets(buf);
```

```
 // process each line of information
 while (scanf("%lf %lf %lf %s", &pertime, &cumsecs, &selfsecs, &name) == 4) {
   proc_sym *fname = gs->lookup_proc(name);
   if (!fname) {printf("Lib function?  %s\n",name); continue;}
   immed_list *il = new immed_list();
   il->append(immed(pertime));
   il->append(immed(cumsecs));
   il->append(immed(selfsecs));
   fname->prepend_annote(k_pcdata,(void *) il);
 }
 delete fileset;
}
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Co., 1986.

[2] Jeremy Brown, André DeHon, Ian Eslick, Michelle Goldberg, Ahmed Shah, and Massimiliano Poletto. The clever compiler: Proposal for a first-cut smart compiler. Transit Note 101, MIT Artificial Intelligence Laboratory, January 1994.

[3] Jeremy Brown, André DeHon, and Massimiliano Poletto. A proposal for first-cut specialization mechanisms. Transit Note 104, MIT Artificial Intelligence Laboratory, May 1994.

[4] Craig Chambers, David Ungar, and Elgin Lee. An efficient implentation of self, a dynamically-typed ojbect-oriented language based on prototypes. *LISP and Symbolic Computation*, 4(3), 1991.

[5] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic code optimizations. *Software - Practice And Experience*, 21(12):1301–1322, December 1991.

[6] William Y. Chen, Pohua P. Chang, Thomas M. Conte, and Wen mei W. Hwu. The effect of code expanding optimizations on instruction cache design. CRHC 91-17, Center for Reliable and High Performance Computing, University of Illinois, University of Illinois, Urbana-Champaign, Illinois, 61801, May 1991.

[7] Thomas M. Conte, Burzin A. Patel, and J. Stan Cox. Using branch hardware to suport profile-driven optimization. In *Proceedings of the 27th Annual Symposium on Microarchitecture*, pages 12–21. ACM, November 1994.

[8] André DeHon. Smart compiler advantages. Transit Note 99, MIT Artificial Intelligence Laboratory, December 1993.

[9] André DeHon, Jeremy Brown, Ian Eslick, and Thomas F. Knight, Jr. Global cooperative computing. Transit Note 105, MIT Artificial Intelligence Laboratory, April 1994.

[10] André DeHon and Ian Eslick. Computational quasistatics. Transit Note 103, MIT Artificial Intelligence Laboratory, March 1994.

[11] André DeHon and Ian Eslick. Starting point for clever-compiler feedback. Transit Note 108, MIT Artificial Intelligence Laboratory, May 1994.

[12] André DeHon, Ian Eslick, John Mallery, and Thomas F. Knight Jr. Prospects for a smart compiler. Transit Note 87, MIT Artificial Intelligence Laboratory, June 1993.

[13] Dawson R. Engler and Todd A. Proebsting. Dcg: An efficient retargetable dynamic code generation system. contact: todd@cs.arizonda.edu, engler@lcs.mit.edu, November 1993.

[14] Joseph Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. ACM SIGPLAN, ACM, June 1982. SIGPLAN Notices, Volume 17, Number 6.

[16] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice Hall, 1993.

[17] Paul Kleinrubatscher, Albert Kriegshaber, Robert Zöchling, and Robert Glück. Fortran program specialization. *ACM SIGPLAN Notices*, 30(4):61–70, April 1995.

[18] Donald E. Knuth. Empirical study of fortran programs. *Software – Practice and Experience*, 1:105–133, 1971.

[19] Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. CMU-CS 93-225, Carnegie-Mellon, December 1993.

[20] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[21] Alan Dain Samples. *Profile-driven Compilation.* PhD thesis, U.C. Berkeley, April 1991. U.C. Berkeley CSD-91-627.

[22] Richard Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139, October 1993.

[23] Larry Wall and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[24] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary Hall, Monica Lam, , and John Hennessy. An overview of the suif compiler system.