# Profile-directed restructuring of operating system code

by W. J. Schmidt
R. R. Roediger
C. S. Mestad
B. Mendelson
I. Shavit-Lottem
V. Bortnikov-Sitnitsky

***In this paper we describe how a profiling system can be successfully used to restructure the components of an operating system for improved overall performance. We discuss our choice of a profiling system and how it was applied to the AS/400® (Application System/400®) operating system for the purpose of reordering code. Previous work in the industry has been mainly useful only for application programs. Our work demonstrates how such techniques can be applied to operating system code, while preserving maintainability of the operating system in the customer's environment.***

It is well-known that performance of processors is increasing at a much faster rate than the performance of their attached memory subsystems.[1] Thus it is increasingly difficult to input data to processors rapidly enough to keep the processors utilized to their maximum capacity. As a result, a great deal of ingenuity has been expended on hardware solutions to improve the access time and throughput of memory references, including caches, prefetch buffers, branch prediction hardware, memory module interleaving, very wide buses, and so forth. Also, software must be optimized to take the best possible advantage of this hardware.

For example, instruction caches are designed to exploit temporal and spatial locality in programs. *Temporal locality* refers to the tendency of programs to execute instructions repeatedly; thus the performance of fetching instructions from main memory can be improved by saving recently executed instructions in a small high-speed cache. Instructions in a program are said to exhibit good *spatial locality* if execution of an instruction tends to be followed quickly by execution of instructions packaged nearby. A program with poor spatial locality will cause unneeded instructions to be fetched into the cache. Thus the cache will not operate at its full potential.

Memory paging systems are likewise designed to exploit spatial and temporal locality. For these systems, volatile memory may be thought of as a medium-speed cache for low-speed persistent memory, such as a disk. Recently used pages are kept in memory to take advantage of temporal locality. Again, good spatial locality is required to avoid bringing unneeded instructions and data into memory. Poor spatial locality thus reduces the efficiency of memory paging.

Unfortunately, "naïve" code generation often results in programs that have poorer spatial locality than is achievable. It is typical, for example, to generate code that branches around infrequently executed error paths. This results in poor utilization of the instruc-

©**Copyright** 1998 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.


0018-8670/98/$5.00 © 1998 IBM

tion cache, since some of the error path code will usually be fetched into the cache along with the branch that bypasses it. It is also typical for procedures to be packaged without consideration for locality, so that although procedure A frequently calls procedure B, A and B are located in different memory pages.

Across the industry, it is becoming more common to use *dynamic profiling* to analyze program behavior during execution. Dynamic profiling (henceforth *profiling*) gathers data about the frequencies with which different execution paths in a program are traversed. These profile data can then be fed back into the compiler to guide optimization of the code.

One of the proven uses of profile data is in determining the order in which instructions should be packaged. By discovering the "hot traces" through a procedure, the optimizer can pack the instructions in those traces together tightly into cache lines, resulting in greater cache utilization and fewer cache misses. Similarly, profile data can help determine which procedures call other procedures most frequently, permitting the called procedures to be reordered in memory to reduce page faults.

**Related work.** Research into reordering programs for better performance dates to the introduction of virtual memory[2-4] in the 1960s. A number of early researchers[5-9] used static analysis to reduce page faults by reordering procedures within a program, while Hatfield and Gerald[10] and Ferrari[11] used dynamic analysis for similar goals, using an instruction trace collected from an execution of the program. Hartley[12] extended the static techniques through the use of procedure duplication and in-line placement. Wu[13] experimented with a trace-based system for repositioning procedures based upon temporal locality, with the goal of improving performance of shared-memory multiprocessors.

With the introduction of instruction caches,[14] focus began to shift to reordering code at a finer granularity. To date, most successful approaches to improving instruction cache performance have used profile data to predict branch outcomes. In contrast to most of the foregoing work on virtual memory performance, these techniques were implemented within the framework of optimizing compilers. McFarling[15] showed how to use profile information to reduce conflict misses in a direct-mapped instruction cache. Mendelson, Pinter, and Shtokhamer[16] also achieved a reduction in conflict misses while requiring only static analysis of the program. Hwu and Chang[17] introduced the idea of using traces of basic blocks to reduce the number of unexecuted instructions brought into the instruction cache (*cache pollution*), thus reducing capacity misses. Pettis and Hansen[18] likewise used traces (or *chains*) to order basic blocks, although their algorithms differ somewhat from those of Hwu and Chang; they also pointed out that infrequently executed traces can be separated entirely from the main procedure body. Gupta and Chi[19] produced two different methods of reordering instructions, one based on the presence of loops, split points, and join points in the control flow graph, and one based directly on the control dependence graph.[20]

Within IBM, Heisch[21,22] suggested that instruction cache performance can be maximized by considering it as a whole-program optimization. Heisch's methods differ from previous approaches by operating as a post-processor on executable program objects. An interesting effect of this method is that basic blocks are allowed to migrate without being constrained by procedure boundaries. Heisch's original algorithm appended the reordered code to the original executable program objects, resulting in reported growth in executable file size of between 5 percent and 41 percent,[22] although this growth had negligible impact on performance. Nahshon and Bernstein[23] later produced an improved algorithm that required less code growth; their techniques, together with those of Heisch, were incorporated into a tool called *FDPR* (*feedback-directed program restructuring*), which has been used to improve performance of executable objects on the AIX* (Advanced Interactive Executive) and OS/2* (Operating System/2*) operating systems.

**Contributions of this paper.** Aware of the performance benefits achieved using FDPR on other platforms within IBM, we began to consider how this technology could be used to improve performance on the Application System/400* (AS/400*) PowerPC AS*-based computer systems. We also wanted to take things a step further: Rather than just applying this technology to applications, our goal was to improve the performance of the AS/400 operating system.

We are not aware of any previous attempt to ship an operating system that has been reordered in this manner. (Experiments with reordering operating system kernels have been described,[22,24] but to our knowledge these techniques have not been used in products that have been shipped to customers.)

There are a number of difficult issues that must be addressed when restructuring an operating system:

- It must be feasible to provide correction code (fixes) to customers should errors be found in the operating system code. An operating system is much too large and complex to be rebuilt from scratch and redelivered to customers. It is necessary to be able to modify smaller amounts of the operating system when fixing problems by replacing a single module (compile unit) or even a single procedure.
- Providing fixes must not result in noticeably degraded performance in a customer environment. If the operating system has been restructured to improve instruction cache and memory paging performance, applying fixes to restore functionality should not undo all or a significant part of those performance gains.
- Special memory requirements of sensitive portions of the operating system must be honored. Many profiling systems (including ours) gather data by "instrumenting" the code to be profiled—that is, inserting snippets of code to record control flow events—and then running the instrumented code using representative inputs. These snippets typically access counters in memory. Some portions of operating system code cannot make data references that will cause a page fault (consider the code whose job it is to *handle* a page fault), while more sensitive code may not even be able to tolerate a miss in the hardware page table. Care must be taken so that the added instrumentation does not violate these requirements.
- Unlike most application code, a single copy of the operating system code may be executing simultaneously on behalf of many tasks. This means that the instrumentation methodology must be sensitive to concurrency issues.
- The process of instrumenting, benchmarking, and optimizing the operating system code must be kept simple enough to avoid delaying product release schedules.

Despite these obstacles, it is very important to be able to improve instruction cache and memory paging performance for operating system code, perhaps even more than for application code. Chen and Bershad[25] have shown that operating system code typically has less instruction locality and is more sensitive to instruction cache performance than is application code. Measuring the performance of two popular workstation operating systems running a number of industry standard benchmarks, Chen and Bershad found that the percentage of instruction cache misses attributable to the operating system exceeded 70 percent for over two-thirds of the benchmarks. This assertion appears to hold independently of application size and execution run time. They also found that the instruction cache penalty, measured as the number of instruction stall cycles divided by the number of executed instructions, was higher in system code than in user code for 20 of their 26 benchmark runs. For larger applications, however, this effect was not always as pronounced.

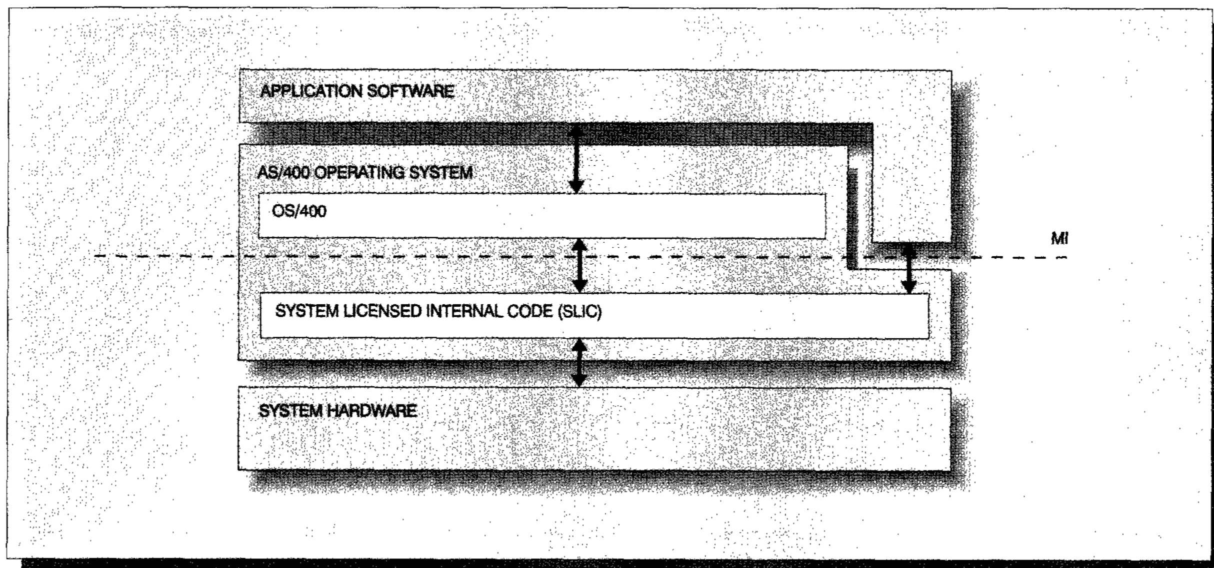
The primary contribution of this paper is to explain how we designed our profiling system to successfully restructure an operating system that could be shipped to customers and maintained in the customer environment without significant loss of performance. Although we describe our techniques with reference to the AS/400 operating system, the problems we faced are representative of those found on any operating system. We also describe improvements to known algorithms for restructuring code, and discuss how we handle issues of concurrency and indeterminate control flow. Although we use profile information in many of our optimization phases, this paper concentrates on its use in reordering code.

The remainder of this paper is organized as follows. We first provide background on the structure of the AS/400 operating system, and describe some of the special requirements it imposed upon our design. We then describe various types of profiling systems that have been developed in the past, and discuss why we chose the one we did. Next follows a detailed description of the profiling process we used on the operating system code, including the algorithms for instrumenting the code, the mechanisms used to collect data, the feedback mechanism for bringing the profile data into the compiler, and the instruction reordering algorithms. We then describe the support mechanism we use that allows us to maintain the code, and the benchmarks on which we measured the operating system. We conclude with some preliminary performance measurements, and thoughts for the future.

## The environment

The environment of the AS/400 is next described, followed by an introductory description of the sampling, trace-based, and instrumented methods for gathering data about the behavior of programs.

Figure 1 The AS/400 operating system and the machine interface (MI) layer




**The operating system for the AS/400.** The AS/400 architecture differs from that of most other computer systems in the level of abstraction exposed to its application software.[26] Whereas software for other systems is targeted directly to the hardware, applications view the AS/400 through an abstract machine layer known as the Technology-Independent Machine Interface (sometimes called TIMI, or just MI). Since the actual hardware and much of the system software are hidden beneath this layer of abstraction, it is possible to completely replace the underlying hardware and software without changing the application software. This is exactly what happened with the recent introduction of new AS/400 systems based on the PowerPC AS RISC (reduced instruction-set computer) architecture: the existing CISC (complex instruction-set computer) processors were replaced with RISC processors without requiring customers to acquire updates for their applications.

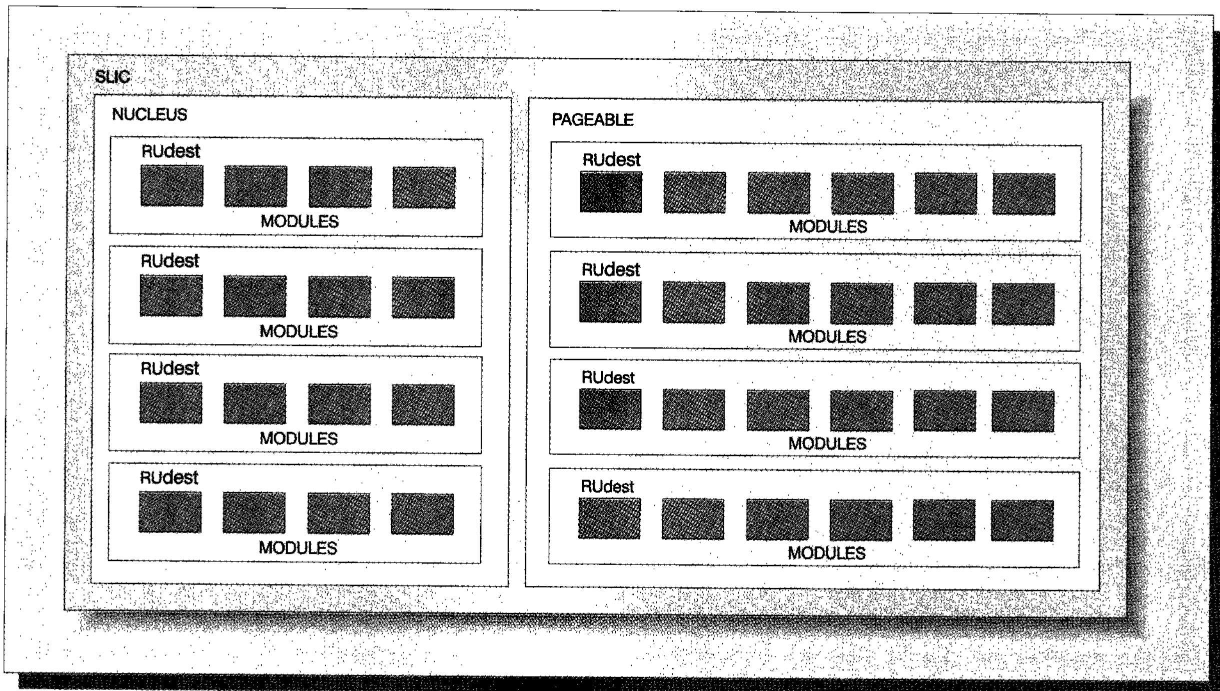
As shown in Figure 1, the operating system for the AS/400 consists of two parts. The portion known as Operating System/400* (OS/400*) resides in a software layer "above the MI." "Below the MI" is a layer known as system licensed internal code (SLIC), which is responsible for implementing the abstract MI functions for a specific hardware architecture.

When an application task runs on an AS/400, part of its time will be spent executing in the application code itself or in OS/400 and the rest will be spent executing within SLIC. Traditional transaction-based business software tends to spend most of its time in the integrated database software below the MI, while more computationally intensive applications spend most of their time above the MI. Because most of our customers primarily use traditional business applications, the performance of SLIC is crucial to the overall performance of the AS/400. SLIC modules, written primarily in C++ and a dialect of PL/I, are compiled through a state-of-the-art optimizing back end called *slicox* (the SLIC optimizing translator).

As shown in Figure 2, SLIC is partitioned into code that must be resident in main memory (the nucleus) and code that need not be (pageable). Each of these code sections is further subdivided into smaller contiguous regions called replaceable unit destinations (RUdests). A RUdest is composed of a number of modules (i.e., compilation units) that share some property requiring them to be kept together. The SLIC link/loader, itself a pageable component of SLIC, is responsible for positioning each module within its assigned RUdest and resolving all external references between modules.

After a release of the operating system has been shipped, it is inevitable that problems will be discovered in the field. Corrections for these problems are packaged into Program Temporary Fixes (PTFs) and

Figure 2 Organization of system licensed internal code (SLIC)




made available to customers. Each SLIC PTF consists of one or more modules that will replace existing faulty modules on customer systems. When customers apply a SLIC PTF to their AS/400, the SLIC link/loader is again responsible for positioning the new modules within the appropriate RUdests, and adjusting all external references to the replaced modules to reference the new ones.

One of our goals was to reorder procedures within SLIC so that their spatial packaging more closely matched their temporal locality. However, the design of SLIC and the PTF process impose some constraints on allowable procedure order.

- Each procedure is required to remain within its target RUdest; therefore two procedures in different RUdests cannot be juxtaposed, even if one calls the other very frequently.
- The process for applying PTFs was designed to replace entire modules. If procedures from a module were not placed contiguously, the job of the link/loader during PTF application would be greatly complicated.

Because of these considerations, we decided to reorder procedures within module boundaries, and order modules within each RUdest according to temporal affinity.

In the future, it may be worthwhile to allow free reordering of procedures across module boundaries. An early study[27] of SLIC behavior while running an internal workload approximating the Transaction Processing Performance Council (TPC-C**) benchmark[28] indicated that about 75 percent of all dynamic procedure calls occurred across module boundaries. Unfortunately, the study did not determine how many of these calls also occurred across RUdest boundaries, which would forbid reordering. The data also indicated a high affinity between pairs of procedures: On average, each procedure was called 83 percent of the time from a single caller, and each procedure made 60 percent of its calls to a single callee. This study, though preliminary, indicates that full procedure ordering would provide some incremental benefit over our current scheme that preserves module boundaries.

**Types of profilers.** The term *profiling*, as used in this paper, refers to using information collected about the dynamic behavior of a program to improve optimization of that program. The program is measured while running one or more benchmarks believed to be representative of the way the program will be used in practice. There are three typical models of profiling, distinguished by the method of gathering data about the behavior of programs.

- A *sampling* profiler operates using a hardware timer, periodically waking up a process that records the address of the currently executing instruction. Although sampling profilers can be adequate for recording which procedures are executed frequently, they do not work well for recording more granular information, such as how frequently a given branch is taken, or which procedures often call which other procedures.
- A *trace-based* profiler collects a hardware execution trace of the instructions executed by the program during the benchmark trials. It then reduces this information to a manageable size to determine branch and procedure call frequencies.
- An *instrumenting* profiler operates by recompiling the program with special instrumentation "hooks" placed at important branch points. As the instrumented program executes, these hooks cause data counters to be updated, recording the branch frequency information directly.

We considered the trace-based and instrumenting profiler models as candidates on which to base our design. We eventually decided upon an instrumenting profiler, because of the difficulties we perceived in using a trace-based profiler. First, a full instruction trace for a nontrivial benchmark would be quite large and time-consuming to process; in order to be practical, we would require a real-time reduction tool to compress the instruction traces into branch frequencies. Second, it would not be easy to map information from the instruction traces into the compilation process. Finally, we did not have a practical way to make a full execution trace tool available to our customers. Since our intent in subsequent releases is to make this technology available for customers to use on their own programs, this was a key consideration in our decision.

In contrast to these problems, the only major drawback of using an instrumenting profiler is its invasive nature: an extra compilation step is required to insert the instrumentation hooks. Although this is a nontrivial consideration because of the number of SLIC modules requiring instrumentation, this only needs to be done once per release of the operating system, whereas the drawbacks of instruction traces were more serious.

## Restructuring the operating system

We next discuss our process for gathering data and using those data to reorder and improve portions of the AS/400 operating system.

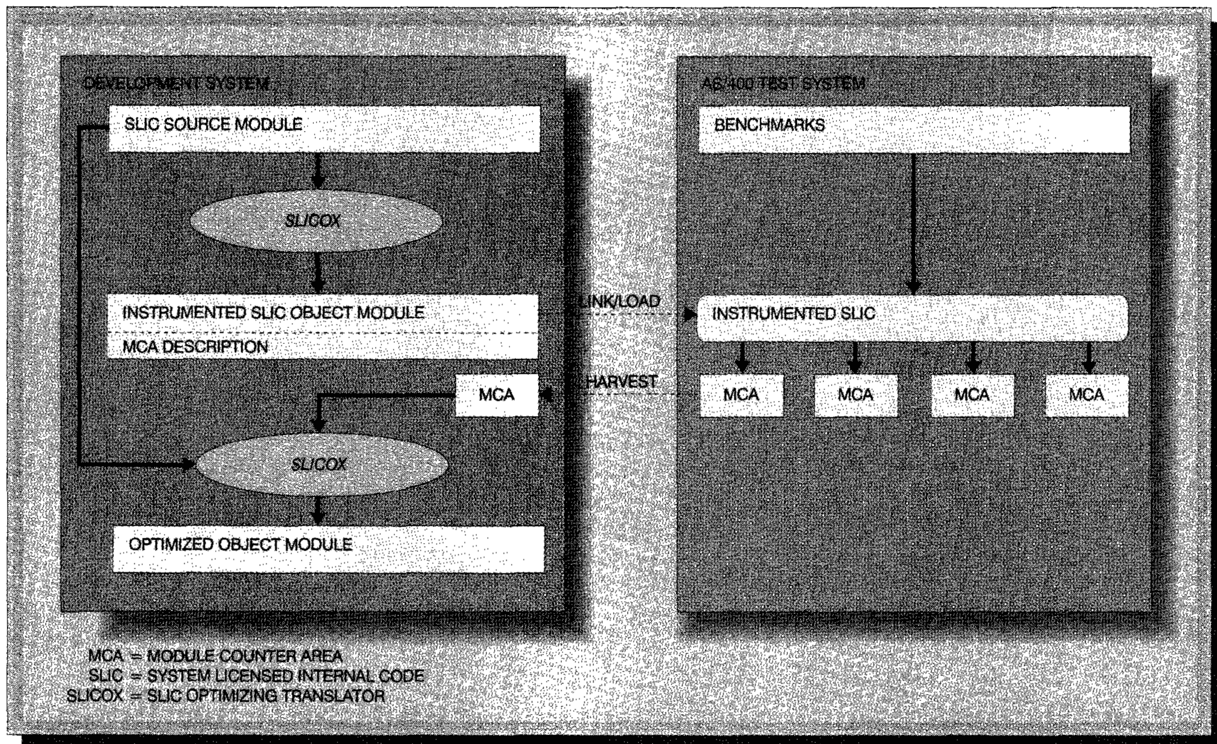
**FDPR design overview.** Our goal of feedback-directed program restructuring (FDPR) was to improve the performance of SLIC by more fully exploiting certain hardware characteristics of the AS/400 processors and memory hierarchies. By reordering instructions within a procedure so that they are likely to be executed in sequence, we can improve performance of the instruction cache by reducing cache pollution, improve the efficiency of sequential instruction prefetching, and reduce the penalty associated with taken or mispredicted branches. By packaging procedures in an order that reflects their temporal locality, we can also reduce the SLIC working set size, taking better advantage of the memory paging system.

FDPR is a three-phase process: instrumentation, benchmarking, and feedback-directed optimization. The following is a brief overview of this process, as illustrated in Figure 3.

Only those SLIC modules that are known to be crucial to some aspect of system performance are considered candidates for profiling. First each of these modules is instrumented by the *slicox*. The instrumentation process analyzes each procedure in the module at a particular point during translation, and inserts *snippets* of instrumentation code. Each snippet (or "hook") contains code that increments a counter whenever a particular control flow event (branch decision or procedure call) occurs. The counters for each module are stored in a static data object associated with the module, known as a module counter area (MCA).

After all modules to be measured have been instrumented, they are loaded onto an AS/400 test system, replacing the uninstrumented versions of those modules. As a result, space is also allocated for the MCAs of the instrumented modules. Special tools are used to clear all counters to zero, and to disable the instrumentation snippets from incrementing the counters until representative benchmarks are ready

Figure 3 Overview of feedback-directed program restructuring (FDPR)




to be executed. The snippets are enabled while the benchmarks run, and then disabled again when the benchmarks are finished, thus freezing the counters. Another tool then "harvests" the profile data by locating all MCAs on the system, saving each as a file, and transferring the files back to the development system.

There is now one file of profile data for each performance-critical module. Each module is once again compiled, this time using the profile data to guide optimizations. Each counter is read from the file and associated with the control flow event that it was monitoring. Based on these data, the *slicox* determines an optimized order for the instructions for each procedure, favoring sequential flow of control through heavily traversed paths. It also determines an optimized packaging order for the procedures within the module, based upon measured intramodular procedure calls. A separate tool then produces a suggested module ordering within each RUdest, based upon measured intermodular procedure calls. The optimized modules are loaded according to the module ordering to form a system image for distribution to customers.

**Module counter areas.** Traditional profiling mechanisms typically do not require any sophisticated organization for the profile counters. They are designed to optimize a single executable object, which is not expected to be modified after profile data have been applied, so a simple linear array of counters can suffice. There are two considerations that render this impractical for our purposes. First, fixes to SLIC are made at the module level; an entire module must be replaced for every change. If modules that have been optimized using profile data are replaced by corrected versions of those modules without such optimization, a noticeable loss of performance might result over time. To reduce the potential for performance loss, we designed our module counter areas to segregate and identify the counters pertinent to each procedure. Thus when a single procedure is modified, or when procedures are added to or deleted from a module, the profile data for the un-

changed procedures can still be considered valid and used to direct optimization.[29]

Second, much of the code in the nucleus is not permitted to incur a page fault on any memory access, so counters for nucleus code must reside in "pinned" (unpageable) memory. Since pinned memory is a finite resource, however, it would not be prudent to place counters for nonnucleus code in pinned memory. At a minimum, the nucleus and nonnucleus counters must be kept separate.

A simple solution to this problem is to allocate the counters inside a data object created by the compiler in the static data space associated with the module. In SLIC, static data for all nucleus code are allocated in pinned memory, while static data for nonnucleus code are allocated in pageable memory. Each module then addresses its counters via offsets from the base address of this module counter area. In addition to solving the forbidden page fault problem, this allocation scheme also eliminates any dependency on the compilation order; the counters for each module are always at fixed positions from the beginning of its own MCA.

The layout of a module counter area is shown in Figures 4 and 5. Each MCA consists of an initial header area, followed by a procedure counter area (PCA) for each procedure contained in the associated module. The header area contains information used to interpret the rest of the MCA, including the total size of the MCA, and the number and location of the PCAs within the MCA. Each PCA consists of a header and several groups of counters, used to measure branching events within a procedure (control flow counters), direct calls to known procedures (direct call flow counters), pointer-based calls to possibly unknown procedures (indirect call flow counters), and invocations of the current procedure (prologue counter).

**Figure 4** Contents of a module counter area (MCA)

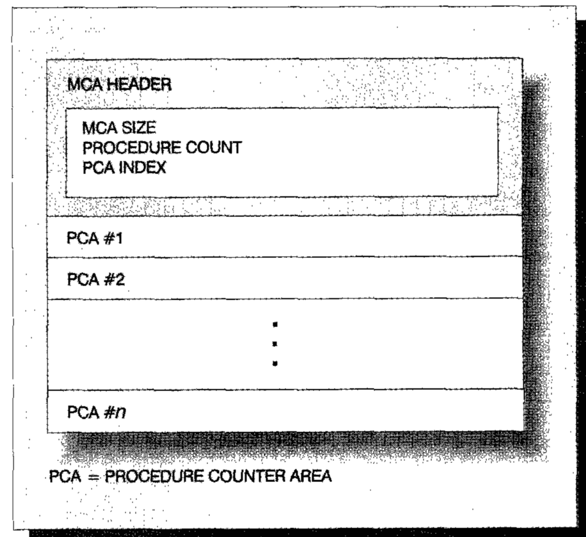
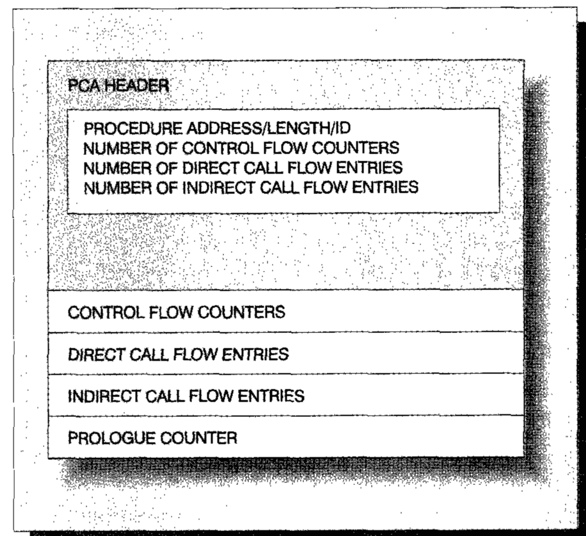



**Figure 5** Contents of a procedure counter area (PCA)




**Instrumentation methodology.** The purpose of instrumentation is to determine the number of times particular control flow events occur during the execution of a program. The data collected will later be used to guide optimizations. Obviously, the compiler must view a procedure to be identical when it inserts the instrumentation snippets and when it subsequently reads the profile data back in. Since modern compilers perform many optimizations that can alter the control flow of a procedure, it is mandatory that the instrumentation and feedback phases take place at the exact same point during compilation, and that all prior phases of compilation produce precisely the same results when the instrumentation and feedback options are selected.

There are four types of instrumentation hooks that match the four kinds of counters: control flow hooks, direct call flow hooks, indirect call flow hooks, and prologue hooks. Direct and indirect call flow hooks

are placed just prior to the corresponding procedure calls in the instruction stream. A prologue hook is placed in the prologue code for each procedure (compiler-generated code that sets up for each invocation of a procedure). Control flow hooks are placed along arcs in the control flow graph (CFG) for the procedure.

A control flow graph for a procedure is an abstraction produced by a compiler to represent possible flow of control through an instruction stream. It is constructed as follows. First the compiler partitions the instructions of the procedure into *basic blocks*. A basic block is a contiguous sequence of instructions that will always be executed together. That is, a branch into a basic block can only target the first instruction of that block, and any branch appearing in a basic block must be the last instruction in that block. Each basic block is represented by a node in the control flow graph. There is a directed arc from block A to block B if and only if block B can be executed immediately after an execution of block A. An example of a CFG appears in Figure 6A.

Note that Figure 6A contains two artificial nodes marked Start and Exit, and an artificial arc from Exit to Start. Any block representing an entry point into the procedure is made a successor of the Start block. Any block at which control may leave the procedure (by returning or by an unhandled exception, for example) is made a predecessor of the Exit block. The use of the artificial blocks and arc ensures that, for every node in the graph, there is a path from that node to itself; that is, the graph is strongly connected. This is a necessary property for use of the spanning tree algorithm described in a later subsection on controlling instrumentation cost.

Profile data indicating path frequencies can be represented in the CFG in one of two ways. *Block weights* indicate the frequency with which each block is executed, while *arc weights* indicate the frequency with which each arc is traversed. Block weights can always be derived from arc weights, but the converse is not generally true. Since the frequency with which a conditional branch is taken is important to the basic block reordering optimization, we directly instrument the control flow arcs. This is done by inserting new basic blocks containing instrumentation hooks (diamond-shaped boxes in Figure 6C) along selected control flow arcs.

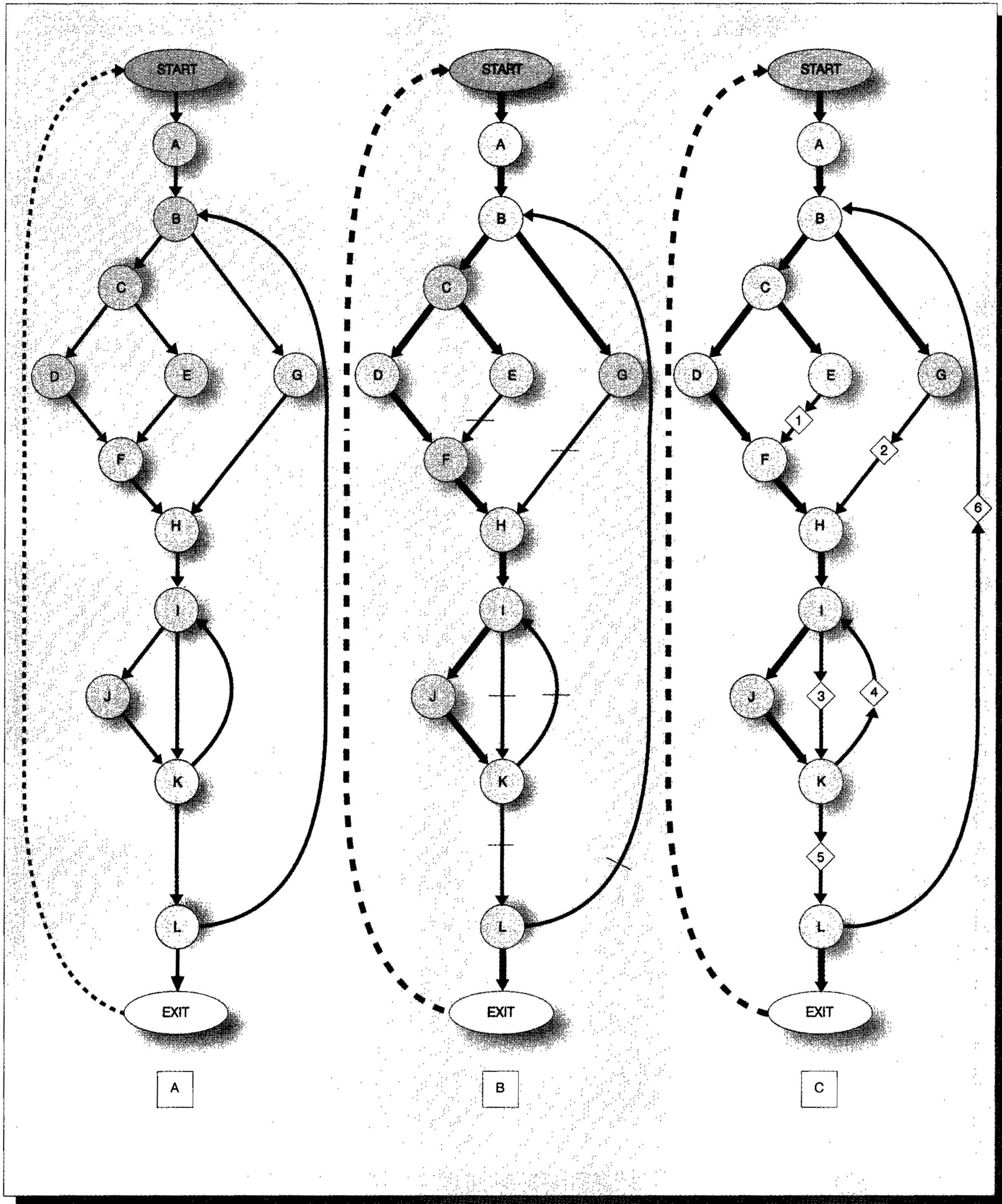
*Instrumentation hooks.* In our implementation, a control flow hook typically consists of three instructions to load, increment, and store a 64-bit counter. The counter is addressed at a fixed offset from a base register. Prologue code executed when the procedure is invoked stores the address of the PCA for the procedure in the base register. The base address of a PCA is determined by adding an offset to the static address of the MCA for the containing module, determined by the link/loader at load time. The prologue code also increments a counter to record the number of invocations of the procedure.

The code inserted prior to a direct procedure call is identical to that for a control flow hook. The only difference is where the counters are stored. Recall that direct call flow counters are segregated from the control flow counters. This is because each call flow counter contains an additional field identifying the procedure being called. This information is used later to reconstruct the system-wide weighted call graph for the benchmark.

The code for an indirect call site, however, is quite different. In general, we cannot know which procedures, or even how many different procedures, may be called at an indirect call site. Previous researchers[18] have ignored indirect call sites because of this difficulty. We chose to create a fixed-size table of callees and counts for each indirect call site. The management of the table is embodied in a system-wide subroutine that takes, as parameters, the address of the procedure to be called, and the table in which counts are to be recorded for the call site. Since the number of called procedures may exceed the size of the table, a method is needed to ensure that the most frequently executed procedures are kept in the table. The management of the table is beyond the scope of this paper.

The question of when during compilation to insert instrumentation code is an interesting one. The answer often depends on the intended use of the profile data. Obviously profile data should be collected prior to performing those optimizations that can benefit from use of the data. In our case, the initial use of profile data within the *slicox* was to reorder basic blocks within each procedure; this is a very late optimization, so for its purposes profile data could be collected just prior to final assembly of the instruction stream. However, we also had heuristic uses for profile data in the register assignment and global instruction scheduling phases of the *slicox*, so we chose to collect data prior to these phases.

**Figure 6** Instrumenting a control flow graph (CFG). (A) CFG example. (B) CFG with spanning tree and potential places for hooks identified (hash marks). (C) CFG enhanced with instrumentation blocks (diamond-shaped boxes) along nonspanning-tree arcs.



Since many optimization phases execute after instrumentation code has been inserted, the amount of code that must be handled by those phases is increased. To minimize this bloat, instrumentation hooks are initially treated as single-instruction macros in the intermediate representation (IR), and are only expanded into the form described above during final instruction assembly. To provide for efficient register utilization, each macro is annotated with unique virtual registers to represent the registers needed when the macro is expanded. The register allocator is free to select appropriate hardware registers for these virtual registers according to the usual methods.

*Controlling instrumentation cost.* Care must be taken to minimize the cost of the instrumentation hooks, since long-running benchmarks are required to provide useful profile data for an operating system. If the instrumented version of the operating system were too slow, profiling would be rendered impractical. Time spent executing instrumentation code can also perturb timings, task queue lengths, and so forth. This means that (1) each instrumentation hook must be as efficient as possible, and (2) the number of hooks must be kept to a minimum.

There is a well-known solution to the hook-minimization problem.[30-33] The idea is to identify a small subset of the arcs in a control flow graph for a procedure such that, if we knew the weights of those arcs, we could infer the weights of all remaining arcs in the graph. The trick is to observe that the flow into a block must equal the flow out of that block. If a given arc is the only one with unknown weight incident to a block, then the weight of that arc can be inferred from the known weights of other arcs incident to that block. We thus only need to instrument this subset of the arcs from which the other weights can be inferred.

Rather than looking for the arcs we want to instrument, it is convenient to identify those that we do not want instrumented. Suppose that we ignore the directions of arcs in the CFG, and that we select some subset of the arcs such that there is no path in the subset from a block to itself; that is, the subset forms one or more *trees*. Suppose further that we have known weights for all arcs *not* in this subset. A property of a tree is that there must be at least one arc in the tree that touches a block touched by no other arc in the tree. As just described, the weight of such an arc can be inferred from the weights of other arcs incident to the block, which are known since those arcs are not in the tree. Once that weight is known, it can be removed from the subset of unknown-weight arcs. The remaining unknown arcs will still form one or more trees, so the process can be repeated until all weights are known. An example can be found in a later subsection on feedback of profile data.

To instrument the fewest arcs, we then need to select the largest possible tree of arcs *not* to be instrumented. In a connected graph having $N$ nodes, the largest possible tree will have $N - 1$ arcs. Such a tree is called a *spanning tree*, since it touches every block in the CFG. The arcs to be instrumented, then, are the arcs *not* in the spanning tree. Many possible spanning trees exist for a strongly connected graph, any one of which can be arbitrarily selected, provided that it includes the artificial arc from Exit to Start (which cannot be directly instrumented).

Figure 6A shows an example control flow graph to be instrumented. In Figure 6B, a spanning tree for the CFG has been arbitrarily selected; the arcs in the spanning tree appear as bold lines to identify them. Those arcs *not* in the spanning tree are identified with a hash mark, and are the ones to be instrumented. Figure 6C shows the modified CFG with the instrumentation blocks inserted (shown as diamonds). Note that the number of instrumented arcs is much smaller than the total number of arcs in the original graph.

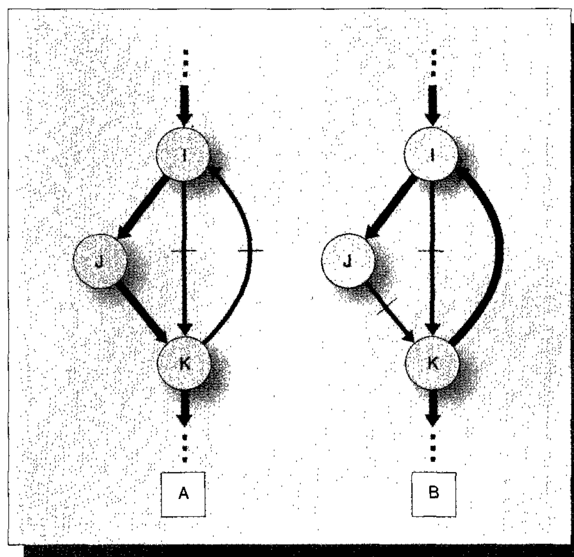
In most cases it is straightforward to add an instrumentation block along an arc in the control flow graph. Suppose that the original arc originates at some block $X$ and targets some block $Y$. Then $X$ will either end with a conditional or unconditional branch that targets $Y$, or $X$ will fall through into $Y$. In the case of fall-through, it is simple to add the instrumentation hook between blocks $X$ and $Y$. If $X$ ends with a relative branch that targets $Y$, a new block $I$ is created to hold the instrumentation hook followed by a branch to $Y$, and the branch in $X$ is modified to target $I$.

A problem not discussed in the literature occurs when $X$ ends with a branch to an unknown code location contained in a register (a nonrelative branch). In some cases, the compiler may be able to statically determine the location or locations that the branch can target, and thus create arcs in the CFG to represent these paths; but in the general case, this is not possible. It is necessary to add artificial arcs to the CFG to ensure that the graph remains connected in

the presence of nonrelative branches, since some targets of such branches may not be reachable by any other path. Since these artificial arcs cannot be executed, the spanning tree must be chosen so that these arcs are not instrumented. In rare cases, though, the complement of every spanning tree will contain an artificial arc. When this occurs, the blocks incident to the artificial arc must be instrumented instead.

There are many possible spanning trees for a control flow graph. Selecting one that results in minimal instrumentation cost is an NP-hard problem,[34] so a heuristic approach is warranted. Ball and Larus[31] made use of static weight estimates to reduce the expected cost of instrumentation hooks. We implemented a very simple heuristic that reduces the cost of instrumentation within loops by avoiding instrumenting back arcs (flow of control from within a loop to the beginning of the loop). Figure 7 shows an example loop from the flow graph of Figure 6. The "natural" depth-first algorithm for finding a spanning tree produces the results shown in Figure 7A, with arcs I → J and J → K in the spanning tree, and arcs I → K and K → I in the complement of the spanning tree; thus the back arc K → I will be instrumented (with diamonds in Figure 6C). Our algorithm prefers to avoid instrumenting back arcs, so it exchanges the back arc K → I with another arc incident to K, in this case J → K. The result is shown in Figure 7B. We only avoid instrumenting a back arc in this manner when we believe the alternate arc (e.g., J → K) will be executed less frequently than the back arc, using static heuristics.

**Figure 7 Avoiding instrumentation of back arcs. (A) Usual spanning tree. (B) Revised spanning tree. Note that J→K will be executed no more frequently than K→I.**



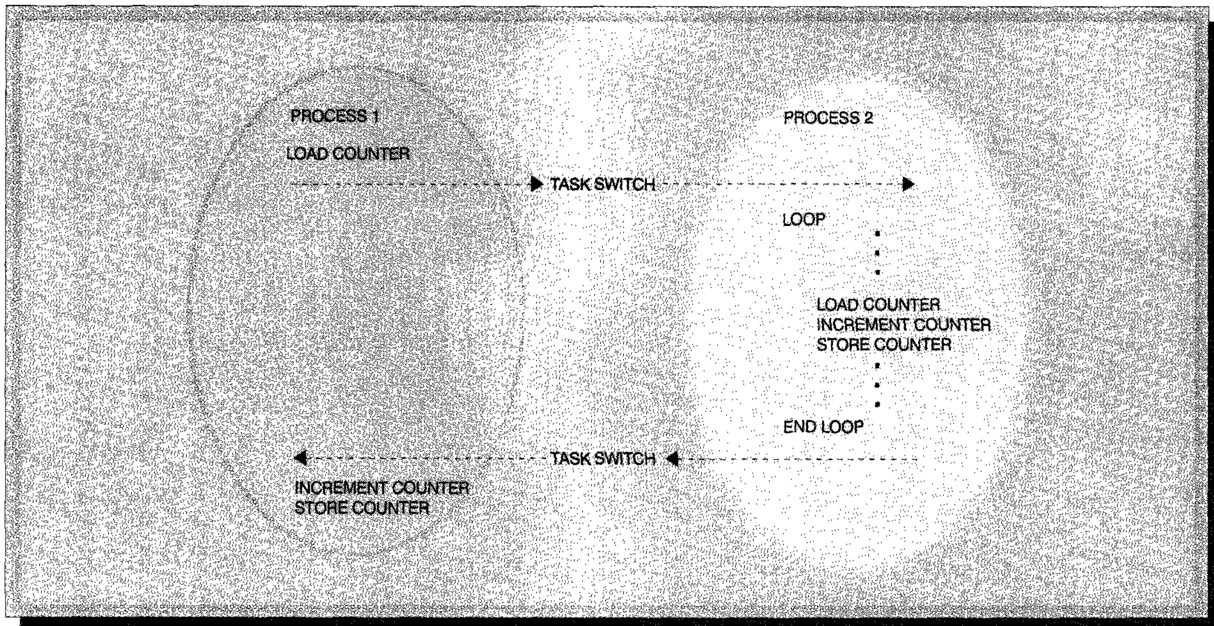

*Concurrency issues.* Another difference between an operating system and most applications is the multitasking nature of the operating system. It is quite common for a single procedure in SLIC to be operating concurrently on behalf of several user processes. This raises the possibility of data loss for any given counter, as illustrated in Figure 8. If the load, increment, and store are not treated as an atomic operation, one process can be switched out after executing the load, another process can execute for a time, possibly updating the counter many times, and then the first process can regain control to execute the increment and store. This means that increments of this counter by the second process will be lost.

Unfortunately the code to perform an atomic update on the AS/400 is much more expensive than a simple increment, requiring roughly three times as much time to execute and over twice as many static instructions. We therefore decided to allow occasional data losses to occur, relying on the length and repetitiveness of our benchmarks to smooth out the losses. This proved to be a successful strategy, but one with implications for the profile feedback step, as discussed in a subsequent section.

More care was needed for the indirect call site instrumentation. In order to manage the policy determining which procedures will occupy the table for a given call site, some static data are maintained within the table. These data must be manipulated atomically in order to avoid corruption of the table. Since indirect calls are relatively infrequent, we implemented a semaphore with each indirect call site table to ensure atomic access. The overhead of the semaphore is less important here, since its cost compared to the time required to manage the table is relatively small.

By implementing only simple increments instead of atomic ones, and by using the spanning tree technique to minimize the number of counters, we were able to limit the execution time overhead of instrumentation to a very acceptable level. For example, for our internal version of the TPC-C benchmark, we measured the number of simulated users required

Figure 8 Loss of data due to concurrent counter access




to achieve CPU saturation. An instrumented version of SLIC attained saturation with roughly 33 percent fewer users compared with an uninstrumented version. The average size of a module increased by roughly 78 percent when adding instrumentation, counting both the added instrumentation and the module counter areas.

*Controlling data collection.* One way in which operating systems and other large multitasking applications differ from smaller executable objects is that a significant amount of setup time may be required before the benchmarks are ready to be run. An operating system in particular must be active during this setup phase. It is undesirable, though, for the profile of the benchmark to be "polluted" by counts that have accumulated during setup. One way to deal with this problem is to provide a method to reset all counters to zero when the setup phase is complete. However, the number of counters used in an operating system can be very large, so that by the time all counters have been cleared, some of them will have again accumulated significant counts. Furthermore, accesses to the counters will cause memory paging activity, which may distort the counts accumulated on behalf of the page fault handling software.

To avoid these problems, we globally dedicated a bit from the condition register of the processor for use as a *profile enabling bit*. This bit is tested within each instrumentation hook to determine whether counts should be accumulated. Although any processor register bit could have been dedicated for this use, a bit in the condition register was the ideal choice, since condition register bits can be tested directly by conditional branch instructions. With this bit available, we were able to disable the instrumentation hooks, set all counters to zero, perform setup for the benchmarks, and then enable the hooks just prior to running the benchmarks. Since one bit is used to control all instrumentation hooks, the effect of enabling or disabling profiling is instantaneous.

**Data collection tools.** After the high-use modules have been compiled to insert instrumentation, they are loaded onto an AS/400 system for the data collection stage. A tool with four functions was created on the AS/400 to facilitate data collection. Two of the functions simply turn the profile enabling bit on and off to determine whether the instrumentation code should be executed. A third function finds all module counter areas on the system and initializes their counter fields to zero. The last function again finds all module counter areas on the system, and extracts

them for transfer to the development machine for use in optimization. Each module counter area is stored in a separate file and given a name based on the module with which it is associated.

**Feedback of profile data.** Once the profile data have been transferred to the development platform, they are ready to be used to guide optimization of the code. Each profiled module is recompiled using a special option indicating that profile data should be read in and used. As each procedure in a module is compiled, the *slicox* searches the MCA for each module to find the PCA for that procedure. It then locates the control flow counters within the PCA (call flow counters are ignored during compilation of a procedure). Recall that each control flow counter corresponds to one of the CFG arcs selected for instrumentation. The algorithm used to determine the arcs to be instrumented is run again during the feedback phase to determine which arcs should be assigned the weights collected in the control flow counters. Figure 9A shows a possible weighting assigned to the instrumented arcs from the example in Figure 6, as modified in Figure 7.

The next step is to use the weights from the instrumented arcs to determine the number of times each of the remaining arcs was traversed. As discussed by Knuth,[30] we can repeatedly select a node with only one incident arc that has not yet been assigned a weight, and determine the weight of that arc by Kirchhoff's first law,[35] which states that flow is conserved at any point in a network. Since the uninstrumented arcs form a tree, there will always be such a node to select, and this algorithm will succeed in determining the weights for all uninstrumented arcs. Figure 9B shows that node K can be selected first in this example, and that the arc from node K to node I is determined to have weight 400 to satisfy conservation of flow. Figure 9C shows the full elaboration of the arc weights for this CFG.

Once all arc weights have been determined, the weights are recorded in the intermediate representation (IR) of the procedure. To facilitate subsequent control flow optimizations, the arc weights are not stored in the CFG, but directly in the IR instruction stream. Each unconditional branch in the instruction stream is annotated with the weight of its corresponding arc. Each conditional branch is annotated with two arc weights, indicating the frequencies with which the branch was taken and not taken. For multiway branches (such as might be generated for a C switch statement), we store the weights in a separate branch table within the IR.

A useful side effect of branch annotation is that SLIC programmers can see the branch frequencies in their program listings. Since each instruction in the low-level IR typically corresponds to a single PowerPC AS machine instruction, we display both sets of instructions in our program listings. By careful examination of these listings, programmers can make inferences about control flow patterns, such as the average number of iterations executed per entrance to a loop. This can be used, for example, to make informed choices among alternative data structures.

Determination of all arc weights according to Kirchhoff's law is easily performed when all arcs can be instrumented. Recall, however, that some arcs cannot be directly instrumented due to the presence of indirect branches. In such cases we inserted control flow hooks directly within basic blocks incident to the uninstrumentable arcs. We therefore needed some place to store these block weights during feedback, since they could not be annotated on branches like the rest. For this purpose we introduced a new IR instruction called a *profweight*. During feedback, each directly instrumented block has a profweight inserted at the beginning or end of the block, indicating the number of times the block was entered or exited (the distinction can be important in the presence of exceptions). In almost all cases, this permits us to infer the weight of each arc in the CFG. In cases such as that shown in Figure 10 where we cannot be certain of the exact weights of certain arcs, we must arbitrarily assign weights to them that satisfy conservation of flow.

A more serious concern arises from the possibility of data loss due to concurrency. Suppose that the counter for arc $G \rightarrow H$ in Figure 9A suffered a loss in count of 60, as shown in Figure 11A. Then the elaborated arc weights would appear as shown in Figure 11B. Note that the data loss has caused a change in perception of the likely path of the branch at the end of block B, and in the weights along paths from B to H, but that this effect is localized and has not changed the rest of the graph. Fortunately, we have found the occurrences of data loss to be relatively rare, and the amounts of data lost relatively low, so that by using long-running benchmarks, the effects of data loss are quite small in practice. (The pedagogical example of Figure 11 is extreme in that a majority of the count for the affected arc was lost.)

Figure 9 Inference of control flow graph (CFG) weights from a subset. (A) Weights assigned to the complement of a spanning tree. (B) Calculating conservation of flow at node K. (C) Fully weighted CFG.

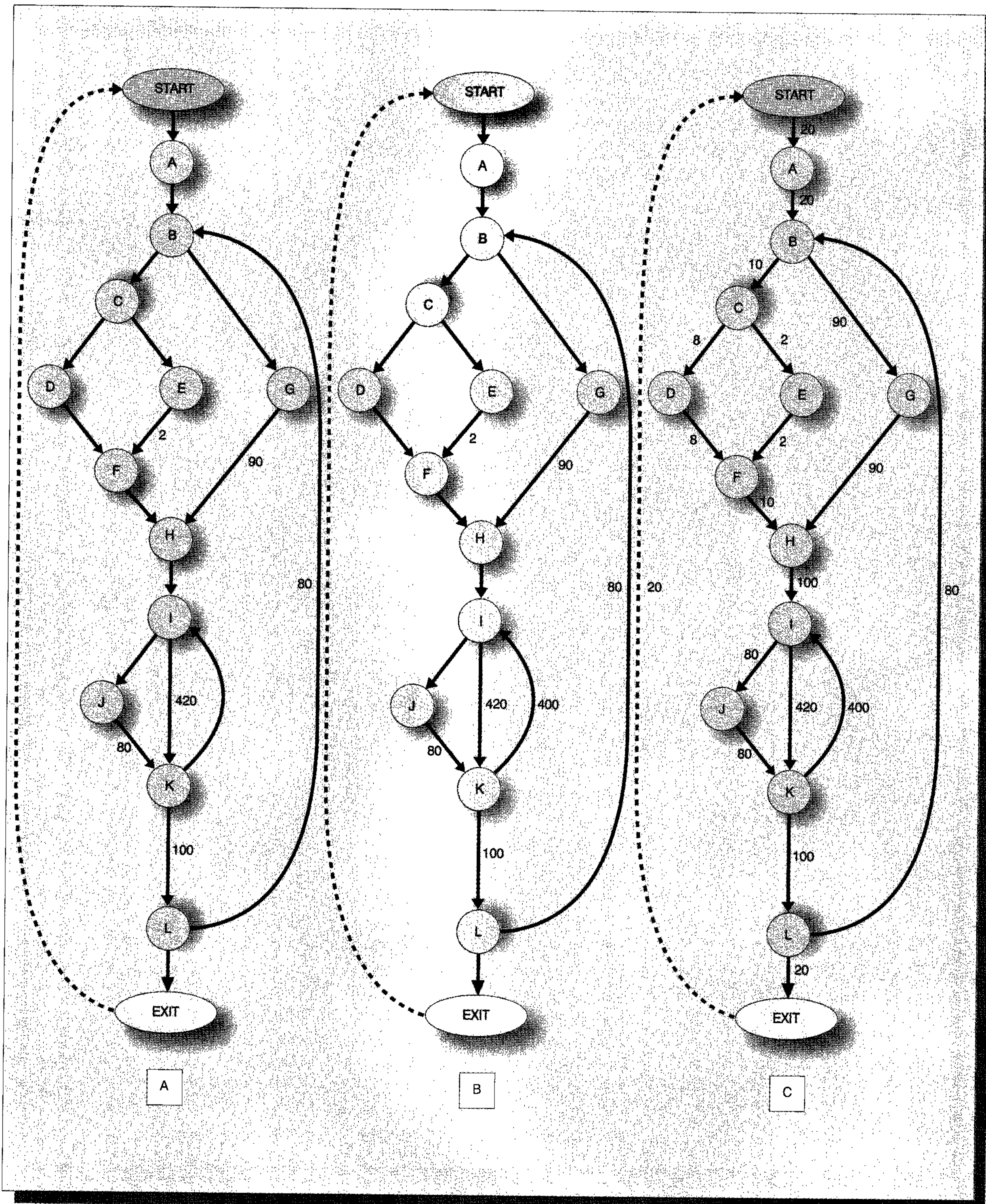
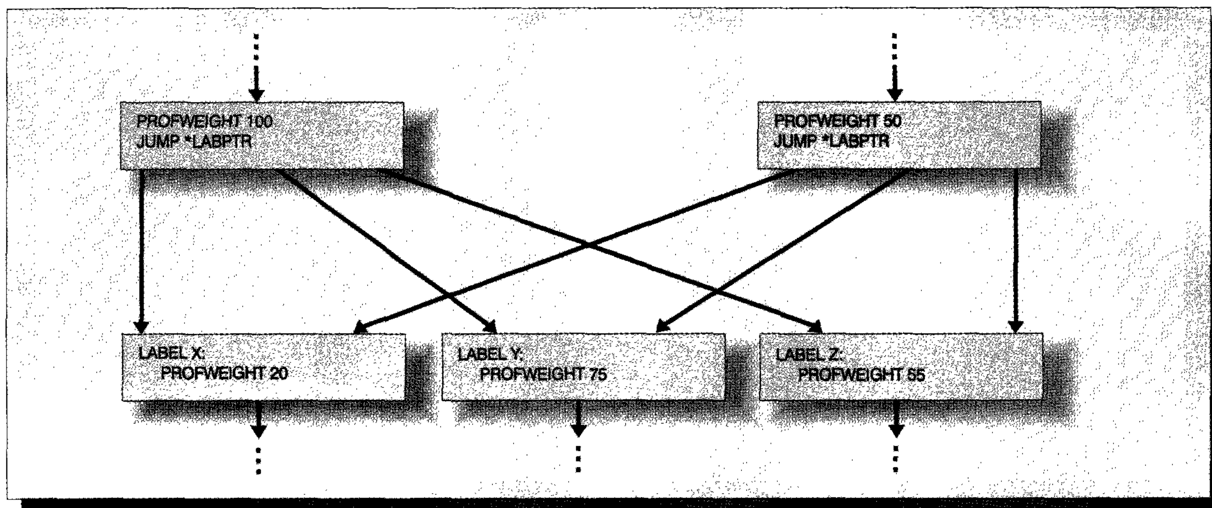


Figure 10 **Indeterminate arc weights. The five node weights are given, but the exact arc weights cannot be determined.**




After branch and profweight annotations have been inserted, subsequent optimizations such as register allocation and instruction scheduling take advantage of the weights. Any optimization that changes the CFG must maintain the branch annotations correctly.

**Reordering basic blocks.** After most other optimization phases have completed, the *slicox* analyzes the weighted CFG to determine an optimized order for basic blocks that attempts to maximize sequential control flow. That is, blocks are positioned so that, insofar as possible, conditional branches will usually not be taken. Our algorithm is largely based on the greedy algorithm identified as algo1 by Pettis and Hansen.[18]

*Use of the greedy algorithm.* The greedy algorithm operates by constructing a new basic block order from the current order, in which it attempts to find long traces of basic blocks that are likely to be executed in sequence. It begins by selecting the procedure prologue (entry point) block as the *seed block* for the first trace. For each trace, the algorithm first attempts to extend the trace backwards by searching for a predecessor of the seed block that has not yet been reordered; if multiple candidates are found, the predecessor whose arc to the seed block has highest weight is selected. This block is placed prior to the seed block in the new order, and the process repeats for the newly selected block. This phase terminates when no unreordered predecessor can be found. Note that this backward extension of the trace will be vacuous for the first trace, since the prologue block for a procedure will not have any control flow predecessors.

The trace is then extended forward from the seed block in a similar manner. Candidates to follow the seed block are those of its control flow successors that have not yet been reordered. If multiple candidates are available, the one with highest weight along the arc from the seed block to the candidate is selected, and the process repeats for the newly selected block. The trace terminates when no unreordered successor can be found.

Each time a trace has been terminated, a new seed block must be determined for the next trace. Since we want to place code together that is likely to execute closely together in time, we consider only those blocks that are successors or predecessors of blocks that have already been placed in a trace. Each such block is assigned a priority value, computed as the sum of weights of all arcs incident to the candidate block and to some previously reordered block. The block with highest priority is selected as the new seed block, and the trace selection algorithm repeats. Each trace is placed contiguously following the previously generated trace.

Figure 12A shows the traces selected for a sample CFG, with the new block order shown in Figure 12C.

Figure 11 **Effect of data loss on weight inference. (A) A count of 60 is lost from arc G→H. (B) Resulting inferred weights (compare with Figure 9C).**

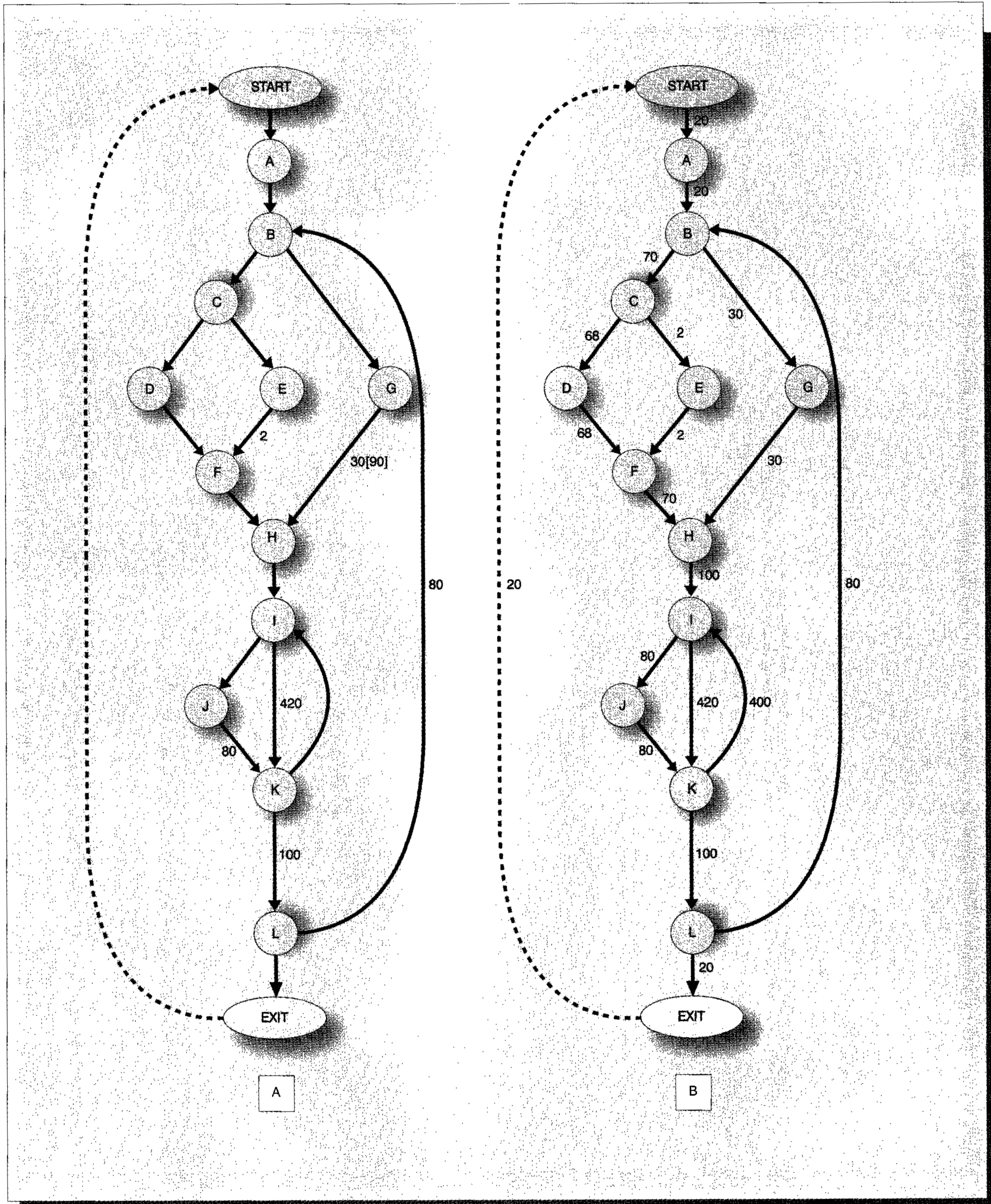
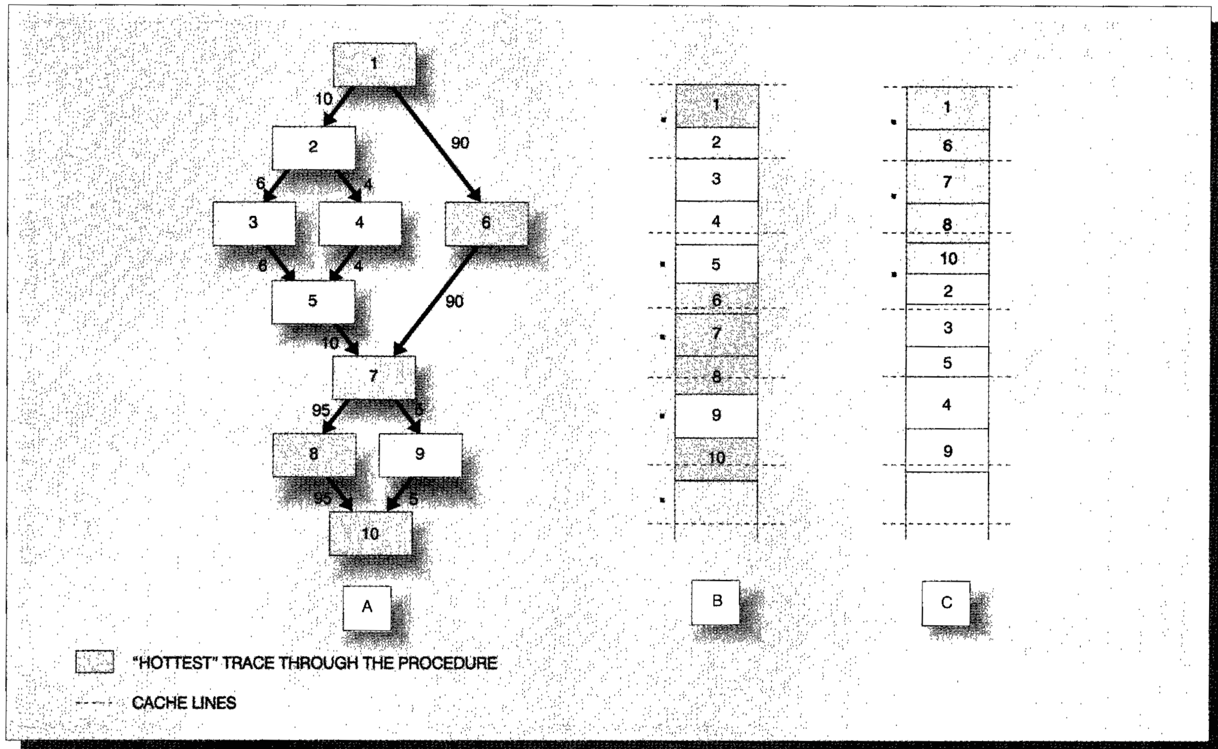


Figure 12 Trace selection in a weighted CFG. (A) Selecting a trace. (B) Mapping to cache lines for naive code generation order. (C) Mapping to cache lines after profile-based reordering.




Compare this to the naïve code generation order of Figure 12B. Note that the blue-shaded blocks represent the "hottest" trace through the procedure. The dashed lines in these figures show how the basic blocks might be mapped to instruction cache lines. Note that executing the hot trace when placed in naïve order requires touching five cache lines, while the optimized order requires only three.

*Modifications to the greedy algorithm.* We chose to limit the greediness of the Pettis and Hansen algorithm by sometimes terminating traces even when there are candidate successors for the last block in the trace. One reason for truncating a trace is if the best candidate successor is executed much less frequently than the last block in the trace. This can occur when the preferred successor of the last block has already been reordered. It is sometimes the case that it would be better to begin a new trace to pick up blocks that are executed much more frequently. Figure 13 shows an example of this. Suppose that the current trace has been extended to include blocks

Figure 13 Limiting the greedy algorithm

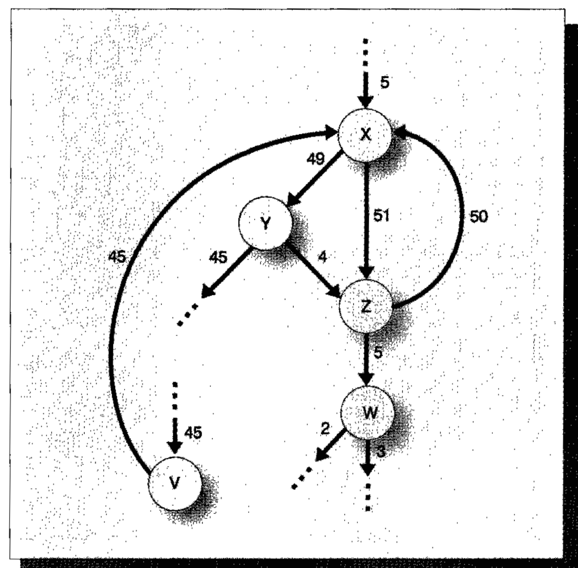


**Figure 14 Limiting the greedy algorithm using the perfect partner heuristic**

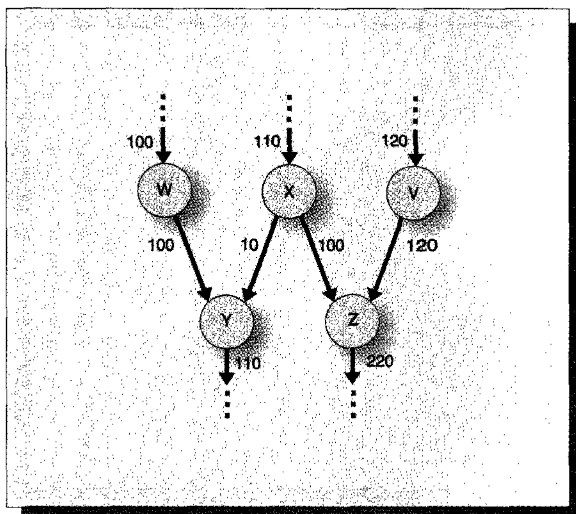
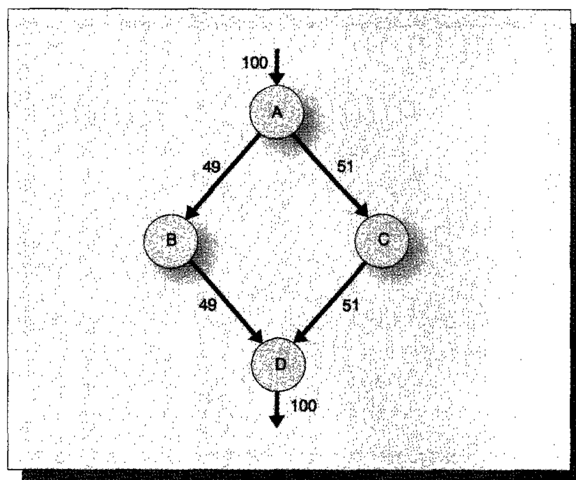



**Figure 15 The isolated block optimization**




X and Z. Note that the preference X has for Z over Y is statistically insignificant, and that block W was executed much less frequently than either Y or Z. Once block Z has been reordered, it will usually be preferable to start a new trace with block Y rather than continuing on with block W, so that the hotter trace beginning at Y is packaged close to its predecessor X.

Another reason to truncate a trace is to avoid reordering a successor of a block when that successor would actually "prefer" to be reordered after a different block that has not yet been reordered. The greedy algorithm always looks at candidates in only one direction; when scanning backwards to extend a trace, it determines which predecessor a given block prefers, and when scanning forward, it determines which successor a given block prefers. No consideration is given to preferences of the candidate blocks. Figure 14 shows an example of this. Assume that blocks V and Z have already been placed in a previous trace, block X has just been added to the current trace, and blocks W and Y have not yet been reordered. The greedy algorithm would reorder block Y after block X, regardless of the preference of block Y to follow block W. Our modified algorithm will only extend a trace when the candidate block is a *perfect partner*—that is, if the current block prefers the candidate block, and the candidate block also prefers the current block.

Another modification to the greedy algorithm was implemented to improve performance for cases such as the one shown in Figure 15. The greedy algorithm will select a trace containing blocks A, C, and D, and possibly many more blocks, before reordering block B, even though arc A → B is executed almost as frequently as arc A → C. If block B is reasonably small, placing block B a long distance away from blocks A and C would clearly not make the most efficient use of the instruction cache. A more efficient ordering would be ACBD. Our algorithm detects isolated blocks such as B as follows. Whenever we consider adding a successor block D to a trace, we check to see if (1) D has an unreordered predecessor B, (2) B is isolated (has no unreordered predecessors and no successors other than D), (3) B contains relatively few statements, and (4) the execution frequency for B is not negligible relative to the frequency of the predecessor for D in the trace (C in our example). If all of these conditions hold, the current trace is ended without adding D, and a new trace is started with B as the seed block.[36]

Sometimes it is important to keep two blocks together in their original textual order. We introduced a ShouldFollow flag to indicate that a relationship of a block with its textual predecessor is important. For example, the instruction prefetching mechanism implemented in the PowerPC AS A30 processor is constrained (due to limited resources) when branch instructions are executed on two consecutive cycles. One way that this can happen is if a return from a

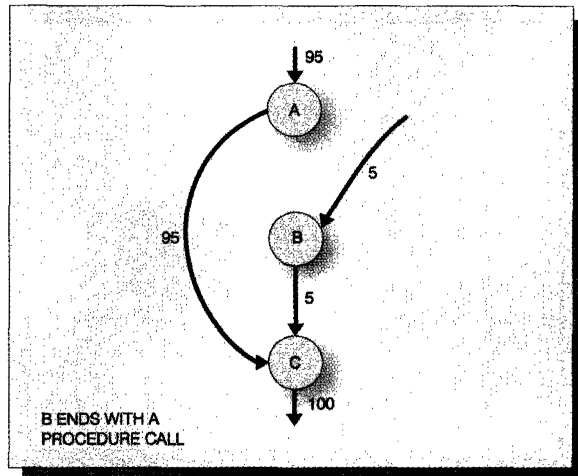
called procedure is immediately followed by a branch. To reduce such occurrences, we detect when a block *B* ends with a procedure call and mark its textual successor *S* with the ShouldFollow flag, since if any other block were placed after B, we would have to add an unconditional branch to S following the procedure call.

The ShouldFollow flag should only be honored when the block containing the call is executed relatively frequently. Consider the example in Figure 16. Even though ordering block C after block A would cause a branch to be added after the procedure call in block B, this is still preferable to the order ABC, which would require keeping the branch at the end of block A, because the arc from A to C is taken much more frequently than the arc from B to C. That is, the penalty of the back-to-back branch occurs rarely, and is therefore not important, whereas the presence of the extra branch in A may be costly.

Whenever a block marked as ShouldFollow is about to be added to a trace, we check whether its textual predecessor was the last block added to the trace. If not, we only add it to the trace if the weight of the arc from its textual predecessor is negligible in comparison with the weight of the arc being followed in the trace. The requirements for negligibility can be tuned heuristically; we currently use a ratio of ten to one.

Although much of SLIC was rewritten in C++ for the PowerPC AS processors,[37] quite a bit of legacy code dating back to the System/38* (the predecessor to AS/400) still remains. Much of this code is written in a variant of PL/I that is very different from languages in common use today. One of the features of this PL/I variant language is the ability to specify that an exception handler should be enabled over a given textual range of the source code. The internal implementation of this exception model requires that the machine code covered by such an exception handler must also remain contiguous. This places severe limitations on block reordering. We were forced to modify the trace selection algorithm to treat each textual exception range within a procedure separately. The algorithm considers as candidates for a trace only those blocks in the CFG that reside in the current textual exception range. Only after all blocks in one range have been placed into traces can blocks in the next textual range be considered. This clearly reduces the opportunity for performance improvements due to code reordering.

**Figure 16 Example motivating the ShouldFollow flag**




**Reordering procedures within a module.** After compiling all procedures for a module, the *slicox* determines a packaging order for those procedures such that their spatial affinity reflects their temporal affinity. This is done by examining the call flow entries for each procedure and using them to construct a weighted *intramodular call graph*. This graph contains one node for each procedure in the module. There is a directed arc from procedure A to procedure B if there is at least one (direct or indirect) call flow entry in A that targets B. The weight of this arc is the sum of the weights of all such call flow entries. Calls to procedures outside the module being compiled are ignored; these will be considered when reordering modules, as discussed below.

Pettis and Hansen's algorithm to construct a procedure order[18] is based on coalescing high-weight arcs in an undirected call graph. Our intuition was that this method successfully places together pairs of procedures that have high affinity for one another, but that this limited view of temporal affinity may not do so well for larger groups of procedures executed close together in time. We chose to implement an algorithm for selecting call traces that is very similar to our algorithm for selecting basic block traces. As in the basic block case, it proves useful to extend traces only when perfect partners are found. When seeding a new trace, we also limit our choice of seed procedure to those procedures adjacent to procedures that have already been reordered, when such procedures exist. The result is that all proce-

dures in each connected component of the call graph are packaged together. This is particularly important for modules (such as dynamic link libraries) that provide several independent services, each of which is implemented using several procedures. More work is needed to compare the effectiveness of our algorithm with that of Pettis and Hansen.

**Reordering modules.** As previously mentioned, we were constrained by our software maintenance processes to permit reordering of procedures only within module boundaries. To reduce this lost performance opportunity, we decided to also order modules so that those modules containing procedures that tend to be executed together in time would have spatial affinity. We developed two tools to produce this packaging order.

The first tool analyzes the call flow entries from all MCAs within SLIC and produces a full weighted call graph for all SLIC procedures. The second tool reads the call graph and reduces it to an *intermodular call graph*. This call graph contains one node for each profiled module in SLIC, with an arc from module A to module B if there is at least one procedure within A that calls at least one procedure within B. The weight of this arc is the sum of the weights of the (direct and indirect) call flow entries from A to B. The tool then analyzes the intermodular call graph and determines an optimized module packaging order, using the same algorithm to analyze the intermodular graph as is used to reorder procedures within a module.

Unfortunately, the module packaging order thus produced cannot be followed to the letter, since each module is constrained to reside in a specific RUdest. If a pageable module would prefer to be packaged next to a module in the nucleus, this request cannot be granted. At the moment, we do not take RUdest constraints into account when determining the suggested module order. Instead, the link/loader is given the preferred module order, and loads modules into their respective RUdests in the order they are presented. An alternative method would be to build a separate intermodular call graph for each RUdest, and produce separate optimized module orders. This may or may not produce a better ordering: our current method can find traces that leave a RUdest and immediately return to it, while using separate graphs for each RUdest would lose this information.[38]

Another advantage of the existing method is that it allows us to analyze placement of modules within RUdests; in some cases, for example, it may pay to move a pageable module into the nucleus if it has strong affinity for a particular nucleus module. We can also analyze the overall SLIC call graph to determine which procedures in a module are used rarely. Many times it is possible to move these low-use procedures into separate modules to improve the effectiveness of procedure and module reordering.

## Field maintenance of restructured code

Clearly the tasks of instrumentation, data collection, and optimization are time-consuming, particularly when applied to a software product of this size. In designing our methods, we felt that it might not be practical to repeat these tasks whenever a new fix needed to be shipped to customers. Therefore we concentrated on ways to reuse existing profile data where possible when compiling fixes, attempting to minimize the performance degradation that might otherwise result.

Profile data for each module are archived together with the source code. As already seen, the hierarchical structure of the counter areas allows existing data to be found for any procedure during the profile feedback phase. This means that procedures whose control flow has not been modified by a fix can continue to use the existing profile data, while changed procedures in the same module are optimized without profile data. This raises the question of how we can detect whether a procedure has been modified.

A very simple test that catches most control flow modifications is to check whether the numbers of control flow and call flow counters in the PCA for a procedure are equal to the numbers of counters that are expected. If there is a mismatch, there have clearly been changes to the control flow of the procedure, and the profile data should be ignored. The obvious drawback to this method is that on occasion a control flow will be changed in such a way that the numbers of expected counters remain identical, but those counters now apply to different control flow arcs. Instead, when creating the PCA during instrumentation, and when reading the profile data during feedback, the *slicox* computes a "signature" from the CFG. This signature consists of a checksum of the block numbers on either end of each control flow arc, and is stored in the PCA during instrumentation. If the stored and computed signatures do not match during profile feedback, the profile data are ignored. The likelihood of two different graphs having the

same signature is extremely small, and can be discounted.

Note that with either of these methods, a simple fix that only alters the instructions within a basic block, and does not change the control flow structure of the procedure, does not invalidate the profile data. Many PTFs satisfy this description (consider fixing a failure to initialize a variable). Thus these methods are better than simply testing whether any source file used to compile a procedure has changed, since the latter would invalidate profile data unnecessarily for such simple fixes.

On the other hand, there are cases where profile data are no longer accurate even though the CFG for a procedure has not changed. For example, reversing the sense of a conditional branch will leave the CFG unchanged, but old profile data will be inaccurate. Similarly, change in the behavior of callers or callees for a procedure may change the behavior of that procedure. Of course, use of invalid profile data does not produce incorrect behavior, just reduced exploitation of performance opportunities. In any case, periodic reprofiling of all instrumented parts is important, and we do this at every release of the AS/400 operating system.

For the benchmarks we currently use to generate a profile for SLIC, we have found a practical way to update the profile information, so that little or no performance is lost. When a programmer provides a fix to a module, a monitoring process checks to see if profile data were invalidated for any procedure in the module. If so, the changed module is automatically reinstrumented, loaded onto a test machine with the most current version of SLIC, benchmarked, and reoptimized. For most modules, the identical benchmarks are run that were used to profile the original release. For modules that affect TPC-C performance, a simpler batch version of the TPC-C benchmark is used. (The standard benchmark requires extensive setup time, significant hardware resources, and human intervention.) Study of the MCAs has indicated that the long-running benchmark and the batch version produce very similar results for most modules.

Although reprofiling has proved practical for our current benchmark suite, the mechanisms to detect profile data that have been invalidated, and to optimize only those procedures that have not changed, are quite important. First, the detection of invalid profile data is used to determine whether a module should be reprofiled; clearly the resources necessary to reprofile a module should not be consumed unnecessarily. Second, in our ongoing efforts to profile other portions of OS/400, we have not always found practical means to automate reprofiling of those modules.

Both the original profiling of SLIC and the reprofiling of fixes are completely transparent to the programmer. Developers need not even be aware that their modules were selected for profiling.

### Benchmark selection

An operating system performs many different functions on behalf of different types of users and applications, so choosing representative benchmarks can be a daunting task. We naturally decided to look at benchmarks that represent areas of performance that are critical to our customers.

Most of our customers use business applications characterized by a traditional transaction processing model. An industry standard benchmark used for measuring transaction processing performance is TPC-C, created by the Transaction Processing Performance Council.[28] We created an internal workload that parallels the functions measured by that benchmark, which for simplicity we refer to here as TPC-C. A benchmark called SPEED also measures transaction processing performance, using a client/server environment.

Another class of customer consists of those that develop applications for the AS/400. For these, performance of program translation and binding within SLIC is very important, so we collected a variety of programs written in different languages to form a Program Model benchmark. Other aspects of the system that were profiled include support for network protocols, such as TCP/IP and APPC (Transmission Control Protocol/Internet Protocol, and Advanced Program-to-Program Communications), primitives for the Integrated File System (IFS), and run-time support primitives for the C language.[39]

These performance areas were also used in determining which modules within SLIC should be profiled. We took measurements using a sampling profiler to determine in which modules the most time was spent when running these benchmarks. After sorting the modules by decreasing contribution, only those modules contributing to the top 95 percent of the time spent in at least one benchmark were se-

Table 1 AS/400 hardware models used in testing

| Model | Processor | Instruction Cache | Data Cache | Level 2 I/D Cache |
|---|---|---|---|---|
| 500 | PowerPC AS A10 | 8K direct mapped 64 byte line size | 4K 2-way associative 32 byte line size | None |
| 510 | PowerPC AS A10 | 8K direct mapped 64 byte line size | 4K 2-way associative 32 byte line size | 1M direct mapped 128 byte line size |

lected for profiling. This cutoff eliminated many modules from consideration whose contribution was too slight for significant payoff from profile-based optimization.

One problem with using multiple benchmarks is the disparate lengths of time needed to run each benchmark. Many portions of the operating system are active during more than one of these benchmarks (for example, the task management and storage management software), and there may be differences in how they act in these different settings. If we were to simply add the weights collected from each benchmark together and use the result to guide optimization, the longest-running benchmark (TPC-C) would have a disproportionate effect on the results.

To avoid this, we built a tool to combine the control flow weights from separate sets of collected data. The tool normalizes the control flow counters for each procedure according to the number of times the procedure was invoked for each benchmark. Suppose that a procedure was invoked $N_i$ times for each of the $i$ benchmarks, and let $N_{max} = max_i\{N_i\}$. Then the combined value $k$ of a control flow counter is computed as $k = \Sigma_i k_i(N_{max}/N_i)$, where $k_i$ is the value of the control flow counter for the $i$th benchmark. This gives equal weight to all benchmarks in determining the final combined profile.

Since not all benchmarks may be considered equally representative of expected customer activity, the tool also permits a weighting factor to be applied to each of the normalized weights in order to adjust the overall contribution of each benchmark. These weights were heuristically determined as follows. First, area experts were consulted to determine a desired weighting $DW_w$ for each benchmark $w$ as a whole. (For example, TPC-C was given a weight of 3, compared to a weight of 2 for TCP/IP.) These desired weights must then be normalized to account for differences in run time and CPU utilization among the benchmarks. This was done by selecting a single procedure $P$ that was highly used by all the benchmarks, and that was known by its designer to have a similar expected profile for all the benchmarks. Note that the decision to use a single procedure $P$ was a heuristic one that was believed to meet our needs; alternatively, a set of such procedures might be chosen. The formula to compute overall benchmark weights is given below.

Let $CPU_w$ be the amount of CPU time spent executing procedure $P$ during workload $w$. Let $I_w$ be the number of invocations of procedure $P$ during workload $w$, and let $I_{max} = \max_w\{I_w\}$. Let $b$ denote a workload such that $I_b = I_{max}$ and assign it an arbitrary weight $W_b$. To determine the desired weight $W_w$ for a workload $w$, we first applied the opinions of the area experts, scaling $W_b$ by the ratio $DW_w/DW_b$. We then calculated the amount of CPU time spent in procedure $P$ per invocation of procedure $P$ on each workload $w$ as $T_w = CPU_w/I_w$, and scaled the previous result by the ratio $T_w/T_b$ to account for differences in usage of $P$. The complete heuristic equation we used for the normalized weights is

$$W_w = \frac{T_w}{T_b}\frac{DW_w}{DW_b}W_b = \frac{CPU_w}{CPU_b}\frac{I_b}{I_w}\frac{DW_w}{DW_b}W_b.$$

## Performance results

For purposes of this paper, we measured the effectiveness of feedback-directed program restructuring (FDPR) on a pre-release version of OS/400 Version 4 Release 1. (FDPR was first used on system licensed internal code, or SLIC, in Version 3 Release 7.) Profile data were collected for the benchmarks previously described, and the combined data were used to optimize the high-use SLIC modules. We then measured the performance improvements of several benchmarks on a number of different AS/400 models. Table 1 shows a comparison of the processors and caches for the models we employed. Because of constraints on machine availability, not all benchmarks were run on all models.

Table 2 shows the reduction in CPU time for the measured benchmarks. (For the IFS benchmark, the reduction in response times is reported.) All of the benchmarks measured are the same ones used in gathering the profile data. The Program Model benchmark was measured both when compiling workloads with minimal optimization and when compiling workloads with full optimization (the profile data contained information gathered using both optimization levels). Improvements for the Program Model benchmark were measured in an unconstrained memory environment (128 MB available) and in a more constrained environment (8 MB available).

**Table 2 CPU time reduction from applying FDPR to SLIC**

| Benchmark | CPU Time Reduction (percent) |
|---|---|
| TPC-C (model 510-2144) | 7.0 |
| Program Model no opt (500-2141, 128 MB) | 16.9 |
| Program Model full opt (500-2141, 128 MB) | 12.3 |
| Program Model no opt (500-2141, 8 MB) | 17.6 |
| Program Model full opt (500-2141, 8 MB) | 13.4 |
| APPC (500-2141) | 6.0 |
| TCP/IP (500-2141) | 7.0 |
| IFS (500-2141) | 3–14* |
| SPEED (510-2144) | 10.9 |
| C Runtime—Heap (500-2141) | 6.4 |
| C Runtime—Math (500-2141) | 7.5 |
| C Runtime—Misc (500-2141) | 5.8 |

*Response time reduction

More detailed hardware performance information is provided in Tables 3 and 4 for many of the benchmarks. These data were recorded using hardware counters aboard the AS/400 processors. Each entry in the table reflects the percentage change of a given measured quantity; for example, the first row shows the percentage decrease in cycles per instruction (CPI) when using a version of SLIC optimized with FDPR as compared to a version of SLIC without FDPR. The column labeled *Average* gives the harmonic mean of the data in the other columns.

Note that the performance improvements from Table 2 are largely explained by changes in CPI. This seems to indicate that, for most of the benchmarks, the effects of basic block reordering dominate those of procedure reordering. By far the largest contributor to the reduction in CPI is the reduction in instruction cache miss rates. This reflects the success of FDPR in reordering basic blocks within procedures to increase sequential control flow. The miss ratios for the translation lookaside buffer (a hardware cache for the SLIC memory page table) appear to be affected largely randomly. This is probably explained by the reordering of procedures in memory, which may increase or decrease the number of hash collisions that occur during memory paging.

Table 4 indicates the effect of FDPR on dynamically executed branches in SLIC. In all cases, the total number of branches executed is slightly reduced, as expected. The percentage of these branches that are unconditional branches is sharply reduced, as is the percentage of conditional branches that are taken. Both of these statistics indicate that basic block reordering is successful in generating long traces of blocks that can execute sequentially.

Table 5 summarizes the effect of FDPR on branch penalties. The first group of statistics indicates the percentage of instruction cache miss cycles that are attributable to taken branches, demonstrating that FDPR significantly reduces this component of the instruction cache (icache) miss cost by ensuring that more branches are not taken. The second group shows the percentage of all cycles that were spent waiting on an icache miss due to any branch. This figure was reduced by an average of 20 percent across all workloads; again this is due to the increased sequentiality of the code. The final group is similar, but here we are interested only in the miss cycles attributable to mispredicted branches. The average reduction of 31 percent indicates that "straightening" the code is important for improving the effectiveness of the branch prediction hardware.

Some additional data were captured from the optimized SLIC by a sampling profiler; these data appear in Table 6. The first row indicates how often the sampler recorded that SLIC was executing code from an FDPR-optimized module. Note that these figures are much smaller than the 95 percent cutoff point that was used in determining which modules to profile. The reason for this is that many of the high-use modules are currently not compiled by the *slicox*. Much of the SLIC legacy code is written in a language that is compiled directly into machine instructions, instead of into an intermediate representation that can be processed by the *slicox*. Clearly this code represents an unexploited opportunity. The second row of Table 6 indicates an upper bound on the estimated performance improvement by FDPR if all code were processed through the *slicox* and pro-

**Table 3 Changes in performance measurements due to FDPR**

| | TPC-C (percent) | APPC (percent) | TCP/IP (percent) | IFS (percent) | Speed (percent) | Average (percent) |
|---|---|---|---|---|---|---|
| Cycles per instruction (CPI) | −6.4 | −5.6 | −5.4 | −5.4 | −7.2 | −6.0 |
| Instruction cache miss ratio | −18.6 | −12.2 | −12.9 | −11.8 | −6.9 | −11.9 |
| TLB miss ratio | +6.3 | −7.5 | +1.0 | −18.6 | −10.0 | Note 1 |
| L2 cache miss ratio | −10.1 | Note 2 | Note 2 | Note 2 | −14.4 | −12.1 |

Note 1 The average has been omitted, since the harmonic mean is undefined for ratios of differing signs, and the arithmetic mean is not meaningful.
Note 2 Run on a Model 500 that has no L2 cache.

**Table 4 Changes in dynamic branch measurements due to FDPR**

| | TPC-C (percent) | APPC (percent) | TCP/IP (percent) | IFS (percent) | Speed (percent) | Average (percent) |
|---|---|---|---|---|---|---|
| Ratio of branches to total instructions | −3.9 | −4.2 | −3.8 | −4.8 | −5.9 | −4.5 |
| Ratio of unconditional branches to all branches | −14.0 | −9.7 | −12.6 | −8.2 | −12.8 | −11.2 |
| Ratio of conditional branches to all branches | +4.0 | +3.2 | +4.4 | +2.7 | +4.1 | +3.6 |
| Ratio of taken branches to all conditional branches | −35.8 | −48.3 | −44.0 | −39.1 | −34.0 | −39.9 |

**Table 5 Reduction in branch penalty due to FDPR**

| | | TPC-C (percent) | APPC (percent) | TCP/IP (percent) | IFS (percent) | Speed (percent) | Average (percent) |
|---|---|---|---|---|---|---|---|
| Percentage of instruction cache miss time due to taken branches | No FDPR | 19.2 | 19.5 | 21.1 | 20.9 | 19.3 | 20.0 |
| | FDPR | 16.0 | 16.6 | 18.2 | 18.2 | 16.7 | 17.1 |
| | Reduction | −16.5 | −15.0 | −13.9 | −12.8 | −13.4 | −14.3 |
| Percentage of cycles due to instruction cache misses for branches | No FDPR | 12.3 | 14.2 | 15.3 | 14.4 | 11.8 | 13.5 |
| | FDPR | 9.0 | 11.4 | 12.4 | 12.8 | 9.4 | 10.9 |
| | Reduction | −27.1 | −19.8 | −19.0 | −11.3 | −20.1 | −19.7 |
| Percentage of cycles due to instruction cache misses for mispredicted branches | No FDPR | 6.9 | 7.0 | 7.9 | 7.3 | 7.8 | 7.3 |
| | FDPR | 4.7 | 4.3 | 5.3 | 5.8 | 5.4 | 5.1 |
| | Reduction | −32.1 | −39.1 | −32.1 | −19.9 | −30.0 | −30.8 |

**Table 6 Profiled module data for optimized SLIC**

| | TPC-C (percent) | APPC (percent) | TCP/IP (percent) | IFS (percent) | Speed (percent) |
|---|---|---|---|---|---|
| Percent of workload profiled | 67.1 | 66.9 | 59.0 | 62.5 | 43.5 |
| Extrapolated CPU time improvement (upper bound) | 10.4 | 9.0 | 11.9 | 4.8–22.4 | 25.1 |

filed, using linear extrapolation from the data in Table 2.

We were initially concerned that optimizing for certain workloads might contribute to degradations in environments not represented by those workloads. In the early stages of development, we used only the TPC-C and Program Model workloads as our profile inputs. We then measured the performance of a number of other workloads. The results, shown in Table 7, demonstrate that most of these benchmarks were improved, and none showed any degradation.[40] It appears that the behavior of much of the operating system is predictable across a wide range of workloads.

An important point about these performance numbers is that they show improvements to code that had already been heavily hand-tuned. Prior to the introduction of FDPR, a good deal of human effort had gone into analyzing and improving hot spots in the code, manually splitting low-use procedures out of high-use modules, inserting compiler directives into code to identify infrequently used code paths, and manually ordering modules in their RUdests according to sampled profiles. That is, a great deal of the opportunity for performance improvements due to FDPR had already been addressed laboriously by hand; the improvements summarized in Tables 2 through 6 are additional gains beyond these manual improvements. One of the anticipated benefits of FDPR is that most of this hand-tuning activity will now be avoided on subsequent releases.

**Table 7** Performance improvements for unprofiled workloads (SLIC version 3 release 7)

| Workload | Performance Improvement (percent) |
|---|---|
| IFS | 3 |
| FSIOP | 4 |
| Batch | 1 |
| Speed | 0 |
| Signon/Signoff | 3 |
| Batch Immed | 2.4 |
| Batch Jobq | 4 |
| Save/Restore | 0 |

## Concluding remarks

In this paper we have demonstrated the feasibility of applying profile-based optimizations, particularly those involving code reordering, to operating system code, in such a manner that the resulting code can be easily supported in the field with a minimum of performance degradation. Although we have concentrated in this paper on the use of profile data for code restructuring, the data are also used today to guide the instruction scheduler and register allocator, and will be used for more optimizations in future releases. Note that the difficulty in applying code-restructuring techniques to operating system code does not apply to many other profile-based optimizations, provided they do not operate across module boundaries.

In an upcoming release, we are providing similar profiling support above the MI for use by our customers, integrating profile-based optimization into the AS/400 native translator and program binder. We anticipate that this will improve performance for many AS/400 applications. Since many applications exhibit better locality than operating system code, improvements for those applications may be less than the results reported here. However, many large business applications suffer from locality problems similar to the operating system; these applications may see results comparable to those measured for SLIC. Even programs with good temporal locality will benefit from code rearrangement to improve their spatial locality, and from the improved efficiency of sequential instruction prefetching that we can provide.

We also plan to expand our uses of profiling within SLIC. For example, we may decide to also profile the initial program loading (IPL) path executed when an AS/400 is rebooted in order to reduce the time that requires. More ambitious would be modifying the link/loader to permit procedures to be reordered across module boundaries while still supporting PTFs in the field. Finally, we might choose to implement Pettis and Hansen's idea of moving infrequently used basic blocks out of a procedure body altogether, although currently too many components in SLIC assume that a procedure body is contiguous.

The Program Model test team (IBM AS/400 Division, Rochester, Minnesota) has found another use for profile data. They have constructed a code coverage tool for analyzing the effectiveness of test suites in covering the code they are intended to test. Although this tool is not capable of determining whether all possible paths through a procedure have been exercised, it is able to indicate those procedures and basic blocks that have not been exercised at all. This tool is proving very useful in improving the quality of testing.

## Acknowledgments

Many outstanding individuals contributed to the success of this project. Itai Nahshon provided substantial guidance and assistance, sharing his experience with developing FDPR for AIX. Mark Novick, Dave Lambert, Scott Hanson, and Jim Holmes developed substantial portions of the *slicox* and the benchmark and feedback tools. David Sandifer and Bob Petrillo provided support for FDPR in other SLIC components, and Bill Seurer and Pat Haugen provided support in the compiler front ends. Brent Hoegh, Dale Peterson, and Mark Sloneker spent many hours planning and implementing methods to add profiling to our product build and test processes. Blair Wyman was key to running the first profiled version of SLIC. Sandy Ryan and Kim Greene helped with performance analysis. Keith Cooper, Jeff DeKelver, Mark Gibbs, and Joe Zoght developed the profile-based test coverage tool. Finally, very special thanks go to Clint Laschkewitsch, Ed Gomez, and Mike Tomashek for their support and encouragement throughout the project.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of the Transaction Processing Performance Council.

## Cited references and notes

1. K. Roland and A. Dollas, "Predicting and Precluding Problems with Memory Latency," *IEEE Micro* **14**, No. 4, 59–67 (1994).
2. L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal* **5**, No. 2, 78–101 (1966).
3. P. J. Denning, "The Working Set Model for Programming Behavior," *Communications of the ACM* **11**, No. 5, 323–333 (1968).
4. P. J. Denning, "Virtual Memory," *Computing Surveys* **2**, No. 3, 153–189 (September 1970).
5. C. V. Ramamoorthy, "The Analytic Design of a Dynamic Look Ahead and Program Segmenting System for Multiprogrammed Computers," *Proceedings of the ACM National Conference*, ACM Pub. P-66, ACM, New York (1966), pp. 229–239.
6. T. C. Lowe, "Automatic Segmentation of Cyclic Program Structures Based on Connectivity and Processor Timing," *Communications of the ACM* **13**, No. 1, 3–9 (January 1970).
7. E. W. Ver Hoef, "Automatic Program Segmentation Based on Boolean Connectivity," *Proceedings of AFIPS 1971 SJCC*, AFIPS Press, Montvale, NJ (1971), pp. 491–495.
8. J.-L. Baer and R. Caughey, "Segmentation and Optimization of Programs from Cyclic Structure Analysis," *Proceedings of AFIPS 1972 SJCC*, AFIPS Press, Montvale, NJ (1972), pp. 23–36.
9. R. Snyder, "On the Application of *a priori* Knowledge of Program Structure to the Performance of Virtual Memory Computer Systems," Ph.D. thesis, University of Washington, Seattle, WA 98195 (November 1978).
10. D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory," *IBM Systems Journal* **3**, 168–192 (1971).
11. D. Ferrari, "Improving Locality by Critical Working Sets," *Communications of the ACM* **17**, No. 11, 614–620 (November 1974).
12. S. J. Hartley, "Compile-Time Program Restructuring in Multiprogrammed Virtual Memory Systems," *IEEE Transactions on Software Engineering* **14**, No. 11, 1640–1644 (November 1988).
13. Y. Wu, "Ordering Functions for Improving Memory Reference Locality in a Shared Memory Multiprocessor System," *Proceedings of the 25th International Symposium on Microarchitecture*, Portland, OR (December 1992), pp. 218–221.
14. A. J. Smith, "Cache Memories," *Computing Surveys* **14**, No. 3, 473–530 (1982).
15. S. McFarling, "Program Optimization for Instruction Caches," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (April 1989), pp. 183–191.
16. A. Mendelson, S. S. Pinter, and R. Shtokhamer, "Compile Time Instruction Cache Optimizations," *ACM Computer Architecture News* **22**, No. 1, 44–51 (March 1994).
17. W.-M. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel (May–June 1989), pp. 242–251.
18. K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, NY (June 1990), pp. 16–27.
19. R. Gupta and C.-H. Chi, "Improving Instruction Cache Behavior by Reducing Cache Pollution," *Proceedings of Supercomputing '90*, New York (November 1990), pp. 82–91.
20. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems* **9**, No. 3, 319–349 (July 1987).
21. R. R. Heisch, "FDPR for AIX Executables," *AIXpert*, No. 4 (August 1994), pp. 16–20.
22. R. R. Heisch, "Trace-Directed Program Restructuring for AIX Executables," *IBM Journal of Research and Development* **38**, No. 5, 595–603 (September 1994).
23. I. Nahshon and D. Bernstein, "FDPR—A Post-Pass Object Code Optimization Tool," *Proceedings of the Poster Session of CC '96—International Conference on Compiler Construction*, Sweden (April 1996), pp. 97–104.
24. S. E. Speer, R. Kumar, and C. Partridge, "Improving UNIX Kernel Performance Using Profile Based Optimization," *1994 Winter USENIX*, San Francisco, CA (January 1994), pp. 181–188.
25. J. B. Chen and B. N. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC (December 1993), pp. 120–133.
26. F. G. Soltis, *Inside the AS/400*, Duke Press, Loveland, CO (1996).
27. M. H. Lipasti, IBM Rochester, MN, personal communication (August 1995).
28. Transaction Processing Performance Council, "TPC Benchmark C: Standard Specification," Revision 3.2 (August 1996). Available at http://www.tpc.org/cspec.html.
29. Note that the data may not be completely valid, since changing the callers or callees for a procedure can change the behavior of the procedure; in practice, however, treating the data as valid gives good results.

30. D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, second edition, section 2.3.4.1, Addison-Wesley Publishing Co., Reading, MA (1973).
31. T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems* **16**, No. 4, 1319–1360 (July 1994).
32. A. Goldberg, "Reducing Overhead in Counter-Based Execution Profiling," Technical Report CSL-TR-91-495, Computer Systems Laboratory, Stanford University, Stanford, CA (October 1991).
33. A. D. Samples, *Profile-Driven Compilation*, Ph.D. thesis (Rep. UCB/CSD 91/627), Computer Science Department, University of California, Berkeley, CA (April 1991).
34. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York (1979).
35. G. Kirchhoff, *Annalen der Physik und Chemie* **72**, 497–508 (1847).
36. In some cases, this transformation interacts badly with the instruction prefetching hardware scheme. The actual predicate we use to determine whether to restart a trace with an isolated block is therefore more complicated than described here.
37. W. Berg, M. Cline, and M. Girou, "Lessons Learned from the OS/400 OO Project," *Communications of the ACM* **38**, No. 10, 54–64 (October 1995).
38. This same criticism might be leveled at our decision to only consider intramodular arcs when determining procedure order within modules. This was a pragmatic decision based on the availability of intramodular call flow data within the MCA for the module being translated. Alternatively, the full SLIC call graph could be analyzed for each module, at the cost of additional compile time and a shared-resource bottleneck (the SLIC call graph file) during distributed compiles.
39. For more detailed information on AS/400 internals, see Reference 26.
40. This experiment was performed on a very early pre-release version of OS/400 Version 3 Release 7, at which time fewer critical modules were eligible to be compiled through the *slicox*. The improvement to TPC-C was roughly 4 percent, and the improvements to Program Model were between 5 percent and 12 percent in various configurations. The numbers in Table 7 should be analyzed in light of these lower performance figures.

*Accepted for publication December 3, 1997.*

**William J. Schmidt** *IBM AS/400 Division, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: wjs@vnet.ibm.com).* Dr. Schmidt is an advisory software engineer with the SLIC Program Model group for the AS/400. He has been developing compiler optimizations for the AS/400 optimizing translator since joining IBM in 1992. He received a B.A. in mathematics and music from Bethel College, North Newton, Kansas in 1984, and M.S. and Ph.D. degrees from Iowa State University, Ames, Iowa in 1991 and 1992, respectively. He holds 19 filed patent applications and two issued U.S. patents, primarily in the area of compiler optimizations. His current interests include profile-based optimizations for Java classes.

**Robert R. Roediger** *IBM AS/400 Division, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: roediger@vnet.ibm.com).* Dr. Roediger joined IBM in 1980. He is presently a senior software engineer with the SLIC Program Model group for the AS/400, working on compiler optimizations for the AS/400 optimizing translator. He has previously worked on various system and computer architecture projects, as well as compiler projects for both the AS/400 and the System/38. Dr. Roediger is a member of the ACM. He received a B.S. degree and M.S. degree in applied mathematics and computer science from Washington University, St. Louis, Missouri, in 1972, and a D.Sc. in computer science from Washington University in 1980. He holds 16 filed patent applications.

**Cynthia S. Mestad** *IBM AS/400 Division, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: Cindy Mestad/Rochester/IBM@IBMUS).* Ms. Mestad is currently a staff software engineer doing performance analysis on the AS/400 at IBM in Rochester, Minnesota. Upon completion of a computer programming course at Brown Institute in Minneapolis, Minnesota, she joined IBM in 1980 as a computer operator on the IBM System/38. She has held various testing, programming, and project management positions within IBM throughout her career. She has focused on the System/36, System/38, and AS/400.

**Bilha Mendelson** *IBM Research Division, Haifa Research Laboratory, Matam—Advanced Technology Center, Haifa 31905, Israel (electronic mail: bilha@vnet.ibm.com).* Dr. Mendelson has been a member of the Haifa Research Laboratory in Israel for several years. She worked on avionic real-time systems at Elbit Ltd. before attending graduate school. In 1990 she joined the Haifa Research Laboratory, where she is now the manager of the code optimization and performance improvements group. She holds a B.Sc. and M.Sc. in computer science from the Technion–Israel Institute of Technology, Haifa, and a Ph.D. in electrical engineering from the University of Massachusetts at Amherst. Her areas of interest include code optimization algorithms, compiler technology, computer architecture, and data-flow systems.

**Inbal Shavit-Lottem** *IBM Research Division, Haifa Research Laboratory, Matam—Advanced Technology Center, Haifa 31905, Israel (electronic mail: ishavit@vnet.ibm.com).* Mrs. Shavit-Lottem is currently a member of the code optimizations and performance improvements group in the Haifa Research Laboratory, where she has been working since 1992. She works on machine-dependent (back-end) compiler improvements and optimizations. She received her B.Sc. degree in computer science from the Technion–Israel Institute of Technology in Haifa. She has four filed patent applications in the area of code optimizations. Her areas of interest include code optimization algorithms, compiler technology, and graphical user interfaces.

**Vita Bortnikov-Sitnitsky** *IBM Research Division, Haifa Research Laboratory, Matam—Advanced Technology Center, Haifa 31905, Israel (electronic mail: vita@haifasc3.vnet.ibm.com).* Mrs. Bortnikov is currently with the code optimizations and performance improvements group in the Haifa Research Laboratory, where she has been working since 1994. She works on machine-dependent (back-end) compiler improvements and optimizations. She received her B.Sc. degree (*cum laude*) in computer science in 1996 from the Technion–Israel Institute of Technology in Haifa. She has four filed patent applications in the area of code optimizations. Her areas of interest include code optimization algorithms, compiler technology, distributed computing, and advanced algorithms and data structures.

Reprint Order No. G321-5677.