Selecting better-performing alternative code using run-time profiling feedback

by

Ping-Shun Huang

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

© Copyright 1995, Massachusetts Institute of Technology. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

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

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

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

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 10 1995

LIBRARIES
Barker Eng

Selecting better-performing alternative code using run-time profiling feedback

by

Ping-Shun Huang

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

In this thesis, I specify several extensions to the C programming language — quasistatic variables and parameters, and the `qif-qelse qinfluence` statements — and implement them by modifying the Stanford University Intermediate Format (SUIF) compiler. Using these constructs, a programmer can specify alternative implementations in the source code. The compiler can then *experimentally* determine the performance characteristics of different combinations of these alternatives, by instrumenting the executables delivered to the end-user to obtain profiles (of real time elapsed) while the program is in actual production usage. Periodically, the compiler replaces the executable with a different version to try out another combination. After some time, the compiler tries to converge to the version of the program which has proven to be the fastest thus far, while continuing to monitor overall program performance to refine and confirm its experimental results. I evaluate the potential benefits of using such a modified compiler to improve program performance, both by modifying an existing benchmark program and by writing several example programs from scratch.

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

Selecting better-performing alternative code using run-time profiling feedback

by

Ping-Shun Huang

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

In this thesis, I specify several extensions to the C programming language — quasistatic variables and parameters, and the `qif-qelse` `qinfluence` statements — and implement them by modifying the Stanford University Intermediate Format (SUIF) compiler. Using these constructs, a programmer can specify alternative implementations in the source code. The compiler can then *experimentally* determine the performance characteristics of different combinations of these alternatives, by instrumenting the executables delivered to the end-user to obtain profiles (of real time elapsed) while the program is in actual production usage. Periodically, the compiler replaces the executable with a different version to try out another combination. After some time, the compiler tries to converge to the version of the program which has proven to be the fastest thus far, while continuing to monitor overall program performance to refine and confirm its experimental results. I evaluate the potential benefits of using such a modified compiler to improve program performance, both by modifying an existing benchmark program and by writing several example programs from scratch.

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

Acknowledgments

This research and the Reinventing Computing group at 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.

The bulk of this thesis has been inspired by numerous technical notes of the Transit project, the predecessor to the Reinventing Computing group. I especially wish to acknowledge the influence of discussions with André DeHon. I'd like to thank my thesis advisor Thomas F. Knight, Jr., for his faith in giving me free rein on my thesis work. Without the research assistantship funding, paying tuition for a fifth year for my master of engineering degree would have been very painful. I would also like to thank Ian Eslick and Jeremy Brown, other graduate students in the Reinventing Computing group, for their comments, feedback, and moral support.

Over the last five years, making time to go on community service projects with Alpha Phi Omega, be it painting at a homeless shelter or wielding power tools of destruction at a Boy Scout camp, contributed greatly to the retention of my sanity. It is healthy to be reminded that there are more important things out there in the real world than passing classes or finishing thesis.

The hundreds of hours sitting at my keyboard during the last several terms passed less painfully thanks to the music of the Chorallaries and Logarhythms (MIT a capella singing groups), whose albums took up semi-permanent residence in the CD-ROM drive on my workstation. Also, while it's not clear that this particular acknowledgement is in the form of a thank you, I should acknowledge the quality soft drink products of the Coca-Cola company, many scores of liters of which I consumed in front of said keyboard.

Last but not least, I would like to thank my parents, who came to a foreign country to give better opportunities to my sister and me; and my friends, who tolerably tolerated my complaints this past year about the job search process and thesis — Kate and Barbara especially come to mind.

Comments, corrections, and feedback are welcome via e-mail to pshuang@MIT.EDU.

Contents

1 Introduction **9**

1.1 Quasistatic Computing 9

1.2 Smart Compilers & Clever Compilers 11

2 Quasistatic Constructs **14**

2.1 Quasistatic Variables and `qif-qelse` Statements 15

2.2 Quasistatic Parameters and `qinfluence` Statements 19

2.3 Justification 22

3 Implementation **24**

3.1 Compiler Selection 24

3.2 Parsing Constructs 27

3.3 Construct Interactions 28

3.4 Profiling 30

3.5 Selecting Alternatives 38

3.6 System Integration 42

4 Examples **44**

4.1 `eqntott` (SPECint92) 44

4.1.1 Attacking `qsort()` 45

4.1.2 Attacking `cmppt()` 48

4.2 Matrix Manipulations 51

4.2.1 Special Forms 51

4.2.2 Blocking 53

5 Conclusions **59**

5.1 Related Work 60
5.2 Practicality Issues 61
5.3 Future Work 62
5.3.1 Bug Fixes 62
5.3.2 Enhancements 63
5.3.3 Additional Research 64

A Compiler Selection Details **65**
A.1 SUIF Output 65
A.2 SUIF Criteria Conformance 65

B Profiling Mechanisms **70**
B.1 prof — PC Sampling 70
B.2 gprof — Call Graph 70
B.3 pixie — Basic Block Counting 71
B.4 Real Time Stopwatches 71

C Further eqntott Transformations **73**

D Source Code **75**
D.1 integrate.cc 75
D.2 add_profiling.cc 82
D.3 select_alternatives.cc 87
D.4 Profiling Implementation 101
D.5 Auxillary routines 107
D.6 Diffs to snoot Pass 112

List of Figures

1-1	Automating the profiling feedback loop.	13
2-1	qif example: sorting.	16
2-2	qif example: graphics hardware assist.	18
2-3	qint example: matrix multiplication.	20
2-4	qint example (alternative syntax).	21
2-5	qint example: load splitting.	21
2-6	qint example (qinfluence syntax).	22
3-1	Clever compiler implementation — overview.	25
3-2	Modified snoot pass example input.	28
3-3	Quasistatic constructs rendered into SUIF by modified snoot.	29
3-4	integrate pass example input.	31
3-5	integrate pass finds qif chains and nested qinfluence statements.	32
3-6	Dynamic interactions between quasistatic variables and parameters.	32
3-7	Example: hello.c.	38
3-8	hello.c after being processed by integrate, add_profiling, and select_alternatives passes.	39
4-1	Comparison routine cmppt()'s inner loop.	49
4-2	A restructuring of the cmppt() inner loop.	49
4-3	Quasistatic version of cmppt() inner loop.	50
4-4	Selecting at run-time between all the special-case matrix routines.	53
4-5	Quasistatically selecting between special-case matrix routines.	54
4-6	Blocked matrix multiply implementation.	56
A-1	countdown.c, to be fed to SUIF.	66
A-2	Output of printsuif countdown.spd for abs() function.	67
A-3	Output of printsuif countdown.spd for main() function.	68
A-4	Partial output of s2c countdown.spd.	69
B-1	test-rdtsc.c: using the RDTSC time stamp instruction.	72

List of Tables

3.1 Comparison of candidate compilers. 26
3.2 Profiling overhead for `eqntott` run on the reference SPECint92 input file. . 34
3.3 Gathering precise profiling data using RDTSC instruction. 36

4.1 Merge sort and median-of-3 quicksort on arrays of integers. 46
4.2 Performance of different rearrangements of boolean terms. 50
4.3 Multiplying 512x512 matrices with different block sizes. 57
4.4 Multiplying 1280x1280 integer matrices with different block sizes. 58

Chapter 1

Introduction

In this chapter, I describe the framework for this thesis, and in particular, the concept of quasistatic computing as embodied by clever and smart compilers. In Chapter 2, I describe in detail quasistatic variables and parameters and the `qif-qelse` and `qinfluence` statements, syntactical extensions to the C language which the human programmer can use to give hints to the clever compiler. In Chapter 3, I describe my implementation of these constructs on top of the SUIF C compiler from Stanford University. In Chapter 4, I evaluate the effects that using these constructs have on the performance of various examples written from scratch and on an existing benchmark program with modifications. Finally, in Chapter 5, I make some conclusions, list related work, and outline possible future directions.

1.1 Quasistatic Computing

The Reinventing Computing group[1] in the MIT Artificial Intelligence Laboratory is working on technologies which take into account the fact that the computing landscape for the 21ST century will be radically different; the key theme is engineering software and hardware technologies that can adapt to inevitable and rapid changes in base hardware and software technologies and in patterns of application usage.

Some specific ideas currently being explored include:

- DPGA-coupled microprocessors: reconfigurable logic arrays integrated into the CPU serving as specializable computational resources. [D94a]

[1] Which evolved from the Transit Project: for more information, see the World Wide Web URL http://www.ai.mit.edu/project/transit/rc_home_page.html.

- Global Cooperative Computing: exploiting the National Information Infrastructure and the World Wide Web to change how software is developed, debugged, supported, optimized, and used. [DBE+94]
- Quasistatic computing. [DE94]

Quasistatic computing is a term which was coined to describe the effects of designing software systems to evolve themselves with minimal user/programmer intervention, in response to the rapid change of base hardware technologies over time, and to the wide variety (at any *given* time) of platforms with very different performance characteristics. In particular, fielded software systems would be able to evolve while in actual use, rather than remain rigid between manual upgrades. Quasistatic computing systems would track the shifts in patterns of software usage and the changes in availability of hardware and other available resources, and optimize software accordingly. Quasistatic computing may manifest itself in processor designs and machine architectures, operating systems, programming languages, and compilers/interpreters; the lattermost is the topic of this thesis.

Key to quasistatic computing is drastically modifying the current model of software delivery, in which a programmer (or team of programmers...) finishes writing a program, and then delivers the same unchanging executable to all the different end-users for consumption. Although this antiquated model is taken to an extreme with mass-market commodity programs (where fifty million users may be running the same executables), even with vertical-market or custom-written programs, there is usually still a notion that at some point there is a compilation[2] which freezes the software for delivery until the next bug fix or version upgrade. By modifying this traditional model so that the program is periodically and dynamically recompiled, a *smart compiler* can incorporate newly acquired knowledge (from any of a number of different sources) to improve program performance, and users see long-term incremental benefits for very little manual intervention effort. The term "quasistatic computing" itself emphasizes the fact that there is a blurring of the line between static (compile-time) computation and dynamic (run-time) computation.

[2]The term "compilation" is used loosely here; even if the software package is written in an interpreted language, there is still a point in time when conceptually the program is no longer viewed as source code but as an application to be used.

1.2 Smart Compilers & Clever Compilers

A traditional optimizing compiler relies entirely on static analysis of the program's source code. The kinds and scope of static analyses which can be practically undertaken have increased dramatically in the last decade. The more sophisticated understanding has made possible more extensive program transformations which have proven their worth in the increased performance of programs, e.g., when benchmarks and real world programs run noticeably faster on the same computing platform solely due to an improved release of a vendor's compiler. However, static analysis by definition can only guess at the performance characteristics of a program. Trying to ascertain the true run-time characteristics of a program would effectively require the compiler to emulate the target architecture and hardware/software run-time environment to simulate running the program — and despite vast improvements in emulation technology, emulation is still relatively impractical. Furthermore, there would still be the problem of not having the *typical* usage input data sets for the program available.

Does it matter that static analyses only give the compiler an incomplete understanding of the program? It can; certain kinds of speculative transformations are not performed or even considered either because they don't *always* improve performance and may even sometimes worsen performance, or because they may cost more in space overhead than can generally be justified — examples include inlining and loop unrolling, techniques which currently are *only* used by agressive optimizing compilers, but only in a very restricted fashion. However, with run-time profiling feedback, a *smart* compiler [BDE94] can try out these more speculative transformations and ascertain which appear to be beneficial in a *particular* instance. In short, a smart compiler can run experiments.

Ideally, a smart compiler would take normal source code, written by a programmer who gave no thought whatsoever to quasistatic computing issues, use profiling feedback to determine what parts of the program are "hot" and are therefore interesting loci, and automatically analyze those parts of the program for potential radical transformations. However, we need stepping stones on the way to that eventual goal: one such stepping stone which has been proposed is a *clever* compiler. A clever compiler relies on programmer annotations in the source code to help it decide how to organize and analyze profiling feedback gathered at program run-time; it is also up to the programmer to indicate potential

transformations by encoding alternative code sequences with the annotations. Thus, a clever compiler can still quasistatically choose between the programmer-indicated alternative code implementations, but falls short of a smart compiler in that it cannot create alternative implementations through its own semantic analysis of source code.

Although taking advantage of a clever compiler requires some additional programmer effort, it should still be much easier than the traditional manual performance optimization process,[3] and retains the quasistatic computing advantage that the program will dynamically evolve in response to changing usage patterns and changing computing platforms without continuing programmer effort.[4] Figure 1-1 illustrates the difference in the feedback loop between a programmer using a traditional compiler and a programmer using a smart compiler. Note that the programmer using the smart compiler is not involved in the iterative loop. Substituting a clever compiler for a smart compiler, the programmer would not be involved in the iterative loop, although there might be an extra step for the programmer to annotate the source code for the clever compiler to work with.

Figure 1-1 also shows that it is important that the profiling mechanisms used by quasistatic computing be very light-weight in nature, since the *end-user* is *always* using an instrumented executable. A number of different mechanisms to gather profiling data exist (e.g., `prof`, `gprof`, `pixie`), with overheads ranging from 3–5% to 40–200%. While a programmer who is explicitly running test cases is usually willing to tolerate that kind of slow-down, an end-user would be unhappy. Admittedly, after a period of time of sustained experimentation, the smart compiler can conclude it has likely converged on a near-optimal program, and reduce the frequency and level of instrumentation and experimentation, thus reducing overhead to near zero. However, it is still necessary to keep overhead low even during that experimental phase. Regardless of the particular mechanism used, a smart compiler should be able to substantially reduce the profiling overhead, since it is *only* interested in gathering performance statistics relevant to the particular experiments it is conducting at any given time, and not detailed statistics about all parts of the program.

[3] In which the programmer examines reams of profiler output to evaluate the resulting program's performance, formulates alternative code for "hot spots", and iterates, and iterates, and iterates....

[4] Another aspect of annotations to allow the programmer to explicitly encode different alternatives in the source code is that such annotations serve to document, in a structured fashion, what alternatives and trade-offs have *already* been considered by the programmer.

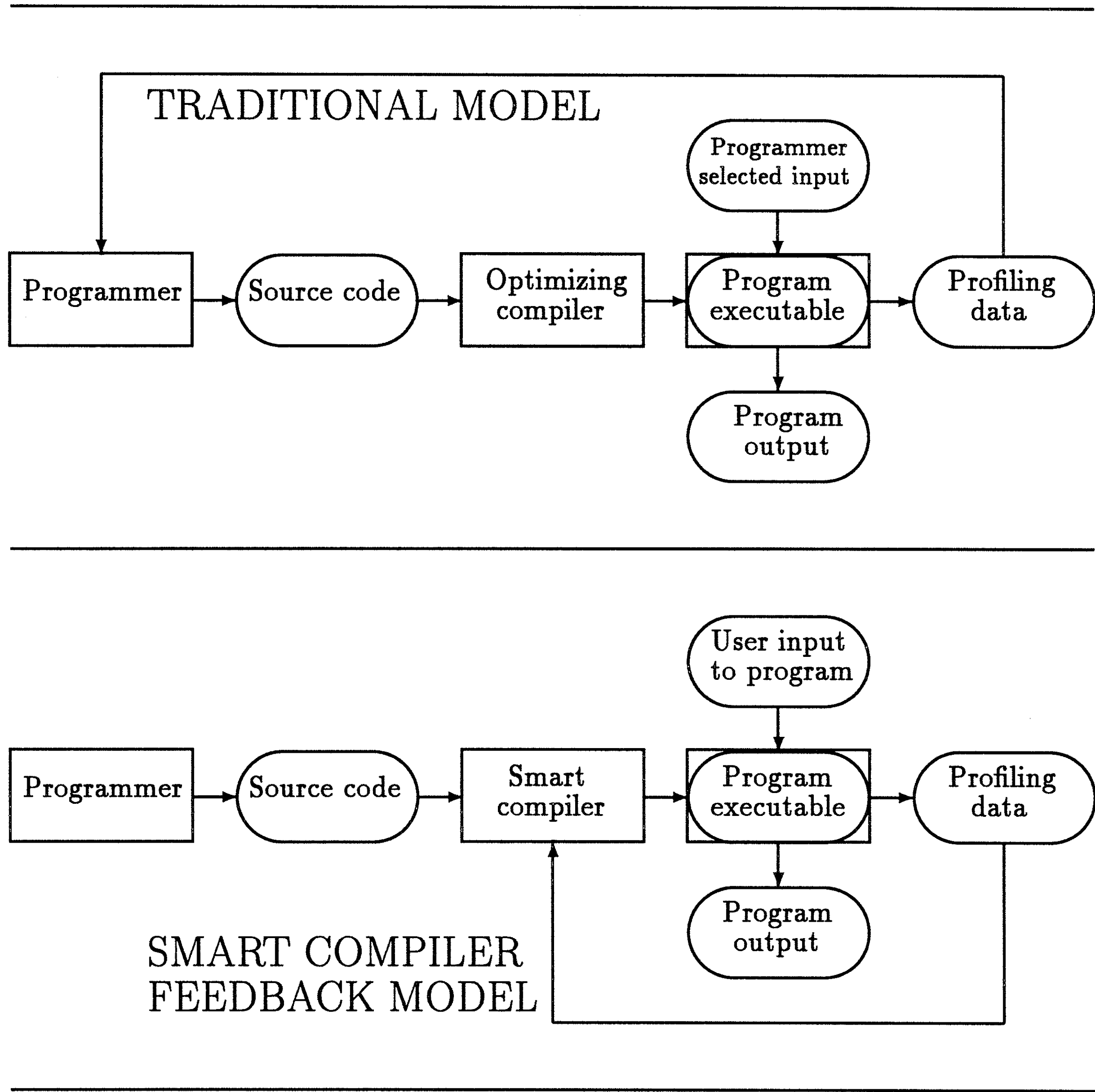

Figure 1-1: Automating the profiling feedback loop.

Chapter 2

Quasistatic Constructs

In the first two sections of this chapter, I define several syntactic constructs for annotating C source code to aid a clever compiler — namely, quasistatic variables and `qif-qelse` statements, and quasistatic parameters and `qinfluence` statements [D94b] — and give some examples to help describe them in detail. Then in the last section of this chapter, I will try to justify this particular approach. Note that I do not claim that these constructs are ideal, necessary, or even sufficient annotations for a clever compiler; however, there is great benefit in having something which can be implemented, tried, and discussed in concrete details rather than vague generalities.

Before proceeding, I give some general background premises.

- When the programmer uses annotations to indicate alternative code sequences in the source code, he or she is telling the clever compiler that there are alternative *versions* of the *program* which can be created by selecting different alternatives code sequences. Furthermore, the programmer is asserting that any of these versions of the program is functionally correct.

- Suppose in function `A()` the programmer indicates (the precise annotation mechanism is unimportant) a decision $D1$ is to be made by the clever compiler between code fragments `A1`, `A2`, and `A3`, and similarly in `B()` the programmer indicates an decision $D2$ between `B1`, `B2`, and `B3`, and in `C()` a decision $D3$ between `C1` and `C2`. If the decisions $D1$, $D2$, and $D3$ can be made independently of each other, then there are already at least 18 different versions of this program. This multiplicative effect means that the number of versions of a program is potentially explosive.

- It is useful to think of the different versions of the program as being located in a decision *space*. What constitutes dimensions of the space may vary; in the above example, one can consider the versions of the program to be lattice points in a three-dimensional space with the dimensions representing respectively the possible outcomes of $D1$, the possible outcomes of $D2$, and the possible outcomes of $D3$.

- Then the clever compiler's task can be formulated as searching the version space for the "best" version. A complete search is likely to be impractical because of the explosive number of versions — however, once cast in this light, standard space-searching techniques (e.g., hill-climbing, simulated annealing) could be applied to this problem, to find a "good" although possibly not "best" version.

- In the interests of making an effective clever compiler possible, annotations should searching the space easier. One way of doing this is to choose annotations that allow the programmer to express whether or not multiple decisions are likely to interact with each other in terms of program performance. This has the drawback that programmers are often wrong about how different parts of their programs interact with respect to performance. However, in the above example, if the programmer somehow indicate to the clever compiler his or her belief that all three decisions are likely to have independent effects on program performance, then a clever compiler can organize its search accordingly. For example, once the compiler has tried $D1$=`A1`, $D1$=`A2`, and $D1$=`A3` versions of the program (with $D2$=`B1` and $D3$=`C1` held constant), it might notice that the $D1$=`A2` version performed best out of those three, and then explore the $D1$=`A2` "plane" of the search space first.

2.1 Quasistatic Variables and `qif-qelse` Statements

quasistatic-variable-declaration:
    `qvar` *identifier*
qif-statement:
    `qif (` *quasistatic-variable* `)` *statement*
    qif-statement `qelse` *qif-statement*
    qif-statement `qelse` *statement*

```
qif(SPECIAL_SORT_INPUT) {
  FOOBARsort(array,N,....,compare_elements);
}
qelse {
  quicksort(array,N,....,compare_elements);
}
```

Figure 2-1: `qif` example: sorting.

A quasistatic `qif-qelse` statement (hereafter referred generically to as `qif`) is resolved at *compile-time* to one of its clauses. I.e., it differs from the C `if` statement in that no selection between the clauses is done at run-time; also, only a single quasistatic variable may appear as the test expression. It differs from the C preprocessor directives `#if defined(...)`[1], `#elseif defined(...)`, and `#endif`, which are sometimes used by programmers to delineate "alternative" code, in that it is the clever compiler which selects one of the `qif` clauses to execute at run-time; by contrast, preprocessor directives are explicitly resolved by the *programmer* when he or she indicates which symbols are to be considered defined by the compiler.

This new syntax allows the programmer to easily provide arbitrary alternative pieces of code, when he or she is uncertain which implementation will provide better performance. Below, I provide some examples to help to define the semantics of the construct; these examples also demonstrate the kinds of situations where the ability to easily specify alternative code might be useful.

Sorting

In Figure 2-1, the programmer thinks that the FOOBAR sort implementation *might* perform better than the Quicksort implementation, but isn't positive. That uncertainty translates into writing a `qif` statement.

Note although `SPECIAL_SORT_INPUT`, the name of the quasistatic variable used in the `qif` statement in Figure 2-1, does not bear any *semantic* meaning to the clever compiler,[2] the compiler *does* use it as an unique tag. At other places in the program where the programmer

[1] Or the less general but more common `ifdef` directive.

[2] Although the programmer certainly should choose a name which he or she finds semantically meaningful, for code readability.

wishes to indicates the choice between the two sorts, the programmer can either use the same tag to force the selection to be the same; or, the programmer can use a different tag[3] to allow the clever compiler to make the selection separately.

Separate optimization decisions might be useful, for example, if one call site is expected to sort nearly random order data, but another call site is expected sort nearly sorted data — the net effect in this example would be a variation of call site specialization. However, whether these *a priori* expectations were correct or not, the clever compiler can still make the right selection for each site so long as the decisions are decoupled at the two sites; thus, decoupled tags are generally preferable. So why should the programmer have the ability to force the clever compiler to couple decisions by using the same tag? A simple example where coupling is necessary is a function that needs to be initialized. Sorting routines generally don't need need initialization, but other kinds of routines may need initialization; if, however, the quasistatic decision is not to use that routine after all, then initialization should not be unnecessarily performed — it might be expensive in terms of computation time (e.g., set-up) or money (e.g., a third-party library if initialized might grab a floating license for the duration of program execution).

Coupling the clever compiler's decisions can be used for more than initialization, however. For example, the ability to force the compiler to couple decisions makes it possible for the programmer to provide different implementations of abstract data structures, but to be manipulated in-line at points of use in the program rather than strictly through an abstraction layer of library function calls. For example, an abstract set might be represented as either an unsorted linked list or as a b-tree (slower insertions, but faster deletions and lookups than the linked list); an abstract complex number might be represented either as in Cartesian form as a pair of real and imaginary numbers, or in polar form as radius and angle (slower additions and subtractions, but faster multiplications and divisions than the Cartesian form). (Note no single implementation of these or other abstract data types is likely to be the best-performing for all possible programs.) It is vital that the quasistatic decisions between alternative code fragments accessing the internals of different representations from different places in the program are coupled, i.e., made consistently, or else an

[3]For convenience, it may be desireable to introduce additional syntax, e.g. `unique:LARGE_SORT`, to allow the programmer to specify that different call sites should be analyzed separately, but still use the same identifier that has semantic meaning to a programmer reading the source code.

```
qif(FULL_HW_ASSIST) {
  communicate_display_list(....);
  /* Pass the display list to the accelerator and tell it to do all
     the rendering. */
}
qelse qif(PARTIAL_HW_ASSIST) {
  process_display_list(....);
  /* Iterate over the display list, decompose each graphics object
     into simpler commands, perform z-order clipping, and then hand the
     commands to the accelerator. */
}
qelse {
  render_display_list(....);
  copy_to_screen(....);
  /* Render the display list into a bitmap, and then send the result
     to the dumb graphics device. */
}
```

Figure 2-2: qif example: graphics hardware assist.

alternative code fragment might end up trying to manipulate data in a b-tree as a linked list, for example.[4]

Hardware Assist

As another example, in Figure 2-2, the programmer is uncertain sure how effective the available graphics accelerator will be at rendering this particular application's display list (e.g., CAD). Hence the programmer decides to let the clever compiler quasistatically choose between having the graphics accelerator do all, some, or none of the work.

The figure demonstrates how the programmer can specify more than two alternatives by adding a `qelse qif(QVAR)` clause. A final `qelse` clause which does not specify a quasistatic variable is optional; therefore, for any particular `qif` statement, no matter how long the

[4]See [S91], which presents Typesetter, a system specifically to select between different implementations of abstract data structures based on profiling feedback. Also, the documentation for `libg++`, a freely distributable C++ library provided by the Free Software Foundation for use with the GNU C++ compiler, includes some pragmatic discussion and statistics about the expected efficiencies of the various operations on different implementations of the abstract data types; also discussed is the improvement in performance when the programmer knows and indicates in the source code exactly which representation is being used for a particular variable of an abstract type at a given point in the code, so that the compiler can skip generating run-time checks to determine the actual representation and instead generate code to *immediately* dispatch to the appropriate implementation.

chain of clauses it contains, either exactly one (no final `qelse` clause) or zero (final `qelse` clause exists) of all the quasistatic variables used must resolve to true, so that in either case exactly one of the clauses in the statement will be selected.

In this example, the code alternatives trade off between doing work locally on the CPU and doing work on the graphics accelerator. It might be the case, for example, that on a workstation model with a slow CPU the `FULL_HW_ASSIST` alternative performs best, whereas on a workstation model with a fast CPU, it's faster to let the CPU take on a larger share of the computation; the sophistication of the graphics subsystem installed on any given machine is of course also a major factor in the quasistatic decision.

2.2 Quasistatic Parameters and `qinfluence` Statements

qint-parameter-definition:
    `qint [` *qint-range-list* `]` *identifier*
qint-range-list:
    qint-range-specifier
    qint-range-list `,` *qint-range-specifier*
qint-range-specifier:
    integer-constant
    integer-constant `:` *integer-constant*
qinfluence-statement:
    `qinfluence (` *qint-parameter* `)` *statement*

A quasistatic parameter (hereafter referred to as `qint`) is an constant integer used by the program in such a way that the initialization value of the constant integer may be selected from the any of the N values specified at the definition of the parameter, without changing the functional behavior of the program; it is similar to the notion of dynamically-valued constants discussed in [S95].

A `qinfluence` statement is used to indicate to the clever compiler what parts of the program should have their performance associated with a given `qint` parameter.

Again, I present examples to demonstrate the intended semantics of the construct.

Matrix Multiplication

The code in Figure 2-3 illustrates that when matrices to be multiplied are too large to work on at once, they can be divided up into sub-matrices and multiplied, and then the results

```
qint[1024,2048,4096] BLOCK_SIZE;
[....]
BlockedMultiply(MatrixA, MatrixB, ResultMatrix, BLOCK_SIZE);
```

Figure 2-3: qint example: matrix multiplication.

composed to get the final answer; the possible values for blocking size are 1024, 2048, and 4096 in this case. Figure 2-4 is semantically equivalent to Figure 2-3, and demonstrates the use of the ":" notation for specifying ranges of values. Exactly which blocking size results in the best performance depends heavily on both static machine characteristics[5] like cache size/associativity, and dynamic characteristics like cache competition introduced by other processes under multitasking operating systems.

The effect of using a qint parameter can be achieved by writing N pieces of alternative code (in each of which the quasistatic parameter symbol is substituted with a literal from the parameter's set of possible values) chained together in a qif statement; however, the net effect achieved is desired commonly enough to merit special treatment. Furthermore, encoding the alternatives using a qint parameter instead of N quasistatic variables in long qif statements potentially allows a clever compiler to more efficiently search the version space by guessing that to a first approximation, performance of a particular version of a program can be estimated by interpolation on the differences in qint parameter values between it and program versions which have already been tried. E.g., if the program versions BLOCK_SIZE = 1024 and BLOCK_SIZE = 4096 have already been tried, then a clever compiler might guess that *to a first approximation*, performance of the BLOCK_SIZE = 2048 version of the program falls somewhere between the performance of those two versions.

Load Balancing

As another example, in Figure 2-5, the programmer has access to remote computation services but is not certain whether their performance is good compared to just performing

[5]Static on any given machine, *sans* field upgrades, but widely varying on different implementations of a binary-compatible instruction set architecture.

```
qint[10:12] BLOCK_SIZE_SHIFT;
#define BLOCK_SIZE (1<<BLOCK_SIZE_SHIFT)
```

[....]

```
BlockedMultiply(MatrixA, MatrixB, ResultMatrix, BLOCK_SIZE);
```

Figure 2-4: qint example (alternative syntax).

```
{
  qint[1024,4096,16384,65536] USE_REMOTE_COMPUTATION_SERVER_LIMIT;
  qint[1:8] REMOTE_PIECES;
  qint[16,64,256] LOCAL_BLOCK_SIZE;

  if (size_of_data < USE_REMOTE_COMPUTATION_SERVER_LIMIT)
    /* Partition data into LOCAL_BLOCK_SIZE-sized blocks
       and process locally */
  else
    /* Partition data into REMOTE_PIECES number of pieces and
       send each piece to a remote computation servers for processing */
}
```

Figure 2-5: qint example: load splitting.

computation locally — the servers are likely to be shared with other users, for example, so if other users are often making heavy demands, the servers will be relatively unresponsive. Using `qint` parameters allows dynamically balancing between client and server processing.

The access scope of a `qint` parameter should be the same as that of a C integer variable definition appearing in the same textual position in the source code file. By default, if there are no `qinfluence` statements in the program which reference a `qint`, for profiling feedback purposes, the entire part of the program where the `qint` is lexically visible is assumed to be influenced by which particular value which the clever compiler assigns to the `qint`. Note that this default can be simultaneously both too broad and too narrow in scope. The value of a qint parameter may not affect program performance for *all* the code within its lexical scope, and including more code than necessary will tend to swamp out any performance

```
{
  qint[1024,4096,16384] USE_REMOTE_COMPUTATION_SERVER_LIMIT;
  qint[1:8] REMOTE_PIECES;
  qint[16,64,256] LOCAL_BLOCK_SIZE;

  if (size_of_data < USE_REMOTE_COMPUTATION_SERVER_LIMIT);
    qinfluence(LOCAL_BLOCK_SIZE) {
      /* Partition data into LOCAL_BLOCK_SIZE-sized blocks
         and process locally */
    }
  else
    qinfluence(REMOTE_PIECES) {
      /* Partition data into REMOTE_PIECES number of pieces and
         ship to a remote computation server to processing */
    }
}
```

Figure 2-6: qint example (qinfluence syntax).

effects of changing the value of the qint parameter; conversely, mutations allow the qint value to influence code performance *outside* its lexical scope.

Where the programmer believes this default would be substantially inaccurate, the qinfluence construct allows him or her to indicate explicitly what parts of the code he or she believes will have their performance influenced. Figure 2-6 demonstrates usage of the qinfluence construct; the programmer has provided more information to the compiler than in Figure 2-5. Note that no qinfluence(USE_REMOTE_COMPUTATION_SERVER_LIMIT) statement is necessary in this example since the entire lexical scope of that qint parameter is the correct scope for the clever compiler to be looking at profiling feedback.

2.3 Justification

C language. My decision to specify (and implement) C language extensions to serve as annotations for a clever compiler was driven primarily by the fact that C is widely used, hence developing a system which can improve performance for programs written in C will be pragmatically more useful than developing such a system for a language which is used for research only. Extending with a functional language with quasistatic constructs would have been simpler than extending C; however, only a small proportion of the total compute

cycles consumed each day are consumed by programs written in functional languages. In addition to being imperative, C was more difficult to work with because the language model of multiple source files compiled independently has traditionally made certain kinds of global program analysis more difficult.

Sufficiency. It is difficult to ascertain whether the constructs described in this chapter are sufficient for the programmer to adequately *communicate* to the clever compiler what the different versions of the program are. There certainly does not seem to be any hope of *proving* anything meaningful about whether they would be sufficient when what they might be sufficient *for* is not formally defined. Actual usage would be necessary so that programmers can provide feedback to the clever compiler writer as to when the existing constructs are inadequate, and what modifications or additional syntax would be necessary. Strict sufficiency aside, there's also the question of how convenient these constructs are to use — and not just by human programmers. Although not explored further in this thesis, human programmers are not necessarily the only "users" of these constructs. Yet another stepping stone between clever and smart compilers might be provided by additional independent compiler passes which generate alternatives based on *partial, limited* analysis and then used the kinds of constructs described in this chapter to express the results of the analysis; i.e., the writers of those kinds of passes would not need to worry about the details of selecting between the alternatives, but only need to know how to generate them. Such usage might generate different criteria for sufficiency or convenience.

Simplicity. Stroustrop claims that one of the criteria he used in evaluating whether particular proposals for extensions to the C++ language was whether they were simple enough that within 10 minutes of explanation to a group of programmers, the audience would understand the proposal well enough to be asking questions about how to use the extensions in particular scenarios with implications which neither Stroustrop nor the proposers had considered. Hopefully, these constructs fall into that category — simple enough to quickly understand, and yet sufficient to provoke interesting questions with implications about the clever compiler and quasistatic computing.

Chapter 3

Implementation

In this chapter, I discuss the selection of a suitable compiler "substrate" upon which to implement the `qif` and `qint` quasistatic constructs, what modifications were made to parse the constructs, the new pass to figure out the interactions between the constructs, how the program is instrumented, and searching the "space" of possible versions of a program. Finally, I discuss how the pieces fit together (see Figure 3-1).

3.1 Compiler Selection

It seemed clear from the outset that modifying an existing C compiler was the most sensible approach for implementing the `qif` and `qint` quasistatic constructs. Whereas many research languages have only one or two compiler implementations to choose from, there are a large number of C compilers which I might choose from. Criteria I was interested in included:

Source code available: Not only should I be able to get my hands on a copy of the source code to modify, but my modifications and the compiler source code should be readily available to those who are interested, without excessive legal restrictions.

Widespread use: If the compiler I modified was already used by a reasonably sized community, people could try out my modifications with relatively little installation effort.

Competitive performance: If the compiler I modify is known to generate output programs with very poor performance as compared to other compilers, then any improvements obtained from the addition of support for quasistatic constructs are suspect as being artifacts of a poor base.

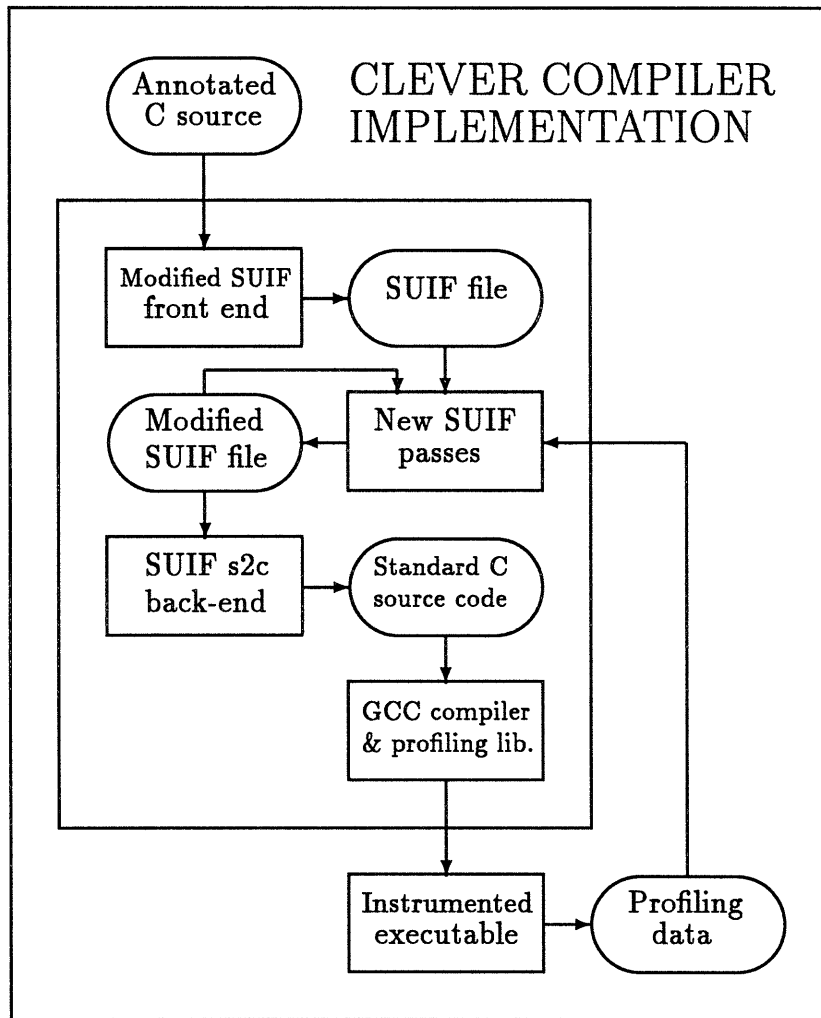

Figure 3-1: Clever compiler implementation — overview.

Compiler	Source code	Wide Use	Competitive	Cross-platform	Easy mod.
`lcc`	Yes	No	(A)	Yes	Yes
`gcc`	Yes	Yes	Yes	Yes	No
SUIF	Yes	No	(A)	(B)	Yes
Vendor	No	Yes	Yes	No	No (C)
C-to-C	Yes	No	N/A	(B)	Yes

(A) Claims are that output programs' performance are approximately 70–90% that of the best available compilers.
(B) SUIF only has a back-end code generator for MIPS; however, it and C-to-C qualify as cross-platform since they can be used as source-to-source compilers.
(C) Some vendor compilers may actually be easy to modify; my *speculation* is that most of them, for performance sake, are not architected to be easily modifiable (e.g., separate passes).

Table 3.1: Comparison of candidate compilers.

Cross-platform: Profiling at run-time lends itself to the possibility of differential optimization on different platforms, hence it would be desireable for the compiler to have been ported to different platforms.

Ease of modification: If the compiler is structured into easy to understand passes, the task of modifying it to support quasistatic constructs would be much simpler than trying to modify a monolithic program.

The compilers which were seriously evaluated were: `lcc`, a ANSI C compiler by Fraser & Hanson at AT&T Bell Laboratories and Princeton University [FH94]; `gcc`, the GNU C compiler from the Free Software Foundation [S94]; and SUIF, the Stanford University Intermediate Format C compiler [WFW+94]. Candidates which were not seriously evaluated included vendor compilers and the C-to-C Translator package.[1] No vendor compilers were investigated more carefully because it seemed from the start an unpromising path to pursue; the latter was not evaluated because it is relatively new; also, strictly speaking, C-to-C is in a different category in that it is *only* intended for use as a source-to-source compiler. Nevertheless, these candidates are included for comparisons purposes in Table 3.1.

Ultimately, I decided to modify the SUIF compiler. See Appendix A for how well SUIF conforms to my criteria. In practice, SUIF has proven to be a good choice. The learning curve on using SUIF's kernel library routines to manipulate the intermediate format was high; however, learning to use them was certainly less effort than writing abstractions to operate on C abstract syntax trees myself.

---

[1]Derived from the MIT LCS `c-parser`, available as ftp://theory.lcs.mit.edu/pub/c2c/c2c-0-7-3.tar.Z.

3.2 Parsing Constructs

SUIF's C front end is based on the `lcc` front-end, heavily modified to produce SUIF output. The first pass, `snoot`, performs initial parsing and lexing and outputs a SUIF file with possibly non-conformant data structures. A second pass, `porky`, eliminates such artifacts, rendering the output of `snoot` into standard SUIF.

Modifying `snoot` to accept the new keywords and constructs was fairly straight-forward, since the syntax for these constructs had been deliberately chosen to resemble existing C syntax forms. Specifically:

`qvar` (At the current time, `qvar` variable definitions are not implemented. Instead, upon first use of a quasistatic variable in a `qif` statement, the symbol name is looked up and if it is not found, is then entered into the symbol table as a constant integer annotated[2] as a quasistatic variable. This should be easily fixable.)

`qif` **and** `qelse` statements are parsed as if the tokens encountered were really `if` and `else` respectively. Annotations are used to flag the resulting intermediate format nodes as quasistatic constructs.

`qint` parameters are entered as constant integers into the global symbol table (as a declaration) and the file symbol table (as a definition); annotations are used to flag the global declarations as quasistatic parameters, and to store the range of possible values.

`qinfluence` statements are parsed as if they were simply a scope-introducing block, and a dummy variable is inserted into the symbol table of the new scope to ensure that the new scope will not be prematurely optimized away. An annotation is used to flag the block as having originated from a `qinfluence` construct.

Figures 3-2 and 3-3 illustrate what the modified `snoot` transforms quasistatic constructs into — for brevity of presentation, I have approximated the SUIF-format annotations in the `s2c` output by using C-style comments. By making both quasistatic variables and quasistatic parameters explicitly `const`, the programmer cannot accidentally mutate their values. Although it would be desirable to prevent the programmer from referencing a quasistatic variable as an integer value, I saw no easy way to prevent such mis-usage. Note

[2] SUIF supports attaching annotations containing arbitrary data to the structures which represent various parts of the parsed C program; annotations can be attached not just to nodes in the abstract syntax trees (representing the statements and expressions), but also to declarations and definitions in symbol tables.

```
qint[10,20,30] EXIT_CODE_BASE;

void foo(void) {
  qinfluence(EXIT_CODE_BASE) {
    qif(FIRST_ALTERNATIVE)
      { exit(1 + EXIT_CODE_BASE); }
    qelse qif(SECOND_ALTERNATIVE)
      { exit(2 + EXIT_CODE_BASE); }
    qelse
      { exit(3 + EXIT_CODE_BASE); }
  }
}
```

Figure 3-2: Modified `snoot` pass example input.

also how a `qif-qelse` chain is parsed into nested `if` statements in Figure 3-3. This would be a matter of concern for run-time efficiency except that `gcc` with the `-O` option is known to do constant-propagation and dead-code-elimination, and does disambiguating such nested `if` statements to a single clause, leaving no residual run-time selection overhead.

3.3 Construct Interactions

The motivation for determining what variables and parameters appear to be dependent or independent of each other is discussed in Section 3.5. The goal of the new `integrate` pass, run after the `snoot` and `porky` passes, is to make this determination possible by gathering more global-level information about what sets of quasistatic variables are used together in the same `qif` statement, and what quasistatic parameters interact with each other due to nested `qinfluence` statements. Figures 3-4 shows a partial program; Figure 3-5 demonstrate what information `integrate` gathers by walking the abstract syntax trees looking for `qif` and `qinfluence` statements; the gathered information is stored in new annotations in the SUIF file. (Each different set is also an assigned integer unique identifier stored in its annotation, for later use.)

Specifically, a `QIF SET` annotation indicates that some part of the program has alternative code tagged by the given list of quasistatic variables. The indicator as to whether there is or is not a final `qelse` clause is important: if there is no final `qelse` in a given set,

[....]

```
extern const int EXIT_CODE_BASE /* ["QINT PARAMETER"] */;
extern void foo(void);
extern const int FIRST_ALTERNATIVE /* ["QIF VARIABLE"] */;
extern int exit();
extern const int SECOND_ALTERNATIVE /* ["QIF VARIABLE"] */;

const int EXIT_CODE_BASE;

extern void foo(void)
  {
    {
      const int QINFLUENCE_BLOCK_HOLDER;

      if /* ["QIF STATEMENT"] */ (FIRST_ALTERNATIVE != 0)
        {
          exit(1 + EXIT_CODE_BASE);
        }
      else
        {
          if /* ["QIF STATEMENT"] */ (SECOND_ALTERNATIVE != 0)
            {
              exit(2 + EXIT_CODE_BASE);
            }
          else
            {
              exit(3 + EXIT_CODE_BASE);
            }
        }
    } /* ["QINFLUENCE STATEMENT"] */
    return;
  }
```

Figure 3-3: Quasistatic constructs rendered into SUIF by modified snoot.

then that introduces the constraint that exactly one of the quasistatic variables must be selected to generate a valid version of the program; if there *is* a final `qelse`, then exactly zero *or* one of the variables must be selected. In this example, exactly one of the quasistatic variables `ALTERNATIVE_1` or `ALTERNATIVE_2` must be selected. It is possible to write source code such that it is impossible to generate a version of the program that satisfies all such constraints; a later clever compiler pass will complain in such a case.

A `QINFLUENCE SET` annotation indicates that there is some part of the program whose performance ought to be associated with the quasistatic parameters listed. Note that the `qinfluence(THREADS)` statement did not introduce a `QINFLUENCE SET` annotation listing just the `THREADS` parameter; instead, since the statement is lexically (textually) contained by a `QINFLUENCE(BLOCK_SIZE)` statement, a `QINFLUENCE SET` annotation listing both `THREADS` and `BLOCK_SIZE` was created instead.

The `integrate` pass at the current time does not record explicit information about other lexical nestings — a `qinfluence` block inside a `qif` statement clause, or a `qif` statement inside a `qinfluence` block or another `qif` statement clause. Nor does the current `integrate` attempt to derive information about possible dynamic nesting — cases where quasistatic variables and parameters may interact to influence the performance of the program through non-lexically-apparent interactions. For example, in Figure 3-6, the performance of the program is likely to depend on the interaction between the quasistatic variables `FOO` and `BAR` and the quasistatic parameter `QUX`, but the fact that `FOO` and `BAR` interact with `QUX` at run-time is not readily apparent through lexical analysis.

3.4 Profiling

> The '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. ...[I]f it is to be a frequently used program the high counts in its profile often suggest basic improvements which can be made. An optimizing compiler can also make very effective use of the profile, since it often suffices to do time-consuming optimization on only one-tenth or one-twentieth of a program.
>
> Knuth, 1973 [K73]

```
int foo(void)
{
  int bar = 0;
  qif(ALTERNATIVE_1)
    { bar = 1; }
  qelse qif(ALTERNATIVE_2)
    { bar = 2; }

  qif(ALTERNATIVE_1)
    { bar += foo(); }
  qelse
    { bar -= foo(); }

  return bar;
}

void partition_work(int);
void dispatch_work_to_new_thread(void);

qint[1024,4096,16384] BLOCK_SIZE;
qint[1:8] THREADS;

void process(void)
{
  qinfluence(BLOCK_SIZE)
    {
      partition_work(BLOCK_SIZE);
      qinfluence(THREADS)
        {
          int index;
          for (index=0; index<THREADS; index++)
            create_new_thread();
          wait_for_threads_finish();
        }
    }
}
```

Figure 3-4: integrate pass example input.

[....]

```
    Global symbol table: 'globals'
```

[....]

```
      Annotations:
        ["QINFLUENCE SET": 0 "BLOCK_SIZE"]
        ["QINFLUENCE SET": 1 "THREADS" "BLOCK_SIZE"]
        ["QIF SET": 2 "ALTERNATIVE_1" "... FINAL QELSE"]
        ["QIF SET": 3 "ALTERNATIVE_2" "ALTERNATIVE_1" "... NO FINAL QELSE"]
```

[....]

Figure 3-5: **integrate** pass finds **qif** chains and nested **qinfluence** statements.

```
qint[1:8] QUX;

void A(void)
{
  qif(FOO)
    { B(); }
  qelse qif(BAR)
    { C(); }
}

void B(void)
{
  qinfluence(QUX)
    { /* .... */ }
}
```

Figure 3-6: Dynamic interactions between quasistatic variables and parameters.

Overhead for Existing Mechanisms

Minimizing the amount of overhead imposed by profiling is critically important for the clever compiler, since in the quasistatic computing model, the end-user (and not just the programmer) will always be subjected to that overhead.

How to instrument a program to extract performance information is a major topic in its own right; I briefly describe the commonly available profiling mechanisms available to C programmers on UNIX platforms[3] — `prof`, `gprof`, and `pixie` — in Appendix B. The different techniques have their pros and cons, but generally speaking, there's a tradeoff between the amount and accuracy of information gathered, and the overhead incurred at program run-time. The percentages shown in Table 3.2 are in accord with the "common wisdom" — 5–10% for `prof`, 10–15% for `gprof`, and 100–200% for `pixie`.

Note that the tremendously outsized amount of overhead for `pixie` is due to the fact that instrumenting every single basic block gathers *far more* information than is necessary most of the time. One way to keep overhead reasonable is to gather exactly what is needed to perform the desired performance analysis and no more; by building the static call graph beforehand, one can reduce the amount of instrumentation added to the program, and still gather a *basis* set of data that can be used to generate a full weighted call graph. [S91, G95] However, even the overhead required to gather such a basis set can still be improved upon: although these general-purpose tools cannot *a priori* predict what information the human programmer will be interested in, most of the time the human programmer doesn't really need to see the *entire* weighted call graph.... However, an user interface that allows the programmer to specify what profiling information is actually desired, thus potentially reducing the run-time overhead, has not been considered or implemented, to my knowledge; not surprising, since under the current paradigm for how these tools are used, even a several-fold performance hit may be acceptable if profiling is only turned on briefly to run the program against some small but hopefully representative sample of program inputs.

Real-Time Stopwatches

Existing profiling mechanisms have several drawbacks when viewed in the light of their use by a smart compiler in a quasistatic computing environment. To minimize run-time

[3]Profilers for MS-DOS and Windows platforms are likely to be using similar techniques.

Profiling type	User time (sec.)	System time (sec.)	Added overhead (%)
DECstation/Ultrix (A)	74.1	0.8	**0.0%**
prof (gcc -p)	83.1	0.7	11.9%
pixie	197.7	1.5	165.0%
Pentium/Linux (B)	19.9	0.3	**0.0%**
gprof (gcc -pg)	22.6	0.3	13.3%
SPARC/SunOS (C)	21.2	0.3	**0.0%**
prof (gcc -p)	22.2	0.2	4.2%
gprof (gcc -pg)	23.7	0.3	11.6%

(A) DECstation 5000/25: MIPS 25Mhz R3000 with 24MB memory running Ultrix 4.3; GNU gcc compiler version 2.5.90. Running in multi-user mode, but with only one user. On this platform, gprof profiling was not available since the vendor supports pixie, which provides a superset of the functionality of gprof.
(B) Dell XPS P90: see 44 for system configuration. On this platform, requesting prof profiling ended up performing gprof profiling; hence numbers are only reported for the latter style. pixie profiling was not available.
(C) Sun SPARCstation 20: TIS390Z50 CPU with 64MB memory running SunOS 4.1.3; GNU gcc compiler version 2.5.7. Running in multi-user mode, but with only one user. pixie profiling was not available.

Table 3.2: Profiling overhead for `eqntott` run on the reference SPECint92 input file.

overhead, the profiling mechanism should permit very selective instrumentation, at exactly the points of interest. None of the mechanisms mentioned up to this point make *selective* instrumentation easy.

In addition, the mentioned mechanisms all attempt to measure or estimate CPU time spent executing the program; however, what the typical user today really cares about is clock time, not CPU time (the days of being charged by how many seconds of CPU time are used being long past for most users). For example, in the load-balancing example on page 20, the goal of the clever compiler should not be to choose the balancing which minimizes the amount of *local* computation done, but rather the balancing which minimizes the time that the user has to wait for the results of the computation; otherwise, the quasistatic decision might end up always pushing all the work to the remote servers, even though this might make the user wait longer on average for the results of their computation.

The UNIX system call `gettimeofday()` could in theory provide the current clock time for profiling purposes; however, system calls are relatively expensive,[4] and hence requiring that added instrumentation code perform a system call implies potentially a substantial

[4]For example, actual timings of 1,000,000 iterations (normalized for the loop iteration overhead) indicate 282 elapsed CPU clock cycles per call to `time(NULL)`; 876 cycles per call to `gettimeofday(&tv,NULL)`; and 766 cycles per call to `getrusage(RUSAGE_SELF, &ru)`. Much of this is probably due to context switches; by contrast, it costs 9 clock cycles per call to a `dummy()` function that simply returns its input parameter — and viewing the assembly code output verifies that gcc has not merely silently inlined the function call.

overhead. Furthermore, frequently the granularity of the time value received back is inherently coarse due to the representation type of the time value, and is not guaranteed to be accurate even to the limit of the representation's resolution.

However, a number of newer processor architectures make the real time available to programs very inexpensively. For example, the new Intel x86 RDTSC instruction[5], very briefly mentioned on page 10-8 and A-7 in the Pentium architecture manual [I94], and more throughly discussed in [M94], is intended to provide a guaranteed monotonically increasing timestamp, but happens to be implemented on the Pentium as a count of the number of clock cycles elapsed since power up. Similarly, SPARC Technology Business indicates in a white paper[6] that the UltraSPARC processor will have a similar feature:

> ...a TICK register has been added, which is incremented once per machine cycle. This register can be read by a user's application to make simple and accurate measurements of program performance."

However, using such timestamps to implement real-time stopwatches to profile program performance also has at least several major flaws. First, it introduces noise which would not be visible for profiling mechanisms attempting to measure CPU time spent executing the program. On a workstation running a multi-tasking operating system and possibly providing services to other machines on the network, the clock time a local-CPU-bound program takes to run to completion with identical inputs can easily fluctuate wildly, depending on other demands are being placed on the workstation. Fluctuations of an order of magnitude is not unheard of under generally heavy load conditions. Possible heuristics to deal with this and other sources of noise will be discussed later. Second, such timestamps are not yet available on most processors in use today, hence real-time profiling is not very portable.

Actual Instrumentation

Table 3.3 demonstrate the kind of information which can be garnered using the RDTSC instruction on a Pentium (see also the source code in Figure B-1 on page 72). Note that the real-time aspect means that the cache effects of calling a function for the first time as compared to later calls is easily visible; also, note that the overhead of the profiling code

[5] Actual timings (normalized for the loop iteration overhead) of 1,000,000 iterations of the RDTSC instruction indicates it executes in 11 CPU clock cycles on the Pentium implementation.

[6] http://www.sun.com/stb/Processors/UltraSPARC/WhitePapers/Performance/Performance.html.

Description	Cycles
1 call to trivial function	55
1 "call" to inline'ed function	0
1,000 calls to trivial function	13,000
1,000 "calls" to inline'ed function	0
Start, stop, and update stopwatch counter	57

Results have been normalized to take into account that attempting to profile a null code sequence resulted in a count of 13 cycles, due to the exact placement of the actual RDTSC instruction in the profiling prologue and epilogue.

Table 3.3: Gathering precise profiling data using RDTSC instruction.

itself is more than an order of magnitude less than making system calls to get the current time, thus making this technique more practical for instrumenting code than using system calls to obtain the real time. Figure B-1 also uses three very useful features of the GNU gcc compiler being used as the back-end for my modified compiler:

- gcc supports extensions that allow the use of instruction-set-specific features by including inlined assembly language statements in the C source code; also, the exact mapping from C `register` variables to the machine's registers can be specified, making for easy manipulation of the timestamp value returned by the RDTSC instruction in Pentium registers EAX and EDX.

- gcc supports open-coded arithmetic operations on 64-bit integers even on 32-bit processors; this allows me to count cycles without worrying about overflow (a 64-bit counter which counts cycles ticking at 100 Mhz, for example, will not overflow until the elapsed time approaches 5,844 years).

- gcc supports an `inline` keyword which allows me to explicitly request that it inline particular functions.

Because of these gcc features, I could write functions to start, stop, and update a stopwatch counter mostly in C. To have a particular region of code profiled, I merely had to (write SUIF manipulation code to) bracket that region in the SUIF file with "calls" to `qp_start()` (start a stopwatch counter) and `qp_stop()` (stop and update a stopwatch counter). The output of s2c would then contain those function calls. However, when I then compiled the output of s2c with gcc after including the `qprofiling.h` header file with the `inline`-marked definitions for `qp_start()` and `qp_stop()`, gcc would take care of

inlining the bodies of the functions wherever they were called. (If inlining were not done, the overhead of profiling a region of code would increase by 26 cycles, 13 cycles for each function call.) This was *much* simpler than having to bracket the region to be profiled with the bodies of `qp_start()` and `qp_stop()` directly in the SUIF file, and made debugging the contents of `qp_start()` and `qp_stop()` relatively painless.[7]

Writing the `add_profiling` compiler pass was not difficult, given `qp_start()` and `qp_stop()`. The pass walks the AST for each function, searching for annotations which indicate a `qif` or `qinfluence` statement, and then brackets the appropriate clauses with the profiling "calls" — the integer UID created by the `integrate` pass for each set of interacting constructs is used in the calls to designate which stopwatch counter should be started and stopped. The `add_profiling` pass also modifies the `main()` function in the SUIF file to call the initialization routine `qp_initialize()` and the termination routine `qp_dump()`; the latter saved profiling information to a file in raw form (to be processed into meaningful format later), and is also responsible for re-invoking the clever compiler when enough profiling information has been accumulated.

One complication only discovered during implementation was non-linear flow control; for example, a `qinfluence` statement's block may contain `longjmp()` calls, `exit()` calls, `goto`, `continue`, `break`, or `return` statements, any of which could cause the `qp_stop()` call matching the entrance `qp_start()` call to not be executed. Calls to `exit()` were particularly problematic, since they would cause the `qp_dump()` routine not to be executed, and no profiling information from the run would be saved.[8] The current `add_profiling` pass has special handling for only one of these non-linear flow control cases, the `return` statement, which I expected to be the most commonly encountered. In principle, the others can be dealt with similarly. A `return` statement which appears inside a `qif` clause or `qinfluence` block of the form

`return` *expression* ;

is transformed into

[7]However, see Table 4.2 on page 50 and related discussion in text for further details on the performance of this implementation.

[8]Although not implemented, it has come to my attention that a number of C compilers provide a library routine `on_exit()` which allows the registration of routines to be run when `exit()` is called. That would have dethorned this particular problem, but the more general problem still remains.

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

qint[1:32000] WAIT_LENGTH;

int main(int argc, char *argv[]) {
  printf("Hello, ");
  qinfluence(WAIT_LENGTH) {
    qif(DO_WAIT) {
      sleep(WAIT_LENGTH);
      printf(", world. (Yawn.)\n");
      return WAIT_LENGTH;
    }
    qelse qif(DONT_WAIT) {
      printf("world.\n");
    }
  }
  qif(DO_WAIT) {
    printf("Did you think I fell asleep?\n");
  }
  qelse qif(DONT_WAIT) {
    printf("See, I'm wide awake!\n");
  }
}
```

Figure 3-7: Example: `hello.c`.

```
suif_tmp0 = expression ;
qp_stop( UID );
return suif_tmp0;
```

The temporary is introduced so that if *expression* would take a long time to evaluate, the time spent would continue to be stopwatched. Any `return` statements in the `main()` function are similarly transformed to ensure that `qp_dump()` is called before program termination. See Figures 3-7 and 3-8.

3.5 Selecting Alternatives

The `select_alternatives` pass performs several tasks: it checks for existence of a raw profiling data file, and if found, processes the file; it decides what experiment to conduct

```
[....]
const int WAIT_LENGTH = 1;
const int DO_WAIT;
const int DONT_WAIT = 1;
[....]
extern int main(int argc, char **argv)
  {
    int suif_tmp0;
    int suif_tmp1;

    qp_initialize("hello.prf", 2, 0);
    printf("Hello, ");
      {
        const int QINFLUENCE_BLOCK_HOLDER;
        qp_start(0, 0);
        if (DO_WAIT != 0)
          {
            qp_start(1, 0);
            sleep(WAIT_LENGTH);
            printf(", world. (Yawn.)\n");
            suif_tmp0 = WAIT_LENGTH;
            qp_stop(0, 0);
            qp_stop(1, 0);
            qp_dump(0);
            return suif_tmp0;
            qp_stop(1, 0);
          }
        else
          {
            if (DONT_WAIT != 0)
              {
                qp_start(1, 0);
                printf("world.\n");
                qp_stop(1, 0);
              }
          }
        qp_stop(0, 0);
      }
    if (DO_WAIT != 0)
      {
        qp_start(1, 0);
        printf("Did you think I fell asleep?\n");
        qp_stop(1, 0);
      }
    else
      {
        if (DONT_WAIT != 0)
          {
            qp_start(1, 0);
            printf("See, I'm wide awake!\n");
            qp_stop(1, 0);
          }
      }
    suif_tmp1 = 0;
    qp_dump(0);
    return suif_tmp1;
    qp_dump(0);
  }
```

Figure 3-8: `hello.c` after being processed by `integrate`, `add_profiling`, and `select_alternatives` passes.

next, or if it is ready to generate a non-experimental version; finally, it creates the actual executable corresponding to its decision.

Experimentation Frequency There are many sources of noise in the profiling data, including short-term variations in usage pattern and temporary variations in machine load (creating noise for the real time, but not CPU time, profiling mechanism). Summing data over repeated program runs can help filter out much of the noise; hence, the current implementation of the clever compiler generates executables which after four runs which successfully result in profiling data saved, will re-invoke the compiler. Four is merely an arbitrarily chosen default, and can be overridden by the `RUNS` environment variable.

It would be preferable to have the default number of runs before recompilation be adaptive — for example, if `select_alternatives` notices that the profiled parts of the program collectively took less than a second of real time to execute in the last four runs, then it could adjust the number of runs upward to reduce the frequency of recompilation, since there is not much benefit to trying to optimize this program, and recompiling less often will help amortize the cost of recompilation. Another factor in how many runs should occur before recompilation is whether the clever compiler is still in an experimental phase, when most of the program version space still unexplored, or whether it has reached an equilibrium phase, when much of the space *has* been adequately explored and the clever compiler therefore has a good idea what version is optimal for current usage. Even in the equilibrium phase, the clever compiler should still monitor program performance, but recompilations can occur at lengthier intervals.

Space Searching Given the formulation of the problem faced by the clever compiler on page 15 as searching a space of program versions, any of a large number of generic techniques for space-searching and function minimization can be applied. Because of time considerations, I chose to implement the most native possible procedure, a combination of the generate-and-test principle and the British Museum procedure: namely, `select_alternatives` generates all possible program versions and tries them all in the generated order. The current implementation of even this procedure is incomplete:[9] to complete this procedure, it would just need to automatically choose the program version which generated

[9] I decided that having some examples written up would be more interesting than having a more sophisticated search algorithm implemented.

the shortest run time, whereas currently it emits a warning to the user when it runs out of program versions to try.

Note that some function minimization techniques such as hill-climbing (alternately, gradient descent) explicitly require functions of nearly continuously variable parameters which have calculable derivative or gradient functions. However, other techniques such as downhill simplex method and Powell's method (direction-set methods) [PFTV88] do not require the derivative or gradient. Still other techniques, like simulated annealing, are designed to solve combinatorial problems, where the dimensions of the space are discrete rather than continuous, and it is explicitly recognized that in such a configuration space, the notion of "continuing downhill in a favorable direction" *may* be meaningless. Thus the latter set of techniques seem to offer a better fit.

Given that the clever compiler is searching in a very specific kind of domain, additional heuristics may be applied to improve the efficiency and the effectiveness of a combinatorial space search. For example, non-interacting sets can be modeled thusly: if a represents the first dimension (a set of 10 quasistatic variables which are always used together in the same `qif` statements), and b represents the second dimension (a quasistatic parameter which can take on any of 4 different values), and c represents the third dimension (a quasistatic parameter which can take on any of 20 different values), then we're seeking to minimize $t(a, b, c)$ in a space of 800 possible program versions. However, if a's effect on the performance of the program and b's appear to be dependent on each other because there is a `qinfluence` statement for b which encloses one of the a `qif` statements, but c's appears to be "completely independent", then we can rewrite what we are trying to minimize as $t(a, b, c) = f(a, b) + g(c)$, in which case $f(a, b)$ and $g(c)$ can be independently minimized, effectively collapsing the search space. Also, as a first approximation, the clever compiler's search engine might assume that $t(a1, b1, c1) < t(a1, b1, c2) < t(a1, b1, c3)$ if $c1$ and $c3$ are relatively close to each other in value and it is known that $t(a1, b1, c1) < t(a1, b1, c3)$ — this is a weak belief that $t(a, b, c)$ is close to continuous when moving onlyu along the dimension corresponding to a quasistatic parameter. Finally, if some profiling indicates that the parts of the program for which a and b influence the performance take up a negligible percentage of the total run-time, then the search engine can to a first approximation ignore the a and b dimensions altogether.

Decision Implementation A decision determines exactly one version of the program, and consists of the quasistatic variables which have been selected for, all quasistatic parameters, each with its selected value. Implementing a decision merely requires creating definitions in the SUIF file with the appropriate initializing value for each quasistatic variable (assigned the integer value 1 if selected for, 0 if not) and for each quasistatic parameter. Then the `create_executable` script is invoked, and it calls the SUIF `s2c` pass and then `gcc` to compile and link the program against the profiling library.

3.6 System Integration

Suppose Alyssa P. Hacker, programmer, wanted to take advantage of the clever compiler implementation described in this chapter, for a program with a single source file, `prog.c`. Here's what Alyssa had to do:

1. Pore over the program and restructure it to take advantage of quasistatic variables and parameters and quasistatic `qif-qelse` and `qinfluence` statements.
2. Add the statement `#include "qprofiling.h"` to `prog.c` before the use of any quasistatic constructs.
3. Invokes the `make` utility on the target `prog.exe`, pointing it at an appropriate makefile which knows to run the following commands:
   - `scc -.i prog.c` (invokes the C preprocessor)
   - `snoot prog.i prog.snt`
   - `porky prog.snt prog.spd`
   - `integrate prog.spd prog.int`
   - `add_profiling prog.int prog.prf`
   - `select_alternatives prog.prf _some_tmp_file_`
   - `create_executable prog.prf`
4. Hand off the executable `prog.prf.exe` created by the `create_executable` pass to the end-user, Ben Bitdiddle.

   [At this point, Alyssa decides to take some of her well-deserved vacation time and leaves for the hackers conference conveniently sited in Tahiti.]

Ben runs the program frequently, and profiling data is stored in `prog.prf.qprofiling_data`. Before long, as the program `prog.prf.exe` is exiting and storing profiling data from the most current run, it decides it has accumulated profiling data from enough runs and re-invokes the `select_alternatives` pass without requiring Ben's intervention; `select_alternatives` incorporates the data in `prog.prf.qprofiling_data` into annotations in

prof.prf. Several weeks (and many iterations of this process) later, when Alyssa returns, the program is running much faster for Ben, but nevertheless Ben clamours at Alyssa, dissatisfied because he wants another half dozen new features. Back to work for Alyssa — a programmer's work is never done.

Chapter 4

Examples

Note: unless otherwise specified, all performance numbers in this chapter are for programs compiled with the GNU `gcc` compiler version 2.5.8 using optimization option `-O`, on a Dell Model XPS90 with an Intel 90Mhz Pentium processor, 256KB 4-way associative L2 cache memory, and 16MB of RAM. This machine was running the Slackware distribution of Linux (kernel version 1.1.59), a freely distributable UNIX-compatible operating system. Where file I/O was performed, local disk storage was used; where the programs being run were memory-intensive, all unnecessary processes (including X Windows) were terminated to make the amount of memory free (approximately 12MB, as reported by the Linux `free` utility) reproducible. No other users were logged in consuming machine resources; the machine does not provide any network services of note.

4.1 `eqntott` (SPECint92)

In a 1975 paper [K75], Donald E. Knuth asserted (without further citation) that at that time computer manufacturers believed that about a quarter of computer time was spent sorting. Regardless of whether that fraction was true then, or what that fraction would be today, sorting is an example of software algorithms as technology. A civil engineer chooses from a wide variety of different building material technologies (e.g., wood frame, reinforced concrete, steel I-beam), based on the desired tradeoffs between cost, aesthetics, and strength, which depend on the needs of the particular project. A software engineer similarly chooses from a wide variety of different sorting algorithms based on the desired tradeoffs between coding complexity, suitability to size of problem (i.e., internal vs. external

sorts), average space consumption, average time consumption, variability of space/time consumption, and sort stability. In both fields, *particular technologies are refined over time, and new technologies may be created with potentially completely different tradeoffs.* Taking advantage of evolutionary and revolutionary changes in sorting technologies sounds like fertile ground for a clever compiler.

The `eqntott` application from the SPECint92 suite of benchmarks [G93] translates a logical representation of a boolean equation to a truth table. It includes the following characterization in its description:

> `2.4: Other performance characteristics: EQNTOTT dynamically allocates about 1.8 megabytes of memory in 450 calls to the library routine malloc, when run with the int_pri_3.eqn input file. In addition, the benchmark spends nearly 95% of its time in the library routine qsort.`

This characterization of `eqntott` execution is strictly speaking correct but rather misleading. Using `gprof` profiling, we see that although `eqntott` does spends about 90% of its time executing `qsort()`, about 85% of the total run-time was actually spent in `cmppt()`, which is only called from one particular call site of `qsort()`.

4.1.1 Attacking `qsort()`

The source code files for `eqntott` include an implementation for `qsort()`, which makes sense, since the stability of standard C library `qsort()` — that is, whether it will or will not preserve the initial ordering of elements being sorted which compare as equal — is intentionally undefined. By including its own `qsort()`, those who submitted `eqntott` to the SPEC organization as a benchmark could provide a reference output file to go with the reference input file; whereas if `eqntott` used the system[1] `qsort()`, then the reference output file might not match `eqntott`'s output on a given platform even though the difference would be entirely due to sort stability, and would not make the resulting output incorrect.[2] Since the `qsort()` routine supplied with `eqntott` dates back to at least 1983 (it claims to be

[1] More specifically, by system I mean the `qsort()` implementation located in the default `libc.a` library used by the default linker when invoked by the C compiler driver.

[2] This is worth noting, because it is important that the programmer supplying alternative code ensure that the alternatives are semantically equivalent. However, sometimes programmers and users are prone to rely on non-guaranteed semantics of a particular implementation — such as the sort stability of `qsort()`, for example.

Sort type	Input is...	Sort time (sec.)	Compares (millions)
Quicksort	unsorted	55.73	115.726
Merge sort	unsorted	60.04	93.372
Quicksort	already sorted	28.87	95.000
Merge sort	already sorted	30.29	49.425

Table 4.1: Merge sort and median-of-3 quicksort on arrays of integers.

better than the system `qsort()` of the day...), I was curious if `qsort()` implementations had noticeably improved since then. Forcing `eqntott` to use the system `qsort()` under SunOS 4.1.3 resulted in nearly identical timings as using the included `qsort()`; however, forcing `eqntott` to use the system `qsort()` on Linux *dramatically* reduced run-time, from just over 20 seconds to 2.4 seconds.[3]

Investigation readily uncovered that the Linux system `qsort()` is supplied by the GNU implementation of the `libc` library; by default, it does not use the quicksort algorithm, but rather uses the merge sort algorithm if it can allocate enough temporary buffer to perform a standard two-space merge sort, only falling back to an in-place quicksort if the attempt to allocate memory fails. Both merge sort and quicksort are on average O(N log N) algorithms; however, in the real world, constant factors and not just asymptotic algorithmic complexity matters. It is generally acknowledged that implementations of some of the quicksort variants have the lowest constant factors of well-known comparison-based sorting routines; this is borne out by some basic experimentation — each row of Table 4.1 was generated by summing the times and comparisons for 5 different runs, each run sorting 1,000,000 integers generated by `random()` with seed values 0, 1, 2, 3, and 4 respectively. (Times were quite consistent for runs which used the same sort and had the same sortedness of input, differing only on the random seed.)

So if the particular quicksort implementation beats the merge sort implementation on both randomized and on already-sorted inputs, what accounts for the nearly an order of magnitude difference in `eqntott` performance? (A difference in the wrong direction to boot: Table 4.1 would suggest that `eqntott` using quicksort would be *faster* than `eqntott` using merge sort, when in fact the reverse is true.) One possibility is that most analyses

[3]Correctness was partially verified by checking the output of `eqntott` using the Linux system `qsort()` against the output of `eqntott` using an insertion sort; the outputs were identical.

of sort algorithms assume that comparisons take constant time; this is not always true. In particular, every comparison routine defined in `eqntott` may take variable amounts of time to execute. Hence it can matter greatly to the total run-time exactly which elements are compared with each other, something which was not true for sorting integers. I.e., one hypothesis is that in this case merge sort tends to be comparing elements which the comparison routine can quickly decide whether $a < b$, $a = b$, or $a > b$, whereas quicksort tends to be comparing elements which the comparison routine cannot quickly determine the ordering relationship. It should be noted that such comparison functions are not particularly unusual: an ordering string comparison function, for example, takes variable time to run, being highly dependent on the nature of input data.

As attractive a hypothesis as this might be, however, it does not stand up to scrutiny. In particular, profiling indicates that there is a 79-fold reduction in the number of calls to `cmppt()`: quicksort calls it 2,841,621 times; merge sort, a mere 35,944 times. *That* is responsible for the reduced run time for `eqntott`, and not a hypothesized reduction in the average latency for a call to `cmppt()`. The `eqntott` call site to `qsort()` which provides `cmppt()` as a comparison function hands `qsort()` an array with just 4,106 items; note that 2,841,621 comparisons to sort 4,106 items seems more nearly $O(N^2)$, despite the fact that the quicksort implementation does have the median-of-3 modification which should prevent most (but clearly not all) cases of such degenerate behavior. The number of calls to `cmppt()` which the merge sort issues, on the other hand, seems much closer to $O(N \log N)$. Is the SPECint92 reference input file particularly unusual in eliciting this kind of behavior from the quicksort implementation? ■

This kind of effort with blind alleys, hypotheses to shoot down, etc., is typical of of a human programmer attempting to optimize a program. If the system `qsort()` had been written with a `qif` statement to quasistatically select between a quicksort implementation or a merge sort implementation, it would not be necessary to go to this kind of effort, nor necessary to *speculate* about whether typical user files provided as input to `eqntott` tend to elicit $O(N^2)$ behavior from the quicksort used; whether or not quicksort often behaved poorly or not on user inputs would be directly observed and taken into account. Furthermore, casual trials with another similarly-sized input file for `eqntott` (modeling a 4-bit 16-input/output multiplexer) suggests that the reference input file for the SPECint92 benchmark use of

`eqntott` is in fact somewhat anomalous for eliciting such poor behavior from quicksort; however, program performance for the multiplexor input file was still about 20% better with merge sort than with quicksort, consistent with the results in Table 4.1 which shows that the merge sort implementation tends to perform fewer comparisons — *and the number of comparisons, when each comparison can be quite expensive, will dominate the running time more than the efficiency of the inner loops of the sorting algorithm.* Hence which sort should be selected in a given usage will depend on both the input data pattern and on the cost of the comparison function provided to `qsort()`. To reiterate, if the library writer had written the system `qsort()` with a `qif` statement, then sorting performance would be improved on average, without any effort from the application programmer's part and without complicating the semantics of `qsort()` usage.[4]

4.1.2 Attacking `cmppt()`

Another obvious approach to improving the performance of `eqntott` would be to try to make `cmppt()` execute faster. Figure 4-1 shows the inner loop of the comparison function. It's not hard to convince oneself that the code in Figure 4-2 is equivalent Figure 4-1, but might execute faster on average if $\{aa == bb\}$, $\{aa == 0,\ bb == 2\}$, or $\{aa == 2,\ bb == 0\}$ were common conditions. However, it's unclear which of the three terms should come first in the conditionally evaluated boolean-OR expression for best performance; presumably, this is dependent on the input to `eqntott`. It would be highly tedious to try them all out by hand; but a quasistatic compiler would not be deterred by impatience. With quasistatic `if` support, the programmer can simply write the code sequence in Figure 4-3 and forget about it.[5] Note the total number of permutations is actually far greater than the three which happen to be shown: the three clauses OR'ed together can be arranged 6 different ways; plus, each of the $\{aa == 0,\ bb == 2\}$ and $\{aa == 2,\ bb == 0\}$ clauses contains

[4]By complication, I mean that the library writer could implement two functions `qsort1()` and `qsort2()`, and document that the latter is preferred over the former for expensive comparison functions for faster performance, but otherwise the two behave similarly. This complicates the semantics since the application programmer now has to think about which function he or she wants to use from each different call site in the program.

[5]It would be even nicer if the compiler could be instructed to generate and try all the different permutations for the programmer, but this would require additional special quasistatic syntax. Note that GNU gcc already supports an extension to C which allows the programmer to designate any programmer-defined function as being `const`, i.e., side-effect free and dependent on the input arguments only. This extension currently allows gcc to perform common subexpression elimination and loop hoisting on such functions; however, this can also be used to get us part of the way to a compiler which understood *when* it could rearrange boolean expressions to arrive at a semantically equivalent formulation with faster average performance.

```
if (aa == 2)
  aa = 0;
if (bb == 2)
  bb = 0;
if (aa != bb) {
  if (aa < bb) {
    return (-1);
  }
  else {
    return (1);
  }
```

Figure 4-1: Comparison routine cmppt()'s inner loop.

```
keep_looking = ((aa == bb)
                || ((aa == 2) && (bb == 0))
                || ((aa == 0) && (bb == 2)));
if (keep_looking)
  continue;
if (aa == 2)
  aa = 0;
if (bb == 2)
  bb = 0;
return ((aa<bb) ? -1 : 1);
```

Figure 4-2: A restructuring of the cmppt() inner loop.

two sub-clauses AND'ed together, which can also be permuted — for a total of 24 possible versions.

Table 4.2 shows the performance numbers for a few of the possible versions; only one of the listed versions performs better than the original version. Note that careless use of quasistatic constructs *in tight inner loops* resulted in spectacular profiling overhead, far greater than even that for pixie. This is in part because the current real-time profiling implementation does not customize the stopwatch start and stop routines for each place where they are used, even though they are inlined; hence the routines still have greater overhead than pixie's instrumentation code per invocation. Furthermore, it so happens that the problem is vastly excaberated by the fact that the current implementation of

[....]

```
qif (EQUALITY_FIRST) {
  keep_looking = ((aa == bb)
                  || ((aa == 2) && (bb == 0))
                  || ((aa == 0) && (bb == 2)));
}
qelse qif(AA_EQUALS_TWO_FIRST) {
  keep_looking = (((aa == 2) && (bb == 0))
                  || (aa == bb)
                  || ((aa == 0) && (bb == 2)));
}
qelse qif(BB_EQUALS_TWO_FIRST) {
  keep_looking = (((bb == 2) && (aa == 0))
                  || ((aa == 2) && (bb == 0))
                  || (aa == bb));
}
```

[....]

Figure 4-3: Quasistatic version of cmppt() inner loop.

Version	Instrumented?	Inner loop cycles	Program user time
Original cmppt()	N/A	N/A	19.95 sec.
EQUALITY_FIRST	Yes	2828759078	136.83 sec.
EQUALITY_FIRST	No	N/A	18.53 sec.
AA_EQUALS_TWO_FIRST	Yes	3122155670	139.80 sec.
AA_EQUALS_TWO_FIRST	No	N/A	20.19 sec.
BB_EQUALS_TWO_FIRST	Yes	3154715805	140.21 sec.
BB_EQUALS_TWO_FIRST	No	N/A	20.08 sec.

Table 4.2: Performance of different rearrangements of boolean terms.

real-time profiling has a bug whereby the GNU gcc code generator believes two registers contain live data between the stopwatch start and stop routines, and this causes several of the inner loop variables to be spilled from the unusually small register set of the Intel Pentium processor. Fortunately, the overhead did not in this case mask which alternative would run faster when instrumentation is turned back off. We see that although there is some improvement, the `cmppt()` comparison function did not offer great opportunity for optimization using quasistatic constructs. However, it does serve as an example where the programmer can use the clever compiler's capabilities as a *tool* to simplify the task of exploring what-if scenarios to improve program performance, although judiciously (i.e., not in inner loops). It's less tedious and less error-prone to let the clever compiler manage program versions than to do so by hand.

Other approaches. Several other possible transformations to improve `eqntott` performance, which might benefit from quasistatic constructs, were not fully explored for lack of time. They are described in Appendix C.

4.2 Matrix Manipulations

4.2.1 Special Forms

Much of scientific and engineering computation boils down to calculations based on linear systems of equations, commonly represented as matrices. Hence there is great interest in developing sophisticated packages with routines carefully tuned for accuracy[6] and performance. *Numerical Recipes* [PFTV88], a popular resource for those interested in numerical computation, mentions that such packages (e.g., LINPACK, NAG) often provide many versions of particular routines, which are specialized for special matrix forms (e.g., symmetric, triangular, etc.) and perform faster than the generalized routine on such cases. One can imagine coding using predicates to check for special forms and thus taking advantage of special form routines, even when one is uncertain in advance what special forms the matrices being dealt with might conform to — see Figure 4-4.

[6]Mostly, detecting when round-off errors in performing computations are likely to make the numeric results meaningless. Numerical accuracy issues will not be considered for the remainder of this section.

Is coding in this fashion worthwhile? Suppose in writing numeric crunching applications, the programmer absolutely knows if matrix A is going to take on a special form or not. If so, then testing at all is completely pointless — the programmer can directly call the the correct special form routine, or the general routine if no special forms apply. However, if there is some uncertainty in the programmer's mind, as is likely to be the case for example with someone writing an application-area-specific library, then which strategy — Figure 4-4, Figure 4-5, or just always calling `LU_decompose_general()` directly — gives the best performance requires more careful consideration. The writers of libraries such as LINKPACK and NAG were presumably reluctant to put in additional "unnecessary" checking overhead into the general routines themselves, thinking "Why should we slow down the general library routine with checks for special cases every time, especially if the general form algorithm will always give the correct answer anyway? Let the user check for special cases and call the specialized routines if they want." However, the user may similarly be reluctant to slow down his or her code by writing something like Figure 4-4. Even if each of the `is_*_matrix()` predicate functions takes just $O(N^2)$ execution time, string enough tests together and there will be noticeable slow-down.

On the other hand, in the quasistatic version in Figure 4-5, only if matrix A *is* often one of the special forms will the price of even a single test continue to be paid at equilibrium (e.g., after the initial experimentation phase); and, if matrix A is not any of the special forms being tested for, then *no* testing for special forms will be performed at all at equilibrium. Furthermore, note that the typical input matrix might be simultaneously several special forms, and in such cases the quasistatic version would automatically pick out the specialized routine that performs the best on the statistically likely inputs. However, the quasistatic version has the draw-back that if the input matrix is one special form about half the time and another special form the other half of the time (i.e., the entropy of the question "What special form is this matrix?" is high), then for Figure 4-5 the clever compiler must choose the clause for one of the special forms over the other; however, better performance could be had by pay the price of an extra testing predicate to test for both equally-likely special forms before defaulting to the general form. It would be advantageous to at least include one more clause, which simply contains the chain of tests in Figure 4-4.[7] Also, whereas in

[7]Observation: once again the `qif` statement syntax is not powerful enough to permit an elegant expression. What we'd like to be able to do is to tell the compiler that there are N predicates and N actions

```
extern Matrix A;
Matrix AA = Matrix_copy(A);
Permutations P;

if (is_tridiagonal(A))                          /* O(N^2) */
  LU_decompose_tridiagonal(AA,P);               /* O(N) */
else if (is_band_diagonal(A))                   /* O(N^2) */
  LU_decompose_band_diagonal(AA,P);             /* O(f(N, band width)) */
else if (is_bordered_band_diagonal(A))          /* O(N^2) */
  LU_decompose_bordered_band_diagonal(AA,P); /* O(f(N, band width,
                                                   border width) */
else if (is_triangular(A))                      /* O(N^2) */
  LU_decompose(AA,P);                           /* O(N^3), smaller constant */
else
  LU_decompose_general(AA,P);                   /* O(N^3) */
}
```

Figure 4-4: Selecting at run-time between all the special-case matrix routines.

Figure 4-4, the price for considering another special form is $O(N^2)$ during every execution of this piece of code, the price in the quasistatic case is that there are more versions of the program for the clever compiler to try out, and it will therefore take longer for the clever compiler to converge on a reasonably optimal program.

4.2.2 Blocking

A well-known optimization technique is blocking: given multiple levels of memory hierarchies with different latencies, blocking improves the reuse of data loaded into the faster levels of the hierarchy and thus overall program throughput. One application of blocking is to matrix multiplication: Figure 4-6 contains a simple implementation of blocked matrix multiplication for illustration purposes. In essence, the `BlockedMultiply` routine takes an $N \times N$ matrix and chops it up into blocks of size $B \times B$; at any given time, the routine places less demand on the cache than an unblocked version because in the innermost loop, it is only accessing information from $B \times B$-sized sub-matrices (blocks) of the source matrices X and Y and the destination matrix Z. [LRW91] observes that in the worst case (no block-

to take should the corresponding predicate be true, plus a default should none of the other predicates be true. Then the compiler could construct all possible orders in which to evaluate all possible subsets of the predicates. This is similar to the boolean clause ordering problem with `eqntott`.

```
qif(MATRIX_OFTEN_TRIDIAGONAL) {
  if (is_tridiagonal(AA))
    LU_decompose_tridiagonal(AA,P);
  else LU_decompose_general(AA,P);
}
qelse qif(MATRIX_OFTEN_BAND_DIAGONAL) {
  if (is_band_diagonal(AA))
    LU_decompose_band_diagonal(AA,P);
  else LU_decompose_general(AA,P);
}
qelse qif(MATRIX_OFTEN_BORDERED_BAND_DIAGONAL) {
  if (is_bordered_band_diagonal(AA))
    LU_decompose_bordered_band_diagonal(AA,P);
  else LU_decompose_general(AA,P);
}
qelse qif(MATRIX_OFTEN_TRIANGULAR) {
  if (is_triangular(AA))
    LU_decompose_triangular(AA,P);
  else LU_decompose_general(AA,P);
}
qelse {
  LU_decompose_general(AA,&P);
}
```

Figure 4-5: Quasistatically selecting between special-case matrix routines.

ing), $2N^3 + N^2$ words may have to be accessed from main memory, but if B is chosen such that the $B \times B$ block of Y and a row of length B of matrix Z can fit in the cache without interference, then the number of memory accesses drops to $2N^3/B + N^2$. Clearly, then, it is desirable to maximize the value of B; the speed of microprocessors has increased to the point in the last few years where the number of main memory accesses (i.e., cache misses) and not the actual computation on the data retrieved can prove to be the performance bottleneck. [LRW91] goes on to develop models of cache interference for such blocked algorithms, to come up with a more sophisticated way of choosing the block size to maximize reuse; the rationale for such work is that simply choosing B such that a $B \times B$ block plus a row of length B might just barely fit within the cache (i.e., considering size only) was demonstrated to be inadequate, as data self-interference under certain cirumstances greatly reduced the cache hit rate and thereby the effective throughput [MGA95].

The results in Table 4.3 show (for both integer and double-precision floating point numbers) that as the blocking size increases, the number of CPU cycles elapsed before the computation finishes decreases nearly monotonically, with most of the performance benefit achieved at relatively small values for the block size. Surprisingly, even after I went past the block size that should have meant the the cache had to have overflown (for example, the integer $B = 256$ case should have just about overflowed the cache, since $256 * 256$ 32-bit integers take up 256KB of memory), we see that the performance was hardly impacted at all when we moved to $B = 512$. This argues for experimental verification of timings on actual machines where the code will be run, as this is an result that would not have been expected from the models — my speculation is that the cache pre-fetch was sufficiently effective on this machine that the additional memory demands of too-large a blocking size did not create a bottleneck. In any case, the clever compiler, when allowed to choose from a wide range of blocking values as expressed by a qint parameter, can experimentally determine that it is safe to use very large blocks. However, another machine which was also i386-binary compatible, but with different latencies and bandwidths for transfers between the cache and main memory, might well have choked on the larger block values, hence it is desireable to set the block size quasistatically and not statically.

Table 4.4 generally corroborates the conclusion that large block sizes are quite safe on this machine; the details demonstrate that measuring elapsed real time rather than UNIX process user time can be important, because they sometimes tell different stories. Certain

```c
#define MIN(a,b) ((a)<(b) ? (a) : (b))
#define ELEMENT(a,b,c) (*(a+(b*N)+c))

ELEMENT_TYPE *BlockedMultiply(ELEMENT_TYPE *X, ELEMENT_TYPE *Y,
                              ELEMENT_TYPE *Z, int N, int B)
{
  int kk, jj, i, j, k;
  ELEMENT_TYPE r;

  /* Allow me to do base-1 addressing instead of base-0 addressing;
     This is a similar trick to what Numerical Recipes does. */
  X = X-(N+1);
  Y = Y-(N+1);
  Z = Z-(N+1);

  for (kk=1; kk<=N; kk+=B)
    for (jj=1; jj<=N; jj+=B)
      for (i=1; i<=N; i++)
        for (k=kk; k<=MIN((kk+B-1),(N)); k++)
          {
            r = ELEMENT(X,i,k);
            for (j=jj; j<=MIN((jj+B-1),(N)); j++)
              ELEMENT(Z,i,j) += r*ELEMENT(Y,k,j);
            /* gcc -O is capable of hoisting unnecessary calculations
               out of this innermost loop */
          }
  return (Z+(N+1));
}
```

Figure 4-6: Blocked matrix multiply implementation.

Data	Block size	Cycles (10^9)	Data	Block size	Cycles (10^9)
Double	1	25.076	Integer	1	22.929
Double	2	10.769	Integer	2	9.382
Double	3	8.086	Integer	3	6.693
Double	4	5.639	Integer	4	5.189
Double	5	5.814	Integer	5	4.920
Double	6	5.198	Integer	6	4.444
Double	7	5.026	Integer	7	4.247
Double	8	4.482	Integer	8	3.763
Double	16	3.993	Integer	16	3.359
Double	32	3.858	Integer	32	3.161
Double	64	3.871	Integer	64	3.133
Double	128	3.929	Integer	128	3.143
Double	256	3.988	Integer	256	3.149
Double	512	3.960	Integer	512	3.149

Table 4.3: Multiplying 512x512 matrices with different block sizes.

values for block size (e.g., $B = 5$, $B = 6$, $B = 7$) resulted in drastically poorer elapsed real time performance as compared to neighboring block sizes (mostly due to delays due to page faults), although their user times continued to drop monotonically; somewhat surprising, though, is the elapsed real time performance drop at $B = 320$. (Ironically, $B = 320$ achieved one of the best user times in the table.) A sophisticated model which was given the exact parameters for the virtual memory subsystem for the operating system kernel might or might not have anticipated this lattermost result; however, experimentation by the clever compiler would have revealed it without the need for such detailed knowledge. Another example of the utility of performing measurements of real time as well as user time is comparing $B = 256$ and $B = 1280$: although $B = 256$ took nearly 50 seconds less user time, $B = 1280$ caused sufficiently fewer page faults that it was faster in terms of elapsed time by about 3.25 billion cycles, or about 36 seconds. On this machine $B = 128$ appears to be the overall the fastest, combining the three criteria real time, user time, and page faults; however, on a similar machine outfitted with more or less memory, or on a machine which was configured to swap over the network, a different value might have proved to be optimal.

Block size	Cycles (10^9)	User time (sec.)	Page faults
1	415.661	4535.27	7391
2	166.012	1741.84	8591
3	129.969	1174.76	26430
4	88.316	880.94	8692
5	112.771	819.91	40829
6	90.070	743.83	26271
7	104.778	708.32	42682
8	66.948	638.46	8718
16	61.818	575.68	9216
32	58.378	542.88	8967
64	54.341	528.72	8248
128	53.014	527.25	8625
256	60.773	526.70	17372
320	91.283	533.96	50052
640	60.963	547.51	15121
1280	57.006	573.51	6582

Table 4.4: Multiplying 1280x1280 integer matrices with different block sizes.

Chapter 5

Conclusions

In this thesis, I first briefly described the concept of quasistatic computing and how it can manifest itself in a clever or smart compiler. The underlying assumption behind the utility of a smart compiler is that a compiler can make profitable use of profiling information of a program's actual end-use to guide (drastic, in this case) transformations of programs to improve average performance. Not only is it unlikely that this decades-old hypothesis will ever be conclusively "proven" true or false — there being too great a variety of programs for any such sweeping statements to be true — but it has never even been adequately tested. My goal, then, was implement a system which would allow programmers to pragmatically explore whether the hypothesis was usually true or not on the kinds of programs which they are writing. By defining some extensions to the C programming language (quasistatic variables and the `qif-qelse` statement, quasistatic parameters and the `qinfluence` statement), and implementing a compiler system based on the SUIF compiler and the GNU gcc compiler that accepts source code with such constructs and emits instrumented executables which periodically call the compiler on themselves, I've demonstrated that it is quite feasible to implement at least a *clever* compiler, if not a *smart* one. Finally, by modifying the SPECint92 benchmark `eqntott` and writing some simple matrix manipulation examples, I've demonstrated that these particular constructs, even if imperfect, can change *how* a programmer goes about optimizing a program, as well as serve as an aid to a programmer performing more traditional forms of profile-driven hand-optimization.

5.1 Related Work

Knuth [K71] argued for profile-driven optimization by compilers back in 1971. Wall [W90] summarized some investigation of how well estimated and real profiles predicted program performance characteristics across multiple runs with different input data; this work is sometimes interpreted as negative support for profile-driven optimization, since it indicated that actual profiles from actual runs were nowhere near perfect predictors of other profiles of actual runs. Nevertheless, numerous commercial C compilers (Digital, SGI – MIPS architecture; Digital – Alpha; Hewlett-Packard – PA-RISC; IBM – POWER2; Convex) today do have options in their compilers to accept profiling feedback information (basic block and function invocation frequencies) during the compilation and link stages[1]. However, it appears that such information is being used primarily in the code generation phase, for such purposes as improving accuracy of conditional branch predictions to generate conditional branch opcodes with the the correct direction "hint", and global register allocation. These compilers do not perform high-level code transformations, nor do the compilers actively experiment to see what output code perform best. See [SW92, FF92, CGL95].

A fair amount of work has been done on profile-driven optimizations such as selective inlining and efficient polymorphic method dispatch, focused mostly on research functional languages. See [DC94, DCG95, HU94]. By comparison, relatively little work has been done with general frameworks for profiling-driven optimizations of programs written in C, at levels other than the code generation pass. However, some specific work on inlining and types selection exists. See [CMCH91, S91].

Dynamic code generation (DCG) is another approach to compiling code based on actual usage. One disadvantage of DCG is that the cost of compilation is paid possibly multiple times every single program run; hence, the full suite of standard optimizations are usually not applied. In the clever compiler framework, the costs of compilation may be amortized over multiple program runs, and eventually the frequency of recompilation may be slowly reduced. One major advantage of DCG is that incorporating code generation into the program executable allows it to re-adapt for changing usage patterns on a *much* finer timescale than is possible with a clever compiler. See [EP93].

[1] Also, a number of operating system's linkers will rearrange functions and static data structures to reduce the number of cache collisions based on profiling feedback; some third-party products can perform this same function on already-linked executables.

5.2 Practicality Issues

Program distribution. The clever/smart compiler model described throughout this thesis presumes that source code is available to the end-user for recompilation. Although a fair amount of software have freely distributable source code, much software is owned by companies which refuse to make source code available except under tightly controlled conditions, in order to protect their trade secrets and proprietary rights in their capital investment. A possible solution to this particular problem might be the wide adoption of some intermediate format which could be made available instead of the human-readable representation of the source code. Such an intermediate format, to satisfy the needs of a clever/smart compiler, would need to preserve a reasonable amount of high-level programming language structure, but *still* inhibit reverse-engineering. (SUIF, for example, would be unsuitable for distribution purpose, since a "reverse-engineering" pass is not just *possible* given the information stored in the intermediate format, but part of the standard distribution.) The Architecture-Neutral Distribution Format (ANDF) technology being worked on by the Open Software Foundation [OSF93] offers a possible solution to this dilemma. ANDF is a language-independent and machine-independent intermediate language that came into existence largely to address the problem of commercial software distribution for multiple instruction set architectures; hence, some of its design goals address this source code access problem. Namely, ANDF attempts to preserve as much semantic information as possible so that the translator from ANDF format to the native platform binary format can perform as much optimization as possible, while still inhibiting reverse engineering (for examples, identifiers are replaced with unique tags). ANDF is complex because it attempts to solve many other problems as well, and its success in the marketplace seems questionable. However, the corpus of ANDF-related work should be considered with respect to any further evaluation of the practicality of distributing "source code".

Compiler availability. The clever/smart compiler model also presumes that the end-user has access to the compiler. This is increasingly not a safe assumption. Although users of UNIX workstations frequently have access to the system cc compiler, the vast majority of computer users use systems which do not include a compiler in the base system software (e.g., IBM compatibles running DOS, Windows, OS/2, or NT; or Macintoshes running the

MacOS).[2] Even in the UNIX world, it is becoming increasingly common for users to be given accounts which do not have access to compilers (on the grounds that they have no need for access to a compiler), and for system vendors to unbundle their optimizing compilers from the base system software in order to generate additional revenue by selling separate licenses to developers, bundling only an older less-capable compiler. If this trend does not reverse itself, the viability of the clever/smart compiler model would be seriously impaired.

Debugging and testing. A correct smart compiler would be tricky to write — correct meaning that the compiler only made program transformations which were semantics-preserving, no matter what the input program. However, it would not be any more difficult for a programmer to *use* than a normal compiler. A correct clever compiler is much easier to write, but the obligation placed on the programmer using the clever compiler is substantial. While writing the program, more emphasis will probably have to be given to data invariants and attempting to ensure that all legal combinations of alternatives result in a program which preserves invariants. Debugging will be more difficult since the programmer now needs to keep track of the settings of the quasistatic variables and parameters in order to reproduce a bug and in order to know what version of the program to examine. Comprehensively testing the program will become that much more intractable when there are an explosive number of alternative versions of the program. However, not all is bleak: formalizing ways of expressing alternative code, and thus replace current *ad hoc* methods, may result in some improvement in source code comprehensibility.

5.3 Future Work

5.3.1 Bug Fixes

Multiple source code files support. The snoot pass should be further modified to support the `qvar` construct, and support use of the `extern` type modifier so as to permit `qvar` and `qint` declarations and not just definitions. Once that is done, then the implemented clever compiler could in theory support multiple source code files. However, the SUIF compiler system itself at the current time only has limited support for multiple source

[2] Out of these example systems, some versions of DOS do at least provide a BASIC interpreter, and OS/2 provides a REXX interpreter.

files. In particular, version 1.0.1 does not have a supported, functional `linksuif` pass which would make the global symbol tables of multiple SUIF files consistent, among other things making declarations which *ought* to be globally visible available in all global symbol tables.

Default scope for `qint` parameters. Chapter 2 specifies that if there are no `qinfluence` statements referencing a particular `qint` parameter, then that `qint` parameter should be associated with the performance of all parts of the program where the definition is visible. This has not yet been implemented. However, this is not a major problem but rather a minor convenience for the programmer.

5.3.2 Enhancements

Smarter searching. The `select_alternatives` pass currently does not perform any kind of smart searching of program version space, and simply searches the entire space. It needs to incorporate both standard space searching techniques and domain-specific heuristics enabled by the information gathered by `integrate`. The current `integrate` pass only finds and records relationships between `qif` variables which are used in the same `qif` statement and `qint` parameters which are referenced in nested `qinfluence` statements. A more sophisticated `integrate` would record relationships between sets of quasistatic variables and sets of quasistatic parameters resulting from `qinfluence` statements nested inside `qif` clauses and `qif` statements nested in `qinfluence` statements as well as other `qif` statements. In addition, `integrate` should record dynamic interactions such as in Figure 3-6.

Program context. A given run of the program has a great deal of context which can be detected by instrumentation code in the program — a few examples of such context items include time of day, identity of person using the program, exactly which machine is being used, load of machine during the program run, network bandwidth and latency, command-line parameters. Some of these context items may be worth storing with more standard forms of profiling data during each run, to give the clever compiler more information upon which to base program version selection. Determination of whether this would be worthwhile is a matter for pragmatic exploration.

Cross-platform implementation, extending support to other profiling mechanisms. Cross-platform availability is especially desireable because one potential benefit of

the quasistatic computing model is to optimize programs differently on different machine platforms. Porting the clever compiler implementation will allow actual evaluation of how much benefit can be obtained. Related to porting, the clever compiler's reliance on real-time stopwatch profiling should be eliminated since many platforms will not support such a mechanism; the clever compiler should be able to make use of `prof/gprof`-like mechanisms which are widely supported. A clever compiler should be able to use program performance information from any of the supported mechanisms on a platform.

5.3.3 Additional Research

Real-time profiling. *Combining* real-time profiling techniques with CPU-time profiling techniques potentially gives additional synergistic insight into a program's performance characteristics. This may be worth exploring independently of the clever compiler implementation.

Constructs. A clever compiler, which requires extending the C language with new syntax and constructs, may or may not be the best stepping stone toward a smart compiler. However, even in developing the examples in Chapter 4, several times the syntax and constructs defined in Chapter 2 proved to not be sufficiently expressive to conveniently encode the code alternatives. Perhaps a few simple extensions would suffice; however, perhaps entirely different mechanisms of encoding code alternatives in source code should be considered and evaluated.

Integration It will be desireable to eventually integrate work on this clever compiler implementation with other manifestations of quasistatic computing ideas, such as feedback-directed specialization of C [B95]. Integration might entail coordinating profiling efforts (i.e., standardizing data collection) and preventing different modules which wish to conduct experiments on different program versions from interfering with each other. Work on such integration would be a step toward a quasistatic computing "environment" [DE94, DBE+94].

Appendix A

Compiler Selection Details

A.1 SUIF Output

Figures A-1, A-2, A-3, and A-4 illustrate what the intermediate format actually looks like when pretty-printed.

A.2 SUIF Criteria Conformance

Source code available Source code is readily available via Internet anonymous FTP from suif.stanford.edu:/pub/suif/suif-1.0.1.tar.Z. Permission is explicitly granted in the copyright notice to modify the software for non-commercial purposes. No FSF-style "copyleft" conditions apply.

Widespread use. SUIF is not in widespread use. Probably something on the order of several dozen sites have the SUIF compiler installed to play with it or to investigate possibly using it for a platform for research. SUIF's authors make it clear that they expect SUIF to remain a research platform; they assert that they do not plan to evolve SUIF into a production compiler — it will never become a practical, general-purpose replacement for the native platform compiler.

Competitive performance. The only backend code generator supplied with SUIF 1.0.1, for the MIPS architecture, achieves 70–90% of the performance of the commercial-available compilers on the MIPS platform for those SPEC92 benchmarks which it is able to compile with the scalar optimizer turned on. [WFW+94] However, this statistic is obviated by the fact that I am using SUIF as a source-to-source compiler and compiling the resulting output with gcc. The run-time performance of gcc compiled code should be adequately competitive. (Also, a few incomplete trials suggest that the output source code of SUIF (used as a source-to-source compiler) when compiled by gcc performs only slightly worse than the original source code compiled by gcc.)

Cross-platform. The SUIF 1.0.1 release does not explicitly delineate what platforms it is available for; however, the compiler and associated libraries appear to be quite portable. As mentioned previously, the only backend code generator is for the MIPS architecture;

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

int abs(int i) {
  if (i<0)
    return -i;
  else
    return i;
}

int main(int argc, char *argv[]) {
  signed int loop;
  for (loop = -10; loop <= 0; loop++) printf("%d ", abs(loop));
}
```

Figure A-1: `countdown.c`, to be fed to SUIF.

when used as a source-to-source compiler, the output from SUIF is ANSI C and thus quite portable.[1]

Ease of modification. Some of the features which SUIF has that makes it easy to modify and extend, and thus suitable for compiler research, include:

- Kernel library routines defining an intermediate representation format for programs that can support both high-level (i.e., tree-structured) and low-level (i.e., flattened) program representations. The kernel library routines include a reasonable set of manipulation primitives.

- Compiler passes are structured as separate programs which communicate via files containing the intermediate representation; hence the interface between passes is well-delineated. This makes the modification of existing passes and creation of new additional passes straightforward.

- Non-kernel tools like `printsuif` (a program that outputs a human-readable representation of the intermediate representation) and the `builder` library (a library which takes advantage of C++ syntax to make generating new C statements for insertion into a SUIF file relatively painless) provide helpful functionality during development.

- An annotations facility allows the attachment of additional and arbitrary pass-specific information *directly* to the relevant parts of intermediate format.

[1]There are minor problems with transporting intermediate format files from one platform to another because the intermediate format files contain declarations from system header files, the exact types of which may vary slightly from platform to platform.

[....]

```
  PROC P:.abs
    Procedure symbol table: 'abs'
      Types:
        <None>
      Symbols:
        s:p1: 'i' var_sym with t:g17; param userdef
        s:p2: 'L1' label_sym
      Definitions:
        <None>
      Annotations:
        ["enable_exceptions": "int_divide-by-0"]
      Parameters:
        abs.i
 1: mrk
      ["line": 5 "countdown.c"]
 2: IF (Jumpto=L:abs.L1)
    IF HEADER
 3:   bfalse  e1, L:abs.L1
 4:     e1: sl t:g28 (i.32) abs.i, e2
 5:       e2: ldc t:g17 (i.32) 0
    IF THEN
 6:   mrk
        ["line": 6 "countdown.c"]
 7:   ret  e1
 8:     e1: neg t:g17 (i.32) abs.i
    IF ELSE
 9:   mrk
        ["line": 8 "countdown.c"]
10:   ret  abs.i
    IF END
  PROC END
```

[....]

Figure A-2: Output of `printsuif countdown.spd` for `abs()` function.

[....]

```
  PROC P:.main
    Procedure symbol table: 'main'
      Types:
        <None>
      Symbols:
        s:p1: 'argc' var_sym with t:g17; param userdef
        s:p2: 'argv' var_sym with t:g129; param userdef
        s:p3: 'loop' var_sym with t:g17; userdef
        s:p4: 'L1' label_sym
        s:p5: 'L2' label_sym
      Definitions:
        <None>
      Annotations:
        ["enable_exceptions": "int_divide-by-0"]
      Parameters:
        main.argc
        main.argv
 1: mrk
      ["line": 13 "countdown.c"]
 2: FOR (Index=main.loop Test=SLTE Cont=L:main.L1 Brk=L:main.L2)
    FOR LB
24:   ldc t:g17 (i.32) -10
    FOR UB
16:   ldc t:g17 (i.32) 0
    FOR STEP
18:   ldc t:g17 (i.32) 1
    FOR LANDING PAD
    FOR BODY
 3:   mrk
        ["line": 13 "countdown.c"]
 4:   mrk
        ["line": 13 "countdown.c"]
 5:   mrk
        ["line": 13 "countdown.c"]
 6:   cal t:g17 (i.32) <nullop> = e1(e2, e3)
 7:     e1: ldc t:g211 (p.32) <P:.printf,0>
23:     e2: ldc t:g46 (p.32) <.__tmp_string_0,0>
10:     e3: cal t:g17 (i.32) e4(main.loop)
11:       e4: ldc t:g214 (p.32) <P:.abs,0>
    FOR END
19: mrk
      ["line": 13 "countdown.c"]
20: mrk
      ["line": 14 "countdown.c"]
21: ret  e1
22:   e1: ldc t:g17 (i.32) 0
  PROC END
```

[....]

Figure A-3: Output of `printsuif countdown.spd` for `main()` function.

[....]

```
extern int abs1(int);
extern int main(int, char **);
extern int printf();

extern int abs1(int i)
  {
    if (i < 0)
      {
        return -i;
      }
    else
      {
        return i;
      }
  }

extern int main(int argc, char **argv)
  {
    int loop;

    for (loop = -10; loop <= 0; loop++)
      {
        printf("%d ", abs1(loop));
      }
    return 0;
  }
```

[....]

Figure A-4: Partial output of `s2c countdown.spd`.

Appendix B

Profiling Mechanisms

B.1 prof — PC Sampling

To prepare for using the `prof` program, the programmer gives the C compiler and linkers an option (usually `-p`) at compile- and link-time. The compiler generates additional code in each function's prologue, including a call to the function `mcount()`, and creates an unique storage slot for each function[1]; the linker links in a special startup function. During program execution, the startup function enables a periodic operating system timer interrupt, which typically operates at 100 Hertz. The handler for the interrupt samples the program counter (PC) value when the interrupt occurs and uses it to index to the appropriate bin, which stores a count that it increments. Data gathered is thus of a statistical nature. The `prof` program post-processes the raw bin counts into human-readable output by totalling the counts in the bins which correspond to each function symbol in the executable's symbol table.

B.2 gprof — Call Graph

`gprof` [GKM82] profiling is similar in some ways to `prof` profiling. Instead of `prof`'s option `-p`, the usual option to enable `gprof` profiling is `-pg`. The linker links against a different `mcount()` function which maintains exact counts of entries into each function by individual call sites, probably by walking the stack at run-time to find the address the called function will return to. The `gprof` post-processor then constructs the call graph for the program, and propagates function execution time (obtained from the PC sampling) through the call graph, proportionally to the number of calls from each call site for the function. The resulting weighted call graph gives a more thorough picture of inefficiencies in the program; however the call graph may be substantially inaccurate when:

- Propagating execution time meaningfully is difficult when there is recursion (i.e., the call graph is not a tree).
- The heuristic of allocating execution time of a function to its call sites proportionally to the number of calls from each call site fails because different call sites made substantially different demands on the function. E.g., a function might be called equal

[1] Possibly under some implementations the additional prologue code and the storage slot created even for `prof`-style profiling are a no-op and go unused, respectively.

number of times from location A and B, but the average latency for calls from A might be 100 times longer than the average latency for calls from B; nevertheless, gprof would assign equal amounts of time to be propagated up the call graph to locations A and B.

B.3 pixie — Basic Block Counting

Unlike prof or gprof profiling, pixie profiling does not require giving an option at compile- or link-time. The pixie utility directly operates on an executable — it analyzes the code to find the basic blocks boundaries, and then instruments the code to keep exact counts of how many times each basic block is executed. No PC sampling is performed. Instead, the post-processor (often a pixie-aware prof post-processor) converts the basic block counts into ideal time by summing the ideal cycle times of the instructions in each basic block. One commonly acknowledged flaw of this technique is that ideal time may be considerably skewed from real time due to cache effects; another less commonly discussed flaw is that the cycle times of instruction sequences may dramatically shift between different implementations of the same instruction set architecture.[2]

Unlike prof and gprof, which are nearly universally available, pixie is available only for some MIPS workstations and Digital OSF/1 Alpha workstations. Some compilers (e.g., GNU gcc 2.6.2; SunOS 4.1.3 cc; however, many do not) have an option -a which cause them to emit code that count basic block execution. However, such counts are currently only used by the tcov program (which displays code coverage) and not by available prof or gprof profiling data post-processors.

B.4 Real Time Stopwatches

Figure B-1 demonstrates how stopwatch counters can be implemented using the RDTSC instruction on a Pentium for timestamps. See also Appendix D for the actual stopwatch code used for the clever compiler implementation.

[2]For example, [A94] spends several chapters discussing the dramatically different performance characteristics of instructions sequences on an Intel i386 vs. i486 vs. Pentium. A i486 is RISC-like in that many more instructions will execute in just one cycle than on an i386, and the Pentium is dual-issue superscalar; any pixie-like tool would have to take this into account in order to calculate meaningful ideal times lest a sequence of instructions carefully tuned for the Pentium's two pipelines be unfairly assigned far more time than it actually takes to execute because pixie was using cycle counts for instructions executing on an i386. Along similar lines, [C+91, CSW94] look at instruction set utilization and the difference in SPEC92 benchmark results for MIPS- and SPARC-based workstations when compilers were given command-line options to permit them to assume later implementations of those chip architectures.

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

#define RDTSC ".hword 0x310F" /* db 0Fh 31h */

INLINE int function(int fiddle)
{
  fiddle++;
  return (fiddle*2);
}

int main(int argc, char *argv[])
{
  register int eax asm ("%eax");
  register int edx asm ("%edx");
  int start_hi, start_lo, finish_hi, finish_lo;
  int dummy_start_hi, dummy_start_lo, dummy_finish_hi, dummy_finish_lo;
  unsigned long long int cycles, dummy_cycles;

  asm volatile (RDTSC);
  start_hi = edx; start_lo = eax;

#ifndef DO_NOTHING
  #ifdef TIME_PROFILING_CODE
    /* Duplicate all the code which would be wrapped around
       something which I wanted to profile */
    asm volatile (RDTSC);
    dummy_start_hi = edx; dummy_start_lo = eax;
    /* CODE WHICH I WANTED TO PROFILE WOULD GO HERE */
    asm volatile (RDTSC);
    dummy_finish_hi = edx; dummy_finish_lo = eax;
    dummy_cycles = ((dummy_finish_hi-dummy_start_hi)*(1<<31)
     + (dummy_finish_lo-dummy_start_lo));
  #else
      {
        int i = 0;
    #ifndef JUST_ONCE
    #define ITERATIONS 1000
        for (; i<ITERATIONS; i++)
    #endif
          function(i);

      }
  #endif
#endif

  asm volatile (RDTSC);
  finish_hi = edx; finish_lo = eax;

  cycles = ((finish_hi-start_hi)*(1<<31) + (finish_lo-start_lo));
  printf("Time elapsed: %i (%X:%X) cycles, %.f nanoseconds.\n",
 (int) cycles,
 (finish_hi-start_hi), (finish_lo-start_lo),
 cycles / 90000000.0 * 1000000000.0);
  /* Prevent gcc from optimizing dummy_* variable operations away altogether */
  if (dummy_cycles == 0) return 1;
}
```

Figure B-1: `test-rdtsc.c`: using the RDTSC time stamp instruction.

Appendix C

Further eqntott Transformations

Copy the table. Semantically, cmppt()'s inner loop is comparing rows in a truth table where the values 0 or 1 represent boolean false or true for that particular table entry, and 2 represents "don't care". The "don't care" values are being "flattened" on the fly to the value 0 for comparison ordering purposes. Such flattening is being done repeatedly on each row: merge sort's 35,944 calls to cmppt() (for the reference input file) means that an average of 17.5 flattenings are performed on each of the 4106 truth table entries; quicksort's 2,841,621 calls to cmppt() means an average of 1384 flattenings are performed on each entry. It may or may not prove faster (depending on whether $O(N^2)$ behavior was common) to make a copy of the entire truth table, mutate all the entries in it once to flatten them, and then re-write cmppt() to perform faster comparisons by using entries from the copy of the table, and write a customized sort routine that perform element swaps on entries in both the copy and the original table.[1] This adds both a per-program-execution cost of $O(N)$ time and $O(N)$ space (probably with fairly large constants, however) to copy and flatten entries in the copied table, and additional overhead ($O($*number of* cmppt() *calls*$)$) in the new sorting routine to swap four elements every time the unmodified sort routine would have swapped two elements. The choice to quasistatically choose between this transformation and the original might look something like this:

```
qif(FLATTEN_FIRST)
  {
    initialize_truth_table_copy();
    qsort_using_copy(..., cmppt_using_copy);
  }
qelse
  {
    qsort(..., cmppt);
  }
```

[1] qsort() is not sufficiently parameterized; if qsort()'s interface had included passing in a function to perform element swapping, then it would be possible to use qsort() itself for this purpose.

More mileage from each compare. Once we consider compressing the "don't care" value down to 0, another transformation suggests itself. If each truth table entry is considered a vector of bits, then even `cmppt_using_copy()` is extremely inefficient, because it is performing the comparison between two bit vectors one bit at a time, when the hardware of the processor is capable of doing exactly the kind of comparison it is doing many bits at a time. To modify `initialize_truth_table_copy()` and `cmppt_using_copy()` to collapse this inefficient representation of bit vectors by using shifts and OR's to coalesce every K truth table entries in each row into a single unsigned integer in the copy of the truth table would greatly increase the constant in the $O(N)$ run time for `initialize_truth_table_copy()`, but it should greatly decrease the run time of `cmppt_using_copy()`. Furthermore, "natural" values for K that come to mind include 8, 16, and 32, but really any value from 2–32 should be considered. Using quasistatic constructs to code all this would relieve the programmer of the task of trying to determine whether this parameterized transformation is on average advantageous or not.

Appendix D

Source Code

The following source code was written to modify and extend the SUIF 1.0.1 compiler package, freely distributable for non-commercial uses as ftp://suif.stanford.edu/pub/suif/suif-1.0.1.tar.Z (see also the World Wide Web URL http://suif.stanford.edu/suif/suif.html).

D.1 integrate.cc

```
#include "integrate.h"

/***********************************************************************/
/* Code */
/***********************************************************************/

/* Before and after hooks to be passed to my_suif_proc_iter */

void before_hook(void)                                                    10
{
  /* Do nothing */
}

void after_hook(void)
{
  count_qints();
  delete_duplicate_sets(k_qinfluence_set);
  delete_duplicate_sets(k_qif_set);
}                                                                         20

/***********************************************************************/

// [....]

/***********************************************************************/

void define_qifs(tree_node *t, int *num_qifs)
{
  annote *an;                                                             30
  char *an_string;
  block_symtab *symbol_table;
  var_sym *qif_variable;
  base_symtab *file_symbol_table;
```

```
  if (t->kind() != TREE_IF)
    return; /* After this: t->kind() == TREE_IF */

  an = t->annotes()->peek_annote(k_qif_statement);
  if (an == NULL)                                                          40
    return; /* After this: an != NULL */

  (*num_qifs)++;

  an_string = (an->immeds()->tail()->contents).string();

  symbol_table = (block_symtab *) t->scope();

#define LOOK_IN_PARENT_TABLES TRUE
  qif_variable = symbol_table->lookup_var(an_string, LOOK_IN_PARENT_TABLES);  50
  assert(qif_variable != NULL);

  file_symbol_table = symbol_table;
   // In some ways it would be nicer if definitions for qif vars and
   // qints were pushed into the global symbol table instead of the
   // file symbol table.  However, can't do that, since SUIF rightfully
   // objects that in the C paradigm, there is no meaning to a global
   // variable definition not associated with a source file.
  while (file_symbol_table->kind() != SYMTAB_FILE)
    file_symbol_table = file_symbol_table->parent();                       60

  if (qif_variable->annotes()->peek_annote(k_qif_variable_defined) == NULL)
    {
      var_def *vd;
#define VD_DATA_ALIGNMENT 32
      vd = new var_def(qif_variable,VD_DATA_ALIGNMENT);

      immed_list *il = new immed_list();
      il->push(new immed_list_e(*(new immed(0))));
      il->push(new immed_list_e(*(new immed(32))));                        70
      il->push(new immed_list_e(*(new immed(1))));
      /* Note: the k_repeat_init annotation [1 32 0] specifies initialization
           to zero; however, s2c will not actually explicitly convert these back
           to "int i = 0;" since ANSI C specifies that static variables which
           are not initialized must be initialized as if they were assigned
           0, and s2c apparently takes advantage of this. */
      vd->append_annote(k_repeat_init, (void *) il);

      file_symbol_table->add_def(vd);
      attach_annotation_string                                             80
           (qif_variable, k_qif_variable_defined,
            file_symbol_table->name());
      if (debug_level >= 2)
           printf("  Definition created for qif variable '%s'.\n", an_string);
    }
  else
    if (debug_level >= 2)
      printf("   ...qif variable '%s' already given definition.\n", an_string);
}
                                                                           90
/**********************************************************************/

void qif_set_accumulate(tree_node *t, CStringSet *pSet)
```

```
{
  attach_annotation_string(t, k_qif_set_analyzed, "analyzed");

  annote *an = t->annotes()->peek_annote(k_qif_statement);
  pSet->add(strdup((an->immeds()->tail()->contents).string()));

  tree_node_list *pTNL = ((tree_if *) t)->else_part();                       100

  /* pTNL points to (unfortunately, rather dependent on SUIF particulars):
     (0) Nothing (no final qelse).
     (1) Final qelse, with need for a block:
             BLOCK
               mrk
            ....
           BLOCK END
     (2) Another qif in the chain:
           mrk                                                                110
             TREE_IF
     (3) Final qelse with no need for a block:
             mrk
             <statements expressed in SUIF>
  */

  if (pTNL->count() == 0)
    {
     pSet->add("...  NO FINAL QELSE");
     return;                                                                   120
    }

  if ((pTNL->count() == 1) && (pTNL->head()->contents->is_block()))
    {
     pSet->add("...  FINAL QELSE");
     return;
    }

  assert(pTNL->count() > 1);
  tree_node *pTN2 = pTNL->head()->next()->contents;                           130
  if (pTN2->is_if() && (pTN2->annotes()->peek_annote(k_qif_statement) != NULL))
    {
     qif_set_accumulate(pTN2, pSet);
     return;
    }
  else
    {
     pSet->add("...  FINAL QELSE");
     return;
    }                                                                          140
}

void find_qif_sets(tree_node *t, int *num_qif_sets)
{
  if (t->kind() != TREE_IF)
    return; /* After this: t->kind() == TREE_IF */

  if ((t->annotes()->peek_annote(k_qif_statement) == NULL)
      || (t->annotes()->peek_annote(k_qif_set_analyzed) != NULL))
    return; /* After this: t is an unanalyzed qif statement. */               150

  CStringSet set;
```

```
  qif_set_accumulate(t, &set);
  assert(! set.empty());

  t->append_annote(k_qif_set, (void *) pCStringSet2pIL(&set));
  global_symbol_table->append_annote
    (k_qif_set, (void *) pCStringSet2pIL(&set));
                                                                          160
  if (debug_level >= 2)
    {
      printf("   Found qif set {%s}.\n", pCStringSet2pChar(&set));
    }
  (*num_qif_sets)++;
}

/**********************************************************************/

void count_qints(void)                                                    170
{
  int num_qints = 0;

  sym_node_list_iter *iter = new sym_node_list_iter
    (global_symbol_table->symbols());
  while (! iter->is_empty())
    {
      sym_node *sn = iter->step();
      annote *an = sn->annotes()->peek_annote(k_qint_parameter);
      if (an != NULL)                                                     180
          {
            num_qints++;
            if (debug_level >= 2)
              printf("   Found qint declaration '%s' [%s].\n",
                     sn->name(),
                     (an->immeds()->tail()->contents).string());
          }
    }

  if (debug_level >= 1)                                                   190
    printf(" File set had %d qint parameters total.\n", num_qints);
}

/**********************************************************************/

void qinfluence_set_process_TNL(CStringSet *pOuterSet, tree_node_list *pTNL)
{
  tree_node_list_iter *iter = new tree_node_list_iter(pTNL);
  while (! iter->is_empty())
    {                                                                     200
      tree_node *tn = iter->step();
      tree_node_dispatch(pOuterSet, tn);
    }
}

void tree_node_dispatch(CStringSet *pOuterSet, tree_node *t)
{
  switch (t->kind())
    {
    case TREE_BLOCK:                                                      210
      if (t->annotes()->peek_annote(k_qinfluence_statement) != NULL)
```

```
        {
         if (t->annotes()->peek_annote(k_qinfluence_set) == NULL)
           mark_qinfluence_set_and_traverse(pOuterSet, (tree_block *) t);
         else
           assert(0); /* shouldn't happen, traversal order must have gotten
                      messed up.  */
        }
    else
        qinfluence_set_process_TNL(pOuterSet, ((tree_block *) t)->body());           220
    return;

  case TREE_LOOP:
    qinfluence_set_process_TNL(pOuterSet, ((tree_loop *) t)->body());
    return;

  case TREE_FOR:
    qinfluence_set_process_TNL(pOuterSet, ((tree_for *) t)->body());
    qinfluence_set_process_TNL(pOuterSet, ((tree_for *) t)->landing_pad());
    return;                                                                          230

  case TREE_IF:
    qinfluence_set_process_TNL(pOuterSet, ((tree_if *) t)->then_part());
    qinfluence_set_process_TNL(pOuterSet, ((tree_if *) t)->else_part());
    return;

  case TREE_INSTR:
    return;

  default:                                                                           240
    /* I've enumerated all the SUIF-listed TREE_FOO types.... */
    assert(0); /* shouldn't ever be reached.... */
  }
}

void mark_qinfluence_set_and_traverse(CStringSet *pOuterSet, tree_block *t)
{
/*
 "Requires": called on an unanalyzed qinfluence TREE_BLOCK.
 Effects: marks t with its qinfluence set, also side-effects children                250
        of t with their qinfluence sets.
*/
  annote *an = t->annotes()->peek_annote(k_qinfluence_statement);
  assert((t->kind() == TREE_BLOCK)
         && (an != NULL)
         && ((t->annotes()->peek_annote(k_qinfluence_set)) == NULL));

  CStringSet *pMySet = new CStringSet();
  char *str = strdup((an->immeds()->tail()->contents).string());
  char *ptr = strtok(str, ",");                                                      260
  while (ptr != NULL)
   {
    pMySet->add(strdup(ptr));
    ptr = strtok(NULL, ",");
   }
  *pMySet |= *pOuterSet;

  t->append_annote(k_qinfluence_set, (void *) pCStringSet2pIL(pMySet));

  global_symbol_table->append_annote                                                 270
```

```
    (k_qinfluence_set, (void *)pCStringSet2pIL(pMySet));

  if (debug_level >= 2)
    {
      printf("  Found qinfluence set (");
      Pix index;
      index = pMySet->first();
      while (index != 0)
            {
             printf("'%s'", (*pMySet)(index));                                    280
             pMySet->next(index);
             if (index != 0)
               printf(",");
            }
      printf(").\n");
    }

  qinfluence_set_process_TNL(pMySet, t->body());
  delete pMySet;
}                                                                                 290

void find_qinfluence_sets(tree_node *t, int *num_qinfluence_sets)
{
  if (t->kind() != TREE_BLOCK)
    return; /* After this: t->kind() == TREE_BLOCK */

  if ((t->annotes()->peek_annote(k_qinfluence_statement) == NULL)
      || (t->annotes()->peek_annote(k_qinfluence_set) != NULL))
    return; /* After this: t is an unanaluzed qinfluence block. */
                                                                                  300
  CStringSet *pEmptySet = new CStringSet();
  mark_qinfluence_set_and_traverse(pEmptySet, (tree_block *) t);
  delete pEmptySet;
  (*num_qinfluence_sets)++;
}

/**********************************************************************/

void do_proc(tree_proc *tp)
{                                                                                 310
  int num_qifs = 0;
  int num_qif_sets = 0;
  int num_qinfluence_sets = 0;

  if (global_symbol_table == NULL)
    /* Can't put this in before_hook since I need to grab onto tp.... */
    {
      global_symbol_table = tp->scope();
      while (global_symbol_table->kind() != SYMTAB_GLOBAL)
            global_symbol_table = global_symbol_table->parent();                  320
    }

  char *procedure_name = tp->proc()->name();

  if (debug_level >= 1)
    printf(" do_proc:  entering procedure '%s'.\n", procedure_name);

  /* Note that the default parameters for tp->map() are for a preorder
     tree traversal; the preorder nature is important. */
```

```
                                                                          330
  tp->map(define_qifs, &num_qifs);
  if (debug_level >= 1)
    printf(" do_proc:  procedure '%s' has %d qif statements.\n",
           procedure_name, num_qifs);

  tp->map(find_qif_sets, &num_qif_sets);
  if (debug_level >= 1)
    printf(" do_proc:  procedure '%s' has %d qif sets.\n",
           procedure_name, num_qif_sets);
                                                                          340
/*
  XXX Somewhere around here I could use a step to effectively
  insert qinfluence statements as necessary for qints which don't
  have any qinfluences otherwise.

  Or, I could just leave it as is and make the programmer explicitly
  put in qinfluence statements.
*/

  tp->map(find_qinfluence_sets, &num_qinfluence_sets);                    350
  if (debug_level >= 1)
    printf(" do_proc:  procedure '%s' has %d outermost qinfluence sets.\n",
           procedure_name, num_qinfluence_sets);
}

/*************************************************************************/

void delete_duplicate_sets(char *k_annotation_type)
{
  annote_list *annotes = global_symbol_table->annotes();                  360
  annote *an;
  CStringSet set;
  int non_unique_sets = 0;

  while ((an = annotes->get_annote(k_annotation_type)) != NULL)
    {
      CStringSet *pTmpSet = pIL2pCStringSet(an->immeds());
      char *rep = pCStringSet2pChar(pTmpSet); delete pTmpSet;
      set.add(rep);
      if (debug_level >= 2)                                               370
          printf("  delete_duplicate_sets(\"%s\"):  found {%s}.\n",
                 k_annotation_type, rep);
      non_unique_sets++;
    }

  if (debug_level >= 1)
    {
      printf(" delete_duplicate_sets(\"%s\"):  total %d sets, %d unique.\n",
             k_annotation_type, non_unique_sets, set.length());
    }                                                                     380

  Pix index = set.first();
  while (index != 0)
    {
      static UID = 0;

      CStringSet *pTmpSet = pChar2pCStringSet(set(index));
      if (debug_level >= 2)
```

```
        printf("  delete_duplicate_sets(\"%s\"):  putting back {%s}.\n",
              k_annotation_type, set(index));                                   390
    immed_list *il = pCStringSet2pIL(pTmpSet);
    il->push(new immed_list_e(*(new immed(UID++))));
    global_symbol_table->append_annote
        (k_annotation_type, (void *) il);
    delete pTmpSet;
    set.next(index);
  }
}

/*************************************************************************/      400

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

  process_command_line(argc, argv);
  if (argc-optind != 2)
    {
      fprintf(stderr, "%s:  wrong number of filenames given on command line.\n",  410
              argv[0]);
      exit(1);
    }

#define WRITE_BACK_TO_FILE TRUE
  my_suif_proc_iter(argc, argv, do_proc, before_hook, after_hook,
                    WRITE_BACK_TO_FILE);
}

/*************************************************************************/      420
```

D.2 add_profiling.cc

```
#include "add_profiling.h"

/*************************************************************************/
/* Definitions */
/***************/

int global_argc;
char **global_argv;
int max_UID_encountered = 0;                                                     10

/*************************************************************************/
/* Code */
/********/

// [....]

/****************/

void instrument_returns(tree_node *this_node, void *x)                          20
{
  // Converts "ret e1" to "temp = e1; ...insert_tn...; return temp".
```

```
  if (! this_node->is_instr())
    return;

  instruction *this_instr = ((tree_instr *) this_node)->instr();
  if (this_instr->opcode() != io_ret)
    return;

  in_rrr *this_return = ((in_rrr *) this_instr);
  operand return_operand = this_return->src1_op();
  switch (return_operand.kind())
    {
    case OPER_NULL:
      /* Fall through to OPER_SYM case */
    case OPER_SYM:
      this_node->parent()->insert_before
           (((tree_node *) x)->clone(this_node->scope()), this_node->list_e());
      break;
    case OPER_INSTR:
      {
           assert(return_operand.is_expr());

           block b1 = block::new_sym(return_operand.type());
           sym_node *temp_sym = b1.get_sym();

           block b2(b1 = block(return_operand.clone(this_node->scope())));
           tree_node_list *tnl = b2.make_tree_node_list();
           this_node->parent()->insert_before(tnl, this_node->list_e());
           this_node->parent()->insert_before
             (((tree_node *) x)->clone(this_node->scope()), this_node->list_e());

           return_operand.remove();
           this_return->set_src1((var_sym *) temp_sym);
      }
      break;
    default:
      assert(0); /* Should never be reached */
    }
}

/****************/

void instrument_TNL(tree_proc *tp, tree_node_list *tnl, int UID,
                    char *pre, char *post)
{
  proc_sym *qhead_sym =
    ((global_symtab *) global_symbol_table)->lookup_proc(pre);
  proc_sym *qtail_sym =
    ((global_symtab *) global_symbol_table)->lookup_proc(post);
  assert (qhead_sym != NULL);
  assert (qtail_sym != NULL);

  block::set_proc(tp);
  block qhead(qhead_sym);
  block qtail(qtail_sym);
  block uid(UID);
  block verbose(debug_level);

  block call_qhead;
```

30 40 50 60 70 80

```
    call_qhead.set(block::CALL(qhead,uid,verbose));
  tnl->push(call_qhead.make_tree_node());

  block call_qtail;
    call_qtail.set(block::CALL(qtail,uid,verbose));
  tree_node *tn = call_qtail.make_tree_node();
  tnl->append(new tree_node_list_e(tn));
  if (debug_level >= 3) print_TNL(tnl);
  tnl->map(instrument_returns, tn);                                         90
  if (debug_level >= 3) print_TNL(tnl);
    // In cases where the TNL ends with a return, this code inserts
    // multiple calls to qtail.  Could be fixed.... XXX
}

/****************/

void wrap_main(tree_proc *tp, tree_node_list *tnl,
               char *pre, char *base_filename, int counters, int verbose1,
               char *post, int verbose2)                                    100
{
  /* Similar to instrument_TNL in some ways, but not similar
    enough to make it sensible to combine the two */

  proc_sym *qhead_sym =
    ((global_symtab *) global_symbol_table)->lookup_proc(pre);
  proc_sym *qtail_sym =
    ((global_symtab *) global_symbol_table)->lookup_proc(post);
  assert (qhead_sym != NULL);
  assert (qtail_sym != NULL);                                               110

  type_node *char_array = file_symbol_table->install_type
    (new array_type(type_char, 0, strlen(base_filename)-1));

  var_sym *base_filename_sym = file_symbol_table->new_var
    (char_array, "qp_basefilename");
#define VD_DATA_ALIGNMENT 8
  var_def *base_filename_def =
    new var_def(base_filename_sym,VD_DATA_ALIGNMENT);
  immed_list *il = new immed_list();                                        120
  for (int loop = 0; loop<strlen(base_filename); loop++)
    il->append(new immed_list_e(*(new immed(base_filename[loop]))));
  il->push(new immed_list_e(*(new immed(8))));
  base_filename_def->append_annote(k_multi_init, (void *) il);
  file_symbol_table->add_def(base_filename_def);

  block::set_proc(tp);

  block call_qhead;
    call_qhead.set(block::CALL(new block(qhead_sym),                        130
                               new block(base_filename_sym),
                               new block(counters),
                               new block(verbose1)));
  tnl->push(call_qhead.make_tree_node());

  block call_qtail;
    call_qtail.set(block::CALL(new block(qtail_sym),
                               new block(verbose2)));
  tree_node *tn = call_qtail.make_tree_node();
  tnl->append(new tree_node_list_e(tn));                                    140
```

```
  if (debug_level >= 3) print_TNL(tnl);
  tnl->map(instrument_returns, tn);
  if (debug_level >= 3) print_TNL(tnl);
    // In cases where the TNL ends with a return, this code inserts
    // multiple calls to qtail.  Could be fixed.... XXX

  /* XXX: deal with calls to exit(); extra headache: multiple counters
    may still be "ticking".  I could rewrite qp_dump() to deal with
    that, I suppose. */
}                                                                    150

/*********************************************************************/

int lookup_set_UID(annote *the_an, char *k_annotation_type)
{
  CStringSet *the_set = pIL2pCStringSet(the_an->immeds());
  annote_list_iter anli = annote_list_iter(global_symbol_table->annotes());
  while (! anli.is_empty())
    {
      annote *an = anli.step();                                      160
      if (an->name() == k_annotation_type)
          {
            immed_list *il = an->immeds();
            immed_list_e *ile = new immed_list_e(il->pop());
            assert(ile->contents.is_integer());
            CStringSet *set = pIL2pCStringSet(il);
            il->push(ile);
            if (*set == *the_set)
              {
                int UID = ile->contents.integer();                   170
                if (UID > max_UID_encountered)
                     max_UID_encountered = UID;
                return UID;
              }
          }
    }
  assert(0); /* Shouldn't ever be reached */
  return -1; /* Make compiler happy */
}
                                                                     180
/*********************************************************************/

void check_block(tree_node *t, tree_proc *tp)
{
  annote *an_statement = NULL;
  annote *an_set = NULL;
  int UID = 0;

  switch (t->kind())
    {                                                                190
    case TREE_IF:
      an_statement = t->annotes()->peek_annote(k_qif_statement);
      if (an_statement == NULL)
           return; /* Not a quasistatic if at all */
      an_set = t->annotes()->peek_annote(k_qif_set);
      if (an_set == NULL)
           return; /* Not the first quasistatic if in a chain */
      (void) t->annotes()->get_annote(k_qif_set_analyzed);
        /* The k_qif_set_analyzed annotation was useful during the
```

```
                stage of calculating the set, but is now useless.   The              200
                get_annote function is a mutator, not just observer.  */
        UID = lookup_set_UID(an_set, k_qif_set);
        instrument_TNL(tp, ((tree_if *) t)->then_part(),
                        UID, "qp_start", "qp_stop");
        while (1)
            {
              tree_node_list *pElseTNL = ((tree_if *) t)->else_part();
              if (pElseTNL->count() == 0)
                break; /* "... NO FINAL QELSE" */
              if ((pElseTNL->count() == 1)                                       210
                  && (pElseTNL->head()->contents->is_block()))
                {
                  instrument_TNL(tp, pElseTNL, UID, "qp_start", "qp_stop");
                  break;
                }
              assert(pElseTNL->count() > 1);
              tree_node *pTN = pElseTNL->head()->next()->contents;
              if (pTN->is_if()
                  && (pTN->annotes()->peek_annote(k_qif_statement) != NULL))
                {                                                                220
                  t = pTN;
                  instrument_TNL(tp, ((tree_if *) t)->then_part(),
                                  UID, "qp_start", "qp_stop");
                  /* Do *NOT* break here */
                }
              else /* "... FINAL QELSE" */
                {
                  instrument_TNL(tp, pElseTNL, UID, "qp_start", "qp_stop");
                  break;
                }                                                                230
            }
        break;
      case TREE_BLOCK:
        an_statement = t->annotes()->peek_annote(k_qinfluence_statement);
        if (an_statement == NULL)
            return; /* Not a qinfluence statement at all */
        an_set = t->annotes()->peek_annote(k_qinfluence_set);
        assert(an_set != NULL);
        UID = lookup_set_UID(an_set, k_qinfluence_set);
        instrument_TNL(tp, ((tree_block *) t)->body(),                           240
                        UID, "qp_start", "qp_stop");
        break;
      default:
        ; /* Do nothing */
      }
}

/**********************************************************************/

void do_proc(tree_proc *tp)                                                      250
{
  if (file_symbol_table == NULL)
    {
      file_symbol_table = tp->scope();
      while (file_symbol_table->kind() != SYMTAB_FILE)
          file_symbol_table = file_symbol_table->parent();
    }
  if (global_symbol_table == NULL)
```

```
    {
      global_symbol_table = file_symbol_table;                                        260
      while (global_symbol_table->kind() != SYMTAB_GLOBAL)
          global_symbol_table = global_symbol_table->parent();
    }

  tp->map(check_block, tp);
  /* Important that tp->map comes before a call to wrap_main,
    in case main itself needs to be instrumented. */

  if (strcmp(tp->proc()->name(),"main") == 0)
    wrap_main(tp, tp->body(),                                                         270
             "qp_initialize", global_argv[global_argc-1],
                max_UID_encountered+1, debug_level,
             "qp_dump", debug_level);
}

/*********************************************************************/

int main(int argc, char *argv[])
{
  global_argc = argc;                                                                 280
  global_argv = argv;

  start_suif(argc, argv);
  register_qannotations();

  process_command_line(argc, argv);
  if (argc-optind != 2)
    {
      fprintf(stderr, "%s:  wrong number of filenames given on command line.\n",
              argv[0]);                                                               290
      exit(1);
    }

#define WRITE_BACK_TO_FILE TRUE
  my_suif_proc_iter(argc, argv, do_proc, NULL, NULL, WRITE_BACK_TO_FILE);
}

/*********************************************************************/
```

D.3 select_alternatives.cc

```
#include "select_alternatives.h"

/*********************************************************************/
/* Definitions */
/***************/

char *profile_data_filename = NULL;

/*********************************************************************/                10
/* Code */
/********/

// [....]
```

```
/*************************************************************************/

boolean qif_constraints_met(CStringSet *sets[], int set_count,
                            CStringSet *on_set)
{                                                                              20
 int loop;
 for (loop = 0; loop < set_count; loop++)
  {
   CStringSet *set = sets[loop];
   Pix index;
   int vars_on = 0;

   if (set->empty())
       continue; /* Bizarre, but we'll call it OK */
   index = set->first();                                                       30
   while (index != 0)
       {
        char *qif_var = (*set)(index);
        if (on_set->contains(qif_var))
         vars_on++;
        set->next(index);
       }
   if ((set->contains("... NO FINAL QELSE")) && (vars_on == 1))
       continue; /* OK so far */
   if ((set->contains("... FINAL QELSE")) && (vars_on <= 1))                   40
       continue; /* OK so far */
   return FALSE; /* Won't be reached if constraints are being met.
                    Also note this WILL be reached if the set
                      doesn't contain one of the possible qelse's. */
  }
 return TRUE;
p}

/****************/
                                                                               50
// [....]

/*************************************************************************/

void get_current_decision(CStringSet **return_qif_decision,
                          CStringSet **return_qint_decision)
{
 annote *an =
  global_symbol_table->annotes()->peek_annote(k_qdecision_current);
 if (an == NULL)                                                               60
  {
   (*return_qif_decision) = NULL;
   (*return_qint_decision) = NULL;
   return;
  }

 CStringSet *decision = pIL2pCStringSet(an->immeds());
 CStringSet *qif_decision = new CStringSet();
 CStringSet *qint_decision = new CStringSet();
 Pix index = decision->first();                                                70
 while (index != 0)
  {
   char *str = (*decision)(index);
```

```
    char *equal = strchr(str, '=');
    if (equal == NULL)
        qif_decision->add(str);
    else
        qint_decision->add(str);
    decision->next(index);
  }                                                                   80
  (*return_qif_decision) = qif_decision;
  (*return_qint_decision) = qint_decision;
}

/****************/

void set_current_decision(CStringSet *all_qif_vars, CStringSet *all_qint_vars,
                          CStringSet *qif_decision, CStringSet *qint_decision)
{
  annote_list *annotes = global_symbol_table->annotes();              90
  if (annotes->get_annote(k_qdecision_current) != NULL)
   {
    if (debug_level >= 1)
        printf(" set_current_decision:  error?  called when "
            "there was existing decision.\n");
   }

  CStringSet *decision = new CStringSet();
  (*decision) |= (*qif_decision);
  (*decision) |= (*qint_decision);                                    100
  global_symbol_table->append_annote
   (k_qdecision_current, pCStringSet2pIL(decision));

  /* Implement qif definitions */
  Pix index = all_qif_vars->first();
  while (index != 0)
   {
    var_sym *vs = global_symbol_table->lookup_var((*all_qif_vars)(index));
    assert(vs != NULL);
    var_def *vd = file_symbol_table->lookup_var_def(vs);              110
    assert(vd != NULL);
    (void) vd->get_annote(k_repeat_init); /* strip off any existing */
    if (qif_decision->contains((*all_qif_vars)(index)))
        {
         immed_list_e *ile1 = new immed_list_e(*(new immed(1)));
         immed_list_e *ile2 = new immed_list_e(*(new immed(32)));
         immed_list_e *ile3 = new immed_list_e(*(new immed(1)));
         immed_list *il = new immed_list();
         il->append(ile1); il->append(ile2); il->append(ile3);
         vd->append_annote(k_repeat_init, il);                        120
        }
    all_qif_vars->next(index);
   }

  /* Implement qint definitions */
  index = all_qint_vars->first();
  while (index != 0)
   {
    char *value_str;
    var_sym *vs = global_symbol_table->lookup_var((*all_qint_vars)(index));   130
    assert(vs != NULL);
    var_def *vd = file_symbol_table->lookup_var_def(vs);
```

```
    assert(vd != NULL);
    (void) vd->get_annote(k_repeat_init); /* strip off any existing */
    if ((value_str = retrieve_rhs(qint_decision,(*all_qint_vars)(index),'='))
          != NULL)
        {
          immed_list_e *ile1 = new immed_list_e(*(new immed(1)));
          immed_list_e *ile2 = new immed_list_e(*(new immed(32)));
          immed_list_e *ile3 = new immed_list_e(*(new immed(atoi(value_str))));      140
          immed_list *il = new immed_list();
          il->append(ile1); il->append(ile2); il->append(ile3);
          vd->append_annote(k_repeat_init, il);
        }
    all_qint_vars->next(index);
  }
}

/****************/
                                                                                  150
boolean decision_already_tried(CStringSet *qif_decision,
                               CStringSet *qint_decision)
{
  /* This function strongly begs to have some caching.... */

  CStringSet this_decision;
  this_decision |= (*qif_decision);
  this_decision |= (*qint_decision);

  annote_list_iter anli = annote_list_iter(global_symbol_table->annotes());       160
  while (! anli.is_empty())
   {
    annote *an = anli.step();
    if (an->name() == k_qdecision_history)
        {
          immed_list *il = an->immeds();
          immed_list_e *ile = new immed_list_e(il->pop());
          CStringSet *past_decision = pIL2pCStringSet(il);
          il->push(ile);
          if ((*past_decision) == this_decision)                                  170
            {
              delete past_decision;
              return TRUE;
            }
          else
            delete past_decision;
        }
   }
  return FALSE;
}                                                                                 180

/**********************************************************************/

CStringSet *dumpfile2pCStringSet(char *filename)
{
  CStringSet *set = new CStringSet();

  if (debug_level >= 2)
    printf("  dumpfile2pCStringSet:  called on file '%s'.\n", filename);
  FILE *file = fopen(filename, "r");                                              190
  if (file == NULL)
```

```
  {
    if (debug_level >= 1)
        printf(" dumpfile2pCStringSet:  error!  Could not open file.    "
               "Returning empty set.\n");
    return set;
  }
unsigned int int_buffer;

fread(&int_buffer, sizeof(unsigned int), 1, file);                                200
if (int_buffer != 0xEFBEADDE)
  {
    if (debug_level >= 1)
        printf(" dumpfile2pCStringSet:  error!  file doesn't start with magic "
               "cookie value.    Returning empty set.\n");
    return set;
  }
fread(&int_buffer, sizeof(unsigned int), 1, file);
if (int_buffer != 0x00000001)
  {                                                                               210
    if (debug_level >= 1)
        printf(" dumpfile2pCStringSet:  error!  file version not understood "
               "by this program.    Returning empty set.\n");
    return set;
  }
fread(&int_buffer, sizeof(unsigned int), 1, file);
fseek(file, 0, SEEK_END);
int file_size = ftell(file);

if (file_size % (3*sizeof(unsigned int) +                                         220
                 int_buffer*sizeof(unsigned long long int)) != 0)
  {
    if (debug_level >= 1)
        printf(" dumpfile2pCStringSet:  error!  file wrong size.    "
               "Returning empty set.\n");
    return set;
  }

fseek(file, 0, SEEK_SET);
                                                                                  230
unsigned long long int *totals = (unsigned long long int *)
  calloc(int_buffer, sizeof(unsigned long long int));
assert(totals != NULL);

int loop;
while (! feof(file))
  {
    fseek(file, sizeof(unsigned int)*3, SEEK_CUR);
    unsigned long long int *pass = (unsigned long long int *)
        malloc(int_buffer * sizeof(unsigned long long int));                      240
    assert(pass != NULL);
    if (fread(pass,sizeof(unsigned long long int),int_buffer, file)
          != int_buffer)
        {
          if (debug_level >= 1)
            printf(" dumpfile2pCStringSet:  error!    Problem reading file.    "
                   "Returning empty set.\n");
          return set;
        }
    for (loop = 0; loop < int_buffer; loop++)                                     250
```

```
        totals[loop] += pass[loop];
    free(pass);
    ungetc(getc(file),file); /* Force EOF condition */
  }

  fclose(file);
  if (unlink(filename) != 0)
    fprintf(stderr, "Error!  Could not remove profile data file.\n");
  else
    {                                                                         260
    if (debug_level >= 1)
        printf("  dumpfile2pCStringSet:  unlinked profile data file.\n");
    }

  char sprintf_buffer[1024], lltoa_buffer[1024];
  for (loop = 0; loop < int_buffer; loop++)
    {
    if (totals[loop] != 0)
        {
         lltoa(totals[loop], lltoa_buffer);                                  270
         sprintf(sprintf_buffer, "%u=%s", loop, lltoa_buffer);
         set->add(strdup(sprintf_buffer));
        }
    }

  if (debug_level >= 2)
    printf("  dumpfile2pCStringSet:  returning {%s}.\n",
           pCStringSet2pChar(set));
  free(totals);
  return set;                                                                280
}

/****************/

void move_to_history(void)
{
  annote_list *annotes = global_symbol_table->annotes();
  annote *an = annotes->get_annote(k_qdecision_current);

  if (an == NULL)                                                            290
    {
    if (debug_level >= 1)
        printf(" move_to_history:  no current decision to be moved.\n");
    return;
    }

  immed_list *il = an->immeds();
  if (debug_level >= 1)
    printf(" temp...:  moving decision {%s} to history.\n",
           pCStringSet2pChar(pIL2pCStringSet(il)));                          300

  var_sym *sym = file_symbol_table->lookup_var("qp_basefilename");
  assert(sym != NULL);
  var_def *def = file_symbol_table->lookup_var_def(sym);
  assert(def != NULL);
  an = def->annotes()->peek_annote(k_multi_init); //  reuse of variable "an"
  assert(an != NULL);
  char *basefilename = (char *) malloc(an->immeds()->count());
    //  basefilename needs an extra byte of storage space for
```

```
  // the terminating '\0', but an->immeds() has one more immediate          310
  // than is going to get converted back to string representation anyway

 immed_list_e *ile = an->immeds()->head();
 assert(ile->contents.is_integer() && (ile->contents.integer() == 8));
 ile = ile->next();
 int loop = 0;
 while (ile != NULL)
  {
   basefilename[loop++] = (char) ile->contents.integer();
   ile = ile->next();                                                       320
  }
 basefilename[loop] = '\0';
 char buffer[1024];
 sprintf(buffer, "%s.qprofiling_data", basefilename);

 il->push(new immed_list_e(*(new immed
  (pCStringSet2pChar(dumpfile2pCStringSet((profile_data_filename != NULL) ?
  profile_data_filename: buffer))))));
 global_symbol_table->append_annote(k_qdecision_history, il);
}                                                                           330

/*********************************************************************/

int load_qif_sets(CStringSet ***return_qif_sets)
{
 int sets_count = 0;
 CStringSet **qif_sets;

 annote_list_iter anli = annote_list_iter(global_symbol_table->annotes());
 while (! anli.is_empty())                                                  340
  {
   annote *an = anli.step();
   if (an->name() == k_qif_set)
       {
        sets_count++;
        if (sets_count == 1)
          assert((qif_sets = (CStringSet **) malloc(sizeof(CStringSet **)))
                 != NULL);
        else
          assert((qif_sets = (CStringSet **)                                350
                  realloc(qif_sets, sets_count * sizeof(CStringSet **)))
                 != NULL);
        immed_list *il = an->immeds();
        immed_list_e *ile = new immed_list_e(il->pop());
        qif_sets[sets_count-1] = pIL2pCStringSet(il);
        il->push(ile);
       }
  }
 (*return_qif_sets) = qif_sets;
 return sets_count;                                                         360
}

/***************/

CStringSet *load_qint_set(void)
{
 CStringSet *qint_set = new CStringSet();
```

```
  annote_list_iter anli = annote_list_iter(global_symbol_table->annotes());
  while (! anli.is_empty())                                                    370
    {
      annote *an = anli.step();
      if (an->name() == k_qinfluence_set)
          {
            immed_list *il = an->immeds();
            immed_list_e *ile = new immed_list_e(il->pop());
            (*qint_set) |= (*pIL2pCStringSet(il));
            il->push(ile);
          }
    }                                                                          380
  return (qint_set);
}

/****************/

char *retrieve_rhs(CStringSet *set, char *key, char divider)
{
  /* I could use libg++'s Map containers for this, but the hassle
     may not be worthwhile.... */
                                                                               390
  Pix index = set->first();
  while (index != 0)
    {
      char *str = (*set)(index);
      if (strncmp(str,key,strlen(key)) == 0)
          return (rindex(str,divider)+1);
      set->next(index);
    }
  return NULL;
}                                                                              400

IntSet *range_string_to_set(char *range_str)
{
  IntSet *set = new IntSet();
  char *token = strtok(strdup(range_str), ",");
  while (token != NULL)
    {
      char *colon_position = strchr(token,':');
      assert(colon_position == strrchr(token,':'));
      if (colon_position == NULL)                                              410
          {
            int singleton;
            sscanf(token,"%d",&singleton);
            set->add(singleton);
          }
      else
          {
            int start,end,loop;
            sscanf(token,"%d:%d",&start,&end);
            if (start>end)                                                     420
              { int tmp = start; start = end; end = tmp; }
            for (loop=start; loop<=end; loop++)
              set->add(loop);
          }
      token = strtok(NULL, ",");
    }
  return set;
```

```
}

int maximum_of_range(IntSet *set)                                              430
{
  /* Not very efficient.... */

  assert(set->length() != 0);
  int maximum;
  Pix index = set->first();
  while (index != 0)
    {
      maximum = (*set)(index);
      set->next(index);                                                        440
    }
  return maximum;
}

int minimum_of_range(IntSet *set)
{
  assert(set->length() != 0);
  return ((*set)(set->first()));
}
                                                                               450
int next_range_value(IntSet *set, int value)
{
  /* Not very efficient.... */

  assert(set->length() != 0);
  Pix index = set->first();
  while (index != 0)
    {
      int current_value = (*set)(index);
      if (current_value > value)                                               460
            return current_value;
      set->next(index);
    }
  assert(0); /* should never be reached */
  return -1; /* Make compiler happy */
}

/****************/

QintDecisionsGenerator::QintDecisionsGenerator(CStringSet *set)                470
{
  /* I'll want QintDecisionsGenerator to deal well with being passed
     empty sets, since that is perfectly valid. */

  this->set = new CStringSet();
  (*(this->set)) |= (*set);
  {
    ranges_set = new CStringSet();
    sym_node_list_iter snli =
      sym_node_list_iter(global_symbol_table->symbols());                      480
    while (! snli.is_empty())
      {
          sym_node *sn = snli.step();
          annote *an = sn->annotes()->peek_annote(k_qint_parameter);
          if (an != NULL)
            {
```

```
        char *buffer = (char *) malloc(1024);
        assert(buffer != NULL);
        strcpy(buffer, sn->name());
        strcat(buffer, "=");                                                    490
        strcat(buffer, an->immeds()->head()->contents.string());
        buffer = (char *) realloc(buffer, strlen(buffer)+1);
        ranges_set->add(strdup(buffer));
        free(buffer);
      }
    }
  }
  current = NULL;
}
                                                                                500
boolean QintDecisionsGenerator::increment_at_position(Pix index)
{

  char *key = (*set)(index);
  assert(key != NULL);
  char *range_str = retrieve_rhs(ranges_set,key,'=');
  char *value_str = retrieve_rhs(current,key,'=');
  assert(range_str != NULL); assert(value_str != NULL);

  int value;                                                                    510
  sscanf(value_str, "%d", &value);
  IntSet *range = range_string_to_set(range_str);

  char *sprintf_buffer = (char *) malloc(1024);
  assert(sprintf_buffer != NULL);
  if (maximum_of_range(range) == value)
   {
    set->seek(key);
    set->next(index);
    if (index == 0)                                                             520
        assert(0); /* should never be reached, since increment_at_position
                      should not have been called if we're done */
    /* Carry over to the next "column" */
    sprintf(sprintf_buffer, "%s=%d", key, value);
    current->del(sprintf_buffer);
    sprintf(sprintf_buffer, "%s=%d", key, minimum_of_range(range));
    current->add(strdup(sprintf_buffer));
     /* Note that this sequence of del() and add() will work
         even if the range only has a singleton value, even though
         it does much more work than strictly necessary. */                    530
    increment_at_position(index);
   }
  else
   {
    /* Increment this "column" */
    int new_value = next_range_value(range, value);
    sprintf(sprintf_buffer, "%s=%d", key, value);
    current->del(sprintf_buffer);
    sprintf(sprintf_buffer, "%s=%d", key, new_value);
    current->add(strdup(sprintf_buffer));                                       540
   }
  free(sprintf_buffer);
}

boolean QintDecisionsGenerator::done(void)
```

```
{
  if (current == NULL)
    return FALSE;
  if (set->length() == 0)
    return TRUE;                                                          550
  Pix index = set->first();
  while (index != 0)
    {
      char *key = (*set)(index);
      assert(key != NULL);
      char *range_str = retrieve_rhs(ranges_set,key,'=');
      char *value_str = retrieve_rhs(current,key,'=');
      assert(range_str != NULL); assert(value_str != NULL);

      int value;                                                          560
      sscanf(value_str, "%d", &value);
      IntSet *range = range_string_to_set(range_str);

      if (value < maximum_of_range(range))
            {
              delete range;
              return FALSE;
            }

      delete range;                                                       570
      set->next(index);
    }
  return TRUE;
}

CStringSet *QintDecisionsGenerator::yield(void)
{
  if (done())
    return NULL;
                                                                          580
  if (current == NULL)
    {
      current = new CStringSet();
      Pix index = set->first();
      while (index != 0)
            {
              char *key = (*set)(index);
              char *range_str = retrieve_rhs(ranges_set, key, '=');
              assert(range_str != NULL);
              IntSet *range = range_string_to_set(range_str);             590
              char *sprintf_buffer = (char *) malloc(1024);
              assert(sprintf_buffer != NULL);
              sprintf(sprintf_buffer, "%s=%d", key, minimum_of_range(range));
              current->add(strdup(sprintf_buffer));
              free(sprintf_buffer);
              set->next(index);
            }
      return shallow_copy_CStringSet(current);
    }
                                                                          600
  increment_at_position(set->first());
  return shallow_copy_CStringSet(current);
  /* There would be considerable rep exposure if I returned current itself. */
}
```

```
QintDecisionsGenerator::~QintDecisionsGenerator()
{
  delete set;
  delete ranges_set;
  delete current;                                                          610
}

// [....]

/****************/

void choose_best_from_history(void)
{
/*
  annote_list_iter anli = annote_list_iter(global_symbol_table->annotes());  620
  while (! anli.is_empty())
    {
      annote *an = anli.step();
      if (an->name() == k_qdecision_history)
           {
             immed_list *il = an->immeds();
             immed_list_e *ile = new immed_list_e(il->pop());
             CStringSet *past_decision = pIL2pCStringSet(il);
             XXX
             il->push(ile);                                                630
           }
    }
*/
  fprintf(stderr, "WARNING!!!    choose_best_from_history() not implemented.\n");
  fprintf(stderr, "(De facto, most recent decision still implemented.)\n");
}

/****************/

void make_decision(void)                                                   640
{
  annote *an =
    global_symbol_table->annotes()->peek_annote(k_qdecision_current);
  assert(an == NULL);

  CStringSet **qif_sets;
  int sets_count = load_qif_sets(&qif_sets);

  if (debug_level >= 1)
    printf(" make_decision:  %d qif sets found.\n", sets_count);          650

  CStringSet *all_qif_vars = new CStringSet();
  int loop;
  for (loop = 0; loop<sets_count; loop++)
    (*all_qif_vars) |= *(qif_sets[loop]);
  all_qif_vars->del("...  NO FINAL QELSE");
  all_qif_vars->del("...  FINAL QELSE");

  CStringSet *all_qint_vars = load_qint_set();
                                                                           660
  /* The following looping constructs are rather inefficient.  Even
    after adding the "(! decision_was_made)" condition to the while
    loops' tests, this is extremely inefficient.  For efficiency, one
```

```
  would want to be able to initialize the SubsetsGenerator and the
  QintDecisionsGenerator from the value of k_qdecision_current
  that was just moved into k_qdecision_history.  However, it's
  probably not worth the effort to make that optimization now.
  Eventually I'll want to rewrite this whole thing anyway, and the
  focus should be on smarts and not on speed.  */
                                                                          670
boolean qif_constraints_meetable = FALSE;
boolean decision_was_made = FALSE;
SubsetsGenerator sg = SubsetsGenerator(all_qif_vars);
while ((! decision_was_made) && (! sg.done()))
 {
  CStringSet *qif_decision = sg.yield();
  if (qif_constraints_met(qif_sets,sets_count,qif_decision))
     {
      qif_constraints_meetable = TRUE;
      if (debug_level >= 2)                                               680
       printf("  make_decision:  {%s} would be a valid qif_decision.\n",
              pCStringSet2pChar(qif_decision));
      if (debug_level >= 2)
       {
        print_QintDecisionsGenerator_results(all_qint_vars);
       }
      QintDecisionsGenerator qd = QintDecisionsGenerator(all_qint_vars);
      while ((! decision_was_made) && (! qd.done()))
       {
        CStringSet *qint_decision = qd.yield();                           690
        if (debug_level >= 2)
            printf("  make_decision:  {%s} would be a valid "
                "qint_decision.\n", pCStringSet2pChar(qint_decision));
        if (decision_was_made)
            {
             if (debug_level >= 2)
              printf("  make_decision:  ...but another decision has "
                     "already been set during this pass.\n");
            }
        else                                                              700
            {
             if (decision_already_tried(qif_decision, qint_decision))
              {
               if (debug_level >= 2)
                   printf("  make_decision:  ...but was already tried.\n",
                        pCStringSet2pChar(qint_decision));
              }
             else
              {
               set_current_decision(all_qif_vars, all_qint_vars,           710
                                    qif_decision, qint_decision);
               decision_was_made = TRUE;
               if (debug_level >= 1)
                   printf(" make_decision:  ...decided on qif {%s} "
                        "and qint {%s}.\n",
                        pCStringSet2pChar(qif_decision),
                        pCStringSet2pChar(qint_decision));
              }
            }
        delete qint_decision;                                             720
       }
     }
```

```
      delete qif_decision;
    }
  if (! qif_constraints_meetable)
    fprintf(stderr,
            "Error!  Not possible to satisfy qif semantic constraints.\n");
  if (! decision_was_made)
    {
      if (debug_level >= 1)                                                    730
          {
            printf(" make_decision:  All qif decisions have been tried.\n");
          }
      choose_best_from_history();
    }
}

/*********************************************************************/

void before_hook(void)                                                         740
{
  /* Do nothing */
}

/****************/

void after_hook(void)
{
  move_to_history();
  make_decision();                                                             750
}

/*********************************************************************/

void do_proc(tree_proc *tp)
{
  /* Wish I could set file_symbol_table and global_symbol_table
    in before_hook() instead, but there isn't anything to grab on
    to cleanly....  The below will get run a lot more than it has
    to, but that shouldn't be a big deal. */                                   760
  if (file_symbol_table == NULL)
    {
      file_symbol_table = tp->scope();
      while (file_symbol_table->kind() != SYMTAB_FILE)
          file_symbol_table = file_symbol_table->parent();
    }
  if (global_symbol_table == NULL)
    {
      global_symbol_table = file_symbol_table;
      while (global_symbol_table->kind() != SYMTAB_GLOBAL)                     770
          global_symbol_table = global_symbol_table->parent();
    }
}

/*********************************************************************/

int main(int argc, char *argv[])
{
  start_suif(argc, argv);
  register_qannotations();                                                     780
```

```
  process_command_line(argc, argv);
  if (argc-optind != 2)
    {
      fprintf(stderr, "%s:  wrong number of filenames given on command line.\n",
              argv[0]);
      exit(1);
    }

  if (debug_level >= 3)                                                          790
    {
      test_SubsetsGenerator();
      test_qif_constraints_met();
      test_QintDecisionsGenerator_aux_routines();
    }

#define WRITE_BACK_TO_FILE TRUE
  my_suif_proc_iter(argc, argv, do_proc, before_hook, after_hook,
                    WRITE_BACK_TO_FILE);
                                                                                 800
#define RENAME_SUCCESS_CODE 0
  if (rename(argv[optind+1], argv[optind]) == RENAME_SUCCESS_CODE)
    {
      if (debug_level >= 1)
          printf(" Successfully renamed '%s' to '%s'.\n",
                 argv[optind+1], argv[optind]);
      char buffer[1024];
      sprintf(buffer, "create_executable %s", argv[optind]);
      system(buffer);
    }                                                                            810
  else
    {
      fprintf(stderr, "%s:  error renaming file, output left in '%s'.\n",
              argv[optind+1]);
      exit(1);
    }
}

/************************************************************************/
                                                                                 820
```

D.4 Profiling Implementation

qprofiling.h

```
extern int qp_initialize(char *base_filename, int counters, int verbose);
extern int qp_dump(int verbose);

#ifdef _GNUC_
extern unsigned int *qp_timestamps_hi;
extern unsigned int *qp_timestamps_lo;
extern unsigned int *qp_counters;
extern unsigned long long int *qp_totals;
extern unsigned int qp_counters_count;                                           10
extern unsigned int qp_tmp_hi;
extern unsigned int qp_tmp_lo;
#define qp_RDTSC ".hword 0x310F" /* Assembly form: db 0Fh 31h */
#endif
```

```
/**********************************************************************/

/* See qprofiling.c for info about how gcc handles the inline keyword. */

#ifdef _GNUC_                                                              20
inline extern void qp_start(int counter, int careful)
{
#if 0
  if (careful) /* Prevents accessing memory past malloc'ed amount */
    counter = counter % qp_counters_count;
#endif
  qp_counters[counter]++;
  if (qp_counters[counter] == 1)
    {
     {                                                                     30
          register unsigned int eax asm ("%eax");
          register unsigned int edx asm ("%edx");

          asm volatile (qp_RDTSC);
          qp_tmp_hi = edx; qp_tmp_lo = eax;
     }
     qp_timestamps_hi[counter] = qp_tmp_hi;
     qp_timestamps_lo[counter] = qp_tmp_lo;
    }
}                                                                          40
#else
extern void qp_start(int counter, int careful);
#endif

/**********************************************************************/

#ifdef _GNUC_
inline extern void qp_stop(int counter, int careful)
{
#if 0                                                                      50
  if (careful) /* Prevents accessing memory past malloc'ed amount */
    counter = counter % qp_counters_count;
#endif
  if (qp_counters[counter] == 1)
    {
     /* Diff timestamps[counter] with the current time, and
          add to qp_totals[counter] */
     register unsigned int eax asm ("%eax");
     register unsigned int edx asm ("%edx");
                                                                           60
     asm volatile (qp_RDTSC);
     qp_tmp_hi = edx; qp_tmp_lo = eax;

/*
      printf("Start (EDX:EAX): %x:%x.\n",
                qp_timestamps_hi[counter], qp_timestamps_lo[counter]);
      printf("Finish (EDX:EAX): %x:%x.\n",
                qp_tmp_hi, qp_tmp_lo);
*/
                                                                           70
     if (qp_tmp_hi == qp_timestamps_hi[counter])
          /* Common case */
          qp_totals[counter] += (qp_tmp_lo - qp_timestamps_lo[counter]);
```

```
    else
        {
         if (qp_tmp_lo < qp_timestamps_lo[counter])
           /* Need to borrow to perform subtraction... */
           qp_totals[counter] +=
             (((unsigned long long int)1<<32) - qp_timestamps_lo[counter])
             + qp_tmp_lo                                                        80
             + (((unsigned long long int)1<<32)
                 * ((--qp_tmp_hi) - qp_timestamps_hi[counter]));
         else
           qp_totals[counter] +=
             (qp_tmp_lo - qp_timestamps_lo[counter])
             + (((unsigned long long int)1<<32)
                 * (qp_tmp_hi - qp_timestamps_hi[counter]));
        }
    qp_counters[counter]--;
    return;                                                                     90
   }
  else if (qp_counters[counter] > 1)
   {
    qp_counters[counter]--;
    return;
   }
  else /* qp_counters[counter] < 0 */
   qp_counters[counter] = 0;
}
#else                                                                          100
extern void qp_stop(int counter, int careful);
#endif

/********************************************************************/
```

qprofiling.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/********************************************************************/

unsigned int *qp_timestamps_hi = NULL;
unsigned int *qp_timestamps_lo = NULL;
unsigned int *qp_counters = NULL;                                               10
unsigned long long int *qp_totals = NULL;
unsigned int qp_counters_count;
unsigned int qp_tmp_hi, qp_tmp_lo;
char *qp_basefilename = NULL;
char *qp_filename = NULL;
const qp_SUCCESS = 1;
const qp_FAILURE = 0;
#define qp_RDTSC ".hword 0x310F" /* Assembly form: db 0Fh 31h */

/********************************************************************/             20

int qp_initialize(char *base_filename, int counters, int verbose)
{
```

```
  FILE *file;

  qp_counters_count = counters;
  if (((qp_timestamps_hi = (unsigned int *)
          calloc(counters,sizeof(unsigned int))) == NULL)
     || ((qp_timestamps_lo = (unsigned int *)
          calloc(counters,sizeof(unsigned int))) == NULL)
     || ((qp_counters = (unsigned int *)
          calloc(counters,sizeof(unsigned int))) == NULL)
     || ((qp_totals = (unsigned long long int *)
          calloc(counters,sizeof(unsigned long long int))) == NULL))
    {
     if (verbose)
          fprintf(stderr, _FILE_ ":  failed allocating profiling buffers.\n");
     return qp_FAILURE;
    }

  qp_basefilename = strdup(base_filename);
  if ((qp_filename = (char *) malloc(1024)) == NULL)
    return qp_FAILURE;
  sprintf(qp_filename, "%s.qprofiling_data", qp_basefilename);
  qp_filename = (char *) realloc(qp_filename, strlen(qp_filename)+1);

  file = fopen(qp_filename,"a+");
  if (file == NULL)
    {
     if (verbose)
          fprintf(stderr, _FILE_ ":  failed opening output file '%s'.\n",
                  qp_filename);
     return qp_FAILURE;
    }
  else
    fclose(file);
  return qp_SUCCESS;
}

/*****************************************************************/

/* Adapted from itoa(), K&R second edition, page 64 */
void lltoa(unsigned long long int n, char s[])
{
  int i = 0;
  do {
    s[i++] = (n % 10) + '0';
  } while ((n /= 10) > 0);
  s[i] = '\0';
  {
    int c,i,j;
    for (i = 0, j = strlen(s)-1; i<j; i++, j--)
      {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
      }
  }
}

/****************/
```

```
int qp_dump(int verbose)
{
  FILE *file;
  unsigned int int_buffer;
  long int new_file_size;
  int output_problem = 0;
  int bytes_written = 0;
  int runs = 0;                                                           90

  /* Diagnostic output */
  if (verbose)
    {
      int some_output_flag = 0;
      int i;
      for (i=0; i<qp_counters_count; i++)
            {
              if (qp_totals[i] != 0)
                {                                                         100
                  static char buffer[1024];
                  lltoa(qp_totals[i], buffer);
                  fprintf(stderr,"Slot %u:  %s.  ", i, buffer);
                  some_output_flag++;
                }
            }
      if (some_output_flag)
            {
              fprintf(stderr,"\n");
            }                                                             110
      else
            {
              fprintf(stderr,"All profiling slots had zero values.\n");
            }
    }

  /* WARNING: at this time, this code does not do *ANYTHING* about
     file locking.  If two copies of a profiled program start up and
     are running at the same time, eit. */
                                                                          120
  file = fopen(qp_filename,"a+");
  if (file == NULL)
    {
      if (verbose)
            fprintf(stderr, __FILE__ ":  failed opening output file '%s'.\n",
                        qp_filename);
      return qp_FAILURE;
    }

#define qp_OUTPUT(addr,obj_size,obj_count) \                             130
  if (fwrite(addr,obj_size,obj_count,file) < obj_count) \
    output_problem++; \
  else \
    bytes_written += obj_size * obj_count;

  int_buffer = 0xEFBEADDE; /* Magic cookie */
  qp_OUTPUT(&int_buffer, sizeof(unsigned int), 1);

  int_buffer = 0x00000001; /* File format version identifier */
  qp_OUTPUT(&int_buffer, sizeof(unsigned int), 1);                        140
```

```
  int_buffer = qp_counters_count;
  qp_OUTPUT(&int_buffer, sizeof(unsigned int), 1);

  qp_OUTPUT(qp_totals, sizeof(unsigned long long int), qp_counters_count);

  fflush(file);
  new_file_size = ftell(file);
  if (fclose(file) != 0) output_problem++;
                                                                                150
  runs = atoi(getenv("RUNS"));
  if (runs == 0)
    runs = 4;

  if (new_file_size >= (runs * bytes_written))
    {
      char buffer[1024];
      sprintf(buffer,"select_alternatives %s -p %s %s %s.s_a.temp",
              (verbose == 0) ? "" : "-d",
              qp_filename, qp_basefilename, qp_basefilename);           160
      if (verbose)
           fprintf(stderr, "Accumulated enough results, "
                   "invoking select_alternatives....\n");
      system(buffer);
    }

  return (output_problem == 0)? qp_SUCCESS : qp_FAILURE;
}

/*********************************************************************   170
```

Library documentation (draft)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The "qp_" prefix is an attempt to reduce the chance of name collision.

*==== int qp_initialize(char *base_filename, int counters, int verbose)*

Performs any set—up necessary. Filename argument is used as the base name for the data file to store profiling output; qp_initialize may check immediately if such a file is writeable. Counters argument indicate how many different "stopwatch" counters will be allocated. Function returns qp_SUCCESS if it is successful, qp_FAILURE if there is a problem. In the latter case, the function may write diagnostic information out to stderr if the verbose argument is nonzero, and further calls to qp_* functions will result in undefined program behavior. [Alternatively, we could have the other qp_* functions defined to do nothing if qp_initialize fails; that would make for nicer semantics -- the program will still work, even if saving profiling data is going to fail -- but at the price of greater run—time overhead, since the implementation which comes immediately to mind is to set a flag which is checked every time qp_start and qp_stop is called. Indirecting calls to qp_start and qp_stop through global variables which qp_initialize could set to different functions strikes me as roughly the same magnitude of overhead (admittedly, small).]

==== int qp_dump(int verbose)

Writes out profiling data to file designated by filename argument to qp_initialize. This function should only be called once; if it is called multiple times, each time it will overwrite any data previously

180 190 200

```
written-out during this program execution. Function returns
qp_SUCCESS if it is successful, qp_FAILURE if there was a problem. In
the latter case, the function may write diagnostic information to
stderr if the verbose argument is nonzero.

 ==== void qp_start(int counter, int careful)
 ==== void qp_stop(int counter, int careful)

For any given value for the counter argument, the time elapsed between
the first call to qp_start and the matching call (for example, think          210
matching parentheses) to qp_stop is recorded. The meaning of elapsed
time is purposely left undefined here; in particular, it may mean wall
clock time *OR* the equivalent of UNIX user+system time, or something
completely different. Calls to these functions with different values
for the counter argument may be freely interleaved. If the careful
argument is non-zero, some sanity checking will be performed: the
value for the counter argument will be bound-checked against the value
of the counters argument to qp_initialize, and an error message will
be issued to stderr if the number of calls to qp_stop for a given
counter value exceeds the number of calls to qp_start.                       220

************************************************************************/

// [....]
```

D.5 Auxillary routines

```
#include "pshuang.h"

/************************************************************************/
/* Definitions */
/************************************************************************/

int debug_level = 0;
base_symtab *file_symbol_table = NULL;
base_symtab *global_symbol_table = NULL;                                     10

char *k_qif_statement;
char *k_qif_set_analyzed;
char *k_qif_set;
char *k_qif_variable;
char *k_qif_variable_defined;

char *k_qint_parameter;
char *k_qinfluence_statement;
char *k_qinfluence_set;                                                      20

char *k_qdecision_history;
char *k_qdecision_current;

/************************************************************************/
/* Code */
/************************************************************************/

/* The code for my_suif_proc_iter was lifted from suif/include/suif/misc.cc.
  I needed the functionality of being able to call a function after all      30
```

```
  the SUIF file proc's had been processed, so I threw in the ability
  to call a function before, too.  Modifications are clearly marked. */

void my_suif_proc_iter(int argc, char * argv[], prociter_f fun,
                       /* ADDED */ void (*before_hook)(void),
                       /* ADDED */ void (*after_hook)(void),
                       boolean writeback,
                       boolean exp_trees,
                       boolean use_fortran_form)
{

// [....]

   /* ADDED */ if (before_hook != NULL) (*before_hook)();

// [....]

   /* ADDED */ if (after_hook != NULL) (*after_hook)();

// [....]

}

/*********************************************************************/

void attach_annotation_string(suif_object *obj,
                              char *the_annotation,
                              char *string_data)
{
  immed *i = new immed(string_data);
  immed_list_e *ile = new immed_list_e(*i);
  immed_list *il = new immed_list();
  il->push(ile);

  obj->append_annote(the_annotation, (void *) il);
}

/*********************************************************************/

immed_list *pCStringSet2pIL(CStringSet *pSet)
{
  Pix index;
  immed_list *il = new immed_list();
  if (pSet == NULL)
    return il;

  index = pSet->first();
  while (index != 0)
    {
      immed *i = new immed((*pSet)(index));
      immed_list_e *ile = new immed_list_e(*i);
      il->push(ile);
      pSet->next(index);
    }
  return il;
}

CStringSet *pIL2pCStringSet(immed_list *il)
{
```

```
  CStringSet *pSet = new CStringSet();                                         90

  immed_list_iter *iter = new immed_list_iter(il);
  while (! iter->is_empty())
    {
      immed i = iter->step();
      if (i.is_string())
           pSet->add(strdup(i.string()));
    }

  return pSet;                                                                 100
}

/*******************************************************************/

char *pCStringSet2pChar(CStringSet *pSet)
{
  if (pSet == NULL)
    return NULL; /* Seems fitting, no? */

#define INITIAL_SIZE 1024                                                      110
#define INCREMENT 1024
  int length_used = 0;
  int buffer_size = INITIAL_SIZE;
  char *buffer = (char *) malloc(INITIAL_SIZE);
  assert(buffer != NULL);
  buffer[0] = 0;

  Pix index;
  index = pSet->first();
  while (index != 0)                                                           120
    {
      char *str = (*pSet)(index);
      int len = strlen(str);

      while ((length_used + (len+3)) >= buffer_size)
           /* The constant 3 is for the two single quotes and potential comma */
           {
             buffer_size = buffer_size + INCREMENT;
             buffer = (char *) realloc(buffer, buffer_size+1); /* +1 for '\0' */
             assert(buffer != NULL);                                           130
           }
      sprintf((char *)(buffer+length_used), "'%s'", str);
      length_used = length_used + (len+2);
      pSet->next(index);
      if (index != 0)
           {
             sprintf((char *)(buffer+length_used), ",");
             length_used++;
           }
    }                                                                          140
  buffer = (char *) realloc(buffer, length_used+1); /* +1 for '\0' */
  assert(buffer != NULL);
  return buffer;
}

CStringSet *pChar2pCStringSet(char *input_str)
{
  /* This function isn't meant to be able to handle the general case
```

```
   of converting strings to CStringSet's; it can handle the output
   format of pCStringSet2pChar, provided that none of the constituent          150
   strings contains single quotes and/or commas. */
  CStringSet *pSet = new CStringSet();
  char *str = strdup(input_str);
  char *ptr = strtok(str, "',");
  while (ptr != NULL)
   {
    pSet->add(strdup(ptr));
    ptr = strtok(NULL, "',");
   }
  return pSet;                                                                 160
}

/**********************************************************************/

void register_qannotations(void)
{
#define WRITE_ANNOTATION TRUE
  ANNOTE(k_qif_statement, "QIF STATEMENT", WRITE_ANNOTATION);
  ANNOTE(k_qif_set_analyzed, "QIF SET ANALYZED", WRITE_ANNOTATION);
  ANNOTE(k_qif_set, "QIF SET", WRITE_ANNOTATION);                              170
  ANNOTE(k_qif_variable, "QIF VARIABLE", WRITE_ANNOTATION);
  ANNOTE(k_qif_variable_defined, "QIF VARIABLE DEFINED", WRITE_ANNOTATION);

  ANNOTE(k_qint_parameter, "QINT PARAMETER", WRITE_ANNOTATION);
  ANNOTE(k_qinfluence_statement, "QINFLUENCE STATEMENT", WRITE_ANNOTATION);
  ANNOTE(k_qinfluence_set, "QINFLUENCE SET", WRITE_ANNOTATION);

  ANNOTE(k_qdecision_history, "QDECISION HISTORY", WRITE_ANNOTATION);
  ANNOTE(k_qdecision_current, "QDECISION CURRENT", WRITE_ANNOTATION);
}                                                                              180

/**********************************************************************/

SubsetsGenerator::SubsetsGenerator(CStringSet *set)
{
  this->set = new CStringSet();
  (*(this->set)) |= (*set);
  current = NULL;
}
                                                                               190
boolean SubsetsGenerator::done()
{
  if (current == NULL) return FALSE;
  return (((*set) == (*current)) ? TRUE : FALSE);
}

void SubsetsGenerator::increment_at_position(Pix index)
{
  char *var = (*set)(index);
  assert(var != NULL);                                                         200
  if (current->contains(var))
   {
    set->seek(var);
    set->next(index);
    if (index == 0)
        return; /* Current is already at 11...11 */
    current->del(var);
```

```
    increment_at_position(index);
   }
 else                                                              210
   {
    current->add(var);
   }
}

CStringSet *shallow_copy_CStringSet(CStringSet *set)
{
 CStringSet *new_set = new CStringSet();
 Pix index = set->first();
 while (index != 0)                                                220
   {
    new_set->add((*set)(index));
    set->next(index);
   }
 return new_set;
}

CStringSet *SubsetsGenerator::yield()
{
 if (current == NULL)                                              230
   {
    current = new CStringSet();
    return shallow_copy_CStringSet(current);
   }
 if (done())
   return NULL;

 Pix index = set->first();
 increment_at_position(index);
 return shallow_copy_CStringSet(current);                          240
 /* There would be rep exposure if I returned current itself.... */
}

SubsetsGenerator::~SubsetsGenerator()
{
 delete set;
 delete current;
}

// [....]                                                          250

/***********************************************************************/

/* Adapted from itoa(), K&R second edition, page 64 */
void lltoa(unsigned long long int n, char s[])
{
 int i = 0;
 do {
   s[i++] = (n % 10) + '0';
 } while ((n /= 10) > 0);                                          260
 s[i] = '\0';
 {
   int c,i,j;
   for (i = 0, j = strlen(s)-1; i<j; i++, j--)
     {
       c = s[i];
```

```
      s[i] = s[j];
      s[j] = c;
    }
  }                                                                    270
}

/*************************************************************************/
```

D.6 Diffs to snoot Pass

```
*** 1.1 1995/01/17 18:48:49
--- token.h 1995/02/02 03:26:39
***************
*** 121,130 ****
  yy(0, 97, 0, 0, 0, 0, 0)
  yy(0, 98, 0, 0, 0, 0, 0)
  yy(0, 99, 0, 0, 0, 0, 0)
! yy(0, 100, 0, 0, 0, 0, 0)
! yy(0, 101, 0, 0, 0, 0, 0)
! yy(0, 102, 0, 0, 0, 0, 0)
! yy(0, 103, 0, 0, 0, 0, 0)
  yy(0, 104, 0, 0, 0, 0, 0)
  yy(0, 105, 0, 0, 0, 0, 0)
  yy(0, 106, 0, 0, 0, 0, 0)
--- 121,130 ----
  yy(0, 97, 0, 0, 0, 0, 0)
  yy(0, 98, 0, 0, 0, 0, 0)
  yy(0, 99, 0, 0, 0, 0, 0)
! zz(QINT, 100, 0, 0, 0, CHAR, "qint")
! zz(QIF, 101, 0, 0, 0, IF, "qif")
! zz(QELSE, 102, 0, 0, 0, IF, "qelse")
! zz(QINFLUENCE, 103, 0, 0, 0, IF, "qinfluence")
  yy(0, 104, 0, 0, 0, 0, 0)
  yy(0, 105, 0, 0, 0, 0, 0)
  yy(0, 106, 0, 0, 0, 0, 0)
```

```
*** 1.1 1995/01/17 18:48:49
--- keywords.h 1995/02/02 03:26:40
***************
*** 166,171 ****
--- 166,208 ----
   return LONG;
   }
   goto id;
+ case 'q':
+  if (rcp[0] == 'e'
+  && rcp[1] == 'l'
+  && rcp[2] == 's'
+  && rcp[3] == 'e'
+  && !(map[rcp[4]]&(DIGIT|LETTER))) {
+  cp = rcp + 4;
+  return QELSE;
+  }
+  if (rcp[0] == 'i'
+  && rcp[1] == 'f'
+  && !(map[rcp[2]]&(DIGIT|LETTER))) {
+  cp = rcp + 2;
+  return QIF;
+  }
+  if (rcp[0] == 'i'
+  && rcp[1] == 'n'
+  && rcp[2] == 'f'
+  && rcp[3] == 'l'
+  && rcp[4] == 'u'
+  && rcp[5] == 'e'
+  && rcp[6] == 'n'
```

```
+ &&  rcp[7] == 'c'
+ &&  rcp[8] == 'e'
+ && !(map[rcp[9]]&(DIGIT|LETTER))) {
+  cp = rcp + 9;
+  return QINFLUENCE;
+  }
+  if (rcp[0] == 'i'
+  &&  rcp[1] == 'n'
+  &&  rcp[2] == 't'
+  && !(map[rcp[3]]&(DIGIT|LETTER))) {
+  cp = rcp + 3;
+  tsym = inttype_sym;
+  return QINT;
+  }
+         goto id;
  case 'r':
   if (rcp[0] == 'e'
   &&  rcp[1] == 'g'
***************
*** 314,320 ****
  case 'n':
  case 'o':
  case 'p':
- case 'q':
  case 'x':
  case 'y':
  case 'z':
--- 351,356 ----
```

```
*** 1.1 1995/01/17 18:48:49
--- stmt.cc 1995/02/03 02:34:01
***************
*** 43,48 ****
--- 43,52 ----
                              var_sym *for_index, operand *for_ub);
  static void ifstmt(label_sym *continue_lab, label_sym *break_lab,
                     struct swtch *swp, int lev);
+ static void qifstmt(label_sym *continue_lab, label_sym *break_lab,
+                     struct swtch *swp, int lev);
+ static void qinfluencestmt(label_sym *continue_lab, label_sym *break_lab,
+     struct swtch *swp, int lev);
  static var_sym *localaddr(genop p);
  static void return_void(void);
  static void return_value(genop to_return);
***************
*** 53,58 ****
--- 57,64 ----
  static void swstmt(label_sym *continue_lab, int lev);
  static void whilestmt(struct swtch *swp, int lev);

+ extern char *k_qif_statement;
+ extern char *k_qinfluence;

  /* branch - jump to lab */
  static void branch(label_sym *target)
***************
*** 879,888 ****
--- 885,1032 ----
          statement(continue_lab, break_lab, swp, lev);
        }

+     if (t == QELSE)
+       {
+         static char follow[] = { IF, ID, '}', 0 };
+         error("[PSH] if followed by a qelse, skipping.\n");
+         skipto(0, follow);
+       }
```

```
+
+     curr_list = old_list;
+
+   }
+
+ int currently_parsing_qif_expression = 0;
+   /* flag: when set, the expression parsing code doesn't consider
+      use of an undeclared identifier (i.e., the qif variable) an error. */
+ char qif_buffer[1024];
+   /* storage buffer: if there are a set of parentheses after the qif
+      keyword, what they enclose is stored here for the annotation. */
+
+ /* qifstmt - qif ( expression ) statement [ qelse qifstmt|statement ] */
+ static void qifstmt(label_sym *continue_lab, label_sym *break_lab,
+                     struct swtch *swp, int lev)
+   {
+     {
+       int status;
+
+       qif_buffer[0] = (char) 0;
+       status = sscanf(cp, " (%[^)])", qif_buffer);
+       if ((strlen(qif_buffer) == 0) || (status != 1))
+         warning("[PSH] Couldn't find '(...)' after a qif.\n");
+     }
+
+     t = gettok();
+     expect('(');
+     definept(NULL);
+
+     currently_parsing_qif_expression++;
+     genop e = conditional(')');
+     currently_parsing_qif_expression--;
+     e.clean_up_bit_field_refs();
+
+     label_sym *else_label = genlabel();
+     tree_node_list *if_test = e.precomputation();
+     operand test_operand = e.suif_operand();
+     instruction *branch_instr = new in_bj(io_bfalse, else_label, test_operand);
+     if_test->append(new tree_instr(branch_instr));
+
+     tree_node_list *then_part = new tree_node_list;
+     tree_node_list *else_part = new tree_node_list;
+     tree_if *the_if = new tree_if(else_label, if_test, then_part, else_part);
+     assert(curr_list != NULL);
+     curr_list->append(the_if);
+
+     {
+       immed *i = new immed(qif_buffer);
+       immed_list_e *ile = new immed_list_e(*i);
+       immed_list *il = new immed_list();
+       il->push(ile);
+
+       the_if.append_annote(k_qif_statement, (void *) il);
+     }
+
+     tree_node_list *old_list = curr_list;
+     curr_list = then_part;
+     refinc /= 2;
+     if (refinc == 0)
+         refinc = 1;
+     statement(continue_lab, break_lab, swp, lev);
+
+     if (t == QELSE)
+       {
+         curr_list = else_part;
+         t = gettok();
+         statement(continue_lab, break_lab, swp, lev);
+       }
```

```
+
+     if (t == ELSE)
+       {
+         static char follow[] = { IF, ID, '}', 0 };
+         error("[PSH] qif followed by a else, skipping.\n");
+  skipto(0, follow);
+       }
+
      curr_list = old_list;

    }

+ tree_block *qinfluence_block = NULL;
+   /* At outermost scope so that stabblock() can modify it; the
+      use of previous_qinfluence_block inside qinfluencestmt() is
+      intended to store values of qinfluence_block in stack variables
+      for nested qinfluences. */
+ int just_parsed_qinfluence_statement = 0;
+
+ /* qinfluencestmt - qinfluence ( expression ) statement */
+ static void qinfluencestmt(label_sym *continue_lab, label_sym *break_lab,
+     struct swtch *swp, int lev)
+   {
+     char qinfluence_buffer[1024];
+     /* storage buffer: if there are a set of parentheses after the qinfluence
+       keyword, what they enclose is stored here for the annotation. */
+     tree_block *previous_qinfluence_block;
+
+     previous_qinfluence_block = qinfluence_block;
+
+     {
+       int status;
+
+       qinfluence_buffer[0] = (char) 0;
+       status = sscanf(cp, " (%[^)])", qinfluence_buffer);
+       if ((strlen(qinfluence_buffer) == 0) || (status != 1))
+         warning("[PSH] Couldn't find '(...)' after a qinfluence.\n");
+     }
+
+     t = gettok();
+     expect('(');
+     definept(NULL);
+
+     /* I don't need to set a flag while parsing the contents of '(...)'
+        since it *IS* an error if there is a qinfluence variable which
+        hasn't already been introduced by a qint. */
+     genop e = conditional(')');
+     e.clean_up_bit_field_refs();
+
+     refinc /= 2;
+     if (refinc == 0)
+         refinc = 1;
+     just_parsed_qinfluence_statement++;
+       /* stabblock() will do the just_parsed_qinfluence_statement-- */
+     statement(continue_lab, break_lab, swp, lev);
+
+     {
+       immed *i = new immed(qinfluence_buffer);
+       immed_list_e *ile = new immed_list_e(*i);
+       immed_list *il = new immed_list();
+       il->push(ile);
+
+       qinfluence_block->append_annote(k_qinfluence, (void *) il);
+     }
+
+     qinfluence_block = previous_qinfluence_block;
+   }
+
```

```
  /* localaddr - returns q if p yields the address of local/parameter q;
     otherwise returns NULL */
  static var_sym *localaddr(genop p)
***************
*** 1007,1012 ****
--- 1151,1162 ----
          case IF:
              ifstmt(continue_lab, break_lab, swp, lev + 1);
              break;
+         case QIF:
+      qifstmt(continue_lab, break_lab, swp, lev + 1);
+      break;
+  case QINFLUENCE:
+      qinfluencestmt(continue_lab, break_lab, swp, lev + 1);
+      break;
          case WHILE:
              whilestmt(swp, lev + 1);
              break;
```

```
*** 1.1 1995/01/18 19:11:03
--- expr.cc 1995/05/21 01:23:52
***************
*** 1172,1177 ****
--- 1172,1180 ----
      return genop(operand(), new tree_node_list);
    }

+ extern int currently_parsing_qif_expression;
+ extern char *k_qif_variable;
+
  /* primary - parse a primary expression */
  static genop primary(void)
    {
***************
*** 1209,1220 ****
                  }
                else
                  {
!                   error("undeclared identifier '%s'\n", q->name);
!                   q->sclass = AUTO;
!                   q->type = inttype;
!                   q->suif_symbol =
!                           get_current_symtab()->new_var(q->type, q->name);
!                   q->suif_symbol->set_userdef();
                  }
                if (xref)
                   use(q, src);
--- 1212,1244 ----
                  }
                else
                  {
!      if (currently_parsing_qif_expression)
!                     {
!  base_symtab *global_st;
!  global_st = get_current_symtab();
!  while (global_st->kind() != SYMTAB_GLOBAL)
!    global_st = global_st->parent();
!
!  q->sclass = EXTERN;
!  q->type = qual(CONST, inttype);
!  if ((q->suif_symbol = global_st->lookup_var(q->name))
!      == NULL)
!    {
!      q->suif_symbol =
!        global_st->new_var(q->type, q->name);
!      q->suif_symbol->set_userdef();
!      q->suif_symbol->append_annote(k_qif_variable);
```

```
!    }
!        }
!                         else
!                           {
!  error("undeclared identifier '%s'\n", q->name);
!  q->sclass = AUTO;
!  q->type = inttype;
!  q->suif_symbol =
!    get_current_symtab()->new_var(q->type, q->name);
!  q->suif_symbol->set_userdef();
!        }
                      }
                  if (xref)
                      use(q, src);
```

```
*** 1.1 1995/01/18 20:10:47
--- decl.cc 1995/04/14 15:08:31
***************
*** 44,49 ****
--- 44,57 ----
  static type_node *parse_type(int lev, int *sclass);
  static boolean control_reaches_list_end(tree_node_list *node_list);

+ extern char *k_qint_parameter;
+ int just_parsed_qint_type = 0;
+   /* flag: when set, the symbol of the declaration being parsed
+      will be annotated with qint_parameter. */
+ char qint_buffer[1024];
+   /* storage buffer: if there are a set of brackets following the
+      qint keyword, what they enclose is stored here for the annotation. */
+
  /* checklab - check for undefined labels; called at ends of functions */
  static void checklab(Symbol p, Generic cl)
    {
***************
*** 648,653 ****
--- 656,668 ----

      pt = src;
      type_node *ty = parse_type(level, &sclass);
+
+     if ((just_parsed_qint_type) && (t != ID))
+       {
+  warning("[PSH] qint not followed by identifier, faking int.\n");
+  just_parsed_qint_type--;
+       }
+
      if ((t == ID) || (t == '*') || (t == '(') || (t == '['))
        {
          Coordinate pos;
***************
*** 690,696 ****
              else if (sclass == TYPEDEF)
                  (void)deftype(id, ty1, &pos);
              else
!                 (*dcl)(sclass, id, ty1, &pos);
              if (t != ',')
                  break;
              t = gettok();
--- 705,728 ----
              else if (sclass == TYPEDEF)
                  (void)deftype(id, ty1, &pos);
              else
!        {
!  if (just_parsed_qint_type)
!    {
!      Symbol p = dclglobal(sclass, id, qual(CONST,ty1), &pos);
```

```
!
!      immed *i = new immed(qint_buffer);
!      immed_list_e *ile = new immed_list_e(*i);
!      immed_list *il = new immed_list();
!      il->push(ile);
!
!      assert(p->suif_symbol != NULL);
!      p->suif_symbol->append_annote(k_qint_parameter,
!    (void *) il);
!      just_parsed_qint_type--;
!    }
!  else
!    (*dcl)(sclass, id, ty1, &pos);
!        }
               if (t != ',')
                   break;
               t = gettok();
***************
*** 903,908 ****
--- 935,947 ----
        {
          static char follow[] = { IF, CHAR, '}', 0 };
          type_node *ty1 = parse_type(0, NULL);
+
+         if (just_parsed_qint_type)
+    {
+      warning("[PSH] qint type while parsing fields, faking int.\n");
+      just_parsed_qint_type--;
+    }
+
          do
            {
              char *id = NULL;
***************
*** 1434,1439 ****
--- 1473,1486 ----
                   error("missing parameter type\n");
                 }
               ty = dclr(parse_type(PARAM, &sclass), &id, lev + 1);
+
+      if (just_parsed_qint_type)
+        {
+  warning("[PSH] qint type while parsing parameters, "
+  "faking int.\n");
+  just_parsed_qint_type--;
+        }
+
               if ((Aflag >= 1) && !hasproto(ty))
                   warning("missing prototype\n");
               if ((ty == voidtype) && ((last != NULL) || (id != NULL)))
***************
*** 1625,1630 ****
--- 1672,1695 ----
               case SHORT:
                   p = &size;
                   break;
+              case QINT:
+  {
+    int status;
+
+    qint_buffer[0] = (char) 0;
+    status = sscanf(cp, " [%[^]]]", qint_buffer);
+    if ((strlen(qint_buffer) == 0) || (status != 1))
+      warning("[PSH] Couldn't find '[...]' after a qint.\n");
+    else
+      {
+        just_parsed_qint_type++;
+        while (*cp != '[') cp++;
```

```
+         cp++;
+         while (*cp != ']') cp++;
+         cp++;
+       }
+  }
+  /* deliberately falling through */
              case VOID:
              case CHAR:
              case INT:
```

```
*** 1.1 1995/02/02 20:31:55
--- gen.cc 1995/05/03 16:41:21
***************
*** 36,42 ****
  #include "c.h"

  RCS_BASE(
!     "$Id: gen.cc,v 1.1 1995/02/02 20:31:55 pshuang Exp $")



--- 36,42 ----
  #include "c.h"

  RCS_BASE(
!     "$Id: gen.cc,v 1.3 1995/05/03 16:41:17 pshuang Exp pshuang $")



***************
*** 278,283 ****
--- 278,285 ----
   * BEGINNING/END OF BLOCK FUNCTIONS                                    *
   ********************************************************************/

+ extern tree_block *qinfluence_block;
+ extern int just_parsed_qinfluence_statement;

  // enter/exit a block in the emit phase, with its user-defined locals
  void stabblock(int enter_or_exit, int level)
***************
*** 301,306 ****
--- 303,321 ----

              tree_block *b = new tree_block(new_list, new_symtab);
              curr_list->append(b);
+
+       if (just_parsed_qinfluence_statement)
+         {
+  just_parsed_qinfluence_statement--;
+  qinfluence_block = b;
+  new_symtab->new_var(qual(CONST,inttype),
+      "QINFLUENCE_BLOCK_HOLDER");
+  /* Entering this dummy variable prevents SUIF from optimizing
+     the block away; otherwise, a qinfluence() whose body
+     does not introduce any scope variables will be reduced
+     to the statements in the body, and the annotation
+     I attach to the block will vanish. */
+         }

              curr_list = new_list;
              if (curr_symtab != NULL)
```

```
*** 1.1 1995/01/17 22:19:42
--- main.cc 1995/02/19 22:20:06
***************
```

```
*** 13,18 ****
--- 13,23 ----

  #include "c.h"

+ char *k_qif_statement; /* annotation for the qif statement itself */
+ char *k_qif_variable; /* annotation for quasistatic variables */
+ char *k_qint_parameter; /* annotation for quasistatic parameters */
+ char *k_qinfluence; /* annotation for quasistatic parameters */
+
  int Aflag;                          /* > 0 if -W specified */
  boolean Pflag;                      /* TRUE if -P specified */
  boolean xref;                       /* TRUE for cross-reference data */
***************
*** 40,45 ****
--- 45,55 ----
      init_bit_ref();
      level = GLOBAL;
      assert(inttype->size() >= voidptype->size());
+
+     ANNOTE(k_qif_statement, "QIF STATEMENT", TRUE);
+     ANNOTE(k_qif_variable, "QIF VARIABLE", TRUE);
+     ANNOTE(k_qint_parameter, "QINT PARAMETER", TRUE);
+     ANNOTE(k_qinfluence, "QINFLUENCE STATEMENT", TRUE);

      if (option_null_check)
        {
```

Bibliography

[A94] Michael Abrash, *Zen of Code Optimization.* Coriolis Group Books: 1994.

[B95] Jeremy Brown, "Feedback-Directed Specialization of C". Master of engineering thesis, Massachusetts Institute of Technology, EECS Department, June 1995.

[BDE94] Jeremy Brown, Andre DeHon, Ian Eslick, et al., "The Clever Compiler: Proposal for a First-Cut Smart Compiler." Transit Note 101, MIT Artificial Intelligence Laboratory, January 1994.

[C+91] Cmelik, et al., "An Analysis of the MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks." Appeared in *Proceedings of the 4*TH *international conference on Architectural support for programming languages and operating systems*, pg. 290–302, 1991.

[CGL95] Brad Calder, Dirk Grunwald, & Donald Lindsay, "Corpus-based Static Branch Prediction." To appear in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, June 1995.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, & Ronald L. Rivest, *Algorithms.* MIT Press, Cambridge, Mass., 1990.

[CMCH91] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, & Wen-Mei Hwu, "Profile-guided Automatic Inline Expansion for C Programs". Appeared in *Software — Practice and Experience*, Vol. 22(5), May 1992, pg. 349–369.

[CSW94] Yin Chan, Ashok Sudarsanam, & Andrew Wolfe, "The Effect of Compiler-Flag Tuning on SPEC Benchmark Performance." Appeared in *Computer Architecture News*, 22:4, Sept. 1994, pg. 60–70.

[D94a] Andre DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century." Transit Note 100, MIT Artificial Intelligence Laboratory, January 1994.

[D94b] Andre DeHon, "A Proposal for Alternative Code: a quasistatic-if-then-else and quasistatic variables." Transit Note 110 (working draft), MIT Artificial Intelligence Laboratory, July 1994.

[DBE+94] Andre DeHon, Jeremy Brown, Ian Eslick, et al., "Global Cooperative Computing." Transit Note 111, MIT Artificial Intelligence Laboratory, Sept. 1994.

[DC94] Jeffrey Dean & Craig Chambers, "Towards Better Inlining Decisions Using Inlining Trials." Presented at *1994 ACM Conference on LISP and Functional Programming*, Orlando, Florida; June 1994.

[DCG95] Jeffrey Dean, Craig Chambers, & David Grove, "Selective Specialization for Object-Oriented Languages." To appear in *SIGPLAN'95: Conference on Programming Language Design and Implementation (PLDI'95)*, June 1995.

[DE94] Andre DeHon & Ian Eslick, "Computational Quasistatics." Transit Note 103, MIT Artificial Intelligence Laboratory, March 1994.

[EP93] Dawson Engler & Todd Proebsting, "DCG: An Efficient, Retargetable Dynamic Code Generation System." November 1993. (Contact: `todd@cs.arizona.edu`, `engler@lcs.mit.edu`.)

[FF92] Fisher and Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program." Appeared in *Proceedings of the 4TH international conference on Architectural support for programming languages and operating systems*, pg. 85–95, 1992.

[FH94] Chris Fraser & David Hanson, *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings: 1994.

[G93] Gray, "SPEC: A Five Year Retrospective." Appeared in *SPEC Newsletter*, 5:4, December 1993.

[G95] Greg McLaren, "QProf: A Scalable Profiler for the Q Back End." Master of engineering thesis, Massachusetts Institute of Technology EECS Department, January 1995.

[GKM82] Susan L. Graham, Peter B. Kessler & Marshall K. McKusick, "gprof: a Call Graph Execution Profiler". Appeared in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pg. 120–126, 1982.

[HU94] Urs Holze & David Ungar, "Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback." Appeared in *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida; June 1994.

[I94] Intel Corporation, *Pentium Processor Family User's Manual Volume 3: Architecture and Programming Manual.* Published 1994.

[K71] Donald E. Knuth, "Empirical Study of FORTRAN Programs." *Practice and Experience*, vol. 1, pg. 105–133.

[K73] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithsm.* Addison-Wesley, 1973.

[K75] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1975.

[KR88] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming Language", 2ND edition. Prentice Hall, 1988.

[LRW91] Monica S. Lam, Edward E. Rothberg, & Michael E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms". Appeared in 4TH *International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS IV)*, Palo Alta, California; April 9–11, 1991.

[M94] Terje Mathisen, "Pentium Secrets." Appeared in *Byte*, July 1994, pg. 191–192. See also the article posted to the Usenet `comp.arch` newsgroup in response to this article: "From: glew@ichips.intel.com (Andy Glew)", "Date: 16 Jul 1994 23:09:24 GMT", "Message-ID: <GLEW.94Jul9235235@pdx007.intel.com>", "Subject: Re: "Pentium Secrets"".

[MGA95] Margaret Martonosi, Anoop Gupta, & Thomas E. Anderson, "Tuning Memory Performance of Sequential and Parallel Programs." Appeared in *IEEE Computer*, April 1995, pg. 32–40.

[OSF93] Open Software Foundation Research Institute, *ANDF Technology Collected Papers.* Volumes 1–4, January/March/June/December 1993. See also WWW URL http://riwww.osg.org:8001/andf/index.html for additional Architecture-Neutral Distribution Format information.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, & William T. Vetterling, *Numerical Recipes in C.* Cambridge University Press, 1988.

[S91] Alan Dain Samples, *Profile-driven Compilation.* UC Berkeley, Computer Science Division, Technical Report UCB/CSD 91/627, April 1991.

[S94] Richard M. Stallman, *Using and Porting GNU CC: Last updated 19 September 1994 for version 2.6.* Free Software Foundation, 1994. Available from ftp://prep.ai.mit.edu/pub/gnu/gcc-2.6.3.tar.gz.

[S95] Jonathan Schilling, "Dynamically-Valued Constants: An Underused Language feature." Appeared in *ACM SIGPLAN Notices*, Volume 30, No. 4, April 1995, pg. 13–20.

[SW92] Amitabh Srivastava & David W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time." DEC Western Research Laboratory, WRL Research Report 92/6, December 1992.

[W90] David W. Wall, "Predicting Program Behavior Using Real or Estimated Profiles." DEC Western Research Laboratory WRL Technical Note TN-18, December 1990.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, et al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." Computer Systems Lab, Stanford University, 1994. Available at the WWW URL http://suif.stanford.edu/suif/suif-overview/suif-overview.html.