Profile-Driven Compilation
by
Alan Dain Samples
Abstract

As the size and complexity of software continues to grow, it will be necessary for software construction systems to collect, maintain, and utilize much more information about programs than systems do now. This dissertation explores compiler utilization of profile data.

Several widely held assumptions about collecting profile data are not true. It is not true that the optimal instrumentation problem has been solved, and it is not true that counting traversals of the arcs of a program flow graph is more expensive and complex than counting executions of basic blocks. There are simple program flow graphs for which finding optimal instrumentations is possibly exponential. An algorithm is presented that computes instrumentations of a program to count arc traversals (and therefore basic block counts also). Such instrumentations impose 10% to 20% overhead on the execution of a program, often less than the overhead required for collecting basic block execution counts.

An algorithm called Greedy Sewing improves the behavior of programs on machines with instruction caches. By moving basic blocks physically closer together if they are executed close together in time, miss rates in instruction caches can be reduced up to 50%. Arc-count profile data not only allows the compiler to know which basic blocks to move closer together, it also allows those situations that will have little or no effect on the final performance of the reorganized program to be ignored. Such a low-level compiler optimization would be difficult to do without arc-count profile data.

The primary contribution of this work is the development of TypeSetter, a programming system that utilizes profile data to select implementations of program abstractions. The system integrates the development, evaluation, and selection of alternative implementations of programming abstractions into a package that is transparent to the programmer. Unlike previous systems, TypeSetter does not require programmers to know details of the compiler implementation. Experience indicates that the TypeSetter approach to system synthesis has considerable benefit, and will continue to be a promising avenue of research.

Acknowledgements

My heartfelt appreciation goes first and foremost to my wife Pat for being patient (almost) every time I reported that I needed just six more months, nine at the most. That she could continue to be encouraging and excited about my work is almost more than I can understand. This dissertation is dedicated to her.

Whatever I have accomplished, it is because I have stood on the shoulders of two of the most loving giants I shall ever know. My dreams could not even have been dreamt without the many accomplishments achieved by my parents. I simply cannot find the words that sufficiently express my thanks and admiration.

I am grateful to Bruce MacLennan for more than he can possibly know. I thank him most for his friendship, his intellect, and encouragment. A lot of him, Gail, and Kimmie are in these pages. Thank you!

I may never have started on this task without the inspiration and downright prodding (goading?) of Dick Hamming and his perspective on the true meaning of a Ph.D.

"[W]e greatly appreciated the valuable suggestions by ... Sue Graham, ... and, especially, Paul Hilfinger — the reviewers who provided most of the comments that kept us busy for so long producing the final draft." [13] I am particularly pleased that I never had to make use of Paul's dimes. I also thank Stuart Dreyfus for his work on my thesis committee.

If I acknowledged everyone the way I want to, these acknowledgements could easily become longer than the dissertation itself. Therefore, I hope a simple thanks suffices for everyone who has helped me achieve this goal. Special thanks to Dr. Rodney Farrow, the unofficial fourth member of my committee, and to Dr. Wendy Sinclair-Brown for helping me see early on what was really important; thanks to Michelle and Allen for their valuable friendship; to Dave Ungar, Richard Probst, and other members of the Stiletto Club; to Eduardo and Vicki; to Michael and Karen; to Charlie and Kendall; to Doug; and especially to Toni, Mara, and everyone in 508-20 for putting up with me and helping me during the Final Days.

This dissertation was supported in part by an AT&T Bell Laboratories Scholarship, and by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292. Their support is very gratefully acknowledged.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> 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 that 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. [28]

In spite of the fact that Knuth made this pronouncement twenty years ago, and in spite of the fact that programmers routinely 'optimize' programs by hand based on profile data, Knuth's Dictum (as we will call it) still has not been fully implemented in an automated profiling system nor shown to be undesirable.

This dissertation examines the issues surrounding the utilization of profile data in the compilation of source code. This is a larger subject than that of generating or collecting profile data. It requires asking, at a minimum, the following questions: *Given that profile data exists for a program, how might a compiler make use of that data to produce better executable code? What kinds of profile data can be generated/collected? What kinds of profile data are useful? How expensive is this profiling?*.

We start with two hypotheses:

1. Collecting profile data need not be prohibitively expensive.

2. Compilers can profitably use profile data at all levels of the compilation process.

Compilers that emit profiling code have been at least partially implemented by most modern systems. But no systems of which I am aware utilize a program's profile data throughout the compilation process. Furthermore, while these hypotheses might be accepted in a general way, there are still some misconceptions about the cost of profiling, and room for improvement in the profiling algorithms themselves.

1.1 Past uses of profile data

The collection and use of profile data has a long history, beginning with Knuth's 1971 paper [28]. In this paper, the term 'profile' was first used, and defined to be the collection of execution frequency counts taken during executions of a program. Since that time, the term 'profile data' has come to mean any quantitative information gathered about the run-time behavior of a program, including execution counts of the program and its sub-parts, reference counts of the program's data objects, and real-time measures of algorithm executions.

Knuth's examination of execution profiles of running user programs uncovered two facts: (1) Most programmers do not know where their programs spend most of the time; and (2) even when programmers analyze their programs, they still don't know where their programs spend most of the time due to the fact that programmers almost never have access to sufficient information about system and library functions to deduce the runtime resources they consume. For example, a major culprit in the FORTRAN environment in which Knuth did his study was the formatting routines in the I/O statements.

Another result of Knuth's study was the rule-of-thumb that said that 90% of a program's time is spent in 10% of the code, variously called the 90-10, or 80-20, rule. Knuth never used either of these numbers but reported that in his studies 50% of the time was spent in 4% of the code. In fact, the actual numbers are different for each program. The 90-10 rule, or whatever you want to call it, is one of the guiding principles on which manual program optimization is based: find that section of your program that takes most of the runtime resources, and either modify the algorithm itself (e.g., change a bubble-sort into a quick-sort) or make the existing algorithm more efficient at the low level (e.g., hoist common expressions out of loops, do strength reduction on the index variables, turn repeated array indexing operations into pointer operations, etc.).

After Knuth's 1971 paper, Dan Ingalls published two papers describing descendants of the FORDAP profile tool used by Knuth. FORDAP was a basic-block counting profiler. The first technical report [23] gives details of how the FORTRAN Execution Time Estimator (FETE) adds execution time estimates to the frequency count displays of FORDAP. This enhancement was prompted by the obvious fact that not all statements are created equal. For example, the FORTRAN statement

```
A = B(I)
```

will execute at vastly different speeds depending on whether `B` is an array or a function. FETE used a value based on 'weights' assigned to expression operators and statement classes to give a rough estimate of the execution time. FETE was not sophisticated enough to handle calls on user functions. If, in the example above, `B` is not an array but a call on a function, FETE will count it as an array reference unless `B` is a standard FORTRAN function.

Ingalls' second paper [22] describes FORTUNE, which is simply a renamed, product version of FETE. FORTUNE and FETE modified the source program so that it contained FORTRAN statements incrementing elements of an array of counters. These counting statements were placed essentially at the beginnings of basic blocks. The analyzer reported statement execution counts and estimates of execution time for each statement.

Prof [5], a profile collector for C, Pascal, and FORTRAN programs on the UNIX system is an example of profiling tools in current use. *Prof* samples the program counter via timer interrupts to estimate the amount of time spent between the symbols of the program. *Prof*'s usability has been enhanced by *Gprof*, a program developed at Berkeley by Graham, Kessler and McKusick [16]. *Gprof* explicitly concentrates on procedure calls, providing both frequencies for calls through counting and estimates of the time spent in each procedure. Timing estimates are derived by sampling the program counter, as in *prof*. After the user's program has run, a separately invoked post-pass analyzer distributes timing estimates to the program's procedures based on the static and dynamic call graphs.

There has been a small amount of research published about the best way to profile a program. In the first volume of his *Art of Programming* series, Knuth gives the algorithm for determining a minimal instrumentation of a program for collecting the execution counts of arcs. Knuth and Stevenson [30] (about which more will be said later) published the definitive algorithm for finding a minimal instrumentation of a program that counts the execution of its basic blocks. Cheung [7] concurrently developed algorithms for finding minimal instrumentations that count the frequency of execution paths through a program. A paper by Sarkar [42], without referencing this body of work, developed an algorithm for instrumenting a program based on its dependence graph.

There has been some research into the potential uses of profile data. Gilbert Hansen's research [17] is an early investigation into behavior-driven optimization. He hypothesized that, for certain classes of software, the optimization of a program could be done at *run*-time more economically than at compile-time. Instead of the usual compile-a-file paradigm that most compiler systems utilize, Hansen's "adaptive" compiler consisted of two "phases". The first phase generated an interpretable form of a FORTRAN program in a fast, one-pass compilation (it produced 'quads' as the interpretable form). The second part consisted of an interpreter and optimizer loaded with the compiled program. When the interpreter detected that a basic block was being executed sufficiently often, interpretation was suspended while the optimizer was invoked to compile the basic block to a lower level. If a basic block were executed often enough, it would eventually be compiled down to machine language.

The one-pass compiler annotated each basic block with information about its size and complexity so the interpreter could predict profitable optimizations. The system was designed to expend effort only on optimizations that had a high probability of paying for themselves through improved execution of the program. There were four levels of optimizations, three of which were performed on the interpretable form (constant folding, common subexpression elimination, and moving invariants out of loops), and the final of which compiled the quads resulting from the first three optimizations into machine language, an 'optimization' he called 'fusion'. Therefore, the optimizer would be invoked up to four times on a basic block, each time optimizing it further. When the basic block had been optimized as much as possible in its interpreted form, the final optimization would compile it to machine language. At this time, several machine-dependent optimizations would also be applied and execution would become 'threaded'; i.e. a mixture of interpretation and direct execution in which the program has been 'fused' into the interpreter.

It is not surprising that Hansen's one-pass compiler executes more quickly than optimizing, multi-pass compilers. What is surprising is that those initial savings were almost never depleted. That is, the time required to do the one-pass compilation to 'quads', plus the time for interpretation and intermittent optimization was almost always less than the time required to do a full optimization of the original program plus the execution time of the optimized program!

Hansen's system is appropriate for compiling and running throw-away, run-once programs; e.g. in a student programming environment, the overall CPU utilization is decreased. It is not an appropriate system for constructing software designed to be run many times (e.g. editors, the operating system, the adaptive compiler itself). The success of this system depended on the ratio of the number of compilations to the number of executions being very close to one. If a program is executed many times, it then becomes profitable to compile and optimize the whole program once.

Hansen's research lends credence to our contention that profile-driven optimization is a useful adjunct to 'traditional' compilation. If his adaptive compiler could use heuristics to predict the future behavior of a program successfully enough, a static compiler using those same heuristics with complete profile data should do better. The major point is that, whereas the adaptive compiler system is forced to make the assumption that the performance of the program in the immediate past is predictive of its performance in the immediate future, a static compiler can accumulate profile data, smooth out anomalous behavior over several runs of the program, and make the more accurate assumption that the average past performance of the program is a good predictor of its average future behavior.

There has been a large amount of research using profile data to improve virtual memory performance. Most of this work has depended on profile data in the form of an address trace and has improved program performance by reorganizing the modules of programs to minimize page faults (Ferrari [12] gives a summary of this early work). Nearly all of this research has concentrated on post-compilation module-level reorganization of a program. There have been some techniques developed to reorganize programs dynamically based on their behavior. For example, K. D. Ryder [39] and J.-L. Baer and G. Sager [2] used dynamically collected profile data to allocate physical memory for programs running on virtual systems.

In spite of all the use of profile data to improve program performance in various ways, modern programmers wishing to improve the performance of their programs using profile data have limited options. In almost every system that provides any profiling capability at all, it is still up to the users to analyze the data and manually reorganize or rewrite their programs based on that data.

1.2 Research Contributions

This dissertation presents several results related to the use of profile data, ranging from exactly how profile data should be collected, to actual uses of profile data in languages and their compilers.

Since at least the mid-seventies the problem of efficient profile collection via code instrumentation has been considered a solved problem. This research discovered problems with the solutions, and results are presented here that show that old assumptions about profiling are incorrect. Specifically, it is not the case that counting the execution frequencies of transfers of control in a program is expensive: it is often cheaper than simply counting execution frequencies of the basic blocks in a program. Also, I show that the algorithms that have heretofore been considered 'optimal' are not only *not* optimal, but that optimality is difficult to achieve. In Chapter 2, I present an algorithm that finds better instrumentations of programs.

A difficult problem from past research has been improving the performance of a program in a paging environment. While paging is no longer the issue that it once was, caching in a memory hierarchy has taken its place. In Chapter 3 I demonstrate that a dramatic percentage of the improvement in the instruction-cache behavior of a program can be obtained by reorganizing a small percentage of the actual code.

Users should be able to declare a variable to be of some abstract type without worrying about the implementation. Unfortunately, it has turned out to be very difficult to design an efficient system with this capability. In Chapter 4 I present the design of a system which uses profile data to assign implementations to variables. With appropriate language extensions that allow the writer of alternative implementations to specify what kinds of profile data are needed and how it is to be evaluated, the user can declare variables to be of a generic type (e.g. `Set(int)`) and let the system decide, based on the profile data, which implementation to use for the variable.

Chapter 2

Profiling Techniques

How do I count thee?
Let me love the ways ...
(apologies to Ms. Browning)

Programmers' intuitions about the runtime behavior of their programs are notoriously bad. Profiling counteracts this deficiency by providing objective measures.

After a brief discussion of various profiling techniques, we will focus on the insertion of counting code in programs as the technique of choice. We will see that the traditional solutions to the minimal instrumentation problem are not optimal. An algorithm that is more nearly optimal is presented, with evidence to show that true optimality may be quite difficult to achieve.

2.1 Profiling techniques

When designing a profile data collection system, two questions must be answered: how, and how much? This work explores the use of software instrumentation to collect profile data. This is in contrast to, say, using specialized hardware to monitor a system from 'outside'. Such specialized hardware has been built, but computer manufacturers that provide it do not provide high-level language programmers with useful tools that can take advantage of such equipment. Tools such as in-circuit emulators or digital oscilloscopes are primarily useful to the electrical engineer or, at best, to the writer of peripheral device drivers. There is no evidence to suggest that such specialized hardware can provide better data than software instrumentation can provide for high-level language applications.

There are three ways profile data can be collected with software instrumentation: monitoring, tracing, and counting. All three methods are discussed in detail below. Monitoring requires some hardware support, usually in the form of a countdown-timer interrupt. Tracing refers to recording in memory or on some external media the sequence of relevant operations of the program or system. Counting is implemented with code inserted into the program to increment (an array of) counters to record the execution frequencies of the program.

The answer to the 'how much' question depends on the granularity of the profile data for any particular program. There are basically four granularities in use: procedure, basic block, statement, and instruction level granularities. There are many variations on these themes, but this gives us enough framework to discuss several profiling techniques.

2.1.1 Monitoring

A histogram of program execution is generated by observing the value of the PC (program counter) at frequent intervals (hence the alternative name of 'PC sampling'). There are three parameters for programmers to specify for monitoring: the area (i.e. address range) of the program that is to monitored, the number of data points to be generated for that address range (the granularity), and the sampling frequency.

For example, a programmer may specify that addresses in the range 0x2000 and 0x100000 are to be monitored, and that 65,536 data points are to be generated. So $0x100000 - 0x2000 = 1,040,384$ addresses are divided into 65,536 regions, meaning that one data point will represent 15 addresses: we say that the measurement granularity is 15. On most machines, more than one instruction can fit in a range of 15 addresses, particularly when the machine is byte-addressable. Therefore, information about multiple basic blocks totally contained in a single 15-address range, or about basic blocks that straddle the boundaries of multiple blocks, will be fuzzy, at best. The goal is to choose a granularity that does not generate too much information[1] and yet captures sufficient information about the program to make reasonable deductions about its performance.

Choosing a good sample rate is also fraught with tradeoffs. If the interrupts occur too infrequently, too much of the program's behavior will occur between the interrupts. If interrupts occur too frequently, the program's execution will be totally swamped by the interrupt overhead. Finding the proper value is hit-or-miss, and there are no published statistical studies showing what range of interrupt frequencies sufficiently capture the behavior of a program.

One problem shared by almost all profiling techniques is that of measuring system overhead. There is no way that a program in a multiprogramming environment can reasonably measure the behavior of the operating system activity due to the executing program. The best that can be hoped for is some measure of the frequency of the system function calls during the program's execution.

All monitoring implementations depend on the sequentiality of instruction execution to extrapolate the statistical samples into information about the program's

[1]Make the granularity one, and the profile data tables will be at least as large as the program and possibly four times larger than the program.

execution. Since there is no single register that can be sampled to derive a profile of a program's data reference behavior, it is extremely difficult to derive measures of a program's use of data with monitoring. Even continuous monitoring of, say, a data bus (which would certainly demand hardware support) cannot provide very interesting information. For example, tracing the data references in the area of memory devoted to the execution stack provides little information. Since the stack varies dynamically as the execution of the program proceeds, functions and their associated data are not guaranteed to map into the same addresses, nor is there any guarantee that a given sequence of addresses will always be used by a given function. To understand the memory reference behavior of a program's stack memory requires some knowledge of the program's dynamic call tree, something a simple memory monitor cannot provide.

2.1.2 Tracing

Very often, simply knowing how many times a function was called, or having an estimate of how much time each function consumed, is not sufficient. Questions such as "how often was event X followed by event Y" turn out to be important in some contexts. In studies of a program's performance in a paging system, the question takes the form "What is the sequence of page references?", and similarly in cache performance studies the question is "What is the sequence of cache line references?". Both kinds of studies have traditionally used address traces (both instruction and data) of actual program executions to answer these questions.

There are three ways to gather traces (without special hardware assistance):

1. *Instrument the code.* This is a messy and laborious technique, particularly if the instrumentation must not interfere with the generated addresses. The trace can be considered legitimate for most purposes only if the recorded addresses are the same as they would be if there were no instrumentation code.

2. *Simulation.* Sometimes it is simpler to write a simulator for the machine in question. Each instruction simulated can then produce a trace of the instruction's address, all data references generated by the instruction, and perhaps other information such as timing estimates.

3. *Single-stepping.* Some microprocessor architectures have a single-step feature that interrupts a process after each instruction executed. A separate process (in a separate address space) handles the interrupt and generates the trace of address references.

Tracing a program usually is quite expensive, causing instrumented programs to run anywhere from 2 (if you're lucky) to 10 times slower, depending on the actual method of tracing and the number of execution features being traced. Furthermore, tracing can produce tremendous amounts of data. With memories and programs getting larger, it may take many millions of instructions of trace data to capture interesting effects. Borg et al. report that some interesting characteristics of traces were not apparent until several billion instructions had executed [6]. Compaction techniques can alleviate some of the problem as I have demonstrated [41], but the management and assimilation of huge amounts of data is always difficult.

Again, even if these problems are solved, the trace of a program on a virtual memory system is not sufficient for capturing effects introduced by the operating system. Tracing the operating system alone may not provide the necessary information either. Tracing a complete system across system calls and interrupts is a gargantuan task, and produces a prodigious amount of information in a small amount of time. Agarwal, Sites, and Horowitz instrumented a processor's micro-code to collect such system-level traces [1], but it is difficult for programmers to utilize this technique to gain an understanding of how their programs and the system interact.

2.1.3 Counting

Many, if not most, compilers now have the ability to insert counting code into users' programs and provide very precise information about the number of times lines, basic blocks, or functions are called. The actual instrumentation is quite simple, even for separately compiled units of a program. Since this is my preferred method of collecting profile data, I discuss it in more detail in the next section.

Counting executions of functions is probably the least useful granularity. Programmers learn which are the most frequently called functions, but that may bear very little relation to the location of the most frequently executed inner loop. Counting lines is not always sufficient since "lines of code" are artifacts of particular languages and the styles of individual programmers. For example, some programs written in C can have many basic blocks concealed on a single line of code due to some programmers' proclivity for using C's conditional expression construct in deeply nested macros. The net result is insufficient information when more than one basic block is on a line, and superfluous counts when basic blocks span lines.

Any modern compiler should implement some form of counting, and at a minimum it should include basic block counting. In the next section, however, I argue that counting execution frequencies of transfers of control (*arc* counting) is best.

2.2 Efficient Counting

Most compilers that implement profiling via the insertion of code count lines or, at best, basic blocks. However, there are some applications where basic block counts are insufficient. Examples include code reorganization to improve the performance of multi-level memory hierarchies [21,35], jump optimization, and code

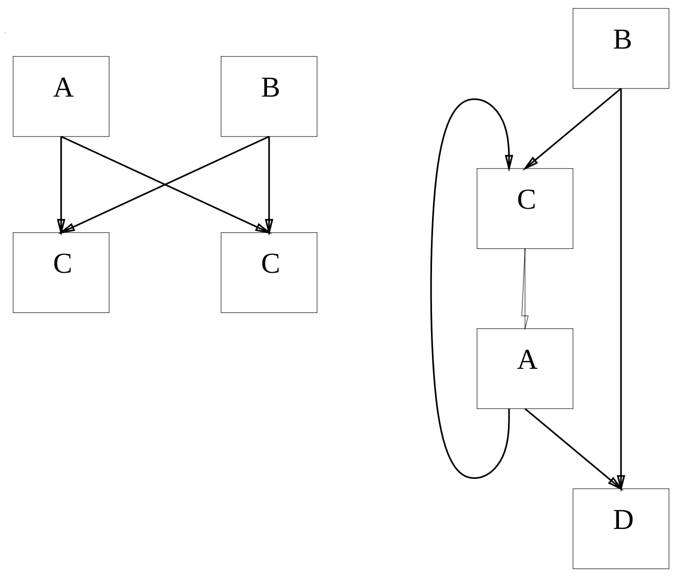

Figure 2.1: Block counts are insufficient. The sub-graph on the right is the same as the sub-graph on the left with the addition of an arc from block C to block A. Knowing the execution counts of the blocks does not allow the derivation of the arc counts.

generation and register assignment [25]. Such optimizations and code transformations depend on knowing branch probabilities (i.e. arc frequencies) and can make only imprecise use of block counts. Arc frequencies frequently cannot be deduced from block frequencies; an example program graph is in Figure 2.1, where knowing that each block is executed n times does not provide enough information to determine the number of times, say, execution of block A is followed by execution of block D. This is not a contrived example. Also shown in Figure 2.1 is an actual program flow-graph (PFG) that contains the problematic graph. This flow graph was generated frequently by a Pascal compiler for **while** loops.

Therefore, arc frequencies, not just block frequencies, are desired, although historically block frequencies have been considered more desirable. To elaborate further on the problem of minimal instrumentation for arc counts, we will need some definitions. We assume that the execution cost of inserted instrumentation code is constant and non-zero; call this cost K_I. In Section 2.5, we will discuss some subtleties in instrumentation code, but for the moment we will assume that each instance of instrumentation code is exactly the same (e.g., a memory-to-memory increment operation, or an equivalent sequence of operations). When instrumentation is inserted in code, it may be necessary to insert a jump instruction to maintain the semantics of the code. We will assume that the execution cost of this jump instruction is also constant, $K_J > 0$.

A program flowgraph is

- a set V of basic blocks, the vertices of the graph;

- a set E of directed edges, which are pairs of vertices, $e = (src(e), snk(e))$. We say that the edge e leaves $src(e)$ and enters $snk(e)$, the *source* and *sink* of the edge, respectively. The edge is an exit arc of $src(e)$ and an entrance arc of $snk(e)$.

- a distinguished edge of the graph, e_0; we assume, without loss of generality, that $snk(e_0)$, the *entrance block* of the graph, has no other predecessors, and $src(e_0)$, the *exit block* of the graph has no other successors; each flowgraph has exactly one entrance block and exactly one exit block.

For each edge e of the graph, $F(e)$ is the frequency count of e; $J(e)$ is a boolean function that is *true* if e is an out-of-line *jump* arc, and *false* if e is a *fall-through* arc; C is a function that maps edges into instrumentation costs. $C(e)$ is the cost required to instrument edge e, and depends on $F(e)$, K_I, and K_J; specifically, $C(e) = C_I(e) = F(e)K_I$ if it is not necessary to insert a jump instruction, or $C(e) = C_J(e) = F(e)(K_I + K_J)$ if a jump instruction is required.

Each block v is the sink of *at most* one fall-through arc, and the source of *at most* one fall-through arc. There is no limit on the number of arcs for which a block is a sink or a source. We define $in(v)$ to be the set of predecessors of v, and $out(v)$ to be the set of successors of v. We say that an arc e is *crowded at its sink* if $|in(snk(e))| > 1$. Likewise, it is *crowded at its source* if $|out(src(e))| > 1$. If an arc is crowded both at its source and at its sink, we simply say that it is *crowded.* We define the predicates $e.crowdedSnk$, $e.crowdedSrc$, and $e.crowded$ on the edge e for these conditions.

The remainder of this discussion assumes that $K_J > 0$ and $K_I > 0$; all of the examples we will display assume $K_J = K_I = 1$.

All of the following algorithms take advantage of the fact that, given a program flow graph, all execution frequencies of the arcs and nodes can be derived if we know an appropriate $|E| - |N| + 2$ arc frequencies. Such a subset of arcs can be selected by finding those arcs that form the complement of a (non-directed) spanning tree in the flow graph. If the arcs have associated costs, then a minimal (maximal) cost subset of arcs can be found by taking those arcs that form the complement of a maximal (minimal) spanning tree in the program flow graph; a proof can be found in Knuth, Vol 1., page 368 [29]. I refer to the algorithm for finding a spanning tree as the SPAN algorithm, and to the algorithm for finding the minimal (maximal) spanning tree as MINSPAN (MAXSPAN). All of this, with the exception of the arc characterization function J and slight differences in notation, is consistent with previous work.

2.2.1 Basic block counts

The most commonly implemented instrumentation technique counts the number of executions of every basic block, which we'll refer to as FULLNODE. Techniques that count every line or every programming language statement are even more inefficient variants of this technique. FULLNODE has the advantage of being the easiest to implement, but the disadvantage of being inefficient: a program can easily be slowed down by as much as 50% to 100%, depending on the execution cost of the instrumentation and the average size of a program's basic blocks.

However, it is not necessary to instrument each and every basic block to get complete block counts. Knuth and Stevenson [30] and Cheung [7] present algorithms that compute a minimal subset of basic blocks that when instrumented provide sufficient data to recover the execution frequencies of all other basic blocks. (Cheung also contains a much more detailed study of minimal instrumentation, including minimal instrumentation for determining path coverage.) A TypeSetter version of Knuth and Stevenson's algorithm is included in Appendix A. I will refer to this algorithm as the K-S instrumentation algorithm.

Conceptually, the K-S algorithm is a graph transformation followed by an application of a spanning tree algorithm. Given a program-flow graph with basic blocks V and edges E, the relation $\equiv$ between basic blocks is defined to be the smallest equivalence relation such that $a \equiv b$ if there exists vertex c and arcs $c \rightarrow a$ and $c \rightarrow b$. A *reduced* graph is produced whose vertices V_r are the equivalence classes of the original graph, and whose edges E_r correspond one-to-one with the basic blocks of the original graph; for each basic block $b \in V$, there exists an edge in V_r from the equivalence class containing b to the class containing the successors of b (by construction, they are all in the same class).

SPAN is applied to the reduced graph to find a spanning tree, and those edges of E_r not in the spanning tree specify the nodes in V that need to be instrumented. The K-S algorithm also computes the expressions that will later allow the frequencies of all nodes to be computed in one pass.

If we have some notion of the execution behavior of the program, then each node v of a program flow graph can be assigned an instrumentation cost, $K_I F(v)$. A minimal cost instrumentation of the program flow-graph can be found by following the steps for the K-S algorithm, but using MAXSPAN to find the spanning tree rather than SPAN. We'll call the minimum cost algorithm MINNODE.

2.2.2 Arc counts

A naive implementation of arc counting in a program flow graph would be to instrument each and every arc of the original graph. This would provide the necessary counts, but rather expensively. The cost comes from two facts: (1) $|E| > |V|$, meaning more space would be required by the instrumentation code; and, (2) to instrument some arcs requires the creation of a new basic block, and the addition of a jump instruction. For such arcs, the instrumentation cost would be $K_I + K_J$, whereas the instrumentation cost for all nodes is simply K_I. These facts have contributed to a commonly held belief that arc counting is too expensive and complicated for practicality. As we shall see, this simply is not the case.

As in the case of basic block counts, it is not necessary to measure each and every arc to derive the number of times each was traversed. A minimum set of arcs to be measured in a graph consists of those arcs not in the spanning tree constructed by SPAN, and the minimal *cost* instrumentation is the set of arcs that form the complement of the spanning tree found by MAXSPAN. We will call the resulting algorithm MINARC.

As it stands, MINARC is too expensive. As noted above, instrumenting an arc requires the creation of a new basic block that is inserted in the arc between its source and sink nodes. For jump arcs, this new basic block must also contain a jump to the original target of the arc; this jump adds to the cost of the instrumentation. This cost is often ameliorated by using transformations to turn arc measurements into node measurements wherever possible. For instance, if an edge that is to be instrumented represents the fall-through of one basic block into another, and the edge is the only edge leaving the source block, then the instrumentation can be inserted in the edge's source block; similarly, if the edge is the only edge entering the target block, then the instrumentation can be inserted in the edge's target block.

Procedure 1 implements a heuristic algorithm using these transformations to reduce the cost of profiling arc traversals, where
ISINK(e)=Add instrumentation code to the front of the block $snk(e)$.
ISOURCE(e)=Add instrumentation code to the front of $src(e)$.
and
ISPLIT(e)=Replace edge e with a new basic block v_e and edges $e_1 = (src(e), v_e)$ and $e_2 = (v_e, snk(e))$. Furthermore, $T(e_1) = T(e_2) = T(e)$, and $F(e_1) = F(e_2) = F(e)$. (As we will see, determining $C(e_1)$ and $C(e_2)$ is problematic.)

These transformations allow us to keep the problem relatively simple. If we added transformations that allowed us, say, to move basic blocks to remove jump instructions, or to make the most frequent arc out of a block the fall-through arc from that block, the problem would be much more complicated. To keep the problem simple, we assume that the linear order of basic blocks in memory will remain the same throughout the process of instrumenting the program. Our only options are to determine *where* to add instrumentation code.

Before the MAXSPAN algorithm can be applied to the program flow graph to find which arcs are to be instrumented, the cost of instrumenting each arc must be estimated. If at all possible, we would like to insert instrumentation code without introducing extra control flow logic or extra basic blocks, so if we can identify where the above transformations can be applied, we can more accurately estimate the cost of instrumenting an arc. This leads to the following heuristic cost estimation algorithm.

Procedure 1 *Instrumenting an arc:*

Input: An arc that is to be instrumented.

Result: Instrumentation code is added to the PFG to count executions of the arc.

Method:

```
Instrument(Arc e)
{
    if not e.crowdedSrc then ISOURCE(e);
    elsif not e.crowdedSnk then ISINK(e);
    else ISPLIT(e);
    endif;
}
```

□

Function 2 *Estimate cost of instrumenting an arc:*

Input: An arc in the PFG.

Output: An estimate of the cost of instrumenting the arc should it be chosen for instrumentation.

Method:

```
InstrumentationCost(Arc e)
{
    if not e.crowded then
        return $K_I \times F(e)$;
    elsif not $J(e)$ then
        return $K_I \times F(e)$;
    else return $(K_I + K_J) \times F(e)$;
    endif;
}
```

□

If the instrumentation code can be appended to a basic block, then the cost of instrumenting the arc e is the cost of the instrumentation code itself, K_I, times the frequency of execution $F(e)$. If the arc is a fall-through arc then the cost is still $K_I F(e)$ even if the arc is crowded: the instrumentation can be inserted between the two blocks. In all other cases, the source block of the arc will be jumping to the instrumentation code, which will itself have to jump to the sink block; therefore, the cost of instrumenting the arc is $(K_I + K_J)F(e)$, where K_J is the cost of the

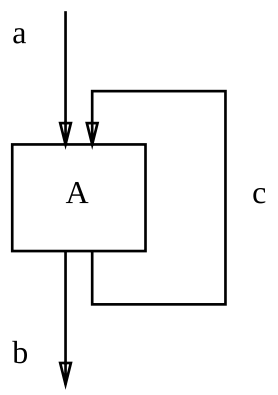

Figure 2.2: Another graph with cheaper instrumentation.

jump instruction. We will call MINARC augmented with the heuristic placement algorithm MINARC$'$.

This approach looks good, and I will present results below to show that it is effective, but there is nothing to suggest that it is complete or produces minimal instrumentations. Rather, it is a set of *ad hoc* rules for utilizing the results of the MINARC algorithm. That it is not complete can be seen from Figure 2.2[2]. If arc c is to be instrumented, and if the execution frequencies of a or b can be derived independently of c, then the instrumentation on c can be moved into the node A. The frequency of arc c is then $F(c) = F(A) - F(a)$, since $F(a) = F(b)$.

This suggests another way of asking the question. We have an algorithm that will find the minimum set of nodes for computing node counts, and an algorithm for finding a minimum set of arcs for computing arc (and therefore node) counts: is there an algorithm that will find a minimum set of nodes *and* arcs for computing execution frequencies for a program flow graph?

An extension to the above algorithms produces a candidate algorithm, which I'll (presumptuously and inaccurately) call OPT. Given a flow graph (V, E), we construct a new one (V', E') as follows. For each node in V we create a corresponding node $v' \in V'$. For each arc $e = v_1 \to v_2 \in E$, we construct a basic block $v'_e \in V'$ and two arcs $v'_1 \to v'_e$ and $v'_e \to v'_2$ in E'. So $|V'| = |V| + |E|$ and $|E'| = 2|E|$.

The K-S algorithm applied to the new graph (V', E') will yield a minimum set of nodes in V' required to compute all frequencies of all nodes in V'. Since each node selected for instrumentation in V' corresponds to either a node or an arc of the original graph, we also have a minimum set of nodes and arcs of (V, E) from whose measurement we can compute all other execution frequencies of nodes and arcs in F. We extend the algorithm to find the minimal cost set of nodes and arcs by assigning the same costs to the nodes in (V', E') as are estimated for the nodes and arcs in (V, E) from which they are constructed. Therefore, if a node $v' \in V'$ corresponds to

[2]Thanks to Jim Wilson of Cygnus Corporation for pointing this example out and for taking the time to convince me it was worth a second look.

MINARC	
$n1 \rightarrow n2$	100
$n3 \rightarrow n4$	27
$n4 \rightarrow n5$	27
$n5 \rightarrow n3$	54
$n5 \rightarrow n6$	81
$n2 \rightarrow n3$	80
$n2 \rightarrow n4$	80
total	449

MINOPT	
n1	100
$n3 \rightarrow n4$	27
$n4 \rightarrow n5$	27
$n5 \rightarrow n3$	54
$n5 \rightarrow n6$	81
n3	67
n4	67
total	423

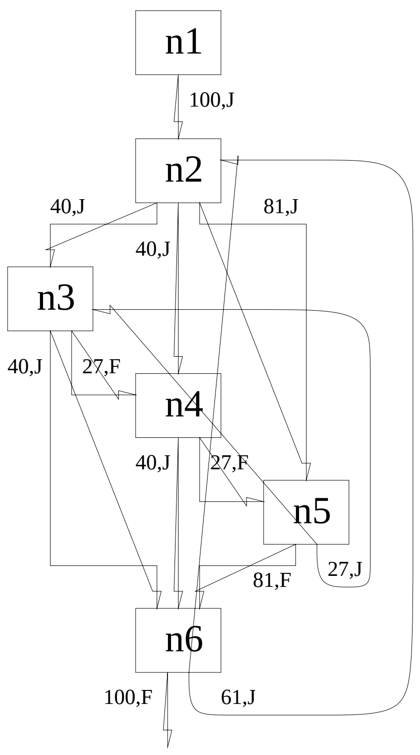

Figure 2.3: A program flow-graph for which MINARC is not optimal

node $v \in V$, then $C(v') = C(v)$ (which is always K_I). If a node $v' \in V'$ corresponds to the arc $e \in E$, then $C(v') = C(e)$. By assigning these costs and invoking a maximal spanning tree algorithm, we find the minimal cost instrumentation using nodes and arcs.

That this algorithm, which we'll call MINOPT, is not equivalent to MINARC$'$, and can sometimes improve on the measurement costs of a program graph is easily proved. In Figure 2.2 MINOPT always instruments node A instead of arc c, unless of course c is executed a lot less than once per execution of A.

Finding PFG instances on which MINOPT produces *different* instrumentations than MINARC$'$ is a bit more difficult. Figure 2.3 is the second simplest subgraph I have found that demonstrates this difference (the simplest is Figure 2.2). In Figure 2.3 each arc is labeled with an execution frequency, and with whether it is a jump arc or a fall-through arc. The two tables show the results of the MINARC$'$ and MINOPT algorithms. The first column contains the objects chosen to be measured, and the second column contains the cost of measuring that object (assuming $K_I = K_J = 1$). The major difference in the results of the two algorithms is that MINOPT has chosen two nodes to measure, while MINARC$'$ can choose only arcs, and then look for transformations to decrease the cost. The instrumentation transformations described previously do not help MINARC$'$ in this example. The only one that applies is ISOURCE($n1 \rightarrow n2$).

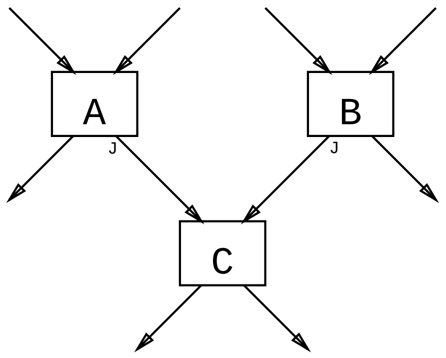

Figure 2.4: The problem configuration

While MINARC′ does almost as well as MINOPT, it is heuristic and not minimal. MINOPT is provably minimal (with respect to a set of instrumentation cost estimates), and can find instrumentations that would be difficult to characterize easily as post-transformations for MINARC. Given that the complexity of both MINARC and MINOPT is $O(|E| \log |E|)$ MINOPT is to be preferred for its simplicity and minimality.

2.2.3 The 'optimal' algorithms are not optimal

The assignment of minimal instrumentation costs to the edges of a program flowgraph has been glossed over in the literature. In fact, such an assignment cannot always be done unambiguously so as to guarantee a minimal solution. That is to say, all optimal solutions shown in the literature are optimal with respect to a specific instrumentation cost assignment on the arcs. But until this time no one has examined the question of how those costs are assigned, or even if they can be assigned, and whether such an assignment still permits an efficient optimal solution. For instance, Cheung mentions that instrumenting some arcs requires extra flow-control instructions, [7, pp. 38-39], but his algorithms assume that these costs can be assigned in linear time, and that the cost of instrumenting one arc does not affect the cost of instrumenting other arcs.

That these assumptions do not hold for even very simple cases can be seen in Figure 2.4 where the arcs to node C are both jump arcs and are both crowded. The cost assignment algorithm described above would assign instrumentation costs of $F(A \rightarrow C)(K_I + K_J)$ and $F(B \rightarrow C)(K_I + K_J)$ to the arcs. However, if either or both arcs are chosen for instrumentation, it is obvious that one of them does not have to jump to block C, but rather can create a new basic block that simply falls through to C. In Figure 2.5 we assume that both arcs going to basic block C are to be instrumented. The arc $B \rightarrow C$ is instrumented by placing its instrumentation code in a separate basic block which falls through to the basic block C. The instrumentation for the arc $A \rightarrow C$ cannot be so configured and hence must use the more

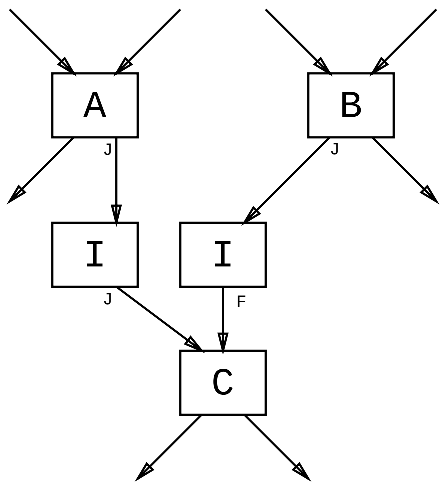

Figure 2.5: ISPLIT vs. ISPLITCHEAP

expensive method for splitting the arc. We can add this transformation to our list of transformation heuristics on page 13:

ISPLITCHEAP(e): Replace edge e with a new basic block v_e and edges $e_1 = (src(e), v_e)$ and $e_2 = (v_e, snk(e))$; $J(e_1) = J(e)$ and $J(e_2) = false$.

If in order to instrument edge e we must use ISPLIT(e), then the instrumentation cost assigned to edge e must be $C_J(e) = F(e)(K_I + K_J)$. If we use ISPLITCHEAP(e), then the cost is $C_I(e) = F(e)K_I$.

Given the above example, it is easy to see that assigning an accurate instrumentation cost to an arc when all that is known is that arc's frequency is not possible: we do not know until the completion of the algorithm which set of arcs must be instrumented, and therefore we don't know whether an arc will need to be ISPLIT or whether it can be ISPLITCHEAP. Surprisingly, it is not possible to assign correct instrumentation costs to the arcs in the above situation even when the frequencies of all arcs entering a node are taken into account. A proof by counter-example is given below in Section 2.4.

The two-step process of assigning instrumentation costs to arcs and then applying a maximal spanning tree algorithm does *not* always produce an optimal instrumentation. There may be a polynomial-time algorithm for finding an optimal instrumentation, but as of this writing, I do not know what it is. If there are p instances of nodes like C in Figure 2.4, then finding the optimal instrumentation could require examining 2^{n^p} spanning trees, trying all n cost assignments at each of the p problem nodes. I conjecture that the problem is NP-complete. A fruitful line of search for a proof might begin with Szymanski's NP-completeness proof for the variable-span branching problem [47], which has some of the same characteristics as the instrumentation-cost problem.

Even so, finding the *optimal* minimal-cost solution in practice does not result in sufficiently more efficient instrumentations to warrant an intense search for a general solution. MINOPT works quite adequately in practice. The non-optimality of the solution matters little because

- we show in Section 2.3 that profiling requires less than 20% overhead, anyway;
- the problem configuration arises only under unusual circumstances (it occurred in only 2.4% of the 20457 basic blocks in our experiments); and,
- of those instances in which it does occur, the performance degradation is estimated to be less than 1%.

Throughout, whenever I refer to 'optimal instrumentation' algorithms, I mean optimal with respect to the instrumentation estimates.

2.3 Empirical data

I created a system that inserts optimal arc-counting in programs and used it to instrument several C programs by some of the algorithms mentioned in the previous section. The programs were first instrumented without the benefit of profile data; the frequencies of all arcs and nodes was one, resulting in a more-or-less random selection of instrumentation points by the algorithms ('random' in the sense that the selected instrumentation points were the result of vagaries of selecting a spanning tree from quick-sorting equi-valued elements). Next, a heuristic was used to assign relative frequencies to arcs: back-arcs and their target nodes were given higher frequencies than the rest on the assumption that a back-arc indicates a loop. Finally, profile data was used to compute a minimal-cost instrumentation.

The results are presented in Table 2.1. Four programs were compiled with `gcc -O` and instrumented: *intmm*, an integer matrix multiply; *compress*, the UNIX compression utility; *troff*, the UNIX typesetting program; and *cc1* of the *gcc* compiler. The first column for each program shows the running time of the program in CPU seconds, while the second column shows the running times of the instrumented versions of the programs expressed as a percentage of the original running time (more precisely, if N is the running time of the program without any instrumentation, and P is the running time of the program with instrumentation inserted, then the second column = $100 \times ((N/P) - 1)$. All programs were run on a Sun 3/140 and were compiled with the GNU C compiler, version 1.37.1. All running times are the average of 10 runs to smooth out system-dependent fluctuations.

The *intmm* program had no input data, and the contents of the matrices were initialized the same for each run (whether they were or not would not have made any difference to the running of the algorithm). The running times shown reflect the best that the various instrumentations could do for that program based

means		intmm		compress		troff		cc1	
not instrumented		29.06		17.03		96.91		28.91	
prof		29.93	3.0%	18.37	7.9%	127.91	32.0%	36.08	24.8%
gprof		31.45	8.2%	21.22	24.6%	164.81	70.1%	46.38	60.4%
FULLNODE		32.08	10.4%	26.78	57.3%	175.18	80.8%	42.40	46.7%
MINNODE	random	31.92	9.8%	20.58	20.8%	138.83	43.3%	36.40	25.9%
	heuristic	31.83	9.5%	21.32	25.2%	139.97	44.4%	37.55	29.9%
	profile	31.73	9.2%	19.90	16.9%	130.44	34.6%	35.72	23.6%
MINARC′	random	33.36	14.8%	20.83	22.3%	131.40	35.6%	36.46	26.1%
	heuristic	33.38	14.9%	20.26	19.0%	131.68	35.9%	37.01	28.0%
	profile	33.32	14.7%	19.94	17.1%	117.40	21.1%	35.21	21.8%
MINOPT	random	33.31	14.6%	20.39	19.7%	131.24	35.4%	36.35	25.7%
	heuristic	32.93	13.3%	20.15	18.3%	141.56	46.1%	37.85	30.9%
	profile	31.89	9.7%	19.89	16.8%	118.99	22.8%	34.90	20.7%

Table 2.1: Profiling overhead

on the profile data: the profile would be exactly the same each run. MINOPT's 9.7% overhead *vs.* MINARC′'s 14.7% reflects the fact that the inner loop in *intmm* mirrors exactly the situation in Figure 2.2. MINOPT was able to find an instrumentation that did not require the expensive arc-splitting that MINARC′ was required to do.

For each of the other three programs, different input was used to create the profile data than was used create the numbers in the table. For *compress*, the profile data was generated by compressing *compress.c*, the source file for the utility. The numbers in the table are from compressing */usr/dict/words*, a 200Kb file containing a sorted list of 25,144 words. *Troff*'s profile data was created by typesetting a 48Kb language reference summary, and the numbers in the table are from typesetting a 190Kb technical report on a bibliographic database browser [49].

The *cc1* profile data was created by compiling *gcc.c*, the 23Kb source file for the process-dispatching front-end of the *gcc* compiler, and *combine.c*, a 46Kb source file for compile-time constant expression evaluation for the same compiler. The measured run compiled *cccp.p*, the 73Kb source file for the Gnu C-preprocessor. The sizes of the source files are *after* all pre-processing commands and all comments were stripped.

Using different input for profiling than for timing runs is necessary to convince us that we simply aren't 'training' the profile algorithms to a specific program. However, it does introduce some anomalies in the numbers in Table 2.1. For instance, MINARC′, using profile data to instrument *troff*, resulted in the instrumented program taking 21.1% longer to run than did the uninstrumented version. This contrasts with MINOPT using profile data on the same program: the instrumented version of *troff* required 22.8% longer (the difference is statistically significant and not due to variations in measurement). Obviously, the instrumentation selected by MINOPT

	intmm	compress	troff	cc1
prof	36813	44189	66309	324357
gprof	20992	25304	42128	204160
FULLNODE	88	1492	10332	68636
MINNODE	56	796	6336	48216
MINARC′	56	808	6408	49040
MINOPT	56	808	6408	49040

Table 2.2: Comparison of profile data size requirements

does not do as well on the set of *troff* input as did the instrumentation selected by MINARC′. Such variation due to variations in the input are to be expected.

The numbers also indicate that the heuristic I used to try to guess which would be the heavily used arcs and nodes in the PFG is quite inadequate: it is better simply to accept a random assignment (i.e., if there is no profile data available). Perhaps a closer analysis of average aggregate program behavior could produce a heuristic that would do better than just random chance.

From the data presented in Table 2.1 we can see that MINOPT is definitely competitive with MINNODE, compares favorably with *prof* and *gprof*, and is definitely better than FULLNODE, the traditional technique for instrumenting programs.

Another benefit of MINOPT is demonstrated in Table 2.2. It shows the number of bytes required for the generated profile data. In all fairness, comparing the output of *prof* and *gprof* with the others is comparing apples and oranges. You can't get execution counts of basic blocks from *prof* or *gprof*, but then you can't get timing estimates from the others. Howver, the difference between FULLNODE and the MIN algorithms is significant.

2.4 Counter-example

In general, it is not possible to assign instrumentation costs to arcs in such a way that an optimal instrumentation can be found using MAXSPAN. To prove this, it suffices to show that there exists one program flow graph for which this is true. To this end, we construct a subgraph and show that for any instrumentation cost assignment algorithm, the subgraph can be embedded in a larger graph that causes MAXSPAN to select a non-optimal instrumentation.

Figure 2.6 shows a portion of a program flow graph that satisfies the criteria. Basic block C has two entering arcs that are both crowded jump arcs. We more-or-less arbitrarily assign execution frequencies to the arcs. The arc $A \to C$ is executed 200 times, the arc $B \to C$ 90 times. We assume that the cost of instrumentation $K_I = 1$ and the cost of a jump instruction $K_J = 1$. The values for these constants could be any non-zero value and we would still be able to find our counter-example,

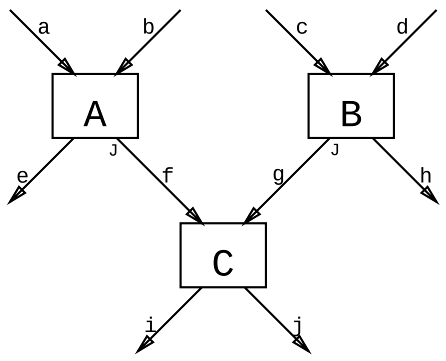

Figure 2.6: The problem configuration.

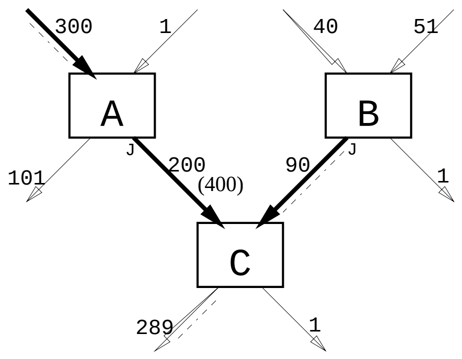

Figure 2.7: Case 1: Estimating that the most frequent arc is split expensively.

but that is not necessary to prove here: we need to show only that there exists one graph for which the algorithm is non-optimal.

The graph surrounding the sub-graph in Figure 2.6 is not shown, but is constructed such that the MAXSPAN algorithm will put nodes A, B, and C in the spanning tree last. This is easily done by creating the surrounding PFG such that each node has at least one entrance or exit arc with a frequency count higher than any arcs in the sub-graph. The nodes in the sub-graph are added to the spanning tree by selecting the arc in the sub-graph with the largest frequency. The arc is chosen from among all of the arcs shown in Figure 2.6, including the entrance and exit arcs of all three nodes. Exactly three of these arcs will be selected by the algorithm to complete the spanning tree.

Before invoking MAXSPAN, instrumentation costs must be assigned to each arc. The question is: does there exist an algorithm for assigning costs to the arcs $A \rightarrow C$ and $B \rightarrow C$ that has as its inputs only the frequencies of those arcs?

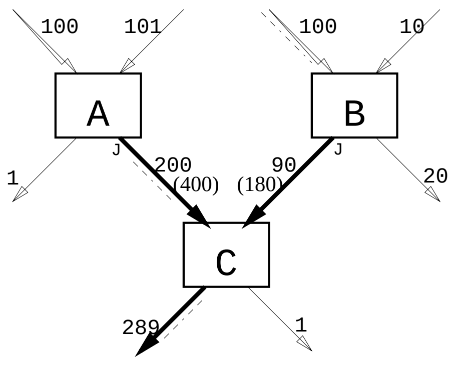

Figure 2.8: Cases 2 and 3: Estimating that the least frequent arc is split expensively, or that both arcs are split expensively.

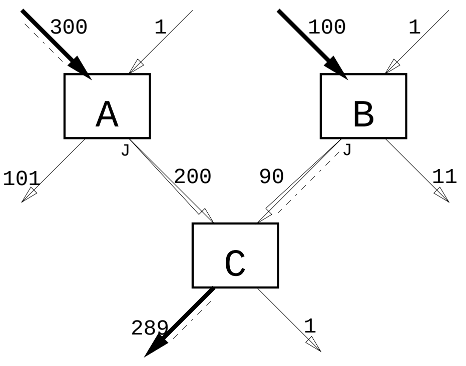

Figure 2.9: Case 4: Estimating that neither arc is split expensively.

If such an algorithm existed it would produce one of four results: it would assign the expensive-split cost to A, assign it to B, assign it to both arcs, or assign it to neither. Figures 2.7 through 2.9 show that no matter which assignment is made, there exists a consistent set of arc frequencies which will cause the MAXSPAN algorithm to misfire and select the wrong arcs for instrumentation – wrong in the sense of being non-optimal. In each figure, each arc is labeled with its frequency, and with its cost estimate in parenthesis if different from the frequency times K_I. For example, in Figure 2.7 the arc $A \rightarrow C$ has frequency 200, but its assigned cost is $K_I + K_J = 2$ times that, or 400. Given these instrumentation cost assignments, the arcs selected by the MAXSPAN algorithm are shown in bold. So in Figure 2.7, the cost of measuring the sub-graph is $1 + 40 + 51 + 101 + 1 + 289 + 1 = 492$. The dashed lines show a better spanning tree resulting in a cheaper instrumentation: in Figure 2.7 that cheaper instrumentation would cost $1+40+51+101+200+1+1 = 395$.

Figure 2.8 shows a set of frequencies for which guessing that the least frequent arc ($B \rightarrow C$) is expensively split, or that both arcs are expensively split, will also fail. Again, the bold arcs are the ones chosen by the MAXSPAN algorithm for inclusion in the spanning tree, while the dashed lines show a better selection, one resulting in a cheaper instrumentation.

Figure 2.9 shows that assuming both arcs will be ISPLITCHEAP (an impossibility in actuality) does not work either. The arcs selected by MAXSPAN result in an instrumentation that costs $1 + 1 + 101 + 200 + \mathit{180} + 11 + 1 = 495$. If neither $A \rightarrow C$ nor $B \rightarrow C$ is put in the spanning tree, then both will be measured. After the MAXSPAN algorithm chooses them for instrumentation, it is easy to see that of the two it is better to ISPLITCHEAP the most heavily used arc; hence the cost of 180 for instrumenting the lesser executed arc $B \rightarrow C$. It would have been better not to measure $B \rightarrow C$, as shown by the dashed lines. This better instrumentation would cost only $1 + \mathit{100} + 1 + 101 + 200 + 11 + 1 = 415$.

Therefore, an algorithm for assigning instrumentation estimates to arcs does not exist that has as its only inputs the frequencies of all arcs entering a node and that depends on the MAXSPAN algorithm to find the minimum instrumentation.

2.5 Problems with counting

There are several complications that must be considered when implementing a profiling system. The first is determining the point during compilation when the profiling code is to be inserted. The system I used operated on the assembly language output of the GNU C compiler. However, if the profiling code is inserted earlier by the compiler, then it is easy to finesse some of the task of counting. For instance simple loops (i.e., reducible sub-graphs with no mid-loop exits) that are executed a compile-time constant number of times do not need to increment a counter on each execution of the loop. In general, if the execution frequency of an arc in a program flow graph is known *a priori* that arc can be removed from the program-flow graph prior to computing the set of instrumentation points. If the removal of the arc results in dis-connected sub-graphs, each sub-graph is treated separately. Even when the loop is not executed a constant number of times, but where strength-reduction can hoist the count increment out of the loop, the loop need not be instrumented. Rather the number of executions can be counted outside the loop and the counter incremented only once. Sarkar [42] shows one method for doing this using dependency graphs. However, it is impossible to tell from his paper exactly how much counting overhead is actually reduced by his technique. Further study and better numbers are needed here.

All of the the results in Section 2.3 computed each function's instrumentation assuming that the number of times each function was called had to be counted. That is, the functions were instrumented one at a time without knowledge of how or from where the function was called[3]. However, if we have profiled the entire program, the number of times a function is executed is simply the sum of the execution frequencies of all call sites that call this function: there is no reason to recompute that number in the instrumentation of the function. Not all programs execute synchronously, as we have implicitly assumed throughout this discussion, nor do all call sites call only one function. If functions are called indirectly, for example by interrupt handling facilities, then it is mandatory that each function be instrumented separately to compute the number of times it was called.

The point to be made here is that these optimizations are possible only if the instrumentation code is inserted prior to the optimizing passes of the compiler.

Another problem for post-pass instrumentation is related to the machine architecture. If the processor's instruction set makes use of condition flags, and if the life of condition flag values extends across basic blocks, then the profile code must preserve those values. This can be done either by saving and restoring the values of the condition flags around the instrumentation code (the method used in our experiments), using an instrumentation sequence that does not change the setting of the condition flags, or by scanning the basic block and inserting the instrumentation just before the first instruction that kills but does not use the condition flags [50]. This situation is much more naturally handled in the compiler proper than in a post-processor.

2.6 Conclusions

I have demonstrated that counting arcs is as cheap as counting nodes, and cheaper than counting every node. The MINOPT algorithm produces instrumentations that take 50-70% of the space required by instrumentations produced by the FULLNODE algorithm, execute in 70-80% of the time and provide significantly more

[3]This method was encouraged by *troff*, the only widely-used non-interactive program I know of that uses inter-procedural *goto*s as a major form of control flow.

information. MINOPT's instrumentations require approximately the same amount of time as MINNODE's and require slightly less space, but, again, they provide significantly more information. (In the next section I present an optimization that uses arc counts and would not work with only node counts.) MINOPT should be the instrumentation algorithm of choice for compilers/systems.

Chapter 3

A Low-level Use: Code Reorganization for Instruction Cache Performance

3.1 Instruction cache utilization

If a compiler has profile data available there are simple optimizations that can take advantage of the data. This chapter explores a simple optimization that requires profile data. Specifically, I will show how to utilize information about the runtime behavior of a program to enhance the performance of that program on architectures with an instruction cache.

There have been many investigations into improving computer performance by reorganizing programs' address spaces on virtual memory machines. In this chapter, I address the question of whether reorganization can be beneficial for machines with caches and examine the costs required to achieve improved performance. If an inexpensive way can be found to reorganize the address space of a program such that a small cache with code reorganization can have the performance of a larger cache without reorganization, the smaller inexpensive caches would be a more competitive choice.

Instructions that are executed close together in time are *temporally* local. Instructions that are close together in the address space are *physically* local. A cache turns temporal locality into physical locality by holding the most recently executed instructions in faster memory. Exactly how a cache should be implemented in hardware and which strategies for replacing the data in the cache are topics that have received a great deal of study. For an overview of cache designs and organizations, see Smith's survey article [45].

If we let C represent a cache, where each line i of the cache has an address $C(i).addr$ and contents $C(i).instr$ (the contents is an instruction for instruction caches), then when address a is referenced, the cache is examined to see if a is

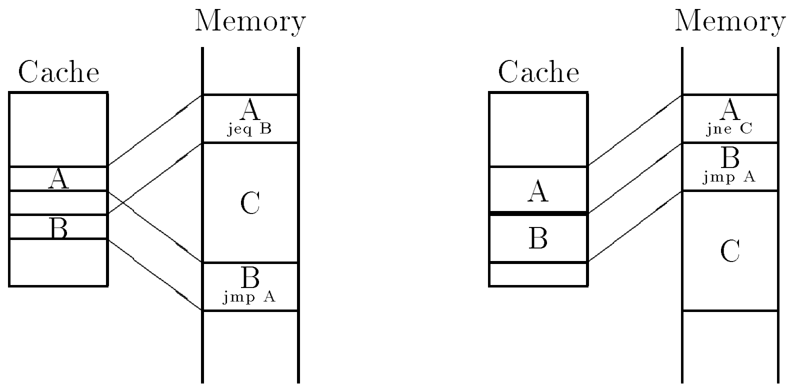

Figure 3.1: Removing cache contention by reorganizing

already in the faster memory. If $a = C(i).addr$ for some i, then the contents of that line is returned as the contents of the referenced memory address.

A fully-associative instruction cache is one that searches in parallel each line of the cache for the referenced address; i.e., if $C(i).addr = a$ for some i, then return $C(i).instr$. In a direct-mapped instruction cache, on the other hand, the address and its contents can be in only one line of the cache. Which line is (usually) determined by the low-order bits of the address; i.e., if $C(\text{lower bits of } a).addr = a$ then return $C(\text{lower bits of } a).instr$. Where in a fully associative cache an address and its contents can be placed in any line of the cache, in a direct-mapped cache, an address and its contents can be put in only one place; hence the name direct-mapped. The hardware required to do the parallel search is expensive to build, while a direct-mapped cache is much simpler and less expensive.

In either case, we are interested in several statistics as indicators of how well a cache performs. A critical statistic is the *miss ratio*: the number of times an address was referenced and it was not found in the cache. The dual of the miss ratio is the *hit ratio*: the number of times an address was referenced and found in the cache. By definition, then, miss ratio $= 1-$ hit ratio.

There are only two ways to improve the performance of a program in a cache: (1) decrease the probability that frequently-executed sections of the program compete for cache resources (Figure 3.1); and (2) increase the amount of useful information in the cache (Figure 3.2).

In Figure 3.1, assume the code at block A and the code at block B are the active portions of a loop. Assume further that, due to the size of the infrequently executed block C, blocks A and B are mapped to the same locations in the direct-mapped cache, as shown on the left. The loop containing these blocks can be made more efficient by moving A and B with respect to one another so that they do not

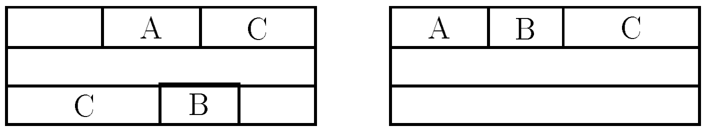

Figure 3.2: Improving cache utilization by reorganizing

conflict in the cache, as shown on the right.

In Figure 3.2 assume the blocks A, B, and C are of such a size that A and B could fit in a cache line. The cache lines on the left represent one way they might map into a direct-mapped cache, with the side-effect of loading infrequently executed code from block C into both lines. In the cache on the right the initial cache miss that loads the code from A also loads B, saving at least one cache miss in the execution of the loop; also, infrequently executed code from C takes up much less space.

For a fully associative cache there may be ways of reorganizing a program to improve its performance with respect to (2); little can be done as far as (1) is concerned. With direct-mapped caches, however, both (1) and (2) suggest easy ways to gain performance improvement. In a direct-mapped cache, contention is a function of the addresses of the competing program segments, which is easily controlled by a loader and/or compiler.

Mark Hill argues that direct-mapped caches are not only cheaper and easier to build [20], they also can give equivalent performance as more complex cache arrangements for the same silicon acreage invested. I have developed an algorithm called *Greedy Sewing* that uses arc counts to reorganize code for improved performance in direct-mapped caches, and that is independent of the parameters of the target cache.

While there have been published results for organizing *data* in memory to improve cache performance, there has been little published regarding rearranging the instruction space. For instance Janet Fabri [11] and K. O. Thabit [48] both discuss methods for improving the cache behavior of a program's data accesses, but say little about the behavior of the code itself.

Some recent work on code reorganization include Scott McFarling's work at Stanford [34], work done at Hewlett-Packard by Pettis and Hansen [36], and Hwu and Chang's work on combining reorganization with the in-lining of procedures [21]. McFarling's work differed from mine by concentrating on positioning basic blocks based on their frequency counts and by utilizing knowledge of the target cache. Hwu and Chang extended my work by doing actual in-lining (as opposed to my pseudo-inlining). Pettis and Hansen pretty much duplicated my work, with the exception that their algorithm reorganizes the entire program instead of just those areas where the vast majority of the improvement is gained.

3.2 Greedy Sewing

A program control-flow digraph (or program flow graph or PFG for short) is a set of basic blocks V and directed arcs E. If $e = v \rightarrow w$ for nodes $v, w \in V$ then we define $src(e) = v$ and $snk(e) = w$. Associated with each arc e (basic block v) in the graph is a positive integer $F(e)$ ($F(v)$) representing the number of times this arc (basic block) was executed during the execution of the program (so $S = \sum_{e \in E} F(e) = \sum_{v \in V} F(v)$). Associated with each basic block v are functions *onThread(v)* that returns the thread that basic block v is on, *onHead(v)* that returns true if the block v is at the head of its thread, and *onTail(v)* that returns true if v is at the tail of its thread. Given threads $t, s \in \mathcal{T}$, we define the procedure *append(t,s_1,s_2,...)* to concatenate the threads s_i in order onto thread t, and delete threads s_i from $\mathcal{T}$. The functions $first(t)$ and $last(t)$ return the first and last blocks, respectively, on the thread t.

The basic idea is to sew threads together such that the order of the basic blocks in a thread tends to improve the correspondence between the static spatial locality of basic blocks with their dynamic temporal locality. We define the function *canStitch(u,v)* to return true if the nodes u and v can be concatenated onto the same thread; this is true only if $u \neq v$ and *onHead(u)* and *onTail(v)* are both true.

We define the procedure *Stitch(e)* to 'sew' two threads together:

Procedure 3 *Stitching Basic Blocks:*

Input: An edge e in a PFG; a set of threads of basic blocks $\mathcal{T}$.

Result: If the source and sink blocks of e can be concatenated, the set $\mathcal{T}$ is modified such that it contains one less thread due to the concatenation of the two member threads.

Method:

```
$Stitch(e\text{: } Arc)$
begin
if $canStitch(e)$ then
    $append(\ onThread(src(e))\ ,\ onThread(snk(e)))$;
end
```

□

Using these functions, we are now ready to lay out a preliminary version of the Greedy Sewing algorithm.

Algorithm 4 *Greedy Sewing Algorithm(1):*

Input: A PFG (V, E), and a parameter p such that $0 \leq p \leq 1$; and a set $\mathcal{T}$ of threads that are initialized such that each basic block is on its own thread; after initialization *onHead(v)* and *onTail(v)* are true for all $v \in V$.

Result: The set of threads $\mathcal{T}$ is modified to indicate the relative ordering of the basic blocks in V. A thread $t \in \mathcal{T}$ specifies the order in which the basic blocks are to be placed contiguously in memory. There is no implied ordering of basic blocks on different threads.

Method: The parameter p is used to specify what portion of the arcs will be examined. That is, a set of arcs $\mathcal{A}$ will be processed where for all $a \in \mathcal{A}, F(a) \geq F(b)$ for all $b \notin \mathcal{A}$, and $\sum_{a\in\mathcal{A}} F(a) < p * \sum_{e\in E} F(e)$. That is, setting $p = .90$ would cause the main loop of the algorithm to be repeated until sufficient arcs had been processed to account for 90% of all arc traversals. This means that usually 5-10% of the arcs, and even fewer basic blocks, need be reorganized.

```
Greedy($p$: real; $\mathcal{A}$: Set(Arc))
begin
assert($0 \leq p \leq 1$);
$S \leftarrow p \times \sum_{e\in\mathcal{A}} F(e)$
$s \leftarrow 0$
while ($s < S$) do
    Select $e \in \mathcal{A}$ such that $F(e)$ is maximum.
    $E \leftarrow E - \{e\}$
    $s \leftarrow s + F(e)$
    if canStitch(src(e), snk(e)) then
        Stitch(src(e), snk(e))
    endwhile
end
```

□

In preliminary tests of the greedy algorithm, several situations were observed that this simple algorithm did not handle adequately. While the majority of program improvement comes from the simple version of Greedy Sewing, it does not take very many special cases to eat into those savings. Based on observations in these preliminary runs, the simple algorithm was enhanced with some checks for special cases. For instance, consider the flow graph in Figure 3.3. If the path $A \rightarrow B \rightarrow D$ is the more frequently executed path, then the thread ABD will be formed (1). Since C cannot be sewn to either A or D now, it remains a singleton thread (2). If it is very infrequently executed, then it makes little difference where C is placed relative to the thread ABD. However, if the path $A \rightarrow C \rightarrow D$ is only slightly less frequently executed than the path $A \rightarrow B \rightarrow D$, and if the sum of the sizes of A, B, C, and D

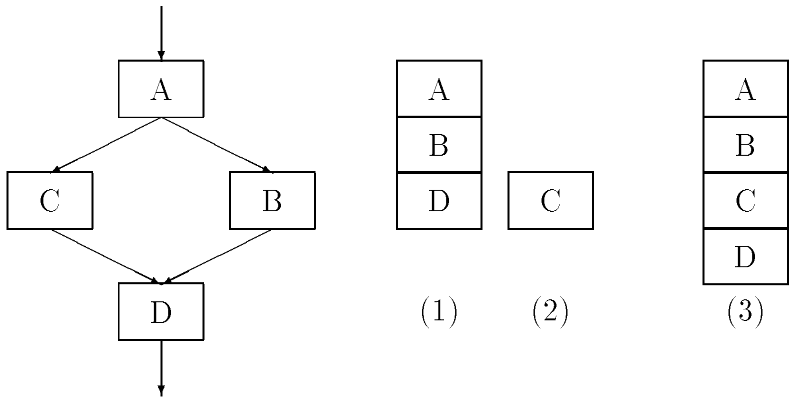

Figure 3.3: Two threads from if-then-else

are small enough that they might all fit in a cache, then the single thread $ABCD$ in Figure 3.3(3) is preferable over the two threads (1) and (2).

Since the Greedy Sewing Algorithm is general and does not depend on any particular cache configuration or size, it cannot know whether any set of basic blocks will fit in a cache, and so uses a heuristic to attempt to capture instances of this configuration of basic blocks. The function *isSmallEquiCondl* checks for basic blocks matching exactly this configuration—i.e. the basic block ends with a conditional jump instruction, the two arms of the conditional have at most one basic block in them and are very nearly equi-probable—and when found the procedure *StitchCondl* creates the longer thread. The actual mechanics of putting a small if-then-else on a thread is straightforward in procedure *StitchCondl* (see the next page).

A second common, but more complicated, situation is pictured in Figure 3.4 where a procedure P is called from a basic block A. We want to concatenate block P_r with block B because the same considerations that applied to the previous if-then-else example apply here: if the frequent path through the procedure is small enough such that A, the frequently executed portions of P, and B could fit in the cache, then we would like to construct the thread shown on the right of Figure 3.4. The procedure *StitchCall* effectively constructs the arc $P_r \rightarrow B$ such that P_r and B are eventually made contiguous. When the bottom of A is sewn to the top of P_0, we say that procedure P has been *pseudo-inlined*. A basic block containing a single jump instruction is inserted between the call and the target to maintain the semantics of the original code.

That is the simple view of pseudo-inlining. It is complicated by the fact that at the time we invoke *StitchCall* on the arc e (using the notation in the example in Figure 3.4, it will be one of arc $A \rightarrow B$ or arc $A \rightarrow P_0$, depending on the vicissitudes of the sorting algorithm) we have not yet encountered the return block P_r, and may

Function 5 *isSmallEquiCondl:*

Input: An arc $e \in E$.

Output: Returns true if $src(e)$ is the head of a small if-then-else with approximately equi-probable true and false arms. A global constant δ defines what is meant by equi-probable.

Method:

```
isSmallEquiCondl(e: Arc) return boolean
begin
   if src(e) ends with a conditional branch (and therefore has
              two exit arcs e and w)
      and snk(e) has one exit arc $e_x$
      and snk(w) has one exit arc $w_x$
      and $snk(e_x) == snk(w_x)$ --they go to the same block
      and $snk(e_x)$ != $src(e)$   --they do not make a loop
      and $snk(e_x)$ != $src(e_x)$
      and $(|F(e) - F(w)|/(F(e) + F(w))) < \delta$
      then
            return true
      else
            return false
end
```

□

Procedure 6 *StitchCondl:*

Input: An arc $e \in E$ that is one arm of a basic block that ends in a conditional branch instruction. Assumes that *isSmallEquiCondl(e)* is true.

Result: A new thread is added to $\mathcal{T}$ that contains the four basic blocks of the if-then-else.

Method: Let w and e_x be defined as in the function *isSmallEquiCondl*. Then a new thread is created by concatenating

```
append(onThread(src(e)),
          onThread(snk(e)),
             onThread(snk(w)),
                onThread(snk($e_x$))).
```

□

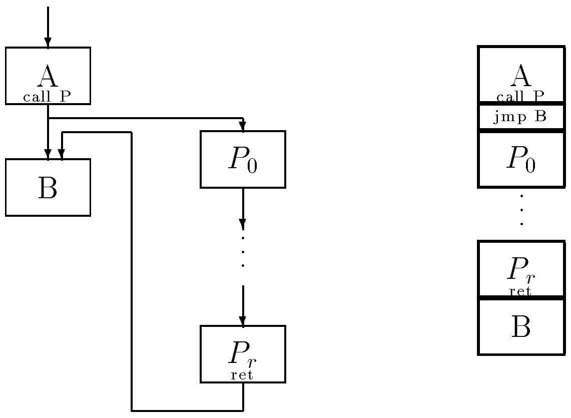

Figure 3.4: Pseudo-inlining

not encounter it if it is not one of the hot spots of the program. During the normal operation of Greedy Sewing all blocks ending with a return instruction will end up at the tail of a thread: there are no exit arcs from a return block. At the same time, we do not want basic block B, the one following the call instruction, to be threaded with a less frequently occurring basic block.

Let the function $target(v)$ return the basic block that is the target of the call instruction that ends block v (undefined otherwise), and let $returnsTo(v)$ be the basic block to which the called procedure returns. Then, whenever an arc e is selected for which *isCall*($src(e)$) is true (i.e. the basic block $src(e)$ ends with a call instruction), the procedure *StitchCall* (Procedure 7) modifies block $B = returnsTo(src(e))$ so that $onHead(B)$ is false (even though B is still (on) a singleton thread) and adds $src(e)$ to a set of remembered basic blocks $\mathcal{R}$. At the end of the Greedy Sewing Algorithm, the procedure *append_R_blocks* (Procedure 8) is invoked to append all of the blocks $r \in \mathcal{R}$ to the appropriate threads.

The entire Greedy Sewing Algorithm used in our experiments is given in Algorithm 9.

3.3 Results

I used the profile-collection techniques of the previous chapter to collect arc frequencies of several programs. After profile data was collected, the program *reorgBBs* then read the original assembly language files and reorganizes them based on that profile.

3.3.1 The Programs and Traces

There were three programs chosen for experimentation and each program had four versions created: the normal, unreorganized version produced by the Gnu

Procedure 7 *StitchCall:*

Input: An arc $e \in E$ such that $isCall(src(e))$ is true.

Result: The target of the call in $src(e)$ is pseudo-inlined into the thread containing $src(e)$. The set of threads T and the set of remembered basic blocks R are modified so that the pseudo-inlining can be completed later.

Method:

```
$StitchCall(e\text{: } Arc)$
begin
assert($isCall(src(e))$)
$t \leftarrow target(src(e))$
$r \leftarrow returnsTo(src(e))$
$append(onThread(src(e),t))$
$onHead(r) \leftarrow$ false;
$add\ src(e)\ to\ \mathcal{R}$
end
```

□

Procedure 8 *append_R_blocks:*

Input: The set R of return blocks; the set of threads T.

Result: The threads containing the return blocks are appended to the threads containing the corresponding call blocks. All return blocks are at the head of a thread, even though *StitchCall* modified them to appear otherwise.

Method:

```
$append\_R\_blocks(\mathcal{R}\text{: } Set(Node))$
begin
for each $e \in \mathcal{R}$ do
    $append(onThread(e),\ onThread(target(e)))$
    endfor
end
```

□

Algorithm 9 *Greedy Sewing Algorithm(2):*

Input: A PFG (V, E); a parameter p such that $0 \leq p \leq 1$; and a set of threads $\mathcal{T}$ that are initialized such that each basic block is on its own thread; after initialization of $\mathcal{T}$ *onHead(v)* and *onTail(v)* are true for all $v \in V$.

Result: The set of threads $\mathcal{T}$ is modified to indicate the relative ordering of the basic blocks in V.

Method: The parameter p is used to indirectly specify what portion of the arcs will be examined.

```
Greedy(p: real; E: Set(Arc))
begin
$S \leftarrow p \times \sum_{e \in E} F(e)$
$s \leftarrow 0$
$\mathcal{R} \leftarrow \emptyset$
while $(s < S)$ 'do
    Select $e \in E$ such that $F(e)$ is maximum.
    $E \leftarrow E - \{e\}$
    $s \leftarrow s + F(e)$
    if isSmallEquiCondl(e) then
        StitchCondl(e)
    else if isCall(e) then
        StitchCall(e)
    else if canStitch(src(e), snk(e)) then
        Stitch(src(e), snk(e))
    endwhile
append_R_blocks($\mathcal{R}$);
end
```

□

name	trace length unreorganized	trace length reorganized $p = .80$	$p = .90$	$p = .95$
SCRUNCH	9,405,156	9,656,437	9,715,946	9,693,277
TROFF	8,059,174	8,343,440	8,325,470	8,338,350
CC1	8,263,593	8,268,313	8,293,376	8,301,226

Table 3.1: Summary of traces: number of instruction words fetched

name	number of basic blocks	number of blocks reorganized $p = .80$	$p = .90$	$p = .95$
SCRUNCH	1,233	22 (1.8%)	27 (2.2%)	32 (2.6%)
TROFF	4,000	149 (3.7%)	207 (5.2%)	318 (8.0%)
CC1	26,407	727 (2.8%)	1,317 (5.0%)	1,939 (7.3%)

Table 3.2: Summary of traces: number of basic blocks

C compiler, and three reorganized by the Greedy Sewing Algorithm with p set to .80, .90, and .95. A summary of the programs, the basic block counts, and trace sizes is in Tables 3.1 and 3.2. I collected a trace of each of these twelve programs which were then used as input to Mark Hill's DineroIII cache simulation program [19]. Each trace was simulated on eleven different cache configurations: 256 bytes with 4 and 8 byte blocks; 1024 byte cache with 4, 8, and 16 byte blocks; and 4096 byte cache with 4, 8, and 16 bytes, each using single associativity (direct-mapped) and two-way set associativity.

The first program was *scrunch*, a Huffman encoding algorithm. The profile was generated by *scrunch*ing a 200K spelling dictionary. The trace was created by *scrunch*ing *scrunch.c*, a 42Kb C source file.

A second program, *troff*, was chosen because of its wide use in UNIX environments. The profile was generated by *troff*ing three separate technical documents, chosen to represent a broad and typical use of the program. The first document consisted of 103K bytes after being preprocessed by *tbl*, *eqn*, and *grn*, This included 1933 lines (32K bytes) of *troff* commands, the remainder being plain text. The other two documents totaled 228K bytes and contained 4004 lines (73K bytes) of preprocessed *troff* commands. The trace was created by *troff*ing a reduced version of the first document of length 7705 bytes, of which 273 lines (2728 bytes) were *troff* commands.

A third program was the Gnu C compiler itself. The profile was collected of the compiler compiling three Gnu C source files: *toplev.c*, *loop.c*, and *recog.c*. They totaled 79K bytes, with 20, 12, and 15 C function definitions, respectively. The trace was collected while compiling *genemit.c*, a 6Kb file containing nine function definitions.

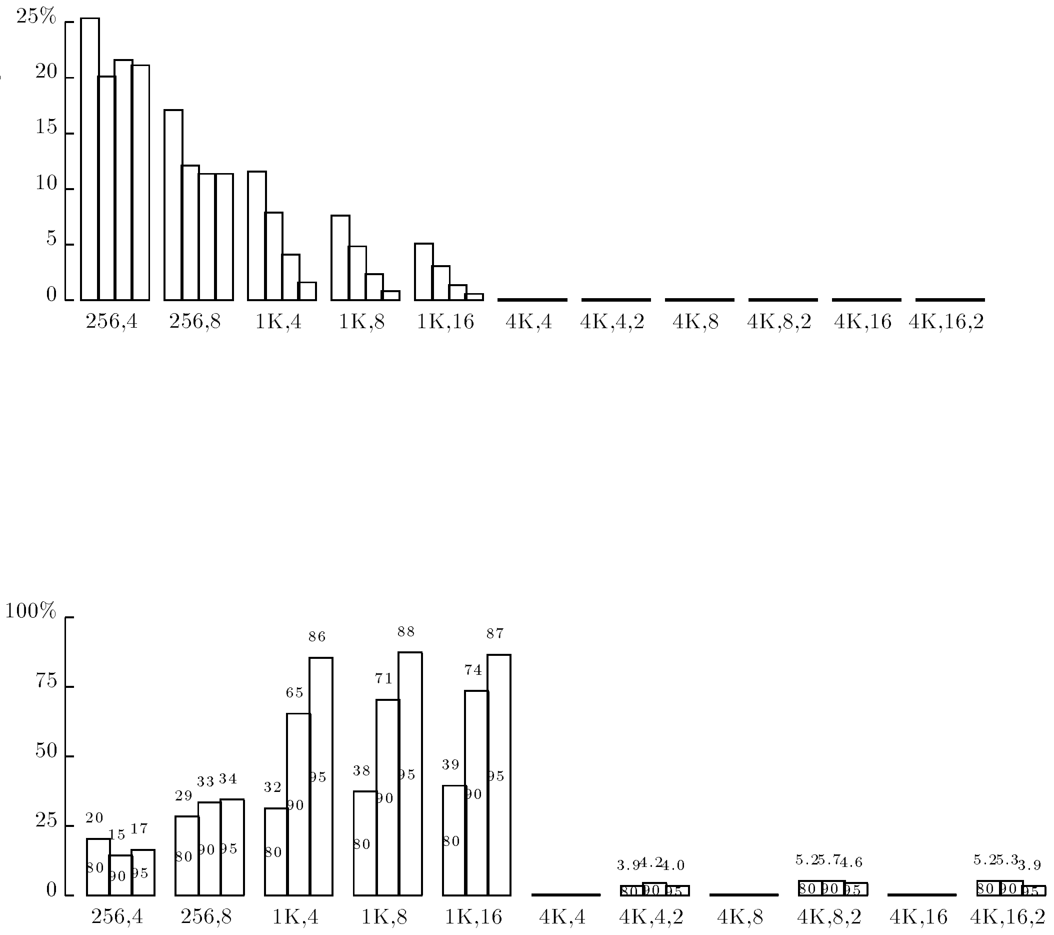

Figure 3.5: Scrunch miss rates and percent improvement

3.3.2 Miss Rates

Figures 3.5, 3.6, and 3.7 are histograms of the results of running the traces of the programs through the cache simulator. (Since the reorganization algorithm didn't take any cache parameters into account, the same traces are used in all eleven simulation runs for each program.) They show the miss rates for the four versions of each program on each of the eleven cache configurations. The leftmost bar of each group of four is the average miss rate of the unreorganized program. The other three of the group, from left to right, are the average miss rates for the reorganized versions for $p = 80\%$, 90%, and 95% respectively. The cache configuration is noted beneath each group. Figures 3.5, 3.6, and 3.7 show the improvement in miss rates of the reorganized versions over the miss rates of the unreorganized versions (i.e. $1 - (M_r/R_r)/(M_u/R_u)$).

There are instances where reorganization can buy the (miss rate) equivalent of a larger cache. For example, looking at Figures 3.5, 3.6, and 3.7 we see that the reorganized programs using a 256 byte cache with 8-byte blocks consistently had as good as or better miss rates than its unreorganized version running in a 1K cache with 4-byte blocks.

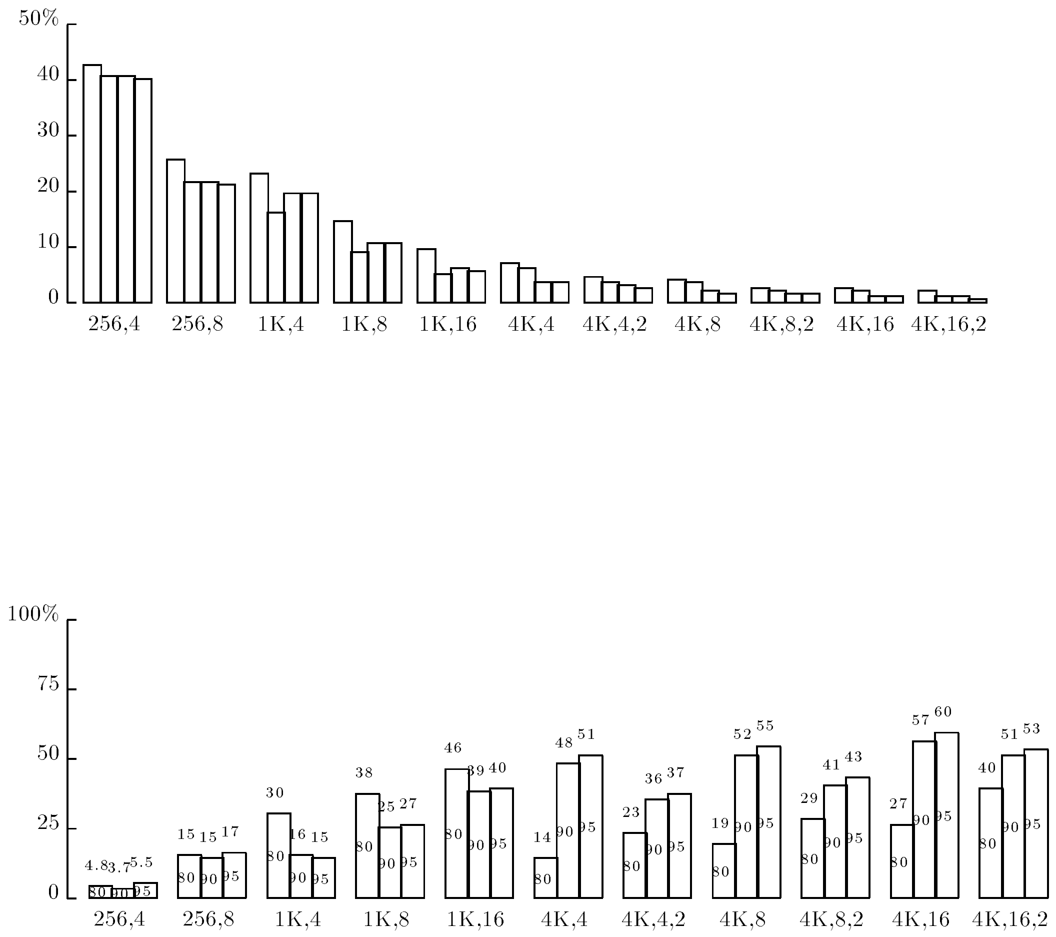

Figure 3.6: Troff miss rates and percent improvement

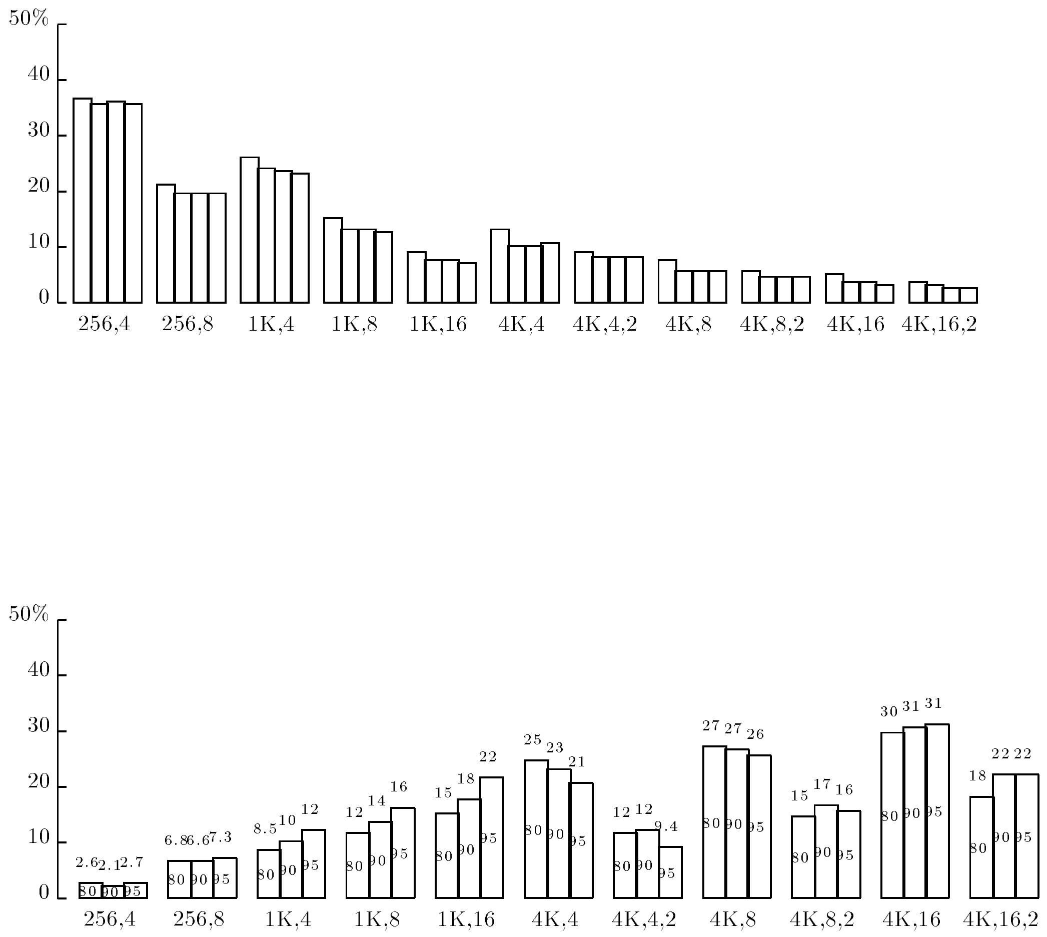

Figure 3.7: Cc1 miss rates and percent improvement

3.3.3 Performance improvement

Let R_u be the number of instruction fetches on the original, unreorganized program, and let R_r be the corresponding number for a reorganized version. For the sake of these estimates, we will assume $R = R_u = R_r$. We see from Table 3.2 that this is not strictly true, but they are sufficiently close for our purposes here. Let M_u be the number of cache misses and H_u the number of hits when the original, unreorganized program is run, and M_r and H_r be the corresponding values for the reorganized program. Define the miss rates $m_u = M_u/R$ and $m_r = M_r/R$, and let $m_\Delta = m_u - m_r$ be the difference in miss rates. Let t_h be the time required to handle a cache hit, and let t_m be the time required to handle a cache miss. The running time of the original program is then $T_u = t_h h_u + t_m m_u$ and the improved running time is $T_r = t_h h_r + t_m m_r$. Finally, define $f = t_m m_\Delta / T_u$, the fraction of the original program's time taken up by cache misses that are turned into cache hits, and $K = t_m/t_h$ the ratio of the cost to handle a miss to the cost to handle a hit. Then by Amdahl's Law:

$$T_r/T_u = (1 - f) + f/K. \tag{3.1}$$

We can now estimate the improvements in performance from reorganization taking our example from the specification of the SPUR memory architecture [18]. In general, the cost of a miss is very high on multi-processor, shared-bus systems due to bus contention or the length of the cache line. SPUR has a 512-byte on-chip cache and 128Kb off-chip cache. According to Mark Hill, a miss in the on-chip cache costs three times as much as a hit, assuming the instruction to be in the off-chip memory cache [20]. Let us assume the on-chip cache shows a normal miss rate of about 20%, and that we can improve that to 15% by reorganizing. Then $f = 3{*}.05/(3{*}.20{+}.80) = .107$. Plugging this into (1) above, we get $T_r/T_u = .929$, i.e. the program executes in only 92.9% of the time of the original, a 7.1% improvement. The maximum possible improvement is 29.6% assuming the unattainable miss rate of 0%.

For the SPUR architecture, an off-chip cache miss will cost 12 to 20 times that for handling a cache hit. SPUR therefore has a very large mixed cache to combat this penalty. If we assume that reorganization can reduce SPUR's miss rate by an absolute 0.25% (e.g. from 1% to 0.75%), then, assuming $K = 17$ (a number lifted from Katz and Eggers [26]), $f = 17 * .0025/(17 * .01 + .99) = .0366$. Plugging this into (1) above, we get $T_r/T_u = .966$, a 3.4% improvement in performance. This is in addition to the performance improvement for the on-chip cache noted above. With these assumptions, we predict reorganization can improve SPUR's performance by about 10%.

This prediction is consistent with other numbers recently published for similar systems. Pettis and Hansen [36] at Hewlett-Packard Laboratories report 10%-26% improvement on a machine with a 16Kb unified cache (the HP-UX 825). When they increased the cache size to 128Kb, the improvements decreased to less than 10% on the HP-UX 835; all five benchmarks' improvements averaged 5%, but with a very wide range of differences (0.8% to 9.3%). Their study concentrated on placing code to improve cache performance utilizing information about the size of the cache. They also used arc counts, rather than node counts, but their algorithm operated over the whole program, not just the hot spots. They also counted each and every arc in the flow graph, which may account for the compiler going twice as slowly when instrumenting the program. Their algorithm for positioning the code slowed the compiler down by about 15-20%.

Figures 3.8, 3.9, and 3.10 show the theoretical improvement in execution performance of the reorganized versions over the original unreorganized versions when the cost factor K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25. Each graph has a column for each of the eleven cache configurations. A line within a column plots the expected performance of the indicated program on that cache when reorganized with the three values $p = .80, .90, .95$ moving from left to right. $K = 2$ is the very top line in each column, and $K = 25$ is the bottom-most line in each column, yielding a range in which I expect reorganization to improve the performance of the programs. So we see that for a 1K cache with 4-byte blocks and $K = 2$, a version of *scrunch* reorganized with $p = 80\%$ would take about 97% as long as the unreorganized version (the left end of the topmost line in the third column from the left). It would take only about 77% as long if $K = 25$ (the left end of the bottom line in that column), and only about 38% as long when $K = 25$ and $p = 95\%$ (the rightmost end of the bottom line in that column).

3.4 Limitations

The algorithm for *StitchCall* is not quite correct, since it does not handle nested procedure calls correctly. The net effect on the numbers is not at all clear, but it should be slight in whichever direction it goes. There simply were not that many sequences of nested procedure calls that could fit in a cache in the code I used as test cases. This would not be true in languages that encouraged the use of many small procedures, e.g., C++.

Furthermore, it is not clear whether pseudo-inlining a procedure in only one location is sufficient. Further tests should be performed to determine whether it is worthwhile to copy the main thread of execution of a frequently executed basic block into multiple locations. If I had to guess as to which would have the most effect on the results—correcting the nested procedure call problem or making multiple copies of threads of procedure execution—I would say that thread copying would probably have more effect.

There is a potential problem in *isSmallEquiCondl* in that it will not handle correctly a small if-then-else where one of the arms of the conditional is empty, and the common exit block loops on itself. This situation never arose in my experiments, and so the potential problem was not detected until this writing.

Figure 3.8: Scrunch performance for K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25

Figure 3.9: Troff performance for K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25

Figure 3.10: Cc1 performance for K = 2, 3, 4, 5, 6, 7, 10, 15, 20, and 25

I have not considered many of the parameters for designing a cache that may be relevant. For example, I have not considered the increase in bus contention caused by going to a larger block size. Nor have I considered alternative cache management strategies such as sub-block placement. My goal was simply to explore improving the performance of the cheapest of these design alternatives with simple compiler enhancements.

I have not considered cache effects such as cold start misses or cache flushes due to system interrupts or context switches. I did not consider any parameters of the target caches when reorganizing code. It was not clear at the beginning of this research how much the parameterization of the algorithms by the cache characteristics would benefit the program's performance; hence I went for the simple solution first.

I haven't solved the problem of *case* statements satisfactorily. Currently, it is possible for the reorganizer to move the code around to such an extent that the jump table can end up quite a distance away from one or more of its targets. On the 68020, this presents a practical problem since jump tables with half-word *pc*-relative entries are much faster than full word entries. If an item of a case is a "hot spot", it is very difficult to relocate it and still satisfy the distance constraint of jump tables. The only program that gave me real problems was the Gnu C compiler, for which I generated full-word jump table entries. This does not change any of the *miss* rate results significantly, but in real life, it would be an unacceptably slow implementation due to the slower execution of the table jump.

Finally, there are architectures that present difficulties for the Greedy Sewing algorithm. An example is the MIPS-X instruction set [8], which has asymmetric conditional branch instructions. Due to the nature of the MIPS-X pipeline, each conditional branch instruction is followed by two instructions that are fetched before the CPU has determined whether the branch will be taken. Each conditional branch instruction also has a *squash* bit that, if 'on', prevents the execution of these two instructions if the branch is not taken. Both of these delay slot instructions are always executed whenever the branch is taken: there is no way to *squash* their execution when the branch is taken. This makes sense if all programs follow the pattern of code generated by most compilers, where the conditional test is at the 'bottom' of the loop and the conditional branch is, therefore, almost always taken.

However, the results of Greedy Sewing's reorganization described here turns those statistics upside down. The algorithm almost guarantees that the head and tail of a frequently-executed loop will be made contiguous, and that execution will almost always fall through the conditional test to the head of the loop: the conditional branch is almost never taken. This leaves three options: (1) fill the delay slots as is currently done by MIPS-X compilers, followed by a jump instruction to the infrequent target (normally the loop-exit); (2) reverse the sense of the conditional and try to fill the delay slots with instructions that don't have to be squashed when the branch is *not* taken (because there is no squash bit for this direction); or, (3) punt and put no-ops in the delay slots.

Option (1) puts infrequently executed instructions right in the middle of high-frequency basic blocks, working against one of the aims of reorganization (better cache utilization in high frequency code). Option (2) sounds plausible, but it is difficult to find instructions that can always be executed no matter which way the branch goes. It may be possible to generate instructions to un-do the effects of the delay-slot instructions when the branch is finally taken, but this begins to get complicated and presents the possibility of really slowing down a frequently executed inner loop. Option (3) is an obvious loss. McFarling [34] implemented option (1), and reports that the size of repositioned code increases about 14%.

Due to the fact that the available MIPS-X compilers filled the delay slots before emitting assembly language code requiring my software to have a MIPS-X assembly language parser, and given the fact that the only profile data I had was the basic block counts generated by their compiler, applying Greedy Sewing to MIPS-X code was too far outside the reach of this research.

3.5 Conclusions

Profile driven code reorganization definitely improves the performance of programs. In envisioned programming environments where profile data is a permanent part of the information manipulated by both programmer and compiler, these improvements would come simply and cheaply. My experiments have shown improvements in miss rates on the order of 30% to 50%, and sometimes as high as 50% to 80%. These figures were obtained by relocating only 3% to 8% of the basic blocks of typical programs.

Chapter 4

A High-level use: Implementation selection of abstract data types

In the previous chapter, I demonstrated one way in which profile data could be used profitably by an optimizing compiler at a low level. Code reorganization is attractive since it improves performance at a small cost, that cost being the time it takes to decide the order in which to emit a few of the basic blocks of a program.

In this chapter, we will look at what a compiler might do with profile data at a very high level. Specifically, I developed TypeSetter, a system for selecting implementations for abstract data type representations and functions. By assuming that profile data exists for a program, we have seen that low-level program transformations can use very simple algorithms to achieve improvements comparable to much more complicated algorithms. With TypeSetter I tested whether comparable simplifications could be made in selecting implementations of high-level abstract data types.

4.1 The Problem

Before stating the problem, it is useful to differentiate between two classes of programmers that would make use of TypeSetter. The *User* of TypeSetter would write a program using only the available abstract data types, and would not be concerned with how those abstractions were eventually implemented (as long as the implementations were relatively inexpensive, of course). The *Implementor* is the programmer that adds implementations to the TypeSetter system. I do not envision that TypeSetter can be (or should be) an extensible language system at the User level. If new implementations are to be added, it is an enhancement to the system, and not simply the shipment of a new library. This difference between User and Implementor allows us to discuss efficiently the difference between using an abstraction and implementing it by referring to the specific programmer.

The general problem can be stated simply: what is the most efficient implementation of a User's program? This includes selecting the most efficient instruction sequences as well as the best implementations for the program's data structures. Most compilers completely side-step this latter problem by giving the programmer a specific set of data types with unique implementations and letting the programmer construct the necessary data structures with the pre-defined data types provided by the system.

For example, there are many ways sets of objects might be implemented: linked lists of various kinds, bit maps, arrays, trees organized by various techniques, etc. Which of these implementation should be used for a particular program depends on the algorithms used in the program and the characteristics of the data. For the most part, letting the compiler select which implementation to use based only on static declarations has proven viable, but difficult and expensive. The problem is difficult even when attention is focused on a small set of abstractions, as the SETL language effort has shown [46,51]. Barstow's PECOS system [3,4] is an attempt to collect a database of rules and heuristics that allow a programmer's specification of a program to be given an implementation. Elaine Kant [24] extended the system to consider rules and heuristics regarding the efficiency of various implementations.

My approach is to assume that programming environments of the future will be collecting and maintaining much more information about a program than the programmer's static declarations. In particular, the collection and utilization of profile data will become a matter of course. In this chapter, I address the following questions:

- Can profile data reduce the complexity of the representation selection problem to a level that compilers can make such choices effectively?

- What information needs to be collected by the profile mechanism so the selector can make effective choices?

- How much control over the collection of profile data can be put in the hands of the designers and implementors of abstract data types?

- Is there a general algorithm for selecting representations that works for a wide variety of abstract data types? That is, can we limit the overall task of the Implementor to implementation of the ADT, specification of what profile data to collect, and specification of the runtime resources used by the implementation?

Obviously all these questions are interrelated: for example, the selection algorithm will influence the kinds of profile data that will be necessary, and possibly the detail to which the Implementor must go to describe the behavior of an implementation.

We are interested in improvements only from data representation selection, in contrast to algorithm transformations, traditional examples of which include such optimizations as code motion, finite differencing, strength reduction, etc. (Low [31] calls these *representation dependent* optimizations.)

As an example, consider the implementation of:

$$S \leftarrow S \cup \{a, b, c\}$$

which may be more efficiently implemented as

$$\begin{array}{c}\text{put } a \text{ in } S\\ \text{put } b \text{ in } S\\ \text{put } c \text{ in } S\end{array}$$

depending on the representation selected for the set S. Consider also the boolean expression:

$$X \in (S1 \cup S2)$$

which is usually much more efficiently computed as

$$(X \in S1) \vee (X \in S2)$$

For the purposes of this research, we will assume that such optimizations are discovered by a high-level optimization module. This ignores problems introduced by the interaction of this 'higher level' optimization and our representation selection mechanism, but allows us to investigate the use of profile data in the selection process. Assuming the latter is possible, the interaction problem can be investigated later.

Unlike Low, we do not want to limit expressions to be homogeneous in representations. That is, in the expression

$$S1 \leftarrow S1 \cup S2;$$

if $S1$ and $S2$ are two sets, it may be the case that they have two different representations. We are interested in seeing if there are situations in which it is profitable to handle the overhead imposed. In this example, either

1. One of $S1$ or $S2$ must be converted to the same representation as the other, or
2. $S1$ and $S2$ are both converted to a third representation, or
3. a routine to take the union of objects of type $itype(S1)$ and $itype(S2)$ must be generated by the compiler, or
4. there must already exist a procedure that can explicitly handle the union of these two representations.

Larry Rowe [38] explored the problem of generating implementations and we will not pursue it in this study. The TypeSetter prototype requires that the explicit function must exist (corresponding to the fourth option above).

TypeSetter has been designed to allow experimentation with various kinds of profile data in addtion to execution counts. For instance, knowing that the attempts to add an element to a set usually have no effect because the element is already in the set may affect which implementation is chosen for that set. It is easy in TypeSetter to count how many times the add-an-element function was called to add an object already a member of the target set. It would be exceptionally difficult to determine analytically when such a situation holds.

The traditional metric (or objective function) used for selecting one representation over another has been the space-time product, but there are complications in dealing with parameters of the system in which the software is to be run. For example, it may be that representation X is better as long as the amount of memory used does not exceed physical memory limits, otherwise using the more compact representation Y will cause less thrashing of the virtual memory system. The inefficiency due to a denser encoding would be offset by the improved performance of the system as a whole. It is problematic how to specify these kinds of limits, particularly those that depend on system parameters that are difficult to profile or can very dynamically and orthogonally to the actions of the program (e.g. dynamic paging rates). Therefore, this study has not attempted to determine the 'best' way of characterizing program behavior. The system is designed such that each interface function has exactly one evaluation function, which returns a real number that represents the relative behavior of the interface function at a particular call site.

Also, this study limits itself to implementations that do not require automatic changes to User-defined data structures; all abstract data types implemented in the system will be 'exomorphic'–only pointing to objects the user is manipulating. For example, an exomorphic implementation of a list would manipulate only references to the objects in the list. In contrast, an *endomorphic* implementation might include the links of the list as part of the User object, one (set of) link(s) for each list to which the object might belong. An endomorphic strategy might be particularly attractive when, for example, it is known that each object can be on only one list at a time.

The problem of generating or modifying structures to take advantage of such implementations is orthogonal to the problem of using profile data to determine which implementation is best. Once the use of profile data is shown to be viable, then the same techniques can be applied to endomorphic implementations.

Programs that exhibit *phase behavior* are problematic. A user's program could exhibit phase behavior by manipulating data one way early in the execution of the program (say, during the initial construction of an aggregate variable) and utilizing that data quite differently in later stages of the program (say, during access and modification of that aggregate variable). An interesting problem is the detection of the behavior and the optimal points for changing the implementation of the ADT from one representation suitable for the first phase into another representation more suitable for the later phase. A general solution to this problem is beyond the scope of our work. In our model, each static instance of a variable will have exactly one implementation. We can approximate some of the advantages of phase behavior detection by limiting conversion from one representation to another at the assignment of a variable. Consider the following:

```
Set(int) A;
:
:  (section one)
:
−−  end of phase one of program
:
:  (section two)
:
```

It may very well be the case that the behavior of the program in section one demands that the variable A be implemented as a singly-linked list, whereas the behavior of section two would be more efficient if A were implemented as a doubly-linked list. In our current model, A will be assigned only one implementation that will minimize the cost of running the program. If the program looked like the following:

```
Set(int) A;
Set(int) B;
:
:  (section one uses A)
:
−−  end of phase one of program
B = A;
:
: (section two uses B)
:
```

then we can look for the possibility of converting representations when B is assigned, and releasing resources used by A. This would, of course, require live-dead analysis, something that is currently beyond the prototype.

Another complication is introduced by user-defined functions. Consider the following:

```
main() {
  Set(int) A;
  Set(int) B;
  :
  user_fcn(A,B);  --  call site 1
  :
  user_fcn(B,A);  --  call site 2
  :
  }

void user_fcn(Set(int) X, Set(int) Y) {
  :
  }
```

Under our model, A and B must have the same implementation since X can have only one implementation (the same is true for Y). It may very well be the case that the program would perform better if there were two user functions, each with a different type signature. However, creation of multiple copies of the user function are beyond the capabilities of the prototype. At any rate, the problem is again orthogonal to the problem of using profile data, so we apply the general rule stated above for variables to the signature of functions: each statically declared object in the User's program will have exactly one representation associated with it during the running of the program.

Real-time applications can impose severe constraints on the behavior of a program, and are not considered further in this work. The work of Kenny and Lin [27] provides an important parallel to our work in the area of real-time control. In particular, we discuss later how their evaluation function generation techniques could be used in our system to improve the precision and portability of evaluation functions (see page 104).

4.2 Previous work

Typesetter is the first system to use a general technique for collecting ADT-specific profile data, and using that data to choose implementations. Almost all previous systems (with the exception of Low's) attempt to synthesize data representations: Typesetter chooses the implementation of a function to use, and thereby indirectly selects, but does not synthesize, the representations of the program's variables.

Low did the original work on implementation selection using profile data [31,32,33]. His system attempted to provide implementations for three abstractions: sets, lists, and a ternary relation which is unique to the base language SAIL. Each

```
partition the variables and expressions into equivalence classes (eq.c.);
determine which operations are performed on which eq.c.;
for each eq.c do
    $S \leftarrow$ all representations ;
    remove from $S$ all representations which are not feasible;
        -- may not have sufficient information at compile time
        -- may require an operation not implemented in a rep.
    predict time and space requirements for all $s \in S$
    for all $s_1, s_2 \in S$
        if $s_1$ requires both more time and space than $s_2$ then
            remove $s_1$ from $S$
        endfor
    rank remaining representations in $S$ by likelihood of being
        the best representation; -- uses a cost fcn
    use a hill-climbing heuristic to finalize implementations
```

Figure 4.1: Low's algorithm

ADT had several implementations that could be used to implement the User's variables. Each of the functions making up the interface of an ADT was written in assembler, and had associated with it an evaluation function that, given a frequency of execution and an aggregate size (e.g., the number of elements in a set), would return an estimate of the cost of using the interface function. His evaluation functions returned an estimate of the number of machine cycles and bytes required on any one invocation of a function.

The system required four passes over a program, with human interaction as one of the passes. The first step ran the subject program (using default implementations for the abstractions) with software monitoring inserted to collect a profile of the performance of the program in terms of statement counts. The system then prompted the user for information too difficult or impossible to derive analytically (e.g. "What is the average size of set *foo*?"). A penultimate static analysis pass computed the possible contents of variables in terms of other variables. This had the side-effect of partitioning the variables of program into equivalence classes; each equivalence class identified the variables that had to have the same implementation as all other variables in the class.

Low's algorithm for selecting representations (Figure 4.1) uses call sites solely for feasibility testing. Once a set of feasible assignments have been established, then an initial set of implementations are assigned. The final heuristic (Rowe called it incremental search) continually 'perturbs' the assignment of implementations by making changes that seem likely to improve the overall performance of the program and keeping only those changes that do.

Low says that his system could not take into account certain features of set insertion (e.g., elements are always inserted in a specific order) and that he thinks it would be hard to include. TypeSetter allows collecting this kind of information (i.e. 'the data is always added in increasing/decreasing/non-increasing/non-decreasing/extremal order) as well as other information (e.g., sizes of sets, average length of lists, etc.).

Low's system did not allow operators to work on multiple representations: a union operator's two operands had to have the same representation. In our approach, a particular implementation function can be assigned to a call site if the actual parameters at the call site can be assigned the types of the implementation function's formal parameters. It is up to the Implementor(s) which of these mixed-representation functions to implement.

TypeSetter explores several aspects of Low's general technique. Low's ADT interface functions were written in assembler so he could make the evaluation of an invocation of one of those functions as precise as possible. He did not want to tackle the problem of writing evaluation functions for compiler-generated code. Given that precision is lost in any estimation of future performance of a real program, and that the performance of a function depends on more than just its frequency of execution and the size of the aggregate-type object, Typesetter's evaluation functions accept inexactness as inevitable, and assume that programs satisfying the 90-10 rule are skewed enough to make the loss of precision irrelevant to the final decisions.

Also, TypeSetter's interface *and* evaluation functions are all in a higher-level language (C++), and the evaluation functions are in terms of this language's constructs. That is, whereas Low's system required the Implementor to count cycles in instructions in order to write an evaluation function, TypeSetter's evaluation functions are written in terms of the high-level language's constructs. TypeSetter has not solved the problem of providing an evaluation of compiler-generated code, rather it finesses the whole problem by admitting up front that evaluation is inexact. Precision is not possible *a priori* with compiled high-level language code (e.g., a different compiler's optimizer will produce different code), but in exchange we get portable, understandable, easily tuned code, both for the implementations of the interface functions, but also for the evaluation functions.

And finally, Low's system concentrated on finding implementations for a program's variables, using program structure solely to determine the feasibility of the various implementations. TypeSetter turns that around, and concentrates on finding implementations for the interface functions, and lets that specify what the implementation of the variables must be. Section 4.4.1 below explains this method of implementation selection in detail.

The other major work relevant to TypeSetter is Hansen's work on adaptive compilation, which we already discussed in some detail in Chapter 1.1. Hansen's work is the primary justification for this research in compiler utilization of profile data, though Hansen's system was targeted toward 'one-shot' compilation.

Other work relevant to what we are doing has concentrated primarily on how to assign implementations analytically. That is, given some program specification, most work has concentrated on finding means for determining implementations solely on the basis of that specification.

Barstow's PECOS system [3,24] processes a program specification via a knowledge database of program transformation rules. The program is declaratively, as opposed to imperatively, specified. Kant's LIBRA system [24] extended PECOS and attempted to apply the same rule-based, synthesizing approach to performance prediction. The knowledge database was enhanced with rules about estimating potential performance of partially constructed programs. Although LIBRA could allow the use of profile data, it was neither integral nor essential to the approach.

TYPESETTER does not attempt to synthesize programs analytically, nor does it attempt to work with program synthesis at as high a level as does PECOS. Rather, my goal was to explore the possibility of providing implementation selection in the context of modern day compilers. Rather than seek a Copernican revolution and invent a totally new language in which to specify programs, I sought a more evolutionary approach to give existing languages and systems as much capability as possible.

Ramirez [37] used zero-one integer programming to assign implementations. His approach required condensing the behavior of a program into two matrices $s(i, j)$ and $t(i, j)$ where s is the estimated storage space consumed by implementation j when used to implement variable (substructure, he calls it) i, and t is the corresponding time estimate. His claim that the behavior of an implementation of an ADT can be summarized by two numbers $s(i, j)$ and $t(i, j)$ is highly suspect. It ignores, for example, how the behavior of a function or operator may change when provided with arguments of differing implementations. That is, he assumes that if implementation j is assigned to variable i, then $t(i, j)$, the amount of time required by that assignment, is independent of any other assignments. This is almost never the case, particularly when operators can accept operands with differing implementations (e.g., a *union* of a set implemented as a bitmap with a set implemented as a linked list). TYPESETTER moves the focus of evaluation functions from the variable to the implementation of the interface functions. This allows the interacting costs of assignments to be taken into account at the expense of losing the ability to use zero-one integer programming to achieve an optimal solution.

Work within the SETL project [9,43] derives representations from declarations in the language and from analysis; e.g., frequencies are estimated by an analysis of the program text. I know of no work using profile data in the synthesis of SETL programs.

The SETL optimizing compiler attempts to determine a good implementation of for the set and mapping abstractions in the language (there is only one representation for tuples). The default representation for sets and maps uses hash tables. If the analysis can determine *bases* for the elements of the sets, or if the programmer declares elements to belong to specific bases, then other more efficient implementations are possible for subsets of the bases. A subset can be represented as a bit in the structures for the elements of the bases (if the bit is one, then the element belongs to that subset, if zero, then not). If all elements of a base set are assigned unique integers, then a subset can be implemented as a bit-map. Or a subset might be represented with a separate hash table of pointers into the base set.

Straub's *Taliere* system improves on the optimization phase of the SETL compiler by considering estimates of performance, including symbolic analysis of execution frequencies. However, since he does not utilize profile data, the User must answer questions[1] of the form *What is the average size of s*t in line 215?*; or even *What is the expected number of iterations in an average execution of the loop starting at line 1235?*. Even worse examples of the kinds of dialogue the system forces on the User are questions about probabilities: *What is the probability of the CASE statement of line 1113 taking the alternative of line 1126?* It seems extremely doubtful to me that a User would know this information with any precision or confidence without profile data.

Weiss [51] worked on finding types of recursive SETL variables, and presented methods for implementing such structures. However, he does not worry about selection of 'best' implementations by numeric criteria.

Sherman's dissertation [44] presents a very comprehensive approach to the problem through language design. The primary contribution of his programming language Paragon is the idea that implementations are subclasses, or refinements, in the ADT hierarchy (with multiple inheritance). That is to say, an implementation is just a refinement of an abstract data type and is specified using the same notation as that used to specify the abstraction. Paragon is an ambitious system that attempts to solve many problems at once, including selection of a refinement of an ADT based solely on the program text. Presumably, profile data could be used, but he does not discuss this in any depth. In the Paragon model, the User (our terminology) is responsible for writing the complete evaluation function (Sherman calls it the *policy* procedure) that selects the implementations of the variables of the program. This puts the onus of selection on the wrong member of our programming duo. We have attempted to design a system that puts the onus of implementation evaluation on the Implementor, and selection of implementations for functions and variables on the system, not on the User.

Rowe's system [38] approached the problem from the direction of selecting an implementation from a description of the desired data relations and functionality. His modeling domain language is implementation independent and is used to search for implementations that satisfy the described relations and operations. In those cases where there does not exist an implementation satisfying the description, Rowe

[1]The questions are taken from his dissertation.

investigated ways of generating an implementation. However, he did not investigate the use of profile data in his work.

4.3 TypeSetter: The System

In the discussion of the TypeSetter system, it is useful to distinguish between different modes of programming which I identify by reference to two different programmer groups: the Users and the Implementors. Since Users need not be as facile with the theory behind TypeSetter as Implementors need to be, the distinction between them is not one simply of convenience of notation. Users write programs that utilize the abstractions provided by the system; the system selects from among the implementations installed by the Implementors.

Examples in the following sections are based on the existing prototype, described in greater detail starting in Section 4.4. Language enhancements required to support TypeSetter are not extensive and the notations should be relatively transparent to anyone who has used an object-oriented programming language like C++.

4.3.1 Formalities

An abstract data type (ADT) is a set of *function signatures* indexed by function names. For each ADT, there is a set of representation types; we'll write $\mathcal{A} \mapsto \mathcal{R}$ to mean that ADT $\mathcal{A}$ can be represented by representation $\mathcal{R}$; $impl(\mathcal{A})$ is the set of possible representations of $\mathcal{A}$. A function signature has the form $T_0, T_1, \ldots, T_n$, for $n \geq 0$ and types T_i. By convention, T_0 is the type returned by the function. For all signatures in ADT $\mathcal{A}$, the T_i are themselves ADTs.

For each abstract function in an ADT, there is a set of *implementation functions*. Like their abstract function counterparts, implementation functions consist of a name and a signature. But where the abstract functions' parameter types T_i are ADTs, the implementation functions' parameter types T'_i are representation types. Furthermore, $T_i \mapsto T'_i$ for all i.

Our task is to assign a representation type to each variable in a user's program, and hence an implementation function to each call site in the program.

Define a program to be a set of variables $\mathcal{V}$ and function call sites $\mathcal{C}$. Each variable $v \in \mathcal{V}$ has been declared to have one of the ADTs in $\mathcal{T}$, *atype(v)*. Each function call site $c \in \mathcal{C}$ consists of the name of an abstract function, *absfcn(c)*, and a list of actual argument variables *actuals(c)*. When an implementation function is assigned as the implementation of the abstract function at a call site, then the implementation type of v_j, the j^{th} actual, is assigned to be T'_j, the j^{th} type in the signature of the implementation function; i.e., $itype(v_j) = T'_j$.

4.3.2 The ideal system

The prototype system is a subset of a larger vision. I have argued previously that profile data should be collected all the time, and that the overhead for doing so is quite minimal. Then the programming system would allow the user to develop a debugged and efficient program in the following steps.

During initial implementation and debugging, the User rarely needs to decide the implementation of many, if not most, of the abstractions used in the program. The TypeSetter system will utilize the simple program execution counts described in chapter 2 to pick a reasonable implementation for the abstractions.

As the User's system is implemented, and its structure is tested with more complex, more complete, and perhaps larger sets of input data, the actual implementation of the abstractions becomes more important (if for no other reason than that debugging a slow program can be particularly aggravating). By this time, the User will have gained enough experience with the program that he can conjecture which implementations of the abstractions may be reasonably efficient for the program. This conjecturing is important only to the extent that it allows the User to determine what additional information might be helpful to supply about the abstractions: TypeSetter will determine which implementations are actually best. This additional information is supplied as part of the declarations of the variables, and is discussed in more detail in section 4.3.5.

It is also true that simple execution counts are insufficient for selecting an implementation. Alternative implementations of abstractions are created by programmers to take advantage of the interaction of properties of the abstractions with specific properties of sets of input data. Therefore, to determine that a bit-mapped implementation of a set is preferred over a linked-list implementation requires knowing not only how the program makes use of the data (e.g., number of insertions *vs.* number of deletions; the mix of element access and destructive operations; etc.) but also requires some information about the input data itself (e.g., is it read in increasing/decreasing order; is it 'sparse'; is it locally dense; etc.). This information can only be gathered directly and intentionally, and cannot be inferred from execution counts except at great expense, if at all.

At this point in the development process, the abstractions in the program are assigned *profiling implementations*, and a couple of runs of the instrumented program (over whatever data it can handle at this stage of development) will provide further data upon which TypeSetter can assign more efficient implementations. Once implementations are determined based on this data, development and debugging of the program can proceed using this more appropriate implementation of its abstractions.

The TypeSetter prototype described here has concentrated on implementing only the intermediate step: using profiling implementations of abstractions to collect abstraction-specific profile data that TypeSetter can use to select from among a set of implementations. Lacking any profile information, the prototype always links in the profiling implementations for all abstractions; running the program then generates profile data for selecting more efficient implementations on a future run of the system.

4.3.3 ADTs

Among the ADTs available to programmers in TypeSetter there are the usual built-in data types (e.g. integer, real, structures, arrays, pointers, etc.), each of which has a fixed implementation in User's programs, and a library of more complex abstract data types (ADTs) with implementations of functions that make up their interface tailored to specific representations. Users, however, make no reference to specific implementations of the functions or variables; they simply make use of the publicly declared interface (or protocol) of the ADT. The compiler will then choose implementations for the variables and functions to minimize a cost function based on data collected by a profiling version of the ADT. TypeSetter supplies three ADTs: sets, lists, and maps. (These correspond closely to the SETL data types of finite sets, tuples, and maps [43].) Their definitions below are in TypeSetter's C++ dialect; in particular the first argument to a function is understood to be a pointer to the object by which the function is invoked.

Sets The Set abstract data type is generic in the type of the contained objects, which is denoted as `Any`. Figure **??** contains the definition of the Set ADT, and Figure 4.2 lists some possible implementations of exomorphic sets. Those with asterisks are currently implemented in the prototype. There are many possible representations of Sets, a few of which are briefly described in Figure 4.2. (The reader may wish to compare this list with the implementations provided by the SETL compiler; see pg. 57.)

Lists The List abstraction is generic in the type of the contained objects. Figure 4.3 lists the functions comprising the interface to the ADT. There are many possible representations of Lists, a few of which are described briefly in Figure 4.4.

Maps Maps, or finite functions, are generic in the type of the domain element and the type of the range element. Figure 4.5 lists the functions forming the interface to the ADT, and Figure 4.6 lists possible implementations. Figure 4.6 lists a few of the many possible implementations of Maps.

∗**Linked list** .

Doubly-linked list .

∗**Sorted linked list:** A linked list on which the elements are maintained in sorted order.

∗**Bit vector:** Elements must be, or map to, integers. Requires knowing max and min integers. Three variations: nofElements kept as part of the set, fast array lookup element count for bytes, ditto for words.

Hash table: Useful when the key is not an integer, but an arbitrary collection of bits. Information about the range of the hash function and the density of the resulting hash values would help select good parameters for the hash table. For sets of arbitrary objects, the programmer must supply a hash function.

Sorted array: Keeps a sorted list of the actual elements of a set. Requires knowing max and min elements; knowing the maximum size of a set, and the average size of sets may help select better parameters for the implementation.

Sorted variable length array: Ditto. Requires extra overhead for the dope vector.

Linked array: Requires knowing max and min elements, as well as the fact that the elements tend to cluster. Optimizes space at the expense of time. To be used in environments where reallocating sets due to growth or memory compaction may be more expensive than just chasing pointers.

Figure 4.2: Possible implementations of sets (∗ in prototype)

`List()`: The constructor of List.

`boolean empty()`: Returns `true` if the list is empty.

`void makeEmpty()`: The list is emptied.

`boolean in(Any elt)`: Returns true if `elt` is in the list.

`int cardinality()`: Returns the size of the list.

`void rest(List L)`: Removes the first element of the list.

`Any first(List L)`: Returns the first element on the list.

`append(Any e)`: The element e is appended to the list.

`prepend(Any e)`: The element is pushed onto the front of the list.

`delete(Any e)`: All instances of the element e are removed from the list.

`sort(CmpFcn f(Any,Any))`: The list is sorted in place using the comparison function f.

`iterInit(Iterator i)`: Initialize an iterator over the list.

`iterate(Iterator i, Any &elt)`: Assign `elt` the next element of the list in the iteration and return *true*, else return *false*.

`iterDone(Iterator i)`: Return `true` if an invocation of `iterate` would return `false`.

`iterCleanup(Iterator i)`: Return resources allocated to the iterator.

`iterCopy(Iterator i, Iterator &j)`: The iterator `i` is copied into a new iterator `j`.

Figure 4.3: Specification of List ADT

∗**Linked list.**

Doubly-linked list.

Fixed-length array: Each list has an array of maximum size allocated for it.

Linked array: The list is kept in a list of arrays, each sub-array allocated/deallocated as the list is manipulated. Requires knowing max and min elements, as well as the fact that the elements tend to cluster. Optimizes space at the expense of time. To be used in environments where reallocating lists due to growth or memory compaction may be more expensive than just chasing pointers.

Figure 4.4: Possible implementations of lists (∗ in prototype)

`Map()`: The constructor of Map.

`boolean empty()`: Returns `true` if the map is empty.

`void makeEmpty()`: The map is emptied.

`boolean inDomain(Any elt)`: Returns true if e is in the domain of the map.

`boolean inRange(Any elt)`: Returns true if e is in the range of the map.

`int cardinality()`: Returns the size of the map the number of elements defined in the range).

`define(Any d, Any r)`: The element d is added to the domain of the map so that it returns the element r.

`delete(Any e)`: All instances of the element e in the domain are removed from the map.

`sort(CmpFcn f(Any,Any))`: The map is sorted in place using the comparison function f.

`iterInit(Iterator i)`: Initialize an iterator over the map.

`iterate(Iterator i, Any &d, Any &r)`: Assign d the next element in the domain of the map, r the corresponding element in the range, and return *true*; else return*false*.

`iterDone(Iterator i)`: Return `true` if an invocation of `iterate` would return `false`.

`iterCleanup(Iterator i)`: Return resources allocated to the iterator.

`iterCopy(Iterator i, Iterator &j)`: The iterator `i` is copied into a new iterator `j`.

Figure 4.5: Specification of Map ADT

$*$**Linked map:** The map may be singly or doubly linked, and consists of pairs of elements as related by calls to *define*.

Fixed-length array: Each map has an array of maximum size allocated for it, if a maximum size is know. The array is two dimensional, one each for the domain and range.

Linked array: The map is kept in a map of arrays, each sub-array allocated/deallocated as the map is manipulated.

Hash table: Hashed by domain elements for faster lookup.

Binary tree: So seeks on domain elements are $O(\log n)$, for n the number of elements in the domain.

Figure 4.6: Possible implementations of maps ($*$ in prototype)

4.3.4 Iterators

In a complete language definition, much of the functional interface for iteration would be hidden from the Users with some syntactic sugar. I envision something close to the Alphard paradigm for iterators, and the User would write something like the following to have the compiler invoke the appropriate iteration functions.

```
Set(SomeType) S;
 ...
for i in S do
    ... -- use of element i
    endfor;
```

The above code would be translated into something like the following using the TypeSetter iterator paradigm:

```
Set(SomeType) S;
...
Iterator S_iter;
S.iterInit(S_iter);
while S.iterate(S_iter,i) do
    ...
    endwhile;
S.iterCleanup(S_iter);
```

The iterators are associated with the object being iterated. Their exact form is never available at the User level and, therefore, the ADT implementation is free to create the iterator object necessary to successfully traverse the aggregate type. In other words, every ADT exports the `Iterator` type, but not the internals of the `Iterator` type.

4.3.5 Optional parameters

Users should be able to write simple declarations of their program variables and have the system select an appropriate implementation of those variables based on that declaration and on knowledge of the behavior of the program containing those declarations.

Set(*Bar*) *foo*;

declares that the variable *foo* contains a set of objects of type *Bar*. TypeSetter recognizes several objects in this declaration. The first is, of course, the use of *foo*, which is the name of the variable being declared, and the second is the name of the ADT. The remaining parameters supply information to the ADT itself, and are of two types: those *required* for even a minimal implementation of the ADT, and *optional* parameters which supply further information that may enable (or exclude) implementations for consideration. Required parameters are positional, and optional parameters are named. In our example, *Bar* is the name of a User type and is required by all implementations of Sets. A declaration of a variable of type Map has two required parameters, the type of the *key*, and the type of the *data*. For example,

Map(*T1,T2*) *vmap*;

declares *vmap* to be a mapping from key objects of type *T1* to data objects of type *T2*.

Use of the variables follows standard object-oriented form. For example,

iset.add(3);

adds the element '3' to the set named *iset*.

Optional parameters (or just *optionals*) are not required for an implementation to be assigned to a variable: there is always at least one implementation of an ADT that can be assigned to any variable of that type. Optionals supply information that allow the system to consider other, possibly more efficient, implementations for a variable. For instance, consider:

Set(**int**) *iset*;

As declared, the variable *iset* could be implemented with one of a variety of bit-map implementations, but with only those implementations that can handle bitmaps of unknown and possibly varying size, and perhaps even negative values as elements. This implies a relatively complex implementation of bitmaps. If the User were aware that the only correct integer values for this set were positive, that information could be provided with the optional parameter *lowerb*:

Set(**int**,*lowerb*=0) *iset*;

The optional parameters are named parameters. The additional information they provide could enable the consideration of other bit-mapped implementations that need it. Obviously, the more information provided about the user's objects, the more implementations that can be considered.

Consider the following declaration:

Set(*Utype*) *uset*;

The User has declared a set of objects of type *Utype*, a user-declared type. In this situation, bit-mapped implementations are not at all feasible since the compiler has no way of mapping objects of type *Utype* onto integers. If such a mapping is possible, the User may declare the mapping function and its inverse:

Set(*Utype,objToInt=f,intToObj=g*) *uset*;

The system is now free to consider bit-mapped implementations as before.

In an ideal system, the User might be given hints as to which optionals might provide performance improvements. How such hints might be automatically generated is a topic for further research. In the TypeSetter system, the Implementor is responsible for providing the documentation for the optionals supported by an implementation. That documentation will include a general description of the possible effects on TypeSetter's ultimate choices of implementations. That is, we depend on documentation to help Users decide which optionals might be beneficial for their programs.

The actual number of optionals required for the prototype has been few. Table 4.7 describes optionals envisioned as useful. Asterisks mark the optionals actually implemented in the prototype. The optionals in the table fall into one of two categories: those that provide information that would difficult to derive from the program even with profile data, and those that are a convenience for the current implementation but could be eliminated with appropriate programming by the Implementor. For instance, the *upperb* optional for set implementations cannot in general be derived from program source and profile data.

The declaration optional *addedDecreasing* could be detected at run time and encoded in the profile data. If different input data were fed to an implementation that attempted a more efficient representation by assuming the elements were added in order, more than likely the User's program would run slower, but would not fail. Of course, it is possible to design an implementation whose correct operation depended on the assumption, in which case detection by a profiling implementation would not be sufficient: a contract with the user in the form of a declaration would be required. The 'order' of the objects in this example is an internal ordering; if the implementation depends on a User-defined ordering, then another optional declaring the order function would need to be defined by the Implementor and declared by the User. Specifically, the implementation *Set_slistord* takes advantage of the fact that sometimes a program creates all the objects that are in a set, and adds them in the order they are created. This often results in the heap allocator allocating the objects such that their memory addresses correlate with their time of creation. *Set_slistord* attempts to cut down on lookup time of elements in a set by keeping the elements on a list in increasing order of the addresses of the objects.

4.3.6 Alternative implementations

The User sees a system complete with a set of possible implementations of abstract data types. These implementations were provided with the system and are, conceptually at least, part of the system. It must be possible for Implementors to specify under what conditions their implementations can be selected, what profile data needs to be collected, and how that data is to be evaluated.

One of the first responsibilities of the first Implementor of an abstraction is

	Optional	value	description
			Sets
*	IntToObj	function	the function that converts integers into objects of the correct type for this set
*	ObjToInt	function	converts objects into integers
*	lowerb	integer	the lower-bound value of the elements of a set of integers
*	upperb	integer	the upper-bound value of the elements of a set of integers
	nofElts	integer	the number of base elements of the set; must equal upperb−lowerb+1, if they are specified
*	ObjsAreInts	(none)	declares that the base objects of this set are in fact integral, and the compiler will perform the correct coercions; this is a convenience optional for the prototype
	compareFcn	function	accepts pointers to two objects and returns −1, 0, or 1 depending on whether the first is less than, equal to, or greater than the second.
	addedDecreasing	(none)	elements are added in decreasing order
	addedIncreasing	(none)	elements are added in increasing order
			Lists
	maxLength	integer	User contracts that list will never be longer than this
			Maps
	IntToObj	function	the function that converts integers into objects of the correct type for this set
	ObjToInt	function	converts objects into integers
	lowerb	integer	the lower-bound value of the elements of a set of integers
	upperb	integer	the upper-bound value of the elements of a set of integers
	hashFcn	function	returns a 32-bit integer that can be used in a hash table implementation of a map
	compareFcn	function	accepts pointers to two objects and returns −1, 0, or 1 depending on whether the first is less than, equal to, or greater than the second.

Figure 4.7: TypeSetter optionals.

```
function Set_P::add(Set this, any e)
{
    profiler
        pcnt, psizeSum,pwasIn;
    Link lp;
    pcnt++;
    psizeSum += length;
    lp = this→first;
    while (lp != nil && e != lp→data) {
        lp = lp→next;
        }
    if (lp == nil) {
        // e not in the set
        Link newp = new Link;
        newp→data = e;
        newp→next = first;
        first = newp;
        }
    else {
        pwasIn++;
        }
}
```

Figure 4.8: Profiling implementation of *add*

to define the functionality of the abstract data type and provide its first implementation. This first implementation must also be the profiling implementation for this ADT, and it must be the most general: it is a requirement in TYPESETTER that no ADT is defined with functions in the interface that cannot be implemented in the profiling implementation; otherwise, there would be no way to collect profile data about that function.

Figure 4.8 shows the code for the profiling implementation of the add-an-element function in the interface for sets. This implementation, called *Set_P*, uses a very general structure (in this case a linked list) to ensure that any function in the interface can somehow be implemented and profiled. (The converse is not true: an alternative implementation does not have to implement every function in the interface. Any program using that function could not have that implementation assigned to the involved variables, however.)

Profile variables (declared as **profiler**s in Figure 4.8) are allocated per call site in the User's program. That is, if the User's program calls *add* from three distinct

```
function Set_bm::add(Set_bm this, any e)
{
    int i = (*this→objToInt)(e);
    int w = (i / (sizeof(integer)*sizeof(byte)));
    int b = (i mod (sizeof(integer)*sizeof(byte)));
    this→setbits[w] |= (1 << b);
}
```

Figure 4.9: An alternative implementation of *add*

sites, then a total of three instances each of *pcnt*, *psizeSum*, and *pwasIn* are allocated. On each call of the *add* function, the invocation counter *pcnt* is incremented, and the *psizeSum* profile variable is incremented by the current length of the set. From this information, evaluation functions can compute the average size of the set per call per call site. Finally, it may be useful to an implementation to know how many times *add* was invoked to add an element that was already a member: the profiling variable *pwasIn* allows us to compute that statistic. As other implementations are added to the collection of implementations for sets, more information may need to be collected by the profiling implementation. The Implementor of an implementation that requires new profile data will modify the profiling implementation to collect it.

4.3.7 Feasibility functions

Continuing with the example of the add-an-element function, Figure 4.9 shows its implementation when sets are implemented as a bit map. Before a User's variable can be assigned *Set_bm* as its implementation, TYPESETTER must first check that this is a feasible assignment. Therefore, the Implementor of *Set_bm* must provide a *feasibility function* that TYPESETTER can call to check feasibility. The function returns either **true** or **false**; the feasibility function for a bit-map implementation having 32 elements or less is shown in Figure 4.10.

4.3.8 Evaluation functions

Traditional profiling techniques cannot capture the wealth of detail required for intelligent selection of implementations. For instance, from knowing the number of times allocation and deallocation routines are executed, it is extremely difficult to deduce the average size of sets, say, at any particular call site in a program. With our profiling schema, it is particularly easy. Furthermore, rather complex information can be acquired such as "Is this a sparse set?", "Does this list ever have items deleted from it?", "Are the elements of this set entered in any particular order that yields advantage to any implementation?", etc.

```
FEASIBILITY
{
  if (!u→IntToObj→def  && !u→ObjToInt→def     &&
      u→ObjsAreInts→def  &&
      u→upperb→def     && u→lowerb→def        &&
      u→upperb→ivalid  && u→upperb→ival > 0   &&
      u→lowerb→ivalid  && u→lowerb→ival ≥ 0   &&
      u→lowerb→ival < u→upperb→ival   &&
      (u→upperb→ival − u→lowerb→ival) < 32
      ) return true;
  else return false;
}
```

Figure 4.10: Feasibility function for implementation *Set_bm*

```
Eval Set_bm::add(CallSite c)
{
    return c.pcnt *
      (idividePwr2_op + modPwr2_op + orAssign_op + array_op + shift_op);
    }
```

Figure 4.11: Evaluation function for *add*

TYPESETTER determines which implementation of an ADT is best based on the estimates returned by the evaluation functions supplied with each implementation. For each function in the interface of an ADT, the Implementor must supply an **Eval** function. For example, the evaluation function for *Set_bm::add* is in Figure 4.11. When called with a call site as a parameter, these functions return an estimate of the runtime resources required by this implementation.

The variable *pcnt*, declared in the profiling implementation as a profile variable, is used here to estimate how much time this implementation of *add* would take at a particular call site. The other profiler variables are not used for evaluating this implementation of *add*. The variables in Figure 4.11 ending in '*_op*' are constants that estimate the relative execution times of each of the indicated high-level language operations. These times will in general be approximate, if only because the execution time of any construct is context-dependent. However, the purpose of these constants is merely to provide an estimate of the time required by this function *relative* to other functions implementing the same functionality for different representations. It is my assertion that such an evaluation technique is 'close enough' to allow reasonably 'correct' assignment of implementations. It pays for itself by allowing the library of implementations to be ported to a new system by simply changing the values of the indicated constants, if necessary.

Therefore, all of the examples here, and all of the implementations in the prototype, estimate the amount of time used in a direct fashion (assuming infinite physical memory, etc.), and do not attempt to handle the complexities of more precise estimates of performance.

Not all functions are as straightforward as the *add* example above. Consider the function for computing the union of two sets. The TypeSetter code for the Set_slist implementation of the two-operand union function (i.e. the union of the two sets is assigned to one of the sets) is shown in Figure 4.12. This implementation of *union* determines for each element in the set *sB* if it is in the current set (the *this* set, in C++ terminology). If not, it is added to the current set. The evaluation function for this function must therefore have access to the evaluation function for the *in* function, passing to it the information it needs to create an estimate of its behavior at the call site in the *union* function. The profiling implementation may not have invoked the *in* function (and in the prototype's profiling implementation for sets, it doesn't) so there is in general no profile data specific to call sites contained in the alternative (i.e., non-profiling) implementations of an abstraction's interface functions. Therefore, in this case, the *union* evaluation function must provide estimates for the profiling data, which it passes as parameters to the evaluation function for *in*.

4.4 TypeSetter: The Implementation

I use the name TypeSetter to refer to the whole system and to the language that results from the enhancements made to C++. The actual implementation of TypeSetter consists of several parts, some of which are written in TypeSetter, the language. The extensions to C++ have been discussed already, and are straightforward. There is the addition of evaluation and feasibility functions to the declaration of class member functions (see pg. 71ff), the use of profiling variables (see pg. 70ff), and the User's ability to optionally declare extra information in a variable declaration (see pg. 67ff).

All of these language features currently take the form of macro invocations. A macro processor first scans all source files and extracts the relevant information via the macro invocations. This information is then fed to the analysis program which makes the actual implementation decisions. The macros are written in m5 [40], a powerful macro language designed for the manipulation of name-scoped text, such as is found in programming language text. In addition to the alternative implementations, the program that makes the actual implementation decisions is also written in TypeSetter; I call this program Therblig after Frank Gilbreth's qualitative unit of work-motion [14]. Figure 4.13 shows the steps necessary to compile

```
void
FUNCTION(union1)(Set_slist sB)
{
  Set_slist_Link *lp = sB.first;
  while (lp != NULL) {
      if (!Set__in(lp->data)) prepend(lp->data);
      lp = lp->next;
      }
}
EVALSUB(Pcnt,PszA,PszB,Povrlp)
{
  @@ executed Pcnt times;
  @@ there were Povrlp elements of s2 already in s1;
  @@ each time loop exec'd avgBsz times;
  @@ and prepend exec'd avgNotIn times;
  if (Pcnt == 0) return 0;
  double avgBsz = PszB / Pcnt;
  double avgIn = Povrlp / Pcnt;
  double avgNotIn = avgBsz - avgIn;
  return Pcnt *
    (assign_op // startup
     + (avgBsz * (cmpZero_op + deref_op + assign_op + not_op))
     + EVALSUBfor(in)(avgBsz, PszA, avgNotIn)
     + (avgNotIn * EVALSUBfor(prepend)()));
}
EVALUATE
{
  return EVALSUB(p_cnt,p_szA,p_szB,p_ovrlp);
}
END_FUNCTION(union1)
```

Figure 4.12: Set union using linked lists

a program written in TypeSetter into C++. The first invocation of *m5* collects information about the declaration of variables and the call sites of abstract functions, and emits a description of the User's program. Therblig analyzes this description, along with profile data about the program (if it exists), and emits an assignment of implementations for all variables and call sites. These assignments, along with a library of ADT implementation sources and the original source of the User program, are processed by another run of *m5* to transform the User program into C++.

Therblig contains all of the functions necessary to evaluate the use of ADT implementations in the User's program, including the evaluation and feasibility functions provided by the Implementors for their contributions to the library of implementations. As can be seen in Figure 4.14, these are compiled as part of Therblig after processing of the implementations declared for each of the ADTs that are part of the system.

These steps are discussed in more detail in the following sections. First, I will discuss the algorithm used in Therblig to assign implementations to variables. Next, I will present the mechanism used in TypeSetter for implementing code sharing among the implementations assigned to the User's variables.

4.4.1 The Implementation Selection Algorithm

In the concluding chapters of his dissertation, Low [31] observed that his hill-climbing heuristic seemed to display the property that implementation decisions were made early and were rarely re-made. Straub [46] notes that even though his representation selection algorithm was run many times with widely varying expected values for the program's variables, "the choice of data structures made by the system tended to be independent of the responses to the queries made to the user." He concluded that this indicates that the selection depends much more on the operations performed than on the expected values of the program variables. (It is curious that he does not even consider collecting profile data as a better source of information than interactive querying of the user. Nor does he consider profile data as a source for finding those important operations.)

Apparently, the representation selection process is being made much more complicated than it really is: good selections depend on a small part of the information that has been utilized in previous research. My hypothesis is that this is due to the 90-10 nature of most programs. That is, if more weight is given to the more frequently executed sections of code (as they are in Low's technique) then the implementation costs of these sections will dominate the overall execution costs of the program. It also means that implementation decisions that are good for these 'hot spots' will be good for the whole program, and, conversely, implementation decisions for very infrequently executed code have little effect on the performance of the program.

This observation, bolstered by the observations of previous researchers, plus

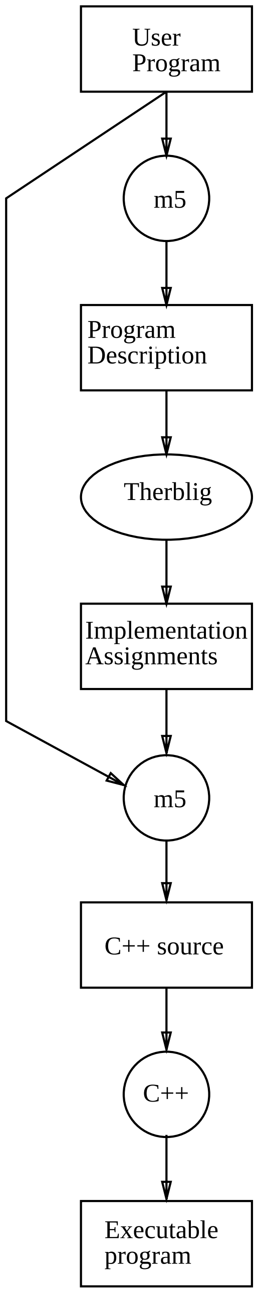

Figure 4.13: Steps to process a User program

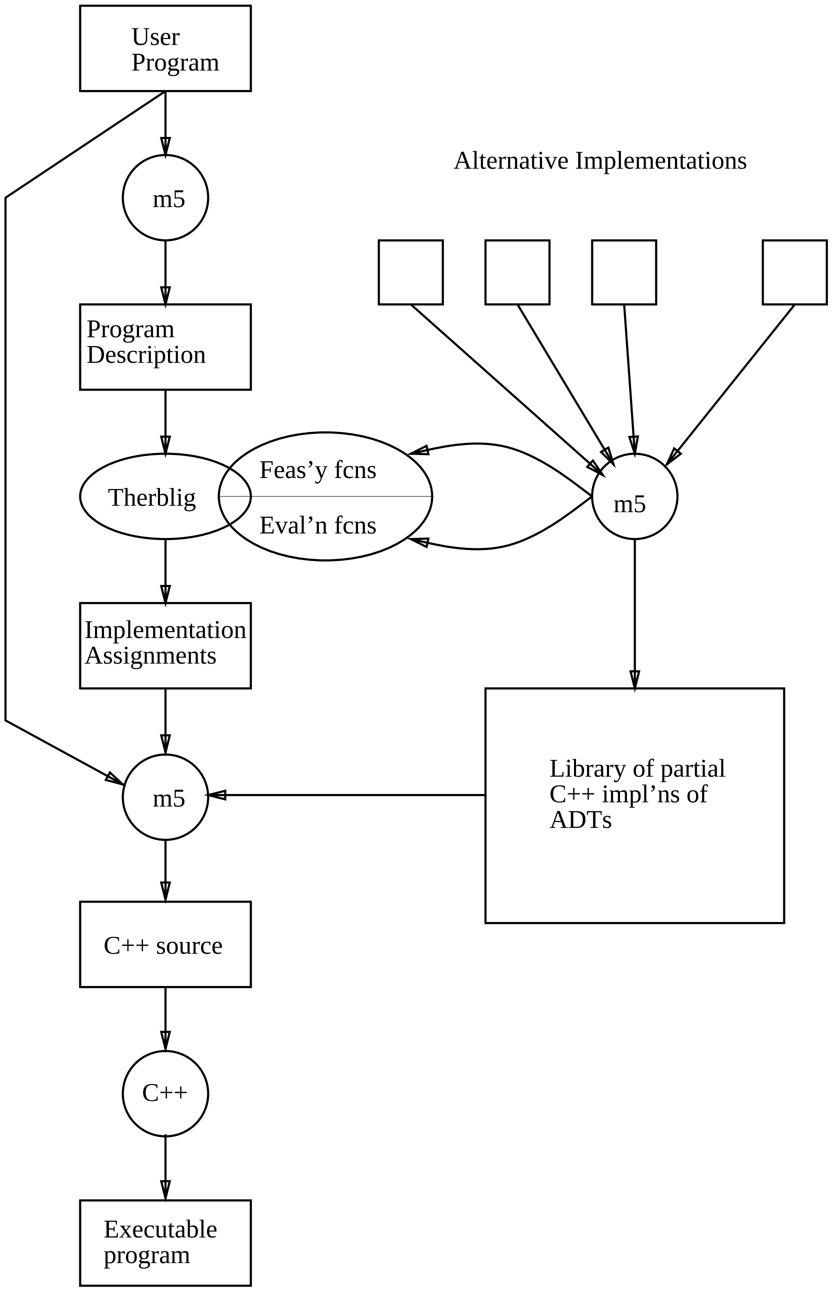

Figure 4.14: Steps to process implementations and build Therblig

the success of the Greedy Sewing algorithm for code reorganization (covered in Chapter 3), plus Hansen's success with focused optimization [17], plus the time-honored success of hand-tailored optimizations based on profile data all suggest that in general a greedy algorithm using profile data will work for assigning implementations to abstract data types. This results in a two stage process: stage one decides which of the existing implementations are feasible, and stage two chooses from among the feasible implementations the one that minimizes the cost of running the program. Furthermore, the algorithm does *not* require analysis of the program flow graph, as does Low's, Rowe's, and Straub's techniques: it reduces to a problem of matching function implementations with function call sites.

Three questions must be answered to find an implementation for a variable.

1. Does enough information exist to take advantage of a representation? That is, does the information exist that would allow variable v to be implemented with implementation R? For example, if a list is implemented as a fixed size array, we have to know the maximum size of the list. If the maximum size of a list is unknown, or there is no maximum size, then the fixed-size array implementation of a list is not feasible.

2. Does enough of a representation exist to implement a variable? That is, if variable v is given implementation R, for each call site c which has v in its actual parameter list, does there exist an implementation f of F_c consistent with that and all previous assignments of implementations?

3. Which implementation of variable v will provide the best performance for the program?

The first two questions come under the realm of *feasibility*: is it possible to select an implementation for the program? The last question seeks to find an implementation that minimizes the cost of executing the program, using the Implementor-provided evaluation functions as objective functions for that minimization.

The assignment algorithm which I have implemented is in Figures 4.15 through 4.19. The pseudo-code in is meant to be descriptive rather than a rigorous program in a well-defined language. A description of some functions that are not otherwise defined can be found in Figure 4.19. The code is not specific as to how the assignment of variables and functions are recorded. We assume a global data structure *assignment* contains a consistent assignment when *chooseImplementation* returns, or it is empty if no assignments were possible.

The basic procedure is summarized as: sort all call sites in decreasing order by some preliminary metric and assign implementations to variables based on the cheapest implementation of the functions called at each site. While I have emphasized the evaluation functions for alternative implementations, the profiling implementation for each ADT also has evaluation functions that are used to estimate

```
proc $chooseImplementation$(Set($callSite$) $\mathcal{C}$)
    List($callSite$) $\mathcal{S}$;
    $\mathcal{S} = sortByImportance(\mathcal{C})$;
    if not $assignable(\mathcal{S})$ then $error$ fi
    endproc
```

Figure 4.15: Main routine for choosing representations

the potential impact of a call site. Consider a program that creates a set of n objects, sorts the set into a list, and then accesses the objects in the list. The call site (assume there is only one) that adds elements to the set is executed n times. The list is accessed kn times, for some integral k. But the sort routine is called exactly once. Nevertheless, that sort routine has complexity $O(n \log n)$ to $O(n^2)$, meaning it has the potential of swamping the significance of the add-an-element function. The evaluation functions of the profiling implementation return values that reflect this potential impact of each function.

The recursive selection algorithm is encoded primarily in the function *assignable* (Figure 4.16) which is passed a list of call sites; the first call site c on the list will be assigned an implementation, if possible. That is, at call site c where the abstract function F_c is invoked, *assignable* will pick the (next) cheapest implementation of F_c, which thereby determines the types of the variables that are passed as parameters to F_c. The function *findCompatible* (Figure 4.18) finds all implementations of F_c that are compatible with this call site. This set of compatible implementations is then sorted in increasing order of the estimated costs provided by the implementations' evaluation functions.

For each function in the sorted sequence of implementations $\mathcal{S}$, if the actual parameters to F can be assigned the implementation types required by the implementation function f, then an assignment is attempted on the next call site on list $\mathcal{L}$. Otherwise, we back out of any implementation assignments made in this invocation of *assignable*, and another implementation function f on $\mathcal{S}$ is tried as the implementation function for this call site. If every implementation function has been tried and no consistent assignment of implementations to variables and call sites has been made, then *assignable* returns false. The function *parmsImplementable* (Figure 4.17) determines if the implementation function f can be used to implement the abstract function F by checking that the actual parameters to F can be assigned the implementation types required by the formal parameters of f.

```
function assignable(List(callSite) $\mathcal{L}$) returns boolean:
    if isEmpty($\mathcal{L}$) then return true;
    $F \leftarrow$ absFcn(first($\mathcal{L}$));
    $\mathcal{S} \leftarrow$ sortByCost( findCompatible( first($\mathcal{L}$) ));
    foreach f on $\mathcal{S}$ do
        if not parmsImplementable(f,F) or
            not assignable(rest($\mathcal{L}$)) then --backtrack
                undoImplementations;
        else return true;
        fi
        endfor
    return false;
    endfcn;
```

Figure 4.16: Routine *assignable* for choosing the implementation of an abstract function

```
function parmsImplementable(impFunc f, absFunc F) returns boolean:
    foreach $v \in$ signature(F), each $t \in$ signature(f) do --parallel
        if not alreadyImpltd(v) then
            if implementable(v,t) then
                implement(v,t);
            else --conflict; backtrack
                return false;
            fi
        fi
        endfor
    return true;
    endfcn

function implementable(variable v, implType t) returns boolean:
    if not atype(v) $\mapsto$ t then return false;
    if not t.feasible(v) then return false;
    return true;
    endfcn
```

Figure 4.17: Mapping parameters onto implemented functions' signature

```
function $findCompatible(callSite\ c)$ returns Set($implType$):
    $I \leftarrow implementations(absFcn(c))$;
    $Ic \leftarrow emptySet$;
    foreach $f \in I$ do
        if $isCompatible(f,c)$ then
            $Ic \leftarrow Ic \cup \{f\}$;
        enddo
    return $Ic$;
    endfcn

function $isCompatible(absFunc\ f,\ callSite\ c)$ returns boolean:
    return if $signature(f) \mapsto actuals(c)$ then $true$ else $false$;
    endfcn;
```

Figure 4.18: Finding compatible function implementations

The function *implementable* checks that v can be assigned the implementation type t. The function call

$t.feasible(v)$

calls the Implementor-supplied feasibility function for the implementation t to verify that t is a feasible representation for v.

Classes of variables due to aliasing in user functions

User-declared functions require some twists on the algorithm as we have presented it so far. User-declared functions can result in equivalence classes of variables caused by aliasing of variables with formal parameters in the signature of these functions. In our example on page 50 if representation R is assigned to $S1$, then it also has to be assigned to $S2$. These equivalence classes are reminiscent of Low's equivalence classes; however, his classes were imposed by the design decision to not allow mixed representation functions in the ADT interfaces and were much more restrictive; the equivalence classes of aliased variable names are much less so.

Figure 4.20 contains the modifications to the algorithm to handle this complication. The prime difference between Figure 4.17 and Figure 4.20 is that the former looks only at the variable, while the latter looks at all variables that are equivalent to the variable because of parameter passing to calls on user functions. This acts as an implementation optimization to make sure implementation selections are compatible without having to backtrack.

```
$absFcn(callSite\ c)$                    $\rightarrow$ $--$abstract function invoked at call site $c$;
$signature(absFunc\ f)$                  $\rightarrow$ $--$signature of abstract function;
$actuals(callSite\ c)$                   $\rightarrow$ $--$list of actual parameters at call site $c$;
$sortByImportance(\mathbf{Set}(callSite)\ \mathcal{C})$
                                         $\rightarrow$ $--$list of call sites sorted in decreasing
                                           $--$order of the results of the evaluation functions
                                           $--$in the profiling implementation;
$sortByCost(\mathbf{Set}(impFunc)\ \mathcal{F})$
                                         $\rightarrow$ $--$list of implementation functions sorted in
                                           $--$increasing order of estimated execution cost;
$first(\mathbf{List}(any)\ T)$           $\rightarrow$ $--$first element of list;
$implement(variable\ v,\ implType\ t)$
                                         $\rightarrow$ $--$update implementation list contained
                                           $--$in global variable $assignment$
$assignImplType(variable\ v,\ implType\ t,\ signature\ s)$
                                         $\rightarrow$ $--$assign all occurrences of variable $v$
                                           $--$the implType $t$ in the signature $s$
$assignImplType(variable\ v,\ implType\ t,\ signature\ s)$
                                         $\rightarrow$ $--$return the signature $s$ with all
                                           $--$instances of $v$ assigned implementation type $t$
```

Figure 4.19: Miscellaneous functions

```
proc $parmsImplementable(impFunc\ f,\ absFunc\ F)$ returns boolean:
    foreach $v \in signature(F)$, each $t \in signature(f)$ do
        if $not\ alreadyImpltd(v)$ then
            if $implementable(w,t)\ \forall w \in class(v)$ then
                $implement(w,t)\ \forall w \in class(v)$;
            else $--$conflict; backtrack
                return $false$;
            fi
        fi
        endfor
    return $true$;
    endproc
```

Figure 4.20: Mapping parameters onto implemented functions' signature with equivalence classes

```
class Set {
    void in(Any e);
    void add(Any e);
    void subset(Set s);
};
```

Figure 4.21: Declarations of generic Set functions

4.4.2 Code sharing

For each user variable declared to be, say, a set of some user-type *UType*, a naive implementation of the generic abstraction of sets would create a new copy of the implementation code for sets with all instances of the generic parameter replaced with *UType*. In general, this is quite unnecessary, and particularly so in the context of TypeSetter where all ADTs are restricted to be exomorphic and *UType* is restricted to be a pointer to a user type. In this case, code can be written once to handle sets of pointers to objects. In the case of sets of various sized integers, only one copy need be created for each size of integer. If pointers are the same size as one of the sizes of integers, the same version of Set code can be used for both.

However, it is important not to give up strong type checking to gain this savings in code space. Users should still be notified when their programs violate the declarations they themselves have made.

For instance, in Figure 4.21 are the generic declarations of some of the interface functions for sets. The abstraction of sets is generic in one type, that of the type of the elements of the sets. If the user declares types Token and String, for instance, and then declares variables to be `Set(Token)` and others to be `Set(String)`, it would be wasteful to have two implementations of the set functions, one for Tokens and one for Strings, since both are actually implemented as a set of pointers to tokens, and set of pointers to strings. All that is needed as an implementation of sets that can handle pointers.

All that is needed to maintain strong type-checking is the declaration of *coercion types* for a set of Tokens and a set of Strings, each of which calls the appropriate function for handling sets of pointers. Therefore, TypeSetter generates one implementation of the set functions capable of handling pointers (and incidentally four-byte integers on many systems) and then generates a coercion class for maintaining strong type checking. The declarations for an implementation of 'set of pointers' are in Figure 4.22; Figure 4.23 contains the coercion implementations. This model of typing and implementation of generic functions builds on the *template* idea first proposed by Stroustroup [10] for C++ but is much more powerful in that it allows the Implementor more control over how much new source code is generated. Such control cannot be created easily in C++ without extending the language fur-

```
class Set_ptr {
    void      in(void* e);
    void      add(void* e);
    boolean   subset(Set_ptr& s);
};
```

Figure 4.22: Declarations of generic Set functions

```
class Set_Token : public Set_ptr {
    void      in(Token* e) { Set_ptr::in((void*) e); }
    void      add(Token* e){ Set_ptr::add((void*) e); }
    boolean   subset(Set_Token& s) { return Set_ptr::subset((Set_ptr&)s); }
};
class Set_String : public Set_ptr {
    void      in(String* e) { Set_ptr::in((void*) e); }
    void      add(String* e){ Set_ptr::add((void*) e); }
    boolean   subset(Set_String& s) { return Set_ptr::subset((Set_ptr&)s); }
};
```

Figure 4.23: Declarations of generic Set functions

ther than is suggested in the latest C++ language reference by Ellis and Stroustrup [10].

4.4.3 Refinements

In order to simplify the above discussion of THERBLIG, I have not included the enhancements added to the code that allows THERBLIG to search the solution space in a controlled manner. In the actual implementation (see Appendix B), using a mechanism very similar to that used in the Greedy Sewing Algorithm, THERBLIG can be invoked with a parameter p that specifies indirectly the portion of call sites that are 'optimally' assigned implementations; i.e., the search for their implementations is exhaustive, with every possible combination of implementations examined.

The set of call sites is sorted in decreasing order of the values returned by the profiling implementation's evaluation functions. Let $S = \sum_i^n C(i)$, where $C(i)$ is the cost estimate returned for the function at the i^{th} location in the list. The sum of these values, S, is multiplied by the parameter p, a number between 0 and 1, to determine a cutoff point in the list of sorted call sites. For a call site at location i in the list of sorted call sites, it is above the cutoff point if $\sum_{j<i} C(j) < p * S$, and it is below the cutoff point if $\sum_{j<i} C(j) >= p * S$. At each point in the assignment algorithm, if the call site being considered is below the cutoff point, only the first consistent implementation is returned, and all others are ignored. If the call site is above the cutoff point, then each consistent implementation is examined to see if it improves the program's implementation.

Invoking THERBLIG with $p = .9$, all possible implementations are examined for the call sites that account for 90% of the estimated runtime resources. By setting $p = 1$, all possible implementations are examined. The results and methods of the algorithms as I have described them above are achieved by setting $p = 0$; i.e., THERBLIG returns the first consistent implementation for the program.

4.5 Examples

I have demonstrated that TYPESETTER code is not difficult to write, either for the User or for the Implementor. No specialized knowledge of compilers or profiling technology is required by either the User or the Implementor. The Implementor specifies the information required to make a reasonable implementation decision with normal-looking programming language statements; the only difference is that profiling variables are allocated per call site rather than per function. In an ideal implementation of TYPESETTER, the User would need only (1) to re-compile the system as directed by the system (although this cycling could certainly be automated), and (2) to be aware of the different kinds of optional information that may be specified for a data type (e.g., upper and lower bounds on elements of sets).

Given profile data and User declarations, TYPESETTER gives the User program a 'reasonable' implementation. I cannot claim that TYPESETTER constructs 'optimal' implementations: the whole process of software construction is too heuristic to allow such a claim. Future work can concentrate on determining exactly how 'optimal' an implementation of a User's program is possible. For this exploratory work, I have concentrated on demonstrating that the implementations chosen are not 'wrong', that is, that TYPESETTER chooses an implementation for an abstract data type that a human programmer would agree is a reasonable candidate.

To convince the reader, I will present some results using three examples to demonstrate TYPESETTER's flexibility. The first example is a small program (approximately 60 lines) that is useless except to the extent it displays some of the capabilities of TYPESETTER. The second example is an implementation of the MINOPT algorithm presented in section 2.2.2. Finally, we will look at TYPESETTER itself, and examine how it chooses its own implementation. The TYPESETTER prototype has nine implementations spread among the three abstractions Set, List, and Map. Set has five implementations, and the other two have two apiece. Since Sets have more possibilities than the other two ADTs, we will concentrate on showing how TYPESETTER performs on variables declared to be sets of User-defined objects.

There are two distinct questions that the prototype was designed to answer.

The first is to test our hypothesis that a greedy assignment algorithm works well. To recap, the implementation assignment algorithm used in TypeSetter sorts the call sites in decreasing order of importance (where importance is estimated by evaluation functions provided by the Implementor), assigns the most efficient implementation to the first call site, and then, in decreasing order of importance, assigns to all other call sites the most efficient implementation that is consistent with previous assignments. We want to know how quickly an initial assignment of implementations is made, and how close that assignment is to the 'optimal' solution, assuming that the performance estimates returned by the evaluation functions is accurate.

The second question is: how accurate are the estimates returned by the Implementor's evaluation functions? Or, in other words, how closely does the final performance of the implemented program correlate with the predictions made by the Implementor's evaluation functions?

4.5.1 Small example

Figure 4.24 contains the TypeSetter code for an example program that constructs three sets of integers. Given that the declarations of the variables contain some optional declarations that tell TypeSetter the sets really are sets of integers, it is not too surprising that TypeSetter selects a bit-mapped implementation for them over a linked-list implementation. The primary point of this example, however, is TypeSetter's ability to share code between instantiations of abstract types. Even though the set *cSet* has more elements than do sets *aSet* and *bSet*, they all three have few enough base elements that they can fit in a 32-bit word, hence they will all use the same implementation of one-word bitmaps. However, because *aSet* and *bSet* were declared to be sets of *short* integers, they will share a coercion class that is different from *cSet*'s.

Figure 4.25 shows the order of priority given to the call sites of our example program based on estimates provided by the profiling implementation's evaluation routines. Each line shows the name of the function being invoked, the line in the file where the invocation occurs, the index in the profiling array for this file (there is one for each source file making up a program), and the values of the profiling variables. Finally, the "profiling costs" (actually an estimate of the cost) is given. The call sites are sorted in decreasing order based on those estimates.

We'll look closely at the information printed for the *Set__add* function on line 35 of our test program (the line numbers are not contiguous due to some irrelevant material not included). It has seven profiling variables corresponding to the numbers given in parentheses: *p_cnt*, *p_szA*, *p_appended*, *p_prepended*, *p_wasIn*, *p_inserted*, and *p_lookedAt*. Respectively, they count the number of times this call site was executed (*p_cnt*=8), the sum of the size of the set at each invocation (*p_szA*=28), the number of times the element could be appended to a list in which the elements of the set were sorted by their memory address (*p_appended*=8), or prepended

```
#define LOOPSIZE 1

DECLARE(aSet, Set, short, ObjsAreInts, upperb=15, lowerb=0);
DECLARE(bSet, Set, short, ObjsAreInts, upperb=15, lowerb=0);
DECLARE(cSet, Set, int, ObjsAreInts, upperb=31, lowerb=0);

main()
{
  int i;
  for (i = 0; i < 16; i++) {
      if ((i & 2) != 0) {
          Set_add(aSet,i);
          }
      }
  Set_add(aSet,1);
  Set_add(aSet,10);
  for (i = 0; i < 16; i++) {
      if ((i & 4) != 0) {
          Set_add(bSet,i);
          }
      }
  //
  for (i = 0; i < 32; i++) {
      if ((i & 15) == 15) {
          Set_add(cSet,i);
          }
      }
  Set_intersect1(bSet, aSet);
  //
  // for every integer in the set c
  //
  for (int j=0; j < LOOPSIZE; j++) {
      forAll(i, cSet,
             if (j == 0) {
                 cout << "Found " << i << "\n";
                 }
             );
      }
}
```

Figure 4.24: Small example

```
Sorted call sites:
Set__intersect1 (line 60 file impltest p[40]=(1,8,9,13,4,4)
                                                        profiling costs 72)
Set__add (line 35 file impltest p[5]=(8,28,8,0,0,0,28) profiling costs 36)
Set__add (line 45 file impltest p[26]=(8,28,8,0,0,0,28)
                                                        profiling costs 36)
Set__add (line 38 file impltest p[12]=(1,8,0,1,0,0,0) profiling costs 9)
Set__add (line 39 file impltest p[19]=(1,9,0,0,1,0,5) profiling costs 9)
Set__add (line 54 file impltest p[33]=(2,1,2,0,0,0,1) profiling costs 3)
Set__iterate (line 72 file impltest p[47]=(3,6) profiling costs 3)
Set (line 25 file impltest p[0]=(1) profiling costs 1)
Set (line 26 file impltest p[1]=(1) profiling costs 1)
Set (line 27 file impltest p[2]=(1) profiling costs 1)
Set__iterInit (line 72 file impltest p[46]=(1) profiling costs 1)
Set__iterCleanup (line 72 file impltest p[49]=(1) profiling costs 1)
```

Figure 4.25: The call sites sorted by profiling estimates of importance

(*p_prepended*=0), the number of times the element being added was already in the set (*p_wasIn*=0), the number of times the element being added had to be inserted into the interior of a list sorted by address (*p_inserted*=0), and the sum of the number of elements that had to be examined over all calls to this functions (*p_lookedAt*=28). The actual code for the profiling implementation for sets is given in Figure 4.26, from which we can see that the profiling implementation also keeps the elements of the set on a list sorted by their memory address.

From Figure 4.25, we can see that the profiling evaluation routines consider the intersection operation on line 60 to be the dominating factor in this program, giving it a weight (72) twice the nearest competitor (the two *add*s, weight 36). Since the intersection function has *aSet* and *bSet* as parameters, assigning an implementation to the intersection function on line 60 will also assign implementations to those two variables. The evaluation functions for the four possible implementations of sets produced the estimates in Figure 4.27 for the *intersect* function. The bitmapped-word implementation is the cheapest, while the most expensive implementation is the one that keeps the elements on a sorted list (in this case, the list is sorted by the values of the integers): apparently, the number of elements in the two sets is not sufficient to pay for the extra overhead of keeping the lists sorted. Therefore, the intersection function on line 60 was assigned *Set_bmwrd_intersect1*, the single word bit-map implementation for sets whose base size is less than or equal to 32. Once this implementation for the function is decided upon, then the arguments to *intersect1* (*aSet* and *bSet*) will be assigned types corresponding to the formal

```
void
FUNCTION(add)( Any e)
{
  Set_P_Link *lp = firstp;
  Set_P_Link **bp = &firstp;
  p_cnt++;
  p_szA += len;
  if (lp != nil && e < lp->data) {
      p_prepended++;
      }
  else {
      while (lp != nil && e > lp->data) {
          bp = &lp->next;
          lp = lp->next;
          p_lookedAt++;
          }
      if (lp == nil) p_appended++;
      else if (e == lp->data) p_wasIn++;
      else p_inserted++;
      }
  if (lp == nil || e < lp->data) {
      Set_P_Link *tp = new Set_P_Link;
      tp->data = e;
      tp->next = lp;
      *bp = tp;
      len++;
      }
  assert(p_cnt == p_prepended + p_appended + p_wasIn + p_inserted);
}
```

Figure 4.26: The actual profiling implementation for the *add* function for Sets

```
Callsite(40): Set_bmwrd::intersect1__=1.5
Callsite(40): Set_slist::intersect1__=68.1125
Callsite(40): Set_bmarr::intersect1__=5.9
Callsite(40): Set_slistord::intersect1__=91.4
```

Figure 4.27: Estimates of the cost of the intersection function

```
Callsite(40): Set_bmwrd::add__=12
Callsite(40): Set_slist::add__=23.4
Callsite(40): Set_bmarr::add__=16.8
Callsite(40): Set_slistord::add__=27.6
```

Figure 4.28: Estimates of the cost of the add function on line 54

types of *Set_bmwrd_intersect1*, which in this case is also *Set_bmwrd*. Our prototype has only one implementation of the intersect function with two parameters defined. TYPESETTER is designed to allow as many implementations as Implementors may deem usable in various situations. This is easily incorporated into TYPESETTER because we concentrate on assigning implementations to *functions*; in other words, implementations of variables occurs as a side effect of assigning implementations to functions.

After assigning an implementation to the intersection function, the only variable implementation remaining to be decided is that of *cSet*. Since it does not interact with either *aSet* or *bSet* in a function call, its implementation is independent of theirs. The call on *Set__add* on line 54 of the program is the most important function, according to the profiling implementation's estimates. Figure 4.28 gives the implementations' estimates of the cost of calling their respective versions of the the Set__add function. Therefore, *cSet* is also assigned the word bitmap implementation. Figure 4.29 shows TYPESETTER's output, specifying the types of the variables of our small program based on these considerations. The specifications are interpreted as follows:

`INSTANTIATE(I,N,P...)` Create source code for implementation I (name the class N) with parameters P. In our example in Figure 4.29, the instantiations require two parameters: the functions for taking an object to an integer and the inverse.

`COERCE(I,N,C,P...)` Create a coercion class C which converts calls on the functional interface of implementation I into the instantiation class N, using the parameters P.

`DECLARE_M(V,C)` Declare variable V to be of type (coercion class) C.

In the sample program is a constant that determines the number of times the loop containing the iteration over the elements of *cSet* is executed. If that constant is set to ten, instead of one, then the profile data induces TYPESETTER to make a different implementation assignment to *cSet*. Figure 4.30 contains a summary of the output from TYPESETTER, emphasizing the differences with the previous run of the program. The intersection function is still the most important, but the iterator functions have moved up in importance. Again, because of the independence

```
INSTANTIATE(Set_bmwrd, Set_bmwrd_int_int, int, int)@;
COERCE(Set_bmwrd, Set_bmwrd_int_int, Set_bmwrd_int_int_of_int, int)@;
DECLARE_M(cSet, Set_bmwrd_int_int_of_int, 32)@;
@;
INSTANTIATE(Set_bmwrd, Set_bmwrd_int_int, int, int)@;
COERCE(Set_bmwrd, Set_bmwrd_int_int, Set_bmwrd_int_int_of_short, short)@;
DECLARE_M(bSet, Set_bmwrd_int_int_of_short, 16)@;
@;
INSTANTIATE(Set_bmwrd, Set_bmwrd_int_int, int, int)@;
COERCE(Set_bmwrd, Set_bmwrd_int_int, Set_bmwrd_int_int_of_short, short)@;
DECLARE_M(aSet, Set_bmwrd_int_int_of_short, 16)@;
```

Figure 4.29: TypeSetter's assignment of types to the program

of *cSet* from the other variables in the program, there is no effect except on *cSet*'s implementation. Now its most important call site is the call on *Set__iterate*, and the various implementations' estimates of cost are shown in Figure 4.30. *Set_slist* and *Set_slistord* evaluate the same since they are both linked-list implementations, differing only in the order in which the elements of the set are returned. Selecting between them more or less at random results in assigning the *Set_slist* implementation to *cSet*.

The above implementations were assigned by Therblig with $p = 0$; that is, the implementations chosen were the first consistent set of implementations. While a reasonable argument can be made for the implementations that were selected, two question still remain. Are they the best implementations possible, given the results of the evaluation functions? And does the performance of the program improve?

To be able to answer the second question, we have to have a program that requires a non-trivial amount of time. To that end, we modify our `LOOPSIZE` macro to 100,000. To answer the first question, we run Therblig on the program with $p = 1$; i.e. all possible assignments of implementations are evaluated. On this small example, it made no difference: an exhaustive search across all possible implementations of the program still assigned *Set_slist* to *cSet*, and *Set_bmwrd* to *aSet* and *bSet*.

Table 4.1 shows the various running times of our small example program when *cSet* is implemented with each of the possible implementations for Set. From it, we can see that Therblig correctly chose *Set_slist* as one of the best implementations possible for *cSet*. Running Therblig with $p = .9$ produced exactly the same result as $p = 1$, corroborating my hypothesis.

```
Sorted call sites:
Set__intersect1 (line 60 file impltest p[40]=(1,8,9,13,4,4)
                                                       profiling costs 72)
Set__add (line 35 file impltest p[5]=(8,28,8,0,0,0,28) profiling costs 36)
Set__add (line 45 file impltest p[26]=(8,28,8,0,0,0,28)
                                                       profiling costs 36)
Set__iterate (line 72 file impltest p[47]=(30,60) profiling costs 30)
Set__iterInit (line 72 file impltest p[46]=(10) profiling costs 10)
Set__iterCleanup (line 72 file impltest p[49]=(10) profiling costs 10)
Set__add (line 38 file impltest p[12]=(1,8,0,1,0,0,0) profiling costs 9)
Set__add (line 39 file impltest p[19]=(1,9,0,0,1,0,5) profiling costs 9)
Set__add (line 54 file impltest p[33]=(2,1,2,0,0,0,1) profiling costs 3)
Set (line 25 file impltest p[0]=(1) profiling costs 1)
Set (line 26 file impltest p[1]=(1) profiling costs 1)
Set (line 27 file impltest p[2]=(1) profiling costs 1)

Callsite(58): Set_bmwrd::iterate__=377.830
Callsite(58): Set_bmarr::iterate__=367.916
Callsite(58): Set_slist::iterate__=105
Callsite(58): Set_slistord::iterate__=105

INSTANTIATE(Set_slist, Set_slist)@;
COERCE(Set_slist, Set_slist, Set_slist_of_int, int)@;
DECLARE_M(cSet, Set_slist_of_int)@;
@;
INSTANTIATE(Set_bmwrd, Set_bmwrd_int_int, int, int)@;
COERCE(Set_bmwrd, Set_bmwrd_int_int, Set_bmwrd_int_int_of_short, short)@;
DECLARE_M(bSet, Set_bmwrd_int_int_of_short, 16)@;
@;
INSTANTIATE(Set_bmwrd, Set_bmwrd_int_int, int, int)@;
COERCE(Set_bmwrd, Set_bmwrd_int_int, Set_bmwrd_int_int_of_short, short)@;
DECLARE_M(aSet, Set_bmwrd_int_int_of_short, 16)@;
```

Figure 4.30: Results from the example with LOOPCOUNT= 10

Set_slist	2.30s
Set_slistord	2.30s
Set_bmwrd	6.32s
Set_bmarr	7.63s

Table 4.1: Small example running times with various implementation assignments for *cSet*

4.5.2 MINOPT

Appendix A contains TypeSetter code for an implementation of the MINOPT algorithm discussed in section 2.2.2. This program, we'll call it *minopt*, has three set variables, two list variables, and one map. The map is a dictionary mapping tokens onto node and arc names. In order to compute execution frequencies of a graph object (arc or node) that is not instrumented, each non-instrumented object maintains a list of other graph objects from which its execution count is com-

Graph	gozintas	gozoutas	time
Set_bmarr	Set_slist	Set_slist	2.50s
Set_slist	Set_slist	Set_slist	2.92s
Set_slistord	Set_slistord	Set_slistord	3.37s
Set_bmarr	Set_bmarr	Set_bmarr	3.79s

Table 4.2: Running times for the K-S algorithm.

puted; this is class member variable *flowlist* in class *GraphObj_obj*. The other list is a sorted list of all graph objects (*sortedObjList*).

The remaining three set variables are: *Graph*, the set of all objects (nodes and arcs) that comprise the current graph; *Node_obj::gozintas*, the set of all arcs that enter a node; and *Node_obj::gozoutas*, the set of all arcs that exit a node.

Profile data was generated by running two PFGs through *minopt*, one is a small five-node graph that Knuth and Stevenson used as an example in their paper [30], and the other is the graph in Figure 2.3. Based on that profile data, THERBLIG,with $p = 0$, selected *Set_bmarr* for the variable *Graph*, and *Set_slist* for the two arc lists, *gozintas* and *gozoutas*. This seems a reasonable assignment of implementations, since *Graph* is added to and iterated over, but nothing else. Since it is a completely full set, there are no penalties to pay in a bitmap implementation for having to check bits in that map that aren't set. This is not the case for the *gozintas* and *gozoutas* variables: the number of arcs coming into or leaving an arc is never more than three in our example graphs; a linked list would do much better for these two variables.

With $p = 1$, THERBLIG makes exactly the same choices, again in support of the hypothesis that implementation decisions made early are close to the 'optimal'. Table 4.2 are *minopt*'s running times when the variables are assigned as shown. The input data is a 364-node graph made by replicating and concatenating the graph in Figure 2.3. The first entry in the table uses the implementations chosen by THERBLIG, and the remainder show that it was indeed a reasonable set of implementations.

4.5.3 Implementing THERBLIG

THERBLIG is the analysis software for the TYPESETTER system. From the descriptions of the available abstractions and their implementations, and the description of the User's program, it selects implementations for the variables declared, and functions invoked, in the User's program. THERBLIG is the most complex software written in TYPESETTER and, therefore, will be our next example.

THERBLIG consists of over 8500 lines of TYPESETTER code and comments. This includes almost 2500 lines of TYPESETTER code for the analysis portion of the software, with the other 6000 lines taken up by the nine implementations of the three abstractions of Sets, Lists, and Maps. Sets has five implementations, including the profiling implementation, while Lists and Maps have two apiece.

In the body of THERBLIG, there are 23 variables utilizing these abstractions: four are Lists, seven are Maps, and eleven are Sets. Given that Sets have a more complete set of implementations than do the other abstractions we will look at how they are implemented by TypeSetter. The complete listing of Therblig is given in Appendix B for reference.

The eleven Set variables in Therblig are:

ADTcalls: The set of all call sites in the User's program.

Ic: A formal parameter to the function *findCompatibleImplementations* (see page 81).

adt_afcns: A member of the class *ADType_obj* representing abstract data types; it is the set of all functions that define the interface to the abstraction.

adtaf_impl_fcns: A member of the class *ADTabsFcn_obj* representing the abstract functions, each of which will have a set of functions that are its implementations; this is that set of implementation functions.

callSites: A variable, local to the function *implementable*, containing all call sites that have a specific variable in their argument lists.

callSitesp: A formal parameter which contains all call sites with a specific variable in their argument lists.

changed: A local variable (in function *assignable*) which keeps track of variables which have been given tentative assignments.

changedp: A formal parameter which keeps track of variables which have been given tentative assignments.

implSet: A local variable to function *assignable* that keeps track of all function implementations that are compatible with the current state of assignments and a particular call site.

ivars: A formal parameter to *undoImplementations* that is a set of variables whose tentative assignments are to be undone.

vd_inSigsOf: A member of the class *VarDecl_obj* that contains the set of all call sites which have this variable as an actual argument.

as_set: Each variable used as a parameter to a User-defined function will be aliased to other variables; this is the set of aliases for a variable. It is a member of the class *AliasSet_obj*.

I have not attempted to solve the problem of detecting input-dependent behavior: that is still up to Users to realize about their own programs, and to take appropriate actions, specifically to run the programs a sufficient number of times with typical input data to cover all the important behaviors of their programs. The problem of determining when sufficient 'typical' input data has been utilized is also beyond the scope of this dissertation. THERBLIG is interesting because it exhibits input-dependent behavior: if there is no profile data for the User's program, it selects only profiling implementations for the program. Because this is a straightforward procedure that does not require much computation, multiple profiling runs of THERBLIG are required to insure that the profiling data is representative of the average behavior of the program. Once profile data exists for THERBLIG to analyze, then the more extensive analysis described in section 4.4.1 is executed to determine an implementation for the program under consideration. Looking at how THERBLIG modifies its idea of a good selection will give us some insight into how it works.

The first step is to build a version of THERBLIG with all variables implemented by the profiling versions of their underlying abstraction. The second step is to run THERBLIG feeding it its own source code. Since there is not yet any profile data, this run simply reads the program description, and writes a specification file giving all of its variables a profiling implementation: this is more-or-less a do-nothing run of THERBLIG.

The third step is to run THERBLIG again, but this time there is profile data generated from its first run. Figure 4.31 shows the call sites of all Set interface functions sorted in the order indicated by the Set profiling implementation's evaluation functions using the profile data generated on the first run. There were a total of 208 call sites in the THERBLIG sources, 174 of which are in the main code. Of these 174, only 53 involve calls on Set interface functions. (Only the non-zero cost call sites that invoke functions in the Set abstraction are shown in the figures.)

The Set profiling implementation's evaluation functions estimate that the call site on line 761, which constructs Set objects, is possibly the greatest bottleneck in the program, with an invocation of *add* trailing a very close second. In general, the profiling implementations' evaluation functions attempt to estimate the *potential* impact of the function at a particular call site—it is a worst case evaluation. In the case of the first *add* on the list, if the implementation chosen were a linked-list implementation, then adding an element could mean having to examine all of the current members looking for duplication; hence the large estimate. Given the number of times the add function on line 1078 was called (*p_cnt*= 208), and the sum of the set sizes across those calls (*p_szA*= 21528), we can estimate the average size of the set for each call to be 103.5 (and we note that 208/2 = 104). Of these 208 calls, exactly *p_appended*= 208 elements were appended to the address-sorted list, and *p_prepended*= 0 were prepended. There were no elements already in the set (*p_wasIn*= 0), and no elements had to be inserted in the list (*p_inserted*= 0). Because all of the elements were appended to the list, then p_lookedAt= 21528.

Based on this data, THERBLIG assigned the implementations to the variables as shown in Table 4.3, first column. (Only the Set variables are shown.)

The next step is to run THERBLIG a third time. The last run added to the existing profile data, and did so while executing code that it did not execute during the first run. The question naturally arises as to whether this would change the as-

```
Sorted call sites:
Set (line 761 file main p[88]=(685) profiling costs 21920)
Set__add (line 1078 file main p[176]=(208,21528,208,0,0,0,21528)
                                               profiling costs 21736)
Set__add (line 1073 file main p[168]=(213,2407,211,0,2,0,2405)
                                                profiling costs 2618)
Set (line 318 file main p[15]=(276) profiling costs 2208)
Set (line 302 file main p[9]=(239) profiling costs 1912)
Set (line 330 file main p[21]=(119) profiling costs 952)
Set__add (line 763 file main p[89]=(685,0,685,0,0,0,0)
                                                 profiling costs 685)
Set__add (line 827 file main p[103]=(56,519,56,0,0,0,519)
                                                 profiling costs 575)
Set (line 310 file main p[12]=(51) profiling costs 408)
Set (line 610 file main p[85]=(56) profiling costs 392)
Set__add (line 882 file main p[121]=(114,126,114,0,0,0,126)
                                                 profiling costs 240)
Set (line 487 file main p[63]=(4) profiling costs 28)
Set__union1 (line 769 file main p[96]=(4,5,4,5,0) profiling costs 20)
Set (line 130 file main p[3]=(1) profiling costs 8)
Set__iterate (line 389 file main p[33]=(8,8) profiling costs 8)
Set__iterInit (line 389 file main p[32]=(4) profiling costs 4)
Set__iterCleanup (line 389 file main p[35]=(4) profiling costs 4)
:
```

Figure 4.31: The sorted call sites of Set functions from one Therblig run

Variable	Assignment 1	Assignment 2	Assignment 3
ADTcalls	Set_bmarr	Set_bmarr	Set_bmarr
Ic	Set_bmarr	* Set_slist	Set_slist
adt_afcns	Set_bmarr	Set_bmarr	Set_bmarr
adtaf_impl_fcns	Set_bmarr	* Set_slist	Set_slist
as_set	Set_slist	Set_slist	Set_slist
callSites	Set_bmarr	Set_bmarr	Set_bmarr
callSitesp	Set_bmarr	Set_bmarr	Set_bmarr
changed	Set_bmarr	* Set_slist	Set_slist
changedp	Set_bmarr	* Set_slist	Set_slist
implSet	Set_bmarr	* Set_slist	Set_slist
ivars	Set_bmarr	* Set_slist	Set_slist
vd_inSigsOf	Set_bmarr	Set_bmarr	Set_bmarr

Table 4.3: Variable assignments based on profileof three runs of THERBLIG with $p = 0$

signments of implementations to variables. And indeed it does, as the second column of Table 4.3 shows. The changed implementations are marked with an asterisk.

The third column of Table 4.3 shows the assignments when THERBLIG is run for a fourth time. But by now the statistics have stabilized, and the assignments do not change.

All of the runs of THERBLIG above were with $p = 0$. Table 4.4 shows the results of running Therblig with $p = 1$. The asterisk beside the entry for the first assignment means that the first choice with $p = 1$ differed from the first choice when $p = 0$. The asterisks in later columns means, as before, that THERBLIG changed the implementation based on more profile data. Again, the assignments have stabilized by the third run.

There are only three differences between the selections made when $p = 0$ and $p = 1$: the variables *callSites*, *callSitesp*, and *vd_inSigsOf* were formerly *Set_bmarr*, a bit mapped array. Looking at all possible combinations of assignment resulted in those implementations being changed to a *Set_slistord*, a simple linked list that keeps its member in the order of their memory addresses.

And finally, Table 4.5 shows the implementations selected when running with $p = .9$. The important point to note here is that running with $p = .9$ means that only about thirty out of 208 call sites (about 15%) are exhaustively analyzed: the remaining 170-some-odd call sites are assigned the first consistent implementation found. The asterisks in the first column indicate that the first assignment differs from the first assignment in Table 4.4. Asterisks in later columns highlight differences from the preceding column. It is interesting to note that the assignments had not stabilized by the third assignment. I did not determine how many iterations $p = .9$ would have required to stabilize.

Variable		Assignment 1		Assignment 2	Assignment 3
ADTcalls		Set_bmarr		Set_bmarr	Set_bmarr
Ic		Set_bmarr	*	Set_slist	Set_slist
adt_afcns		Set_bmarr		Set_bmarr	Set_bmarr
adtaf_impl_fcns	*	Set_bmarr	*	Set_slist	Set_slist
as_set		Set_slist		Set_slist	Set_slist
callSites	*	Set_slistord		Set_slistord	Set_slistord
callSitesp	*	Set_slistord		Set_slistord	Set_slistord
changed		Set_bmarr	*	Set_slist	Set_slist
changedp		Set_bmarr	*	Set_slist	Set_slist
implSet		Set_bmarr	*	Set_slist	Set_slist
ivars		Set_bmarr	*	Set_slist	Set_slist
vd_inSigsOf	*	Set_slistord		Set_slistord	Set_slistord

Table 4.4: Variable assignments based on the profile of three runs of Therblig with $p = 1$

Variable		Assignment 1		Assignment 2		Assignment 3
ADTcalls		Set_bmarr		Set_bmarr		Set_bmarr
Ic		Set_bmarr	*	Set_slist		Set_slist
adt_afcns		Set_bmarr		Set_bmarr		Set_bmarr
adtaf_impl_fcns	*	Set_slist		Set_slist		Set_slist
as_set		Set_slist		Set_slist		Set_slist
callSites	*	Set_bmarr		Set_bmarr	*	Set_slist
callSitesp	*	Set_bmarr		Set_bmarr	*	Set_slist
changed		Set_bmarr	*	Set_slist		Set_slist
changedp		Set_bmarr	*	Set_slist		Set_slist
implSet		Set_bmarr	*	Set_slist		Set_slist
ivars		Set_bmarr	*	Set_slist		Set_slist
vd_inSigsOf	*	Set_bmarr		Set_bmarr	*	Set_slist

Table 4.5: Variable assignments based on the profile of three runs of Therblig with $p = .9$

1	$p = 0$	34.92s
2	$p = .9$	32.60s
3	$p = 1$	35.98s
4	Set_slist	36.21s
5	Set_slistord	36.36s
6	Set_bmarr	152.17s
7	profiling	44.73s

Table 4.6: THERBLIG running times with various implementation assignments

To measure the effectiveness of the assignments, I examined the output from THERBLIG to see if I could have done better with the existing implementations of sets. In essence, I manually performed Low's search heuristic: perturbing an existing assignment of implementations to see if another would be better. For THERBLIG, there are only three implementations of sets that are feasible:

Set_slist: A simple list, with a single link to successive elements, and a single pointer to the first element of the list.

Set_slistord: A singly-linked list as for Set_slist, with the addition of a pointer to the last element of the list, and the elements are kept on the list in the order of their memory addresses.

Set_bmarr: An array of bits; requires functions to map objects to integers and integers to objects.

The *Set_bmwrd* implementation is not feasible since all sets in THERBLIG have more than 32 elements. I ran seven versions of THERBLIG: one that uses only the profiling implementations; one with implementations assigned by THERBLIG running with $p = 0$ (assignment 3 from Table 4.3); one with implementations assigned with $p = 1$ (assignment 3 from Table 4.4); one with implementations assigned with $p = .9$ (assignment 3 from Table 4.5); and three others with all sets assigned *Set_bmarr*, *Set_slist*, and *Set_slistord*. The timing runs had $p = 1$ to exercise THERBLIG as fully as possible. The results are in Table 4.6.

Finally, we look at exactly how much it costs us to profile THERBLIG. A sense of the cost can be had by comparing the running times of the profiling implementation *vs.* the *Set_slist* implementation in Table 4.6. The *slist* implementation of sets was initially derived by removing all profiling code from the profiling implementation. From this, I estimate that profiling using counters in a special implementation slows down the program by 10-20%; that is, it runs 10-20% slower than it would if all the counting code were removed. However, in some cases, this will often be insignificant, particularly where the default implementation is ill-suited to the program being profiled. In that case, the major slow down of the program will be due to algorithmic unsuitability, and not to counting.

Chapter 5

Conclusion

For the last twenty years, no one has agreed with Knuth's Dictum (see page 1) enough to implement the idea, nor has anyone proven the assertion false. There are several assumptions in the Dictum, two of which have formed the central hypotheses of this work. They are:

- that profiling can be done efficiently enough so as not to be perceived as onerous by the programmer; and,
- that compilers and other tools can automatically extract useful information from profile data.

In the process of investigating the first of these hypotheses, I determined that an implicit assumption held by many programmers is false. Most programmers (myself included) have believed that counting executions of basic blocks is sufficient and more efficient than getting the more complete information about arc traversals. I demonstrated in Chapter 2 that this simply is not so. I presented an algorithm MINOPT which finds the 'optimal' instrumentation of a program by automatically placing the instrumentation code in the nodes or on the arcs. Previous algorithms have found optimal solutions for nodes, or for arcs. MINOPT is the first provably minimal algorithm for both nodes and arcs. I also pointed out that the 'optimal' algorithms aren't, that all have assumed the ability to compute instrumentation costs in linear time. I do not know whether there exists such an algorithm or not, but I have shown that if it does exist, it cannot be 'local'. That is, when estimating the instrumentation costs of a node's incoming and outgoing arcs, more information is required than just the execution frequencies of that node and its arcs.

My measurements showed that profiling in the form of in-line execution counts imposes anywhere from 10% to 20% overhead. This can be predicted solely from the observation that most programs' basic blocks average from four to ten instructions in size, and from the not too unrealistic assumption that incrementing a counter in memory requires about the average number of cycles for the execution of an instruction on a machine. Therefore, putting instrumentation in the most frequently executed basic block will produce a slowdown of 10-20% for the program as a whole.

Programmers would complain if a programs were slowed down 10-20% for no reason. That is, if no one (or thing) was making any use of the profile data, then programmers would turn off profile collection. (That is why all compilers today require the programmer to specify when to collect profile data.) However, the second of the hypotheses above would alleviate the problem considerably. If some part of the programming system were able to utilize the profile data to produce superior programs, then the profile collecting overhead is not onerous. It is comparable to the overhead of non-optimized bounds-checking code. While there has been some research in improving the overhead of profile collection (in particular, see Sarkar's paper on using dependency graphs to optimize profile counting [42]), there has yet to be a definitive exploration of the optimization of profile counting.

For there to be such research, it has to be shown that continual collection of profile data is a win. Therefore, I concentrated in Chapters 3 and 4 in exploring ways a compiler might make use of profile data. In Chapter 3 I presented an algorithm I call Greedy Sewing for improving the behavior of programs on machines with instruction caches. By physically moving basic blocks closer together that are executed close together in time, miss rates in instruction caches can be reduced up to 50%. Profile data not only allows the compiler to know which basic blocks to move closer together, it also allows it to ignore those situations where it will not matter to the final performance of the program.

The primary contribution of this work is the development of a programming system that utilizes profile data to select implementations of program abstractions. The TypeSetter system integrates the development, evaluation, and selection of alternative implementations of programming abstractions into a package that is transparent to the User. Unlike previous systems, TypeSetter does not require specialized compiler knowledge of the User or the Implementor. From the data collected so far, the TypeSetter approach to system synthesis appears to be a promising avenue of research.

5.1 Problems and future work

I have only scratched the surface of the body of engineering problems that need to be solved before TypeSetter can be considered a complete system. Some of these are related to problems inherent in using profile data to predict the future performance of a program, but others are related to the specific approach taken by TypeSetter.

Execution counts: During this work, I fell into an assumption that I think is widely shared, but which can cause problems. I had assumed that summing profile counts across multiple runs of a program was a reasonable approach to understanding the behavior of a program. But consider a program that has a function that is called once for each element on a list. For 99% of the elements, the function requires $O(1)$ time to execute. But for 1 out of 100 elements, it requires much more time. For example, let us assume that the occurrence of a certain kind of element requires that it be put in a separate list, and that sorting this list n-element list requires $O(n^2)$ time (it uses an inefficient sorting algorithm) where $n = 1$. If the program is run M times, and the profile counts used as measures of the complexity of this function are summed, then there comes a point where the one-in-a-hundred event dominates the analysis. If we assume a list that is 100 elements long, and one of the elements causes a re-sort, then running the program 100 times could make the list look like it was 10,000 elements long, with 100 re-sorts, implying that the sorting of the special elements requires as much time as the processing of the non-special elements, when in fact it never sorts a list longer than one element.

In general, this problem will rear its head when evaluation functions are non-linear in the values of the profile variable. For profilers like *prof* and *gprof*, this may not cause any particular problem, even though their output does not indicate how many runs of the program produced the data on which they base their analysis. THERBLIG was modified to count the number of executions of a program in addition to the counters specified in the profiling implementations. During analysis, all counters were divided by the number of program runs to try to avoid problems similar to the ones described in the previous paragraph. However, I am not satisfied that this avoids *all* problems of analysis from execution counts derived from multiple runs. This needs to be examined further.

Evaluation functions: The most difficult functions to write in TYPESETTER are the evaluation functions. While some of the difficulty is due to the fact that I've never had to write functions that evaluate the potential performance of other functions in such numbers before, they bring their own set of problems. For one thing, they are hardly ever 'wrong', at least not in the sense that inaccuracies produce obviously aberrant behavior on the part of the program. I have serendipitously discovered several instances where evaluation expressions I have written do not accurately reflect the performance of the actual function; even ignoring the fact that these are all estimates anyway, the results returned were misleading. Debugging these routines to a reasonable level of accuracy is difficult.

Kenny and Lin [27] report a technique for capturing the behavior of functions that might be usable in a THERBLIG-like environment. The Implementor would specify an expression with free variables that he suspects would adequately capture the behavior of the function in question; for example, $A * x + B * y^2 + C$, where x and y are parameters such as the length of a list, or size of set. By executing the function many times on many inputs, an average behavior for the function based on x and y can be found by determining appropriate values for A, B and C with a curve fitting algorithm. While this may be an approach for rigorously and more automatically producing evaluation functions, it will not reduce the amount of work required by the Implementor and may impede the Implementor from taking advantage of *logical* information contained in the profile data. For example, a curve-fitting approach may not be able to handle knowledge about the density of bit vectors, order of presentation of elements to a function, etc.

In general, evaluation functions need to be easier to write and debug.

Evaluating ADT-invoked User functions: There are several optionals that require the names of User-defined functions. The ones I have identified are ObjToInt, IntToObj, and compareFcn. They present problems when used because TypeSetter has no way of estimating the runtime resources of the indicated functions. Presumably, future systems will have the User give some indication of the cost of executing these functions so that the evaluation functions can give better estimates of the cost of using implementations that require them. I would like to avoid forcing the User to write evaluation functions: that is mixing the roles of User and Implementor too much. Exactly how to achieve the same result without User-written evaluation routines is yet to be determined.

The prototype finesses the problem entirely. Currently, the ObjToInt function must always be a reference to an integer field of the object, and IntToObj must be an array reference. This has not been terribly restrictive up to this point, but since the maintenance of the array of objects must be done by the User, it imposes some overhead that should be eliminated. Ideally, a map from integers to objects, and its inverse, should not be in the final implementation of a program unless it is needed. Currently, it will always be there, whether Therblig selects implementations that use them or not.

Second-order effects: Another problem arises when there are dependencies in the User program that are not part of the information available to a Therblig-like analyzer. Consider a program that keeps objects sorted on a list, but has its own sorted-list code rather than using a library routine. The list is created from a set of these objects, the implementation of said set assigned by the system. It could turn out that the implementation of the set causes the elements to be returned in an order that interferes with the efficient execution of the User's code: i.e. one implementation of set returns the elements in the order of their memory address which corresponds to the order in which they were constructed which in turn corresponds to the order data was read from a file. It is easy to see that there could be interference between the User's implementation and any implementation chosen by the system for the set, and no amount of analysis of the User's use of the *set* would uncover it.

This is outside the scope of a Therblig style system. One of its major premisses is that looking at the use of the ADTs alone is sufficient to make a reasonable assignment, and extra-ADT information is simply not made available to it. I have not encountered this kind of second-order effect in any of the programs I have run through THERBLIG, but theoretically it is possible.

Implementation containment: When a bit vector implementation of a set of size N is instantiated, then any declaration of a smaller set could share the code for the larger set. This would decrease the program's memory size further, at the expense of making the space allocated for some sets larger. Discovering and taking advantage of these tradeoffs would require the evaluation functions to consider space as well as time in their analysis. Since Low, for one, has already considered the more complex space-and-time integral objective function for minimization, I felt that duplicating this was not necessary to my objectives and I have concentrated on the simpler time-analysis.

Even if THERBLIG were capable of handling the space analysis, there is nothing in its analysis framework that would allow the kinds of implementation containment described above. In other words, there is no way for the evaluation functions written by the Implementor to conclude "Use implementation X unless condition Y holds, in which case use implementation Z." Again, future work will have to show, first, that this is an optimization that needs to be available and, second, how to obtain it.

Design of implementation libraries: I have barely begun to explore the possibilities in a library of implementations. As mentioned before, it may be desirable to have several profiling implementations, each capable of collecting certain kinds of information that is otherwise difficult to obtain. For example, once a bit vector implementation of a set is determined to be desirable, another bit-vector oriented profiling implementation could be used to determine which of the many bit vector implementations would be best for this program.

In the interests of simplicity, I have also avoided making use of the more complex language features available in my base language, C++. For instance, the implementations List_slist, Set_slist, and Map_slist all use the same implementations of a linked list as their underlying representation. Currently, they each have their own copies of this code, primarily because the *kinds* of profiling information collected differs between the implementations. It is possible that they could all be derived from a linked-list class, increasing even further the possibilities for code sharing. Future work is needed to look at integrating the class hierarchy and attendant inheritance into the library of implementations.

5.2 Summary

I have explored in some detail the proposition that compilers and language systems can make use of profile data in the generation of code for programs, and in the synthesis of large software systems. I have improved the existing 'optimal' instrumentation algorithms, and shown how arc counts can be used to improve the execution time of programs on machines with instruction caches. I have presented the design of a language and attendant system that can select for a User the implementations of variables declared to be of an abstract data type. I have also demonstrated that such a system can make reasonable choices for those implementations based on the profile data collected by abstraction-specific profiling implementations.

References

1. Agarwal, A., Sites, R. and Horowitz, M. ATUM: A New Technique for Capturing Address Traces Using Microcode. *Proceedings 13th Annual Symposium on Computer Architecture* (June 1986).
2. Baer, J-L. and Sager, G. R. Dynamic Improvement of Locality in Virtual Memory Systems. *IEEE Transactions on Software Engineering*, *SE-2*, 1 (March 1976), 54–62.
3. Barstow, D. *Automatic Construction of Algorithms and Data Structures using a Knowledge Base of Programming Rules*. PhD Dissertation, Stanford University, November 1977.
4. Barstow, D. On Convergence Toward a Database of Program Transformations. *ACM Transactions on Programming Languages and Systems*, *7*, 1 (January 1985), 1–9.
5. Bell Laboratories. *UNIX Programmer's Manual*. Murray Hill, NJ, 1984.
6. Borg, A., Kessler, R. E., Lazana, G. and Wall, D. W. Long Address Traces for RISC Machines: Generation and Analysis. DEC Western Research Lab, WRL Research Report 89/14, September 1989.
7. Cheung, R. C-yee A Structural Theory for Improving Software Reliability. University of California, Berkeley, PhD Dissertation, December 1974.
8. Chow, P. MIPS-X Instruction Set and Programmer's Manual. CSLSU, CSL-86-289, May 1986.
9. Dewar, R. B. K., Grand, A., Liu, S-C. and Schwartz, J. T. Programming by Refinement, as Exemplified by the SETL Representation Sublanguage. *ACM Transactions on Programming Languages and Systems*, *1*, 1 (January 1979), 27–49.
10. Ellis, M. and Stroustrup, B. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1990.
11. Fabri, J. *Automatic Storage Optimization*. PhD Dissertation, Courant Institute of Mathematical Sciences, NYU, 1979.
12. Ferrari, D. Improving Program Locality by Strategy-Oriented Restructuring. *Information Processing, Proceedings IFIP Congress 74*, New York, NY, Amsterdam (1974).
13. Fischer, C. N. and LeBlanc, R. J. *Crafting a Compiler*. Benjamin/Cummings, Menlo Park, CA, 1988.
14. Gilbreth, Jr., F. B. and Carey, E. G. *Cheaper by the Dozen*. T.Y. Crowell, New York, 1948.

15. Goldstine, H. H. and von Neumann, J. *Planning and Coding of Problems for an Electronic Computing Instrument*. Institute for Advanced Study, Princeton, NJ, 1947, (3 vols.).

16. Graham, S. L., Kessler, P. B. and McKusick, M. K. An Execution Profiler for Modular Programs. *Software–Practice Experience*, *13* (August 1983), 671–685.

17. Hansen, G. J. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. PhD Dissertation, Carnegie-Mellon University, Pittsburgh, PA, 1974.

18. Hill, M. D., Eggers, S. J., Larus, J. R., Taylor, G. S., Adams, G., Bose, B. K., Gibson, G. A., Hansen, P. M., Keller, J., Kong, S. I., Lee, C. G., Lee, D., Pendleton, J. M., Ritchie, S. A., Wood, D. A., Zorn, B. G., Hilfinger, P. N., Hodges, D., Katz, R. H., Ousterhout, J. and Patterson, D. A. Design Decisions in SPUR. *IEEE Computer*, *19*, 11 (November 1986).

19. Hill, M. D. DineroIII Cache Simulator. University of California, Berkeley, UNIX Programmer's Manual, August 1985.

20. Hill, M. D. Aspects of Cache Memory and Instruction Buffer Performance. Computer Science Division, EECS, University of California, Berkeley, Technical Report UCB/CSD 87/381, PhD Dissertation, November 1987.

21. Hwu, W-meiW. and Chang, P. P. Achieving High Instruction Cache Performance with an Optimizing Compiler. *Proceedings 16th Annual Symposium on Computer Architecture* (June 1989).

22. Ingalls, D. H. H. *The Execution Time Profile as a Programming Tool*. In *Design and Optimization of Compilers*, R. Rustin, Ed. Prentice Hall, Englewood Cliffs, NJ, 1972, 107–128.

23. Ingalls, D. H. H. FETE – A FORTRAN Execution Time Estimator. SUCSD, STAN-CS-71-204, February 1971.

24. Kant, E. Efficiency Considerations in Program Synthesis. Stanford University, PhD Dissertation, 1981.

25. Karr, M. Code Generation by Coagulation. *Proceedings of the ACM-SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices*, *19*, 6 (June 1984).

26. Katz, R. H., Eggers, S. J., Gibson, G. A., Hansen, P. M., Hill, M. D., Pendleton, J. M., Ritchie, S. A., Taylor, G. S., Wood, D. A. and Patterson, D. A. Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses. Computer Science Division, EECS, University of California, Berkeley, UCB/CSD 85/221, January 1985.

27. Kenny, K. B. and Lin, K-J. Performance Polymorphism: Integrating Performance Constraints in a Class Hierarchy. University of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-89-1523, Urbana, Illinois, May 1989.

28. Knuth, D. E. An Empirical Study of FORTRAN Programs. *Software–Practice Experience*, *1* (1971), 105–133.

29. Knuth, D. E. *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*. Addison Wesley, Reading, MA, 1973, 2nd ed..

30. Knuth, D. E. and Stevenson, F. R. Optimal measurement points for program frequency counts. *BIT*, *13* (1973), 313–322.

31. Low, J. R. Automatic Coding: Choice of Data Structures. Computer Science Department, Stanford University, PhD Dissertation, Technical Report CS-452, August 1974.

32. Low, J. R. Automatic Data Structure Selection: An Example and Overview. *Communications of the ACM*, *21*, 5 (May 1978), 376–385.

33. Low, J. R. and Rovner, P. Techniques for the Automatic Selection of Data Structures. *Conference Record of the Third ACM Symposium on Principles of Programming Languages* (January 1976).

34. McFarling, S. Program Optimization for Instruction Caches. *Symposium on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (April 3-6, 1989).

35. Paris, J-F. Application of Restructuring Techniques to the Optimization of Program Behavior in Virtual Memory Systems. PhD Dissertation, University of California, Berkeley, PROGRES report 81.5, May 1981.

36. Pettis, K. and Hansen, R. C. Profile Guided Code Positioning. *Proceedings of the ACM-SIGPLAN 1990 Conference on Programming Language Design and Implementation*, *25*, 6 (June 20-22, 1990), 16–27.

37. Ramirez, R. J. Efficient Algorithms for Selecting Efficient Data Storage Structures. Faculty of Mathematics, University of Waterloo, PhD Dissertation, Technical Report CS-80-18, March 1980.

38. Rowe, L. A. A Formilization for Modelling Structures and the Generation of Efficient Implementation Structures. University of California, PhD Dissertation, 1976.

39. Ryder, K. D. Optimizing Program Placement in Virtual Systems. *IBM Systems Journal*, *13* (1974), 292.

40. Samples, A. D. User's Guide to the M5 Macro Language. Technical Report UCB/CSD 91/621, March 1991.

41. Samples, A. D. Mache: No-Loss Trace Compaction. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Berkeley, CA (May 1989).

42. Sarkar, V. Determining Average Program Execution Times and their Variance. *Proceedings of the ACM-SIGPLAN 1989 Conference on Programming Language Design and Implementation*, *24*, 7 (June 21-23, 1989), 298–312.

43. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. and Schonberg, E. *Programming with Sets: An Introduction to SETL*. Springer Verlag, Berlin, 1986.

44. Sherman, M. S. *Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types*. Lecture Notes in Computer Science, 189, Springer Verlag, Berlin, 1985.

45. Smith, A. J. Cache Memories. *ACM Computing Surveys*, *14*, 3 (September 1982), 473–530.

46. Straub, R. M. *Taliere: An Interactive System for Data Structuring SETL Programs*. PhD Dissertation, Courant Institute of Mathematical Sciences, NYU, May 1988.

47. Szymanski, T. B. Assembling Code for Machines with Span Dependent Instructions. *Communications of the ACM*, *21*, 4 (April 1978), 300–308.

48. Thabit, K. O. *Cache management by the compiler*. PhD Dissertation, Rice University, November 1981.

49. Van De Vanter, M. L. Designing BiblioText: An Experiment in User Interface Design. Computer Science Division, EECS, University of California, Berkeley, 88/454, October 24, 1988.

50. Weinberger, P. J. Cheap Dynamic Instruction Counting. *Bell System Technical Journal*, *63* (1984), 1815.

51. Weiss, G. Recursive Data Types in SETL: Automatic Determination, Data Language Description, and Efficient Implementation. Courant Institute of Mathematical Sciences, NYU, Department of Computer Science, Technical Report 102, March 1986.

Appendix A

The Knuth-Stevenson Algorithm

```
//
// A program that implements the MINOPT algorithm.  MINOPT finds the minimal
// set of arcs and nodes required for implementation in a program that allows
// execution counts of all arcs and nodes to be computed.
//   Syntax:
//         ks < graphDescriptionFile
//
// This program also implements Knuth and Stevenson's (K-S) algorithm for
// finding a minimal set of nodes to instrument in a program.
//   Syntax:
//         ks -o < graphDescriptionFile
//
// The differences between the two algorithms are controlled by the boolean
// flag MINOPTing; these areas are highlighted with a right comment:
//                         f                                      // MINOPT
//

#include <stream.h>
#include <stdio.h>
#include <assert.h>
#include "util.H"
#include "Tokens.H"
#include "userTypes.H"
#include "ks_ADTs.H"
#include "IMPLSRCS.H"

boolean MINOPTing = true;                                         // MINOPT

void
error(char *msg)
{
  cerr <<"Error: " << msg << "\n";
}

void
fatal(char *msg)
{
  cerr << "Fatal error: " << msg << "\n";
  abort();
}

Define(`maxNofGraphObjs',1000)@;
Define(`maxNofGraphObjsm1',`Eval(maxNofGraphObjs - 1)')@;
Define(`go_bminfo',`lowerb=0, upperb=maxNofGraphObjsm1,
                 IntToObj=GraphObjNo, ObjToInt=NofGraphObj')@;

//
// class definitions
//

CLASS(Registration_obj)
{
  friend main();
  Any*  reg;
  int   size;
  int   last;
public:
  Registration_obj(int maxsz) { last = -1; size = maxsz;
                                                     reg = new Any[size]; }
  Any&  operator[](int i) {
      if (i < 0 || i >= size) fatal("illegal registration number");
      return reg[i];
      }
  int   next(Any obj, char* str);
}SSALC

CLASS(Flowitem_obj)
{
public:
  Flowitem_obj(GraphObj o, boolean p) { obj = o; plus = p; }
  GraphObj      obj;
  boolean       plus; // false if value is to be subtracted
}SSALC


CLASS(GraphObj_obj)
```

```
{
  friend class Registration_obj;
  friend int NofGraphObj(GraphObj go);
  friend GraphObj GraphObjNo(int i);
  boolean       IsArc; // false if node
  int           rnum;
public:
  static Registration_obj go_R;
  // data
  int           freq;
  double        cost; // the cost of measuring this object; defaults to 1
  Token         name;
  boolean       instrument; // true if obj is to be instrumented
  DECLARE(flowlist,List,Flowitem);
  // structors
  GraphObj_obj(Token t, boolean b );
  // functions
  boolean       isArc() { return IsArc; }
  boolean       isNode() { return !IsArc; }
  int           sum(); // add up the flowitems for this object
  //
  // data and functions specific to the Knuth-Stevenson algorithm
  //    I have maintained the variable names from the Knuth and
  //    Stevenson paper in BIT 13 (1973), pp 323-337.
  void          arcto(GraphObj); // to make a node for each node and arc
                                 // in the original graph
  GraphObj      equivTo; // A graph object that belongs to the
                         // same equivalence class as THIS object.
                         // == nil if THIS object is the
                         // representative of its equivalence class.
  //
  GraphObj      super(); // Returns the unique graph object
                         // that is the representative object
                         // of the equivalence class to which
                         // THIS graph object belongs (a
                         // function that chases the
                         // equivTo chain).
  //
  GraphObj      follow; // creates an super-arc in the reduction graph
                        // from THIS object to the FOLLOW object.
  //
  boolean       d;
  GraphObj      u;
  GraphObj      compfather;
  // These three attributes are used only for vertices in the reduced
  // graph.  We build the spanning tree of connected components in the
  // reduced graph as we go.  There is one arc in the reduced graph
  // for each vertex and arc in the original graph.  If COMPFATHER ==
  // nil then this vertex represents an (super) arc in the reduced
  // graph, viz an arc from u.super() to u.follow().super().  If d,
  // then that arc goes from THIS to COMPFATHER, else the arc goes
  // from COMPFATHER to THIS.
  //
  GraphObj      comp();
  // The representative of this supervertex's component in the reduced
  // program flow graph constructed so far; analogous to super(), above.
  void          makeComponent();
  void          createRedArc();
}SSALC


CLASS(Node_obj) : public GraphObj_obj
{
public:
  // data
  DECLARE(gozoutas, Set, Arc, go_bminfo);
  DECLARE(gozintas, Set, Arc, go_bminfo);
  // functions
  void          gozinta(Arc);
  void          gozouta(Arc);
  int           sumGozintas();
  int           sumGozoutas();
  // structors
  Node_obj(Token t);
}SSALC

CLASS(Arc_obj) : public GraphObj_obj
{
public:
  // data
  Node          from;
  Node          to;
  // structors
  Arc_obj(Token t);
  // functions
  void          goes(Node F, Node T)
    {
    (from=F)->gozouta(this);
    (to=T)->gozinta(this);
    }
```

```
}SSALC

@@ ================ Registration ================

Registration_obj::next(Any obj, char* str)
{
  if (last == size-1) {
      int newsize = (9*size)/8;
      Any* newreg = new Any[newsize];
      int i;
      for (i = 0; i < size; i++) newreg[i] = reg[i];
      delete [size]reg;
      size = newsize;
      reg = newreg;
      cerr << "Warning: registration for " << str << " increased to " << size
        << "\n";
      }
  reg[++last] = obj;
  return last;
}

// ================ Graph objects ================

Registration_obj GraphObj_obj::go_R(maxNofGraphObjs);
int NofGraphObj(GraphObj go)
{
  return (((GraphObj)go)->rnum);
}
GraphObj GraphObjNo(int i)
{
  return (GraphObj_obj::go_R[i]);
}

CONSTRUCTOR(GraphObj_obj, Token t, boolean b)
{
  name = t;
  IsArc = b;
  freq = -99999999;
  cost = 1;
  instrument = false;
  equivTo = nil;
  follow = nil;
  u = nil;
  compfather = nil;
  rnum = go_R.next(this,"GraphObj");
```

```
}

void
GraphObj_obj::arcto(GraphObj tgt)
{
  if (follow == nil) {
      follow = tgt;
      }
  else {
      // make tgt and follow equivalent
      tgt = tgt->super();
      if (tgt != follow->super()) {
          tgt->equivTo = follow->super();
          }
      }
}

GraphObj
GraphObj_obj::super()
{
  return (equivTo == nil? this : equivTo->super());
}

GraphObj
GraphObj_obj::comp()
{
  return (compfather == nil? this : compfather->comp());
}

void
GraphObj_obj::makeComponent()
{
  // transform the d, u, and compfather attributes of the
  // supervertices so that this supervertex is the representative of
  // its reduction graph spanning tree component.
  if (compfather != nil) {
      compfather->makeComponent();
      compfather->compfather = this;
      compfather->d = !d;
      compfather->u = u;
      compfather = nil;
      }
}

void
```

```
GraphObj_obj::createRedArc()
{
  // create the arc in the reduction graph that will correspond to
  // THIS graph object
  GraphObj v, w;
  v = super();
  v->makeComponent();
  w = follow->super();
  // if v and w not in the same tree of the forest of intermediate
  // spanning trees, ...
  if (v != w->comp()) {
      // put in spanning tree
      v->compfather = w;
      v->d = true;
      v->u = this;
      instrument = false;
      }
  else {
      // update the flow items
      instrument = true;
      while( w != v ) {
          Flowitem fi = new Flowitem_obj(this, w->d);
          List_append1(w->u->flowlist, fi);
          w = w->compfather;
          }
      }
}

// ================ Nodes ================

CONSTRUCTOR(Node_obj, Token t) CC GraphObj_obj(t,false)
{
}

void
Node_obj::gozinta(Arc a)
{
  Set_add(gozintas, a);
}

void
Node_obj::gozouta(Arc a)
{
  Set_add(gozoutas, a);
}
```

```

int
Node_obj::sumGozintas()
{
  Arc a;
  int sum = 0;
  forAll(a, gozintas,
         sum += a->freq;
         );
  return sum;
}

int
Node_obj::sumGozoutas()
{
  Arc a;
  int sum = 0;
  forAll(a, gozoutas,
         sum += a->freq;
         );
  return sum;
}

// ================ Arcs ================

CONSTRUCTOR(Arc_obj, Token t) CC GraphObj_obj(t, true)
{
  from = nil;
  to = nil;
}

//
// ================ the program ================
//

DECLARE(Graph,Set,GraphObj, go_bminfo);
DECLARE(dict,Map,Token,GraphObj);

Node
readNode()
{
  Node n;
  Token t;
  cin >> t;
  if (Map_in(dict,t)) {
      Map_value(dict, t, n);
      }
  else {
      n = new Node_obj(t);
      Map_define(dict, t, n);
      }
  return n;
}

boolean
readArc()
{
  // input is a set of lines of the form:
  // arcname freq cost nodename nodename ,
  //
  Token t;
  double k;
  cin >> t;
  if (cin.eof()) return;
  assert(!Map_in(dict,t));
  Arc a = new Arc_obj(t);
  cin >> a->freq;
  cin >> a->cost;
  a->cost *= a->freq;
  Map_define(dict, t, a);
  // now read the from and to nodes
  Node F = readNode();
  Node T = readNode();
  Set_add(Graph,a);
  Set_add(Graph,F);
  Set_add(Graph,T);
  a->goes(F,T);
  if (MINOPTing) {
      // now do the equivalent of K-S arcto function, except we do it for
      // the nodes and the arcs
      F->arcto(a);                                                  // MINOPT
      a->arcto(T);
      }
  else {
      F->arcto(T);
      }
  // done
  cin >> t;
  if (t == commaToken) return true;
  return false;
```

```
}

void
readGraph()
{
  while (readArc());
}

void
propagateCounts()
{
  GraphObj o;
  forAll(o, Graph,
         if (o->isArc()) {
             assert(o->freq >= 0);
             }
         else {
             Node_obj& n = *((Node)o);
             int t = n.sumGozintas();
             assert(n.freq < 0 || n.freq == t);
             n.freq = t;
             n.cost = (double)n.freq;
             assert(n.freq == n.sumGozoutas());
             }
         );
}

int
gobjCmp(GraphObj f1, GraphObj f2)
{
  if (f1->cost > f2->cost) return -1;
  if (f1->cost < f2->cost) return 1;
  return 0;
}

main(int argc, char **argv)
{
  GraphObj go;
  double sum = 0;
  DECLARE(sortedObjList, List, GraphObj);
  if (argc > 1) {
      if (strcmp(argv[1], "-o") == 0) MINOPTing = false;
      else {
          fatal("Unrecognized option");
          }
      }
  readGraph();
  propagateCounts();
  Set_sort2(Graph, sortedObjList, gobjCmp);
  forAll(go, sortedObjList,
         if (MINOPTing || go->isNode()) {                        // MINOPT
             go->createRedArc();
             }
         );
  forAll(go, sortedObjList,
         if (!MINOPTing && go->isArc()) continue;                // MINOPT
         cout << go->name;
         if (go->instrument) {
             cout << ": instrument cost=" << go->cost << "\n";
             sum += go->cost;
             }
         else {
             // print equation that computes count for this object
             cout << "= ";
             Flowitem item;
             forAll(item, go->flowlist,
                    if (item->plus) cout << "+";
                    else cout << "-";
                    cout << item->obj->name;
                    );
             cout << "\n";
             }
         );
  cout << "Instrumentation cost = " << sum << "\n";
}
```

Appendix B

Therblig

```
@ FILE: types.t
@ terminology:
@   The class that is a specific implementation of an ADT is called
@   a "representation class" of that ADT, since an ADT is not just a
@   set of functions but also a representation of the data.
@
@   The ADT is defined as a set of functions operating on objects of that
@   type.  A representation class will have zero, one, or more
@   "implementations" of those interface functions.
@
#include <stream.h>
#include <stdio.h>
#include "userTypes.H"
#include "Tokens.H"
#include "IMPLSRCS.H"
#include "option_types.H"

Define(‘maxNofVarDecls’,1000)@;
Define(‘maxNofVarDeclsm1’,‘Eval(maxNofVarDecls - 1)’)@;
Define(‘vd_bminfo’,‘lowerb=0, upperb=maxNofVarDeclsm1,
                 IntToObj=VarDeclNo, ObjToInt=NofVarDecl’)@;

Define(‘maxNofADTabsFcn’,200)@;
Define(‘maxNofADTabsFcnm1’,‘Eval(maxNofADTabsFcn-1)’)@;
Define(‘adtaf_bminfo’,‘lowerb=0, upperb=maxNofADTabsFcnm1,
                 IntToObj=ADTabsFcnNo, ObjToInt=NofADTabsFcn’)@;

Define(‘maxNofADTimpFcn’,200)@;
Define(‘maxNofADTimpFcnm1’,‘Eval(maxNofADTimpFcn-1)’)@;
Define(‘afd_bminfo’,‘lowerb=0, upperb=maxNofADTimpFcnm1,
                 IntToObj=ADTimpFcnNo, ObjToInt=NofADTimpFcn’)@;
```

```

Define(`maxNofADTcallSite',225)@;
Define(`maxNofADTcallSitem1',`Eval(maxNofADTcallSite-1)')@;
Define(`acs_bminfo',`lowerb=0, upperb=maxNofADTcallSitem1,
                 IntToObj=ADTcallSiteNo, ObjToInt=NofADTcallSite')@;

CLASS(Registration_obj)
{
  friend main();
  Any*  reg;
  int   size;
  int   last;
public:
  Registration_obj(int maxsz) { last = -1; size = maxsz;
                                                  reg = new Any[size]; }
  Any&  operator[](int i) {
      if (i < 0 || i >= size) fatal("illegal registration number");
      return reg[i];
      }
  int   next(Any obj, char* str);
}SSALC


CLASS(Profarray_obj)
{
  Token         filename; //@N the filename for this array
  long          size;     //@N the size of this array
  long          *iarray;  //@N the array itself
  double        *array;   //@N avg'd over runs
  boolean       Valid;    //@N did the read work?
public:
  Profarray_obj(FILE *pf, Token fn, int sz);
  ~Profarray_obj() { delete [size+1]array; }
  double& operator[](int i) { return array[i]; }
  boolean valid() { return Valid; }
  Token   file() { return filename; }
}SSALC

#define maxNofOptsPerADT 10

CLASS(Optional_obj)
{ @ optionals as they are read in from the input file
  @ this class is actually overloaded.  When the ADT is being defined,
  @ a list of these is created for each optional that is possible on a
  @ variable declaration.  The ival is then the index of that optional for
  @ that ADT.
  @ When the optionals are read on an actual variable declaration,
  @ then the fields are filled in as documented below.
public:
  int   idx;    @ the index of the optional
  optionType *table; @ the table it is an index into
  Token sval;   @ the value represented as a token
  int   ival;   @ the value as an integer, if ivalid
  bool  ivalid; @ is the value a valid integer?
  @ fcns
  Optional_obj(Token val, optType intp, int idx, optionType *tbl);
  Optional_obj(int value);
  Optional_obj();
  ostream& print(ostream&);
  ostream& printForm(ostream&);
}SSALC

CLASS(VarDecl_obj)
{
  void             init_VarDecl(void);
public:
  static Registration_obj vd_R;
  Token         name;           @ my name;
  ADType        vd_ADT;         @ my abstract type;
  boolean       implemented;    @ means that this variable has been
                                @ assigned a representation (i.e.
                                @ vd_repr contains a valid impln type)
  ADTRepr       vd_repr;        @ my representation type; used by assign
  ADTRepr       vd_bestRepr;
  DECLARE(vd_adtParms, List, Token);    @@ parms in my DECLARE
                                        @ (including optionals?)
  DECLARE(vd_inSigsOf, Set, ADTcallSite, acs_bminfo);
  AliasSet      vd_as;
  int           rnum;
  @@
  @@ OPTIONALS
  @@
  boolean       optsParsed;
  Optional      vd_opts[maxNofOptsPerADT]; // not nil if present
  String        instance_name, instance_parm;
  String        coercion_name, coercion_parm;
  String        constructor_parms;
  @ fcns
  VarDecl_obj(Token, ADType, ADTRepr);  @ #1
  VarDecl_obj(Token, ADType, Token);    @ #1
  VarDecl_obj(Token, ADType);           @ #2
  VarDecl_obj(Token, Token); @ check that the 2nd is an ADType name
  void aliasOf(VarDecl);
  void  betterRepr(){assert(implemented); vd_bestRepr = vd_repr; }
  @ Note that the difference between #1 and #2 is that #1 assigns a
  @ representation, while #2 does not.
  boolean operator==(VarDecl_obj &that);
  boolean operator!=(VarDecl_obj &that) { return !(*this == that); }
  ostream& print(ostream&);
  ostream& printForm(ostream&);
}SSALC

CLASS(Signature_obj)
{
public:
  DECLARE(sig_sig, List, VarDecl);
  @ fcns
  Signature_obj();
  void add(VarDecl V);
  void add(Token, Token, Token); @create the VarDecl yourself
  void add(VarDecl, Token, Token);
  void add(Token, ADType, Token); @ ditto
  void add(Token, ADType, ADTRepr);
  void add_dontCare();
  int  len(void);
  ostream& print(ostream&);
  ostream& printForm(ostream&);
}SSALC

CLASS(ADType_obj)
{ @ An AbstractDataType; defined by a set of AbstractFunctionDefinitions.
public:
  bool  adt_inited;
  int   adt_number;
  Token name;
  ADTRepr adt_profileImpl;                     @ my profiling representation
  DECLARE(adt_reprs, Map, Token, ADTRepr);      @ my representations
  DECLARE(adt_afcns, Set, ADTabsFcn, adtaf_bminfo);
                                       @ abstract fcns defining my interface
  @@DECLARE(adt_optionals, Map, Token, Optional);       @
  @ fcns
  ADType_obj(Token);
  ADType_obj(char *);
  ostream& print(ostream&);
  ostream& dump(ostream&);
  boolean operator==(ADType_obj r) { return (name == r.name); }
  boolean operator!=(ADType_obj r) { return (name == r.name); }
}SSALC

CLASS(ADTRepr_obj)
{ @ the representation of an AbstractDataType;
  @ an ADT implementation consists of a set of sets of implementations of the
  @ ADTs functions.
  @ Note: name == adtr_of->name + '_' + adtr_suffix
public:
  bool          adtr_inited;
  Token         name;           @ My name (Set_1, List_3, Map_2, etc.)
  ADType        adtr_of;        @ the ADT I'm a representation of
  Token         adtr_suffix;
  int           adtr_number;    @ for accessing tables
  @ fcns
  ADTRepr_obj(Token);
  ADTRepr_obj(Token, ADType);
  ostream& print(ostream&);
  void  printName(ostream&);
  boolean operator==(ADTRepr_obj r) { return (name == r.name); }
}SSALC

CLASS(ADTabsFcn_obj)
{ @ abstract function (interface function)
public:
  static Registration_obj adtaf_R;
  int           adtaf_uid;      @ for registration
  Token         name;           @ the name by which the user invokes me
  ADType        adtaf_for;      @ the ADT I'm in the interface of
  Signature_obj adtaf_sig;      @ my parameters
  ADTimpFcn     evalFcn;        @ the function representing the profiling
                                @ implemention evaluation function.
  DECLARE(adtaf_impl_fcns, Set, ADTimpFcn, afd_bminfo); @ my implementations
  int           nofProfVars;    @ number of profiling variables for this fcn
  @ functions
  ADTabsFcn_obj(Token, ADType);
  ostream& print(ostream&);
  void printName(ostream&);
  boolean operator==(ADTabsFcn_obj r) { return (name == r.name); }
}SSALC(adtaf_sig())

CLASS(ADTimpFcn_obj)
{ @ An implementation of an abstract (interface) function
public:
  static Registration_obj afd_R;
  int           afd_uid;        @ index into evalFcns
  Token         name;           @ the name by which I am ref'd in the library
  ADTabsFcn     afd_impl_of;    @ the abstract function I implement
  ADTRepr       repr;           @ the representation I'm a member of
  Signature_obj afd_sig;        @ my parameters;
  @ only types, not names, are important in my signature; should
  @ probably be a derived class of Signature_obj
  @ functions
  ADTimpFcn_obj(Token, ADTabsFcn, ADTRepr);
  ostream& print(ostream&);
  void  printName(ostream&);
  void typedName(ostream& o) { o << repr->name << "::" << name; }
  boolean operator==(ADTimpFcn_obj r)
    { return (name == r.name); }
}SSALC(afd_sig())


CLASS(ADTcallSite_obj)
{ @ A call site where the user's program has invoked one of the interface
  @ functions for an ADT.
  boolean implemented;
  ADTimpFcn     implementation;
  ADTimpFcn     BetterImpl;
public:
  static Registration_obj acs_R;
  int           acs_ruid; @ for registration
  int           acs_upid; @ each call site has a unique id which is its
  @ base index in the profile eval-function arrays.
  Profarray     acs_parr; @ the profile array for this call site
  ADTabsFcn     acs_afcn; @ the abstract function being invoked
  Signature_obj acs_sig;  @ my actual parameters; names given, types computed
  int           acs_line; @ the line number of the file I'm in
  double        acs_rank;
  @ constructors
  ADTcallSite_obj(int, Profarray, int, ADTabsFcn);
  @ functions
  double eval(ADTimpFcn);
  double eval() { assert(implemented); return eval(implementation); }
  ostream& print(ostream&);
  ostream& printForm(ostream&);
  void printName(ostream& fout) { fout << acs_line; }
  void implement(ADTimpFcn f) { implemented = true; implementation = f; }
  void unimplement() { implemented = false; implementation = nil; }
  void betterImpl() { assert(implemented && implementation!=nil);
```

```
                      BetterImpl=implementation; }
}SSALC(acs_sig())


CLASS(UserFcnDecl_obj)
{
  @ We have to know the structure of some user functions' signatures;
  @ user fcns can cause either conversion functions to be invoked
  @ or force certain bindings (e.g. if this global is assigned this
  @ representation, then this formal parm must also have it).  Depends
  @ on whether the analyzer has been implemented with conversion-on-calls
  @ implemented.
public:
  int           ufd_upid; @ each user function has a unique id
  Token         name; @ the name of the user function
  Signature_obj ufd_sig; @ my parameters; abstract given, repr computed.
  @ fcns
  UserFcnDecl_obj(int, Token );
  ostream& print(ostream&);
}SSALC(ufd_sig())


CLASS(UserFcnCall_obj)
{
public:
  int           ufc_upid; @ each call site of a user function has a unique id
  UserFcnDecl   ufc_decl; @ the function being called.
  Signature_obj ufc_sig; @ my actual parameters
  @ fcns
  UserFcnCall_obj(int, UserFcnDecl);
  ostream& print(ostream&);
}SSALC(ufc_sig())


CLASS(AliasSet_obj)
{
friend class VarDecl_obj;
public:
  DECLARE(as_set, Set, VarDecl, vd_bminfo);
  @ fcns
  AliasSet_obj(VarDecl);
  void merge(AliasSet);
}SSALC()


```

300

```
@@ FILE: main.t
#include <stream.h>
#include <stdio.h>
#include <assert.h>
#include "util.H"
#include <math.h>

#ifdef DBG_MALLOC
extern void malloc_verify();
#define MALLOCK do { if (DebugMalloc) malloc_verify(); } while (0)
#else
#define MALLOCK
#endif

@@
@@ read the input file that has all the information we need:
@@ (we might note here that EVERY name in this file MUST be unique)
@@
@@ (1) a list of all ADTs available for analysis, and names of
                                                          implementations:
@@      (1.1) a list of the interface functions and abstract parameter
                                                          types;
@@      (1.2) a list of the implementations of the interface functions and
@@            the parameters' implementation types.
@@ (The Therblig system puts these in file <adt>.th in the <adt> directory
@@  impls/<adt>.  The user has a conventional way of creating the
@@  appropriate declaration file, named, ADTs.th, for his program.)
@@ Syntax:
@@   The ADT and its implementations on one line, followed by several
@@   lines of declarations of the abstract functions, followed by the
@@   lines describing the implementations of the abstract functions. I.e:
@@
@@   <ADT> <ADT_i> ... ;
@@   = <AbsFcnName> <signature> ,
@@   :
@@   <AbsFcnName> <ImpFcnName> <signature> ,
@@   :
@@      ;
@@   : another block like the above
@@      ;
@@   .
@@
@@   I'll use the equal sign as a flag that this is an abstract function
@@  declaration, and so I won't have to worry about the order of the
@@  declarations if I decide to change it later.  The abstract defns
@@  will come from the ADT_P.H file, and the implementation defn's will
@@  come from the ADT_i.H files.  They end up in their respective ADT_i.th
@@  files, which are included into one file ADTs.th in the user's
@@  directory by the user's makefile.
@@ E.g.:
@@
@@ Set Set_1 Set_2 ... ,
@@  = union1 Set Set ,
@@  = union2 Set Set Set ,
@@  = add Set ? ,
@@   :
@@ union1 union1_1_1 Set_1 Set_1 ,
@@ union1 union1_1_2 Set_1 Set_2 ,
@@   :
@@ union2 union2_1 Set_1 Set_1 Set_1 ,
@@ union2 union2_2 Set_2 Set_2 Set_2 ,
@@   :
@@  ;
@@ List List_1 List_2 ... ,
@@   :
@@  ;
@@ .
@@
@@
@@ (2) all variable declarations (Therblig puts them in ADT_vars.th).
@@ Syntax: var-name ADT-name p1 p2 p3 ... ;
@@ A Set 10 int ... ;
@@  :
@@ .
@@
@@ (3) all user function declarations of interest
@@      (Therblig puts them in ADT_ufcns.th)
@@ Syntax: fid user-fcn-name p1-name p1-type p2-name p2-type ... ;
@@ 145 userFcn A Set B ? C List ... ;
@@  :
@@ .
@@
@@ (4) all ADT function call sites (Therblig puts them in ADT_csites)
@@ Syntax: upid  ADT-fcn-name  var1 var2 ... ;
@@ 1234 union1 A  B ;
@@  :
@@ .
@@
@@ (5) all user function call sites (Therblig puts them in ADT_ucsites)
@@ Syntax: upid user-fcn-name var1 var2 ... '
@@ 134 userFcn A ? C ... ;
@@  :
@@ .
@@
@@
@@ (1) must be written by the compiler when doing the implementations.
@@
@@ (2)-(5) must be written by the compiler when doing the user's program.
@@
@@


#include "userTypes.H"
#include "main_ADTs.H"
Include(types.t)
@@ also defines ADTinfoTable
#include "OptArrays.H"

#include "EvalFcns.H"

#define openFile(fn,io,mode,filename,die)              \
  name2(io,stream) fn(filename,mode);                  \
  if (die && fn.fail()) fatal("Could not open file");

#define fopenFile(fn,mode,filename,die)                        \
  FILE *fn = fopen(filename, mode);                            \
  if (die && fn == 0) fatal("Could not fopen file");


@@COERCN_CLASSES


@@ ===== GLOBALS =====
@@Debug(Std)
@@DebugStack
@@DebugPools
DECLARE(ADTs,      Map, Token, ADType); // abstract data types
@@Debug(Off)
DECLARE(ADTReprs,  Map, Token, ADTRepr);        // their representations
DECLARE(ADTafcns,  Map, Token, ADTabsFcn);      // abstract functions
@@ DECLARE(ADTifcns,  Map, Token, ADTimpFcn);   // their implementations
DECLARE(ADTcalls,  Set, ADTcallSite, acs_bminfo);// their call sites
DECLARE(Vars,      Map, Token, VarDecl);        // variables in the program
DECLARE(UserFcns,  Map, Token, UserFcnDecl);    // user declared functions
```

```
@@ DECLARE(UserCalls, List, UserFcnCall);       // user call sites
DECLARE(ProfArrays,Map, Token, Profarray);      // the profile data

bool    profileDataValid;
double  curAssignCost;
int     cutOffIndex;
int     cutOffPercent;

ADType dontCareADT;
ADTRepr dontCareRepr;

// The following defines are due to therblig's minimal parsing ability
#define VarDecl_obj_print       (void *)VarDecl_obj::print
#define VarDecl_obj_printForm   (void *)VarDecl_obj::printForm
#define Token_obj_print         (void *)Token_obj::print
#define ADTRepr_obj_printName   (void *)ADTRepr_obj::printName
#define ADTabsFcn_obj_print     (void *)ADTabsFcn_obj::print
#define ADTabsFcn_obj_printName (void *)ADTabsFcn_obj::printName
#define ADTimpFcn_obj_print     (void *)ADTimpFcn_obj::print
#define ADTimpFcn_obj_printName (void *)ADTimpFcn_obj::printName
#define c_print                 (void *)c->print
#define ADTcallSite_obj_print   (void *)ADTcallSite_obj::print
#define ADTcallSite_obj_printName       (void *)ADTcallSite_obj::printName


@@ THE CLASS ROUTINES FOR THERBLIG

@@ ===== print routines ====

#define DefPrinter(type) \
  ostream& operator<<(ostream& fout, type v) { return v->print(fout); }
#define DefPrintForm(type) \
  ostream& operator<<(ostream& fout, type v) { return v->printForm(fout); }
DefPrinter(Optional)
DefPrinter(VarDecl)
ostream& operator<<(ostream& fout, Signature_obj &v)
                        { return v.printForm(fout); }
DefPrinter(ADType)
DefPrinter(ADTRepr)
DefPrinter(ADTabsFcn)
DefPrinter(ADTimpFcn)
DefPrintForm(ADTcallSite)
DefPrinter(UserFcnDecl)
DefPrinter(UserFcnCall)

```

```
@@ ================ Registration ================

Registration_obj::next(Any obj, char* str)
{
  if (last == size-1) {
      int newsize = (9*size)/8;
      Any* newreg = new Any[newsize];
      int i;
      for (i = 0; i < size; i++) newreg[i] = reg[i];
      delete [size]reg;
      size = newsize;
      reg = newreg;
      cerr << "Warning: registration for " << str <<
        " increased to " << size << "\n";
      }
  reg[++last] = obj;
  return last;
}


@@ ===== Profarray_obj =====

CONSTRUCTOR(Profarray_obj, FILE *pfile, Token fn, int sz)
{
  @@ New requirement: the last entry of each profile array is the number
  @@ of times that profile array was written to.  When we read the array in
  @@ it is converted from long to double, with each entry divided by the
  @@ execution count.
  Valid = false;
  filename = fn;
  size = sz;
  iarray = new long[sz+1];
  array = new double[sz+1];
  if (fread(iarray, sizeof(long), size+1, pfile) != size+1) {
      cerr << "Profile data array " << fn << " corruption? Not valid.\n";
      cerr << "   size+1=" << size+1 << "?\n";
      fclose(pfile);
      return;
      }
  double nofExecutions = (double)iarray[sz];
  assert(nofExecutions > 0);
  for (int i = 0; i < sz; i++) {
      array[i] = iarray[i] / nofExecutions;
      }
  array[sz] = iarray[sz];
  delete [sz+1]iarray;
  Valid = true;
}

@@ ===== Optional_obj =====

CONSTRUCTOR(Optional_obj,Token val,optType type,int index,optionType *tbl)
{
  sval = val;
  idx = index;
  table = tbl;
  if (type == int_opt_type) {
      assert(val != nil);
      if (val->type == num_tkn) {
          ivalid = true;
          ival = val->val;
          }
      else {
          cerr << "Warning: " << val << " is not an integer\n";
          ivalid = false;
          }
      }
  else if (type == tmpint_opt_type) {
      cerr << "Warning: " << val << " is not a valid optional\n";
      }
}

CONSTRUCTOR(Optional_obj) // used by feasibility/eval routines
{
  ivalid = false;
  sval = nil;
  idx = 0x4FFFFFFF; // something to cause a problem if used
  table = nil;
}

CONSTRUCTOR(Optional_obj, int value) // used by feasibility/eval routines
{
  ivalid = true;
  ival = value;
  sval = nil;
  idx = 0x4FFFFFFF; // something to cause a problem if used
  table = nil;
}

ostream &
```

```
Optional_obj::print(ostream& fout)
{
  return fout << table[idx].name;
}

ostream &
Optional_obj::printForm(ostream& fout)
{
  return fout << table[idx].name;
}

@@ ===== VarDecl_obj =====

Registration_obj VarDecl_obj::vd_R(maxNofVarDecls);
int NofVarDecl(VarDecl vd)
{
  return (((VarDecl)vd)->rnum);
}
VarDecl VarDeclNo(int i)
{
  return (VarDecl_obj::vd_R[i]);
}

void
VarDecl_obj::init_VarDecl()
{
  for (int i = 0; i < maxNofOptsPerADT; i++) {
      vd_opts[i] = nil;
      }
  vd_bestRepr = nil;
  rnum = vd_R.next(this,"VarDecl");
  vd_as = new AliasSet_obj(this);
}

CONSTRUCTOR(VarDecl_obj, Token t, ADType a, ADTRepr r)
{
  name = t;
  vd_ADT = a;
  vd_repr = r; implemented = true;
  init_VarDecl();
}

CONSTRUCTOR(VarDecl_obj, Token t, ADType a)
{
  name = t;
  vd_ADT = a;
  vd_repr = nil; implemented = false;
  init_VarDecl();
}

CONSTRUCTOR(VarDecl_obj, Token t, ADType a, Token tr)
{
  ADTRepr r = new ADTRepr_obj(tr,a);
  if (tr != dontCareToken && !Map_in(a->adt_reprs, r->name)) {
      fatal("VarDecl passed Token not in ADType's repr List");
      }
  name = t;
  vd_ADT = a;
  vd_repr = r; implemented = true;
  init_VarDecl();
}

CONSTRUCTOR(VarDecl_obj, Token t, Token ta)
{
  ADType a;
  name = t;
  Map_value(ADTs, ta, a); // ADTs->value(ta,a);
  if (a == nil) {
      a = new ADType_obj(ta);
      Map_define(ADTs, ta, a);
      }
  vd_ADT = a;
  vd_repr = nil; implemented = false;
  init_VarDecl();
}

boolean
VarDecl_obj::operator==(VarDecl_obj &that)
{
  if (name != that.name) return false;
  if (vd_ADT != that.vd_ADT) return false;
  if (!List_equal(vd_adtParms, that.vd_adtParms)) return false;
  return true;
}

ostream&
VarDecl_obj::print(ostream& fout)
{
  fout << name << "(" << vd_ADT->name;
  if (implemented) {
      if (vd_repr != nil) fout << "/" << vd_repr->name;
      else fout << "/<nil>";
      }
  if (vd_bestRepr != nil) fout << "/" << vd_bestRepr->name;
  return fout << ")";
}

ostream&
VarDecl_obj::printForm(ostream& fout)
{
  if (implemented) {
      if (vd_repr != nil) fout << vd_repr->name;
      else fout << "<nil>";
      }
  if (vd_bestRepr != nil) fout << "/" << vd_bestRepr->name;
  else fout << "(" << vd_ADT->name << ")";
  return fout << " " << name;
}

void
VarDecl_obj::aliasOf(VarDecl v)
{
  AliasSet AS = v->vd_as;
  VarDecl var;
  if (AS == vd_as) {
      // then this is a redundant call: they are both pointing to the same
      // alias set.
      return;
      }
  forAll(var, v->vd_as->as_set,
         if (var != v) var->vd_as = vd_as; // because iterCheck gets upset
         );
  v->vd_as = vd_as;
  vd_as->merge(AS); // also frees up set AS
}

@@ ===== Signature_obj  =====

CONSTRUCTOR(Signature_obj)
{
  // do nothing special
}

int
Signature_obj::len()
```

```
{
  return List_length(sig_sig);
}


void
Signature_obj::add(VarDecl v)
{
  List_append1(sig_sig, v);
}

void
Signature_obj::add_dontCare()
{
  VarDecl v = new VarDecl_obj(dontCareToken, dontCareADT, dontCareToken);
  List_append1(sig_sig, v);
}

void
Signature_obj::add(Token vn, Token an, Token rn)
{
  ADType a;
  Map_value(ADTs, an, a);
  if (a == nil) {
      fatal("Signature passed non_adt Token");
      }
  VarDecl v = new VarDecl_obj(vn, a, rn);
  List_append1(sig_sig, v);
}

void
Signature_obj::add(VarDecl v, Token an, Token rn)
{
  ADType a;
  Map_value(ADTs, an, a);
  if (a == nil) {
      fatal("Signature passed non_adt Token");
      }
  List_append1(sig_sig, v);
}

void
Signature_obj::add(Token vn, ADType a, Token rn)
{
  VarDecl v = new VarDecl_obj(vn, a, rn);
  List_append1(sig_sig, v);
}

void
Signature_obj::add(Token vn, ADType a, ADTRepr r)
{
  VarDecl v = new VarDecl_obj(vn, a, r);
  List_append1(sig_sig, v);
}

ostream&
Signature_obj::print(ostream& fout)
{
  fout << "{Sig: ";
  List_print(sig_sig, fout, VarDecl_obj_printForm);
  return fout << "}";
}

ostream&
Signature_obj::printForm(ostream& fout)
{
  fout << "(";
  List_print(sig_sig, fout, VarDecl_obj_printForm);
  return fout << ")";
}


@@ ===== ADType_obj =====

int
ADTinfoTableLookup(Token t)
{
  for (int i=0; i < nofADTs; i++) {
      if (*t == ADTinfoTable[i].name) return i;
      }
  cerr << t << ": ";
  fatal("Unknown ADT in ADTinfoTableLookup");
}

CONSTRUCTOR(ADType_obj, Token t)
{
  name = t;
  adt_inited = false;
  adt_number = ADTinfoTableLookup(name);
}
```

```

CONSTRUCTOR(ADType_obj, char *sp)
{
  name = new Token_obj(sp, id_tkn, 0);
  adt_inited = false;
  adt_number = ADTinfoTableLookup(name);
}

ADType
isADType(Token t)
{
  ADType adt;
  Map_value(ADTs, t, adt);
  if (adt == nil) {
      adt = new ADType_obj(t);
      Map_define(ADTs, t, adt);
      }
  return adt;
}


ostream&
ADType_obj::dump(ostream& fout)
{
  fout << "{ADType: " << BOOL(adt_inited)
    << " " << name << "\nreprs = ";
  Map_print(adt_reprs, fout,
              Token_obj_print,
              ADTRepr_obj_printName);
  fout << "\nabs. fcns. = ";
  Set_print(adt_afcns, fout, ADTabsFcn_obj_printName);
  return fout << "}";
}

ostream&
ADType_obj::print(ostream& fout)
{
  return fout << name;
  return fout << "}";
}

@@ ===== ADTRepr_obj =====

CONSTRUCTOR(ADTRepr_obj, Token t, ADType a)
{
  name = t;
  adtr_of = a;
  adtr_inited = false;
  adtr_suffix = t->suffix();
}

CONSTRUCTOR(ADTRepr_obj, Token t)
{
  name = t;
  adtr_of = dontCareADT;
  adtr_inited = false;
  adtr_suffix = t->suffix();
}

// overload isADTRepr;

ADTRepr
isADTRepr(Token t)
{
  ADTRepr adtr;
  Map_value(ADTReprs, t, adtr);
  if (adtr == nil) {
      adtr = new ADTRepr_obj(t);
      Map_define(ADTReprs, t, adtr);
      }
  return adtr;
}

ADTRepr
isADTRepr(Token t, ADType a)
{
  ADTRepr adtr;
  Map_value(ADTReprs, t, adtr);
  if (adtr == nil) {
      adtr = new ADTRepr_obj(t,a);
      Map_define(ADTReprs, t, adtr);
      }
  else {
      if (*adtr->adtr_of != *a) {
          fatal("error in isADTRepr(t,a)");
          }
      }
  return adtr;
}

```

```

void
ADTRepr_obj::printName(ostream& fout)
{
  fout << name;
}

ostream&
ADTRepr_obj::print(ostream& fout)
{
  fout << "{ADTRepr_" << adtr_suffix << ": "
    << BOOL(adtr_inited) << " " << name << "(" << adtr_of << ")" ;
  return fout << "}";
}

@@ ===== ADTabsFcn_obj

Registration_obj ADTabsFcn_obj::adtaf_R(maxNofADTabsFcn);
int NofADTabsFcn(ADTabsFcn af)
{
  return (((ADTabsFcn)af)->adtaf_uid);
}
ADTabsFcn ADTabsFcnNo(int i)
{
  return (ADTabsFcn_obj::adtaf_R[i]);
}

CONSTRUCTOR(ADTabsFcn_obj, Token t, ADType a)
{
  name = t;
  adtaf_for = a;
  adtaf_uid = adtaf_R.next(this,"ADTabsFcn");
}

ostream&
ADTabsFcn_obj::print(ostream& fout)
{
  fout << "{ADTabsFcn(" << name << "(" << adtaf_for << ")" << adtaf_sig
     << ")\nimpls:";
  Set_print(adtaf_impl_fcns, fout, ADTimpFcn_obj_printName);
  return fout << "}";
}

void
ADTabsFcn_obj::printName(ostream& fout)
```

```
{
  fout << name;
}

@@ ===== ADTimpFcn_obj =====

Registration_obj ADTimpFcn_obj::afd_R(maxNofADTimpFcn);
int NofADTimpFcn(ADTimpFcn aif)
{
  return (((ADTimpFcn)aif)->afd_uid);
}
ADTimpFcn ADTimpFcnNo(int i)
{
  return (ADTimpFcn_obj::afd_R[i]);
}

CONSTRUCTOR(ADTimpFcn_obj, Token t, ADTabsFcn af, ADTRepr r)
{
  name = t;
  afd_impl_of = af;
  repr = r;
  afd_uid = afd_R.next(this,"ADTimpFcn");
}

void
ADTimpFcn_obj::printName(ostream& fout)
{
  fout << name;
}

ostream&
ADTimpFcn_obj::print(ostream& fout)
{
  fout << "{ADTimpFcn:" << name << "(" << afd_impl_of << ")"
    << afd_sig;
  return fout << "}";
}

@@ ===== ADTcallSite_obj =====

@@ a presumably invalid value
#define IllegalRank -999999.0

Registration_obj ADTcallSite_obj::acs_R(maxNofADTcallSite);
int NofADTcallSite(ADTcallSite acs)
```

```
{
  return (((ADTcallSite)acs)->acs_ruid);
}
ADTcallSite ADTcallSiteNo(int i)
{
  return (ADTcallSite_obj::acs_R[i]);
}

CONSTRUCTOR(ADTcallSite_obj,int lno,Profarray array,int upid,ADTabsFcn af)
{
  acs_upid = upid;
  acs_afcn = af;
  acs_line = lno;
  acs_parr = array;
  implemented = false;
  acs_ruid = acs_R.next(this,"ADTcallSite");
  acs_rank = IllegalRank;
}

double
ADTcallSite_obj::eval(ADTimpFcn f)
{
  return (*evalFcns[f->afd_uid])(this);
}

ostream&
ADTcallSite_obj::print(ostream& fout)
{
  fout << "{ADTcallSite(" << acs_upid << ")" << acs_afcn
    << acs_sig;
  return fout << "}";
}

ostream&
ADTcallSite_obj::printForm(ostream& fout)
{
  fout << acs_afcn->name << " (line " << acs_line << " file "
    << acs_parr->file() << " p[" << acs_upid << "]=(";
  for (int i = 0; i < acs_afcn->nofProfVars; i++) {
      if (i > 0) fout << ",";
      fout << (*acs_parr)[acs_upid + i];
      }
  fout << ") ";
  if (implemented) {
      fout<<implementation->repr->name<<" cost "<<eval(implementation);
      }
  else if (BetterImpl != nil) {
      fout << BetterImpl->repr->name << " cost " << eval(BetterImpl);
      }
  else {
      fout << "profiling costs " << eval(acs_afcn->evalFcn);
      }
  fout << ")";
  return fout;
}

@@ ===== UserFcnDecl_obj =====

CONSTRUCTOR(UserFcnDecl_obj, int i, Token t)
{
  ufd_upid = i;
  name = t;
}

ostream&
UserFcnDecl_obj::print(ostream& fout)
{
  fout << "{UserFcnDecl(" << ufd_upid << ") " << name << ufd_sig;
  return fout << "}";
}

@@ ===== UserFcnCall_obj =====

CONSTRUCTOR(UserFcnCall_obj, int i, UserFcnDecl uf)
{
  ufc_upid = i;
  ufc_decl = uf;
}

ostream&
UserFcnCall_obj::print(ostream& fout)
{
  fout << "{UserFcnCall(" << ufc_upid << ") " << ufc_decl << ufc_sig;
  return fout << "}";
}

@@ ===== AliasSet_obj =====

CONSTRUCTOR(AliasSet_obj, VarDecl v)
{
  Set_add(as_set, v);
}

void
AliasSet_obj::merge(AliasSet a)
{
  Set_union1(as_set,a->as_set);
  delete a;
}

@@ ===== here is the beginning of the input routines for therblig =====

void
readADTs(char *finame)
{
  Token t;
  int adtrnumber = 0;
  istream fin(finame,"r");
  if (fin.fail()) fatal("Could not open file in readADTs");
  @@ read the names of the abstract data types (ADTs)
  fin >> t;
  while (t != dotToken) {
      @@ - <adt> <adt_i> ... ,
      assert(t == minusToken);
      fin >> t;
      ADType adt = isADType(t);
      if (adt->adt_inited) {
          fatal("Duplicate ADT types declared");
          }
      @@ read the names of the implementations of this adt
      @@ the profiling implemenation is the ADT name appended with '_P'
      @@ and is created automatically.
      @@ It is not added to the dictionary for this type, since it never
      @@ enters into the assignment computation.  But it must exist for
      @@ printAssignments procedure to work generally.
      adt->adt_profileImpl = isADTRepr(t->append("_P"));
      adt->adt_profileImpl->adtr_inited = true;
      adt->adt_profileImpl->adtr_number = adtrnumber++;
      fin >> t;
      while (t!=commaToken) {
          ADTRepr adti = isADTRepr(t);
          if (adti->adtr_inited) {
              fatal("Duplicate ADT Reprs declared");
              }
          adti->adtr_number = adtrnumber++;
```

```
              Map_define(adt->adt_reprs, t, adti);
              adti->adtr_inited = true;
              fin >> t;
              }
      assert(t==commaToken);
      @@ read the names of the (abstract) functions in the interface
      @@ now we have one of two kinds of lines:
@@      = <absfcnname> <Signature> ,
@@      or
@@      <absfcnname> <impfcnname> <Signature> ,
@@      down to the first semi-colon.
@@
      fin >> t;
      while (t!=semiToken) {
          if (t==eqToken) {
              int nofProfVars;
              @@ = <absfcnname> <nofProfVars> <parmtype1> <parmtype2> ... ,
              fin >> t;
              ADTabsFcn adaf = new ADTabsFcn_obj(t, adt);
              Set_add(adt->adt_afcns, adaf);
              if (Map_in(ADTafcns, t))
                fatal("duplicate abstract function names");
              Map_define(ADTafcns, t, adaf);
              @@ read the number of profiling variables for this function
              fin >> adaf->nofProfVars;
              @@ now read the parameters to the abstract function
              fin >> t;
              while (t!=commaToken) {
                  VarDecl v = new VarDecl_obj(dontCareToken, t);
                  adaf->adtaf_sig.add(v);
                  fin >> t;
                  }
              @@ the profiling evaluation functions must be accessible, but
              @@ they are declared implicitly.
              adaf->evalFcn=new ADTimpFcn_obj(t,adaf,adt->adt_profileImpl);
              }
          else if (t == plusToken) {
              @@ + optionalName optionalName ... ,
              do {
                  fin >> t;
                  } while (t != commaToken);
              }
          else {
              @@ <ADT> <reprname> <absfcnname> <impfcnname> <type> <type>
              @@  (1)      (2)        (3)           (4)
```

```
            @@ Each function is associated with several names:
            @@ (1) the name of the ADT it is an impl'n function for;
            @@ (2) the name of the representation it is an an impl'n
            @@     function for;
            @@ (3) the name of the abstract function it implements;
            @@       These must be unique across all ADTs.
            @@ (4) the name of the member function by which it is invoked;
            @@       (this is finessed right now: all member functions
            @@        are invoked by the same name as their abstract
            @@        function)

            ADType adt2;
            Map_value(ADTs, t, adt2);
            if (adt2 == nil)
              fatal("Unknown ADT in imp fcn dcln");
            @@ read the repr name (Set_1, Map_2, etc.)
            Token rn;
            fin >> rn;
            ADTRepr repr = isADTRepr(rn);
            @@ read the abstract fcn name;
            ADTabsFcn adaf2;
            Token aftok;
            fin >> aftok;
            Map_value(ADTafcns, aftok, adaf2);
            if (adaf2 == nil)
              fatal("Unknown abstract function name in imp fcn dcln");
            @@ read an implementation name of an abstract function
            fin >> t;
            ADTimpFcn fd = new ADTimpFcn_obj(t, adaf2, repr);
            Set_add(adaf2->adtaf_impl_fcns, fd);
            @@ read the parameters of the implementation function
            fin >> t;
            while (t != commaToken) {
                if (t == questToken) {
                    fd->afd_sig.add_dontCare();
                    }
                else {
                    ADTRepr adtr = isADTRepr(t);
                    assert(t==dontCareToken || adtr->name!=dontCareToken);
                    fd->afd_sig.add(dontCareToken, adtr->adtr_of, adtr);
                    }
                fin >> t;
                }
            }
        @@ this declaration line processed
```

```
          fin >> t;
          }
      @@ an adt group processed;
      adt->adt_inited = true;
      fin >> t;
      }
  @@ all ADTs are read
    @@ need a check that all ADTs have been inited
      @@ and that all ADTReprs have been inited;
  ADType adtelt; Token name;
  forAll(‘name, adtelt’, ADTs,
         if (!adtelt->adt_inited) {
             error(name->str);
             fatal("An uninitialized ADT");
             }
         );
  ADTRepr reprelt;
  forAll(‘name, reprelt’, ADTReprs,
         if (!reprelt->adtr_inited) {
             error(name->str);
             fatal("An uninitialized ADT representation");
             }
         );
    }

void
readVarDecls(Token ftok, char *finame)
{
  Token var;
  openFile(fin,i,"r",finame,true);
  fin >> var;
  while (var != dotToken) {
      Token typ,parm;
      int cnt = 0;
      fin >> typ;
      @@ assert(isanadt(typ));
      @@ read all the variable names declared in this program
      ADType adt3;
      Map_value(ADTs, typ, adt3);
      VarDecl vd = new VarDecl_obj(var, adt3);
      fin >> parm;
      while (parm != commaToken &&
             cnt < ADTinfoTable[adt3->adt_number].nofReqd) {
          List_append1(vd->vd_adtParms, parm);
          fin >> parm;
```

```
            cnt++;
            }
        assert(cnt == ADTinfoTable[adt3->adt_number].nofReqd);
        @@ just to be on the safe side, we'll check that if the var name
        @@ is already defined, its declaration matches exactly what we
        @@ have already defined.
        if (Map_in(Vars, var)) {
            VarDecl vddefd;
            Map_value(Vars, var, vddefd);
            if (*vd != *vddefd) {
                cerr << vd << "\n" << vddefd << "\n";
                fatal("Duplicate declarations of variable do not match");
                }
            delete vd;
            }
        else {
          Map_define(Vars, var, vd);
          }
        //  is also the place to read the optionals
        // parse_optionals(vd, ADTinfoTable[adt3->adt_number].tbl);
        // the optionals are of the form dd or dd=token
        optionType *tbl = ADTinfoTable[adt3->adt_number].tbl;
        while (parm != commaToken) {
            // the parm better be an integer
            int idx = parm->integer();
            fin >> parm;
            if (parm == eqToken) {
                fin >> parm;
                vd->vd_opts[idx] =
                  new Optional_obj(parm, tbl[idx].type, idx, tbl);
                fin >> parm;
                }
            else {
                vd->vd_opts[idx] =
                  new Optional_obj(nil, tbl[idx].type, idx, tbl);
                }
            }
        fin >> var;
        }
}

void
readUserFcnDecls(Token ftok, char *finame)
{
  Token t;
  openFile(fin,i,"r",finame,true);
  fin >> t;
  while (t!=dotToken) {
      int num = t->integer();
      Token fcnnamet, varnamet;
      fin >> fcnnamet;
      UserFcnDecl ufd = new UserFcnDecl_obj(num, fcnnamet);
      assert(!Map_in(UserFcns,fcnnamet));
      Map_define(UserFcns,fcnnamet,ufd);
      @@ read the parm types
      fin >> varnamet;
      while (varnamet!=commaToken) {
          Token vartypet;
          VarDecl var;
          fin >> vartypet;
          assert(vartypet != commaToken);
          if (varnamet == dontCareToken) {
              assert(vartypet == dontCareToken);
              ufd->ufd_sig.add_dontCare();
              }
          else {
              Map_value(Vars, varnamet, var);
              if (var == nil) {
                  cerr << "Undeclared variable read in readUserFcnDecls: "
                    << varnamet << "\n";
                  exit(1);
                  }
              ufd->ufd_sig.add(var, vartypet, dontCareToken);
              }
          fin >> varnamet;
          }
      fin >> t;
      }
}

void
readADTcallSites(Token ftok, char *finame)
{ @@ call sites of calls on functions in the interface of an adt
  Token t;
  Profarray profileArray;
  openFile(fin,i,"r",finame,true);
  fin >> t;
  Map_value(ProfArrays, ftok, profileArray);
  while (t!=dotToken) {
      Iterator apli;
      Token nt;
      ADTabsFcn aaf1;
      int profnum = t->integer();
      fin >> t;
      int linenum = t->integer();
      fin >> nt;
      Map_value(ADTafcns, nt, aaf1);
      assert(aaf1 != nil);
      ADTcallSite afc =
        new ADTcallSite_obj(linenum, profileArray, profnum, aaf1);
      @@ read the actual parms
      fin >> t;
      List_iterInit(aaf1->adtaf_sig.sig_sig, apli);
      while (t != commaToken) {
          VarDecl vd3;
          VarDecl abvd3;
          assert(!List_iterDone(aaf1->adtaf_sig.sig_sig, apli));
          List_iterate(aaf1->adtaf_sig.sig_sig, apli, abvd3);
          Map_value(Vars,t,vd3);
          @@ if the variable is not found, it is a dontCare;
          @@ That is, there is not a dontCareVar;
          @@ this can be confirmed by seeing if the corresponding parameter
          @@ of the abstract function is a dont care.  Otherwise, error;
          if (vd3 == nil) {
              if (abvd3 != nil) {
                  if (abvd3->vd_ADT == dontCareADT) {
                      afc->acs_sig.add_dontCare();
                      }
                  else {
                      fatal("Unrecognized var in call site parm list");
                      }
                  }
              else {
                  fatal("List_iterate(aaf1->adtaf_sig.sig_sig,apli,abvd3)
                                                                  == nil");
                  }
              }
          else {
              assert(abvd3->vd_ADT != dontCareADT);
              assert(vd3->name != dontCareToken);
              afc->acs_sig.add(vd3); @N type to be computed;
              Set_add(vd3->vd_inSigsOf, afc);
              }
          fin >> t;
          }
      List_iterCleanup(aaf1->adtaf_sig.sig_sig, apli);
      Set_add(ADTcalls,afc);
      fin >> t;
      }
}

void
readUserFcnCalls(Token ftok, char *finame)
{ @@ call sites of calls on (interesting) user functions
  Token t;
  VarDecl fcnParm;
  openFile(fin,i,"r",finame,true);
  fin >> t;
  while (t!=dotToken) {
      int num = t->integer();
      Token nt;
      fin >> nt;
      UserFcnDecl ufd3;
      Map_value(UserFcns,nt,ufd3); @@ must exist;
      assert(ufd3 != nil);
      UserFcnCall ufc = new UserFcnCall_obj(num, ufd3);
      Iterator sigIter;
      Signature_obj& formalParms = ufc->ufc_decl->ufd_sig;
      List_iterInit(formalParms.sig_sig, sigIter);
      @@ read the actual parms
      fin >> t;
      while (t!=commaToken) {
          @@ assert((t == dontCare) || isavar(t));
          VarDecl vd4;
          Map_value(Vars, t, vd4);
          ufc->ufc_sig.add(vd4);
          // alias this variable with the formal
          List_iterate(formalParms.sig_sig, sigIter, fcnParm);
          if (vd4 == nil) {
              assert(fcnParm != nil && fcnParm->name == dontCareToken);
              }
          else {
              vd4->aliasOf(fcnParm);
              if (DebugDetails) {
                  cerr << ">>>Aliases for "
                    << hex((int)vd4) << " " << vd4->name << ":";
                  Set_print(vd4->vd_as->as_set,cerr,VarDecl_obj_printForm);
                  cerr << "\n";
                  cerr << ">>>Aliases for "
                    << hex((int)fcnParm) << " " << fcnParm->name << ":";
                    Set_print(fcnParm->vd_as->as_set,cerr,
                                                 VarDecl_obj_printForm);
                    cerr << "\n";
                    }
                }
            fin >> t;
            }
        fin >> t;
        List_iterCleanup(formalParms.sig_sig, sigIter);
        }
}

void
readProfileData(Token ftok, char *finame)
{
  if (!profileDataValid) return;
  @@ Form of profile data input:
  @@ It is an array of 32-bit integers.  Each slot in the array
  @@ corresponds to a profile variable declared in the ADT profiling
  @@ implementations, or to the static call site of an adt function or
  @@ user function.  The number read with each function in
  @@ readprogramdesc is the beginning location of the profile
  @@ variables for all functions.  E.g. if f1 collects 5 profile
  @@ variables, and f1s number is 15, then locations 15 through 19
  @@ are the locations in the profile array of f1s profile variables.
  @@ these variables are not accessed by anything here except that the
  @@ first location of each function read is its execution frequency count.
  @@ (this is a required convention and is not currently enforced by
  @@ any software checks.  this is easy to fix by always declaring
  @@ p_cnt for ADT interface functions.)
  @@ Note that we are talking about the unique profile id (upid), not the
  @@ function id (fid).
  @@ If the profile data file does not exist, then our choices are simple:
  @@ all implementations are profiling (*_P) implementations.
  @@ New requirement: the last entry of each profile array is the number
  @@ of times that profile array was written to.  When we read the array in
  @@ it is converted from long to double, with each entry divided by the
  @@ execution count.
  long size;
  FILE *pfile = fopen(finame, "r");
  if (pfile == 0) {
      profileDataValid = false;
      if (DebugFiles) {
          cerr << "     Profile file " << finame << " not there.\n";
          }
      return;
      }
  fread((char *)&size, sizeof(long), 1, pfile);
  Profarray pa = new Profarray_obj(pfile, ftok, size);
  if (!pa->valid()) {
      profileDataValid = false;
      return;
      }
  Map_define(ProfArrays, ftok, pa);
  fclose(pfile);
  return;
}

@@ ===============================
@@ end of input section of program
@@ ===============================

@@ main section

@@ global variables
DECLARE(sortedCallSites, List, ADTcallSite);

@@ little functions

boolean
mapsto(ADTRepr r, ADType t)
{
  @@ can t be implemented/represented by r
  if (DebugDetails)
    cerr << ">>>mapsto(ADTRepr " << r << "::ADType " << t << ")\n";
  /* if (r == dontCareRepr) {
      if (t == dontCareADT) return true;
      else return false;
      }
  else if (t == dontCareADT) return false;
  -- the above is done by have a pseudo-ADT called dontCareADT, with
    -- one don't care ADTRepr dontCareRepr
    */
  return Map_in(t->adt_reprs,r->name);
}

boolean
mapsto(VarDecl actual, VarDecl formal)
{
  assert(formal->implemented);
  if (DebugDetails)
    cerr << ">>>mapsto(VarDecl " <<actual<<"::VarDecl "<<formal<<")\n";
  if (actual->implemented) {
      return (*actual->vd_repr == *formal->vd_repr);
      }
  else return mapsto(formal->vd_repr, actual->vd_ADT);
}

boolean
mapsto(Signature_obj &actual, Signature_obj &formal)
{
  @@ do the types of the variables map to the abstract types in the first
  @@ parameter List.  they map if each parameter maps.
  if (actual.len() != formal.len()) return false;
  Iterator nexta, nextf;
  VarDecl av, fv;
  if (DebugDetails)
    cerr << ">>>mapsto(Sig " << actual << "::Sig " << formal << ")\n";
  List_iterInit(actual.sig_sig, nexta);
  List_iterInit(formal.sig_sig, nextf);
  while (List_iterate(actual.sig_sig, nexta, av) &&
         List_iterate(formal.sig_sig, nextf, fv))
    {
    boolean r_mapsto, r_feas;
    if (!(r_mapsto = mapsto(av, fv))
        || (r_feas = (fv->vd_repr != dontCareRepr
               && !(*feasibilityFcns[fv->vd_repr->adtr_number])(av)))) {
        List_iterCleanup(actual.sig_sig, nexta);
        List_iterCleanup(formal.sig_sig, nextf);
        if (DebugDetails) {
            cerr << " >>>Doesn't map because ";
            if (!r_mapsto) {
                cerr << "the actual does not map to the formal\n";
                }
            else {
                assert(r_feas);
                cerr << av->name << " cannot be implemented by "
                  << fv->vd_repr->name << "\n";
                }
            }
        return false;
        }

    }
  List_iterCleanup(actual.sig_sig, nexta);
  List_iterCleanup(formal.sig_sig, nextf);
  return true;
}

// overload isCompatible;

boolean
isCompatible(ADTimpFcn fi, ADTcallSite c)
{
  if (DebugAssign)
    cerr << ">>>isCompatible(" << fi << "," << c << ") == ";
  if (mapsto(c->acs_sig, fi->afd_sig)) {
      if (DebugAssign)  cerr << "true\n";
      return true;
      }
  else {
      if (DebugAssign)  cerr << "false\n";
      return false;
      }
}

DeclareUserFcn(findCompatibleImplementations, ?, ?, Ic, Set)@;

void
findCompatibleImplementations(ADTcallSite &c,
                              DeclareParm(Ic, Set, ADTimpFcn, afd_bminfo))
{ @@ find all implementations of c->acs_afcn compatible with the
  @@ parameters in call site c;
  ADTimpFcn fi;
  Set_makeEmpty(Ic);
  forAll(fi, c->acs_afcn->adtaf_impl_fcns,
         if (isCompatible(fi, c)) {
             Set_add(Ic, fi);
             }
         );
  return;
}

DeclareUserFcn(callSitesContaining, ?, ?, callSitesp, Set)@;

void
callSitesContaining(VarDecl v, DeclareParm(callSitesp, Set, ADTcallSite,
                                           acs_bminfo))
{
  @@ union is overkill: callSitesp is always empty.  An interesting
  @@ data point for therblig analysis, though.
  Set_union1(callSitesp, v->vd_inSigsOf);
}

void
assignImplType(VarDecl v, ADTRepr t)
{
  assert(!v->implemented);
  if (DebugDetails)
    cerr << ">>>assignImplType: " << v << " <- " << t << "\n";
  v->vd_repr = t;
  v->implemented = true;
}


void
unassignImplType(VarDecl v)
{
  assert(v->implemented);
  v->implemented = false;
}

boolean
implementable(VarDecl callSiteVar, ADTRepr r)
{
  @@ The inherent feasibility of assigning callSiteVar the
  @@ representation r was checked in isCompatible.
  assert((*feasibilityFcns[r->adtr_number])(callSiteVar));
  @@
  @@ if callSiteVar is assigned the impl. type impFcnFormalVar->vd_repr,
  @@ then for every call site c that has callSiteVar in its actual parameter
  @@ List, check that there still exists AT LEAST ONE impl'n function
  @@ that can be used to implement the function called at c.
  @@ This check is not absolutely necessary, but I suspect it may cut
  @@ down on the amount of backtracking.
  @@ Side effect: callSiteVar is assigned impln
  @@        impFcnFormalVar->vd_repr, if feasible.
  @@
  DECLARE(callSites, Set, ADTcallSite, acs_bminfo);
  ADTcallSite c;
  @@ by defn of findCompatibleImplementations:
  assert(mapsto(r, callSiteVar->vd_ADT));
  assignImplType(callSiteVar, r);
  if (DebugAssign) {
      cerr << ">>>trying " << r << " for " << callSiteVar->name << "\n";
    }
  CallUserFcn(callSitesContaining, callSiteVar, callSites)@;
  callSitesContaining(callSiteVar, callSites);
  if (DebugAssign) {
      cerr << ">>>callSitesContaining: ";
      Set_print(callSites, cerr, ADTcallSite_obj_printName);
      cerr << "\n";
      }
  if (Set_empty(callSites)) return true;
  @@ for each call site
  Iterate(next, c, callSites,
          ADTimpFcn fi;
          Iterate(nextfi, fi, c->acs_afcn->adtaf_impl_fcns,
                  if (mapsto(c->acs_sig, fi->afd_sig)) {
                      Set_iterCleanup(c->acs_afcn->adtaf_impl_fcns, nextfi);
                      goto SUCCESS;
                      }
                  );
          unassignImplType(callSiteVar);
          if (DebugAssign) {
              cout << "No implementations for " << c->acs_afcn << "\n";
              }
          Set_iterCleanup(callSites, next);
          return false;
        SUCCESS: ;
          );
  return true;
}

DeclareUserFcn(parmsImplementable, ?, ?, ?, ?, changedp, Set)@;

boolean
parmsImplementable(ADTcallSite c, ADTimpFcn f,
                   DeclareParm(changedp, Set, VarDecl, vd_bminfo))
{
  @@ Check that the impl. fcn f can be used to implement the fcn
  @@ called at call site c by checking that the variables in the actual
  @@ parameter List to c can be assigned the
  @@ impl. types required by the formal parms of f.  Note that the
  @@ parallel iteration over the formal parms of f and the
  @@ actual parms of c works by the
  @@ definition of the function findCompatibleImplementations.
  @@ Refinement: this check has to be performed for the actual var and every
  @@ variable to which it is aliased in every function call in which they
  @@ occur.
```

```
assert(Set_empty(changedp));
Iterator nextf;       @@ points to the implementation formal
VarDecl impforml;
Iterator nextv;       @@ points to the call site actual
VarDecl actual;
if (DebugAssign)
  cerr << ">>>parmsImplementable:" << c->acs_sig << f->afd_sig << "\n";
List_iterInit(f->afd_sig.sig_sig, nextf);
List_iterInit(c->acs_sig.sig_sig, nextv);
while (List_iterate(f->afd_sig.sig_sig, nextf, impforml) &&
       List_iterate(c->acs_sig.sig_sig, nextv, actual))
  {
  assert(impforml->implemented); @@ it is a formal impln parm
  if (!actual->implemented) {
      //     for each variable aliased to actual,
      //       for each call site using that variable,
      //         see if it is implementable using the type of impformal
      VarDecl aliasv;
      MALLOCK;
      if (DebugAssign) {
          cerr << ">>>Aliases for " << actual->name << ":";
        Set_print(actual->vd_as->as_set, cerr, VarDecl_obj_printForm);
        cerr << "\n";
        }
      forAll(aliasv, actual->vd_as->as_set,
             if (DebugAssign)
                 cerr << ">>>Alias loop: " << aliasv << "\n";
             if (!aliasv->implemented) {
                 if (implementable(aliasv, impforml->vd_repr)) {
                     // implementable assigns implementation to actual
                     if (DebugAssign) {
                         cout << "Implementing " << aliasv->name <<
                           " as " << impforml->vd_repr->name << "\n";
                         }
                     Set_add(changedp, aliasv);
                     }
                 else {
                     List_iterCleanup(f->afd_sig.sig_sig, nextf);
                     List_iterCleanup(c->acs_sig.sig_sig, nextv);
                     if (DebugAssign) {
                         cout << "Could not implement " << aliasv->name
                           << " as " << impforml->vd_repr << "\n";
                         }
                     return false;
                     }
```

```
                   }
               );
        }
    }
  List_iterCleanup(f->afd_sig.sig_sig, nextf);
  List_iterCleanup(c->acs_sig.sig_sig, nextv);
  return true;
}

DeclareUserFcn(undoImplementations, ivars, Set, ?, ?)@;

void
undoImplementations(DeclareParm(ivars, Set, VarDecl, vd_bminfo), int index)
{
  @@ undo the implementations of the variables in ivars
  VarDecl v;
  forAll(v, ivars,
         if (DebugAssign){
             cout << "   Undoing " << v->name << " " << index << "\n";
             }
         unassignImplType(v);
         );
}

@@ the call site of interest in the current invocation of assignable
@@ an unfortunately necessary global

ADTcallSite curCallSite;

int
compareCosts(ADTimpFcn f1, ADTimpFcn f2)
{
  @@ compare the resource costs of the two functions
  // these computations should be cached in the call site (e.g. a list
  // of costs for each possible afd_uid).
  double f1r, f2r;
  f1r = curCallSite->eval(f1);
  f2r = curCallSite->eval(f2);
  if (DebugCosts) {
      cerr << "Callsite(" << curCallSite->acs_upid << "): ";
      f1->typedName(cerr);
      cerr << "=" << f1r << " and ";
      f2->typedName(cerr);
      cerr << "=" << f2r << "\n";
      }
  if (f1r < f2r) return -1;
  if (f1r > f2r) return 1;
  return 0;
}

void
printSortedCallSites(boolean better)
{
  ADTcallSite cs;
  if (DebugSortCallSites) {
      cout << "\nSorted call sites:\n";
      forAll(cs, sortedCallSites,
             cout << cs << "\n";
             if (better)
                 cs->betterImpl();
             );
      cout << "\n";
      }
}

void
RecordCurAssignments()
{
  VarDecl var;
  Token t;
  printSortedCallSites(true);
  forAll(`t,var', Vars,
         if (t != dontCareToken) {
             cout << "Implemented " << var->name
               << " as " << var->vd_repr->name;
             if (var->vd_bestRepr!=nil && var->vd_bestRepr!=var->vd_repr) {
                 cout << " (vs. " << var->vd_bestRepr->name << ")";
                 }
             cout << "\n";
             var->betterRepr();
             }
         );
}

boolean
assignable( Iterator iter, int listIndex, double cost )
{
  @@ take the next call site c, and assign it the cheapest
  @@ implementation you can.  whether c can be assigned the cheapest
  @@ implementation is determined by parmsImplementable.
```

```
ADTcallSite c;
ADTimpFcn  fi;        @@ a candidate implementation of this abs. fcn
DECLARE(implSet, Set, ADTimpFcn, afd_bminfo); @@ Set of impl'n fcns
                                        @@ compatible with ADTcallSite c
DECLARE(implList, List, ADTimpFcn);@@ implSet sorted;
boolean worked;@@ true if parmsImplementable succeeded;
assert(cost >= 0);
@@ prune this search branch if we are already too costly;
if (curAssignCost != -1.0 && cost >= curAssignCost) {
    cout << ".Pruned at " << listIndex << ".\n";
    return false;
    }
cout << "." << listIndex;
if (List_iterDone(sortedCallSites, iter)) {
    // then we are at the bottom of the file.
    if (curAssignCost == -1.0 || cost < curAssignCost) {
        RecordCurAssignments();
        if (curAssignCost == -1.0) cout << "First ";
        else cout << "Better ";
        cout << "implementation: " << form("%10.2f",cost);
        if (curAssignCost != -1.0) {
            double delta = (curAssignCost - cost ) / curAssignCost;
            cout << "  delta=" << form("%10.8f",delta);
            }
        cout << "\n";
        curAssignCost = cost;
        }
    else {
        cout << "Not better implementation: " << cost << "\n";
        }
    return true;
    }
if (!List_iterate(sortedCallSites,iter,c)) {
    assert(false);
    }
if (DebugAssign) {
    cerr << ">>>LOOP: assignable call site: " << c << "\n";
    }
MALLOCK;
CallUserFcn(findCompatibleImplementations, c, implSet)@;
findCompatibleImplementations(c, implSet);
if (DebugAssign) {
    cerr << ">>>Compatible implementations: ";
    Set_print(implSet, cerr, ADTimpFcn_obj_print);
    cerr << "\n";
    }
curCallSite = c;
Set_sort2(implSet, implList, &compareCosts);
@@ for each implementation fi for c, assign the types implied by
@@ fi‘’s Signature to the variables for c.
@@ if this assignment is feasible, then recurse and try the next
@@ call site on the List.  if the recursion returns true, then
@@ return true, else the assignment is not feasible.
@@ if the assignment is not feasible try the next implementation fcn.
@@ if no impl fcns are feasible then return false.
Iterate(next, fi, implList,
       @@ the Set of all variables assigned
       DECLARE(changed, Set, VarDecl, vd_bminfo);
       @@ on a call to parmsImplementable.
       if (DebugAssign) {
           cerr << ">>>assignable loop on " << fi << " index " <<
             listIndex << "\n";
           }
       CallUserFcn(parmsImplementable, c, fi, changed);
       worked = parmsImplementable(c, fi, changed);
       if (DebugAssign) {
           cerr << ">>>parmsImplementable: " << BOOL(worked) << "\n";
           }
       if (worked) {
           Iterator iter2;
           List_iterCopy(sortedCallSites, iter, iter2);
           c->implement(fi);
           double fcnCost = c->eval();
           if (isnan(fcnCost) || isinf(fcnCost) || fcnCost < 0.0) {
               cerr << "Evaluation function problem:\n";
               cerr << "The fcn for "<<fi->repr->name<<fi->name<<"\n";
               cerr << "fubarred.  It returned " << fcnCost << "\n";
               exit(1);
               }
           if (assignable(iter2, listIndex+1, cost + c->eval())) {
               // we're walking back up the list toward the more
               // important call sites;
               if (listIndex < cutOffIndex) {
                   // then we want to try alternatives;
                   List_iterCleanup(sortedCallSites, iter2);
                   CallUserFcn(undoImplementations,changed,listIndex);
                   undoImplementations(changed, listIndex);
                   c->unimplement();
                   continue; @@ very important: continues the forAll!!!
                   }
```

```
                    else {
                        // then we won‘’t try alternatives;
                        List_iterCleanup(sortedCallSites, iter2);
                        List_iterCleanup(implList, next);
                        if (DebugAssign || DebugCosts) {
                            cerr<<">>>assigned "<< c <<" "<< fi->repr->name
                              << "; cost: " << c->eval(fi) << "\n";
                            }
                        return true;
                        }
                    }
                else {
                    CallUserFcn(undoImplementations,changed,listIndex);
                    undoImplementations(changed,listIndex);
                    c->unimplement();
                    }
                List_iterCleanup(sortedCallSites, iter2);
                }
            else {
                if (DebugAssign) {
                    cout << "Did not implement call site " << c << "\n";
                    }
                CallUserFcn(undoImplementations,changed,listIndex);
                undoImplementations(changed,listIndex);
                c->unimplement();
                }
            );
  return false;
}

//==============================================================

void
findVariableAliases()
{
  // for each call site
  //    alias the formal and the actual
  // until done
  //        for each variable
  //            merge alias sets.
}

void
assignEverythingToProfile()
{
  Token t;
  VarDecl vd;
  forAll(‘t, vd’, Vars,
         vd->vd_repr = vd->vd_bestRepr = vd->vd_ADT->adt_profileImpl;
         vd->implemented = true;
         );
}

int
compareCallSitesImportance(ADTcallSite c1, ADTcallSite c2)
{
  if (c1->acs_rank == IllegalRank)
    c1->acs_rank = (*evalFcns[c1->acs_afcn->evalFcn->afd_uid])(c1);
  if (c2->acs_rank == IllegalRank)
    c2->acs_rank = (*evalFcns[c2->acs_afcn->evalFcn->afd_uid])(c2);
  if (c1->acs_rank < c2->acs_rank) return 1;
  if (c1->acs_rank > c2->acs_rank) return -1;
  return 0;
}

void
sortByImportance(void)
{
  @@ sorts the Set ADTcalls (the Set of callSites) into the List
  @@ sortedCallSites.  the key is the frequency of the call sites.
  Set_toList(ADTcalls, sortedCallSites);
  List_sort1(sortedCallSites, &compareCallSitesImportance);
  @@ better:
  @@     Set_sort(ADTcalls, sortedCallSites, &compareCallSitesImportance);
  @@ now determine how many of these items we’re going to iterate over;
  int size = List_length(sortedCallSites);
  if (cutOffPercent == 100) {
      cutOffIndex = size;
      }
  else if (cutOffPercent == 0) {
      cutOffIndex = 0;
      }
  else {
      double rankSum = 0;
      ADTcallSite cs;
      forAll(cs, sortedCallSites,
             assert(cs->acs_rank != IllegalRank);
             rankSum += cs->acs_rank;
             );
      double rankCutOff = rankSum * cutOffPercent / 100;
      rankSum = 0;
      cutOffIndex = 0;
      forAll(cs, sortedCallSites,
             if (rankCutOff <= rankSum) break;
             rankSum += cs->acs_rank;
             cutOffIndex++;
             );
      }
  printSortedCallSites(false);
}

void
printAssignments(char *foname)
{
  // print out the assignments
  openFile(afile,o,"w",foname,true);
  Token t;
  VarDecl vd;
  afile << m5Comment << "This is automatically created by therblig\n";
  afile << m5Comment << "\n";
  afile << m5PushPool_VarDecl_pool << "VARDECLS_HDR\n";
  afile << m5Comment2;
  forAll(`t, vd', Vars,
         if (vd->vd_bestRepr == nil) {
             afile << "!!!Not implemented: " << vd->name << "\n";
             cerr << "!!!Not implemented: " << vd->name << "\n";
             }
         else {
             @@ first, call the instantiation routine for the ADT
             @@ on this variable.  It will define the strings necessary
             @@ to declare the variable, make sure the sources of the code
             @@ exist, and that the appropriate coercion class exists.;
             (*instantiationFcns[vd->vd_bestRepr->adtr_number])(vd);
             afile << m5Comment2 << m5Comment << "    " << vd->name << "\n"
               << m5Comment2;
             @@;
             @@ first, put out the INSTANTIATE_ADT_i macro;
             @@;
             afile << "INSTANTIATE(" << vd->vd_bestRepr->name << ","
               << vd->instance_name;
             if (!vd->instance_parm->empty())
               afile << "," << vd->instance_parm->string();
             afile << ")" << m5Comment2;
             @@;
             @@ second put out the COERCE_ADT_i macro;
             @@;
             afile << "COERCE(" << vd->vd_bestRepr->name << ","
               << vd->instance_name
                 << "," << vd->coercion_name->string();
             if (!vd->coercion_parm->empty())
               afile  << "," << vd->coercion_parm->string();
             afile << ")" << m5Comment2;
             @@;
             @@ third  put out the DECLARE_M macro for the variable itself.;
             @@;
             afile << m5DECLARE_M << vd->name << ","
               << vd->coercion_name->string();
             if (!vd->constructor_parms->empty())
               afile << "," << vd->constructor_parms->string();
             afile << "`)'" << m5Comment2;
             afile << m5Comment2;
             @@ Token t;
             @@ forAll(t, vd->vd_adtParms, api, afile << "," << t; );
             }
         );
  afile << "VARDECLS_TLR\n";
}


void
initialize()
{
  dontCareADT = isADType(dontCareToken);
  dontCareRepr = isADTRepr(dontCareToken);
  Map_define(dontCareADT->adt_reprs, dontCareToken, dontCareRepr);

  dontCareADT->adt_inited = true;
  dontCareADT->adt_number = 0x7FFFFFFF;

  dontCareRepr->adtr_of = dontCareADT;
  dontCareRepr->adtr_inited = true;
  dontCareRepr->adtr_number = 0x7FFFFFFF;
}

#define ReadThisForAllFiles(varname,fcn)                        \
fnp = argv; do {                                        \
        Token t_f;                                      \
        strcpy(fbuf, *fnp);                             \
        strcat(fbuf, "_");                              \
        strcat(fbuf, varname);                          \
        t_f = new Token_obj(*fnp, id_tkn, 0);           \
        if (DebugFiles) {                               \
            cerr<<"Doing function "<<#fcn<<" on file "<<fbuf<<"\n"; \
            }                                           \
        fcn(t_f,fbuf); } while (*++fnp != 0)

main(int argc, char **argv)
{
  Iterator scs_iter;
  char **fnp;
  char fbuf[64];
  char *vardeclsName = "vardecls.m5";
  curAssignCost = -1.0;
  argv++; argc--;
  cutOffPercent = 0; @@ this should yield the same results as the original
    @@ version;
  while (argv[0] != 0 && argv[0][0] == ch_minus) {
      char *cp = &argv[0][1];
      if (*cp == ch_D) {
          while (*++cp != 0) {
              if (*cp == ch_a) { DebugAssign = true; }
              else if (*cp == ch_c) { DebugCosts = true; }
              else if (*cp == ch_f) { DebugFiles = true; }
              else if (*cp == ch_i) { DebugInput = true; }
              else if (*cp == ch_s) { DebugSortCallSites = true; }
              else if (*cp == ch_d) { DebugDetails = true; }
#ifdef DBG_MALLOC
              else if (*cp == ch_m) { DebugMalloc = true; malloc_debug(1); }
              else if (*cp == ch_M) { DebugMalloc = true; malloc_debug(2); }
#endif
              else fatal("Unknown debugging flag");
              }
          }
      else if (*cp == ch_o) { vardeclsName = argv[1]; argv++; }
      else if (*cp == ch_P) {
          if (*(cp+1) != ch_null) {
              cutOffPercent = atoi(cp+1);
              }
          else {
              cutOffPercent = atoi(argv[1]); argv++;
              }
          assert( 0 <= cutOffPercent && cutOffPercent <= 100 );
          if (cutOffPercent < 0) cutOffPercent = 0;
          if (cutOffPercent > 100) cutOffPercent = 100;
          }
    else fatal("Unknown argument");
    argv++;
    argc--;
    }
if (argc < 1) fatal("Must specify file to work on");
initialize();
readADTs("ADTs.th"); @@ this should really be compiled in and not read.
profileDataValid = true;
ReadThisForAllFiles("profData.dat",readProfileData);
ReadThisForAllFiles("‘ADT_vars’.th",readVarDecls);
ReadThisForAllFiles("‘ADT_ufcns’.th",readUserFcnDecls);
ReadThisForAllFiles("‘ADT_csites’.th",readADTcallSites);
ReadThisForAllFiles("‘ADT_ufcalls’.th",readUserFcnCalls);
findVariableAliases();
if (profileDataValid) {
    cout << "Attempting assignment ...\n";
    sortByImportance();
    cout << "Cutoff: " << cutOffPercent << " results in "
      << cutOffIndex << " of " << List_length(sortedCallSites)
        << " being recursed over.\n";
    List_iterInit(sortedCallSites, scs_iter);
    assignable(scs_iter, 0, 0.0); @@ always returns false
    if (curAssignCost >= 0.0) {
        cout << "Writing " << vardeclsName << ".\n";
        printAssignments(vardeclsName);
        }
    else {
        cout << "\nCannot assign for some reason:
                          assigning default profiling implementations\n";
        assignEverythingToProfile();
        cout << "Writing " << vardeclsName << ".\n";
        printAssignments(vardeclsName);
        exit(1);
        }
    List_iterCleanup(sortedCallSites, scs_iter);
    }
else {
    cout <<
        "No profile data: assigning default profiling implementations\n";
    assignEverythingToProfile();
    printAssignments(vardeclsName);
    }
/*                                        */ MALLOCK;
if (DebugAssign || DebugCosts) {
    cerr<<"Nof VarDecls     = "<< VarDecl_obj::vd_R.last + 1 <<"\n";
```

```
      cerr<<"Nof ADTabsFcns   = "<< ADTabsFcn_obj::adtaf_R.last + 1 <<"\n";
      cerr<<"Nof ADTimpFcns   = "<< ADTimpFcn_obj::afd_R.last + 1 <<"\n";
      cerr<<"Nof ADTcallSites = "<< ADTcallSite_obj::acs_R.last + 1 <<"\n";
      }
  exit(0);
}
```

```
@@ FILE: Tokens.t
#define TOKENS_MAIN
#include <stream.h>
#include <ctype.h>
#include "util.H"
#include "Tokens.H"
#include "charClasses.h"
#include "Tokens_ADTs.H"
#include "userTypes.H"

String_obj::~String_obj()
{
  delete [maxlen]str;
}

void
String_obj::realloc(int add)
{
  // assumes that len is already set to new length
  // if add == 0, then this copies the string.
  char *cp = new char[len];
  strcpy(cp, str);
  delete [len-add]str;
  str = cp;
  maxlen = len;
}

String_obj&
String_obj::operator<<(String_obj& t)
{
  if (maxlen <= (len += t.len)) {
      realloc(t.len);
      }
  strcat(str, t.str);
  return *this;
}

String_obj&
String_obj::operator<<(char *cp)
```

```
{
  int l = strlen(cp);
  if (maxlen <= (len += l)) {
      realloc(l);
      }
  strcat(str, cp);
  return *this;
}

String_obj&
String_obj::operator<<(int i)
{
  char buf[32]; // should be big enough for an int
  sprintf(buf,"%d",i);
  int l = strlen(buf);
  if ((len += l) >= maxlen) {
      realloc(l);
      }
  strcat(str,buf);
  return *this;
}

String_obj&
String_obj::operator<<(Token_obj& T)
{
  if ((len += T.len) >= maxlen) {
      realloc(T.len);
      }
  strcat(str,T.str);
  return *this;
}

#define HASH_SZ MAX_NOF_TOKENS

#define NOF_SECHASH 16
static Token hash[HASH_SZ];
static int hashprime[NOF_SECHASH] = {
13,31,41,71,131,139,149,157,163,173,227,373,389,457,461,499 };

@@ Tokens consist of a type (id_tkn, punct_tkn, num_tkn, ...) and a
@@ string.

Token_obj::Token_obj(char *sp, typetype t, int v)
{
  int hashv;
  int i;
  int l = strlen(sp);
  int first, phash, shash;
  char *cp = sp;
  Token T;
  bool done;
  phash = 0;
  for (i=0; i < l; i++) {
      phash = (phash << 1) + *cp++;
      }
  phash += (int)t + (phash >> 8);
  first = phash & (HASH_SZ-1);
  shash = hashprime[(phash >> 10) & (NOF_SECHASH - 1)];
  done = false;
  do {
      if ((T=hash[first]) == 0) {
          @@ not in table;
          if (this == 0) {
              T = hash[first] = (Token) new char[sizeof(Token_obj)];
              }
          else T = hash[first] = this;
          T->str = new char[l+1];
          strcpy(T->str, sp);
          T->hashv = phash;
          T->val = v;
          T->len = l;
          T->type = t;
          T->uid = Token_nextuid++;
          done = true;
          }
      else {
          if (T->hashv == phash && T->len == l && strcmp(T->str, sp) == 0)
            @@ found it in the table
            done = true;
          else {
              first = (first + shash) & (HASH_SZ-1);
              if (first == (phash & (HASH_SZ-1))) {
                  cerr << "Token table full\n";
                  exit(1);
                  }
              }
          }
      } while (!done);
  this = T;
}
```

```

Token Token_obj::suffix()
{
  // return the suffix of the string of the form: xxxx_ss
  char *cp = strrchr(str, ch__);
  if (cp == nil) return new Token_obj("?", id_tkn, 0);
  return new Token_obj(cp+1, id_tkn, 0);
}

Token Token_obj::append(char *sfx)
{
  char buffer[256];
  strcpy(buffer, str);
  strcat(buffer, sfx);
  return new Token_obj(buffer, type, 0);
}

ostream&
operator<<(ostream &s, String str)
{
  if (str == 0) { return s << "<string not allocated!>"; }
  if (str->str == 0) { return s<<"<char array not allocated for string!>"; }
  else return s << str->str;
}

ostream&
operator<<(ostream &s, Token gt)
{
  gt->print(s);
  return s;
}

static char buf[256];

#define casech(ch,tkn) case ch:{*cp++ = c; s.get(c); toktype = tkn; break;}

istream&
operator>>(istream& s, Token& gt)
{
  char c;
  char *cp;
  typetype toktype;
  int tokval;
  int toklen;
  cp = buf;
```

```
LOOP:
 if (!s.good()) {
     if (s.eof())
       cerr << "Tokens>> should never reach EOF in any input files\n";
     if (s.fail()) cerr << "Token >> => _fail?\n";
     if (s.rdstate() == _bad) cerr << "Token >> => _bad?\n";
     cerr << "Token >> not good!\n";
     exit(1);
     }
 s.get(c);
 switch (c) {

   case_is_C_firstid:
     {
     *cp++ = c;
     s.get(c);
     while (s.rdstate() != _eof && (isalnum(c) || c == ch__)) {
         *cp++ = c;
         s.get(c);
         }
     toktype = id_tkn;
     break;
     }

   case ch_twiddle:
     c = ' ';
     /* NOTE fall through! */

   case_isspace:
     goto LOOP;

   case_isdigit:
     {
     tokval = c - ch_0;
     *cp++ = c;
     s.get(c);
     if (c == ch_x || c == ch_X) { @@ hex numbers
         *cp++ = ch_x;
         s.get(c);
         while (s.rdstate() != _eof && (isxdigit(c))) {
             int h = c - ch_0;
             if (h > 9) h = c - ch_A + 10;
             if (h > 15) h = c - ch_a + 10;
             tokval = tokval*16 + h;
             *cp++ = c;
```

```
                s.get(c);
                }
            }
        else if (isdigit(c)) {
            tokval = tokval * 10 + c - ch_0;
            *cp++ = c;
            s.get(c);
            while (s.rdstate() != _eof && isdigit(c)) {
                tokval = tokval * 10 + c - ch_0;
                *cp++ = c;
                s.get(c);
                }
            }
        toktype = num_tkn;
        break;
        }
        casech(ch_dot,punct_tkn);
        casech(ch_colon,punct_tkn);
        casech(ch_semi,punct_tkn);
        casech(ch_quest,punct_tkn);
        casech(ch_comma,punct_tkn);
        casech(ch_eq,punct_tkn);
        casech(ch_minus,punct_tkn);
        casech(ch_plus,punct_tkn);

      default:
        *cp++ = c;
        s.get(c);
        toktype = unknown_tkn;
        error("Unknown token");
        *cp = ch_null;
        error(buf);
        }

    if (s.rdstate() != _eof) s.putback(c);
    *cp = ch_null;
    toklen = cp - buf;
    gt = new Token_obj(buf, toktype, tokval);
    if (DebugInput) {
        cerr << buf << " ";
        if (gt == commaToken || gt == semiToken) cerr << "\n";
        }
    return s;
}
```