QProf: A Scalable Profiler for the Q Back End

by

Greg McLaren

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

Master of Engineering in Computer Science and Engineering

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .....
Department of Electrical Engineering and Computer Science
March 17, 1995

Certified by .....
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Alejandro Caro
Graduate Student of Electrical Engineering and Computer Science
Thesis Supervisor

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

QProf: A Scalable Profiler for the Q Back End

by

Greg McLaren

Submitted to the Department of Electrical Engineering and Computer Science
on March 17, 1995, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Computer Science and Engineering
and
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

Current profiling tools available in the C environment rely either on program counter sampling or instrumentation of every basic block to generate a performance profile based on actual or ideal execution time, respectively. Program counter sampling (e.g. **prof**) yields only a coarse measure of real time, and makes correct attribution of callee execution time to a caller difficult. However, relying solely on ideal time, as do cycle-counting utilities like **pixie**, ignores delays caused by the memory hierarchy and data-dependencies in execution pipelines. Qprof, a three-phase profiling utility incorporated into the Q back end of the multi-threaded dataflow **Id** compiler, takes a step beyond current schemes by uniting fine-grain real time measurement with ideal time estimates to reveal memory hierarchy effects. Qprof's initial implementation generates live time and actual time spent in each Id code block, actual time spent in each partition, and both actual and ideal times spent in each basic block, taking advantage of the low-overhead timing facilities of the RS/6000 (or PowerPC) processor. Performance measurements are presented graphically by a post-processor operating on files generated by the runtime system. Although Qprof incurs substantial overhead in its initial implementation, groundwork is laid for reducing the overhead to a small fraction of uninstrumented execution time by relocating real-time accumulation points and applying graph-theoretic optimizations to reduce the number of instrumentation points. Qprof is easily scalable to multiple processors with minimal modification, and should not, with the overhead reductions schemes just mentioned, perturb parallel execution significantly.

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Alejandro Caro
Title: Graduate Student of Electrical Engineering and Computer Science

Acknowledgments

This work would not have been possible without the support of Alejandro Caro of MIT's Computation Structures Group.

Contents

List of Figures

Chapter 1

Introduction

The driving goal behind the development of new parallel architectures is their potential for a significant speedup over conventional sequential processors. However, parallel algorithms designed without concern for the implementation architecture can often fail to achieve the desired speedup due to the effects of the network (multicomputers), scheduling, and subtle memory hierarchy delays. Freed from low-level concerns by a high-level parallel language, even an expert programmer can fail to forsee dynamic inefficiencies in a particular program resulting from the interplay of scheduling and the latency of split-phase transactions. Thus, if performance is to be maximized in a parallel environment while preserving high-level abstractions, the programmer must be supplied with a tool to diagnose runtime inefficiencies and guide him toward corrective action.

1.1 The Parallel Language Id

The performance gauging tool Qprof developed to meet these needs is designed around the general-purpose parallel language **Id**, created by members of the Computation Structures Group in MIT's Laboratory for Computer Science. Intended for use in programming dataflow and other parallel machines, Id, at its core, is a purely functional language with non-strict semantics much like the lambda calculus. Layered over the referentially transparent core are Id's state-containing *I-structures* and *M-structures*. While I-structures, which can only be defined once, break referential transparency, M-structures, which can be modified at will, may even be non-deterministic. In certain applications, the expressive power inherent in the I- and M-structures is worth the loss of clean semantics accompanying a break from the purely functional approach.

1.1.1 Parallelism and Storage in Id

An Id program, or module, at the highest level is a collection of Id *procedures*. Procedures, in turn, can be broken down into Id *code blocks*, which are assigned to individual processors for execution; an invocation of a given code block must execute on a single processor. Associated with each invocation of a code block is a *frame* to provide for local storage, and at runtime a *call tree* of *invocation frames* is constructed as shown in Figure 1-1. Within a frame live the active *partitions*, which are subdivisions of a code block that execute atomically. Partitions are the fundamental unit of scheduling in such an implementation of Id, and a the set of active partitions of all frames allocated on a given processor is that processor's scheduling pool. A partition can either be an inlet, which is scheduled upon receipt of certain data values, or a thread, which is scheduled by an explicit instruction from another partition. Important information associated with a code block, such as its frame size, is stored in a permanent location called the *code block descriptor* (CBD). While a procedure can have several activation frames if more than one invocation is live, it always has a unique CBD. In addition to the tree of activation frames generated by Id's purely functional core, also present in the diagram is the *global heap*, where I- and M-structures are stored. The global heap is conceptually distributed across all processors in the system, and can be referenced by individual partitions.

1.1.2 Id's Low-Level Implementation

As Id has evolved, it has been run on both simulated dataflow machines, and more recently on the custom Monsoon tagged dataflow machine. However, as high-performance, general-purpose processor lines such as the POWER and PowerPC[1] have gained acceptance in industry and research, it has become increasing clear that to become a more useful and effective tool, Id must be compiled down to a *network of general-purpose processors* rather than a custom architecture. As described by Culler [3], the idea is to first compile Id down to a standard RISC-type language with network, heap, and scheduling primitives, and then compile this "Portable RISC" code down to assembly level on the target machine. By defining an interface specification, modular solutions can be developed which, on the one hand, efficiently transform Id into "Portable RISC" without regard to hardware issues, and on the other, efficiently implement "Portable RISC" on a given processor.

[1]The PowerPC processor is a single-chip version of the POWER processor chip set designed for personal computer use. The POWER and PowerPC instruction sets have many common members but are not identical.

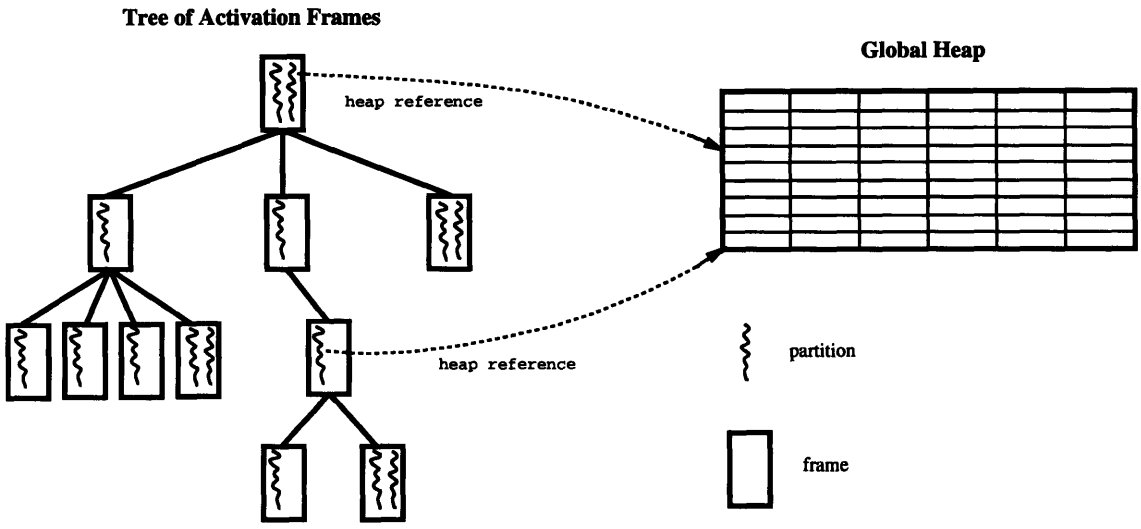

Figure 1-1: The runtime storage of a threaded implementation of Id. Procedure calls generate a tree of activation frames, one frame per code block, and partitions running inside the frames may either reference the local frame or the global heap.

1.1.3 The Q Compiler Project

The Q Project underway at MIT's Computation Structures Group is an effort to implement Id on a network of conventional processors by compiling through just such a "Portable RISC" interface. Conceptually, the Q compiler can be broken down into a five-part model. First, the *front end* translates the original Id source code into a program graph. Next, the *middle end* builds a partitioned program graph from the input graph. In the *back end*, the partitioned program graph, or PPG, is mapped into *RISC (pronounced "star risk"), which is intended to function as the "Portable RISC" interface language for various underlying conventional architectures. In Figure 1-2, some of the data structures used to represent an Id program in *RISC are shown, and links are made to their counterparts in the previous diagram. Note that in the current implementation of Q, a procedure contains a single code block, so that the two terms are interchangeable.

The target hardware in the Q project consists of a *fat tree* of symmetric multiprocessors, each composed of PowerPC processors, as depicted in Figure 1-3. The remaining two modules of the Q compiler must therefore accomplish the translation of *RISC into PowerPC assembly code. To modularize register allocation, this task is broken into two parts. Initially, the fourth module of the Q compiler translates *RISC into "infinite register" PowerPC—PowerPC assembly code using

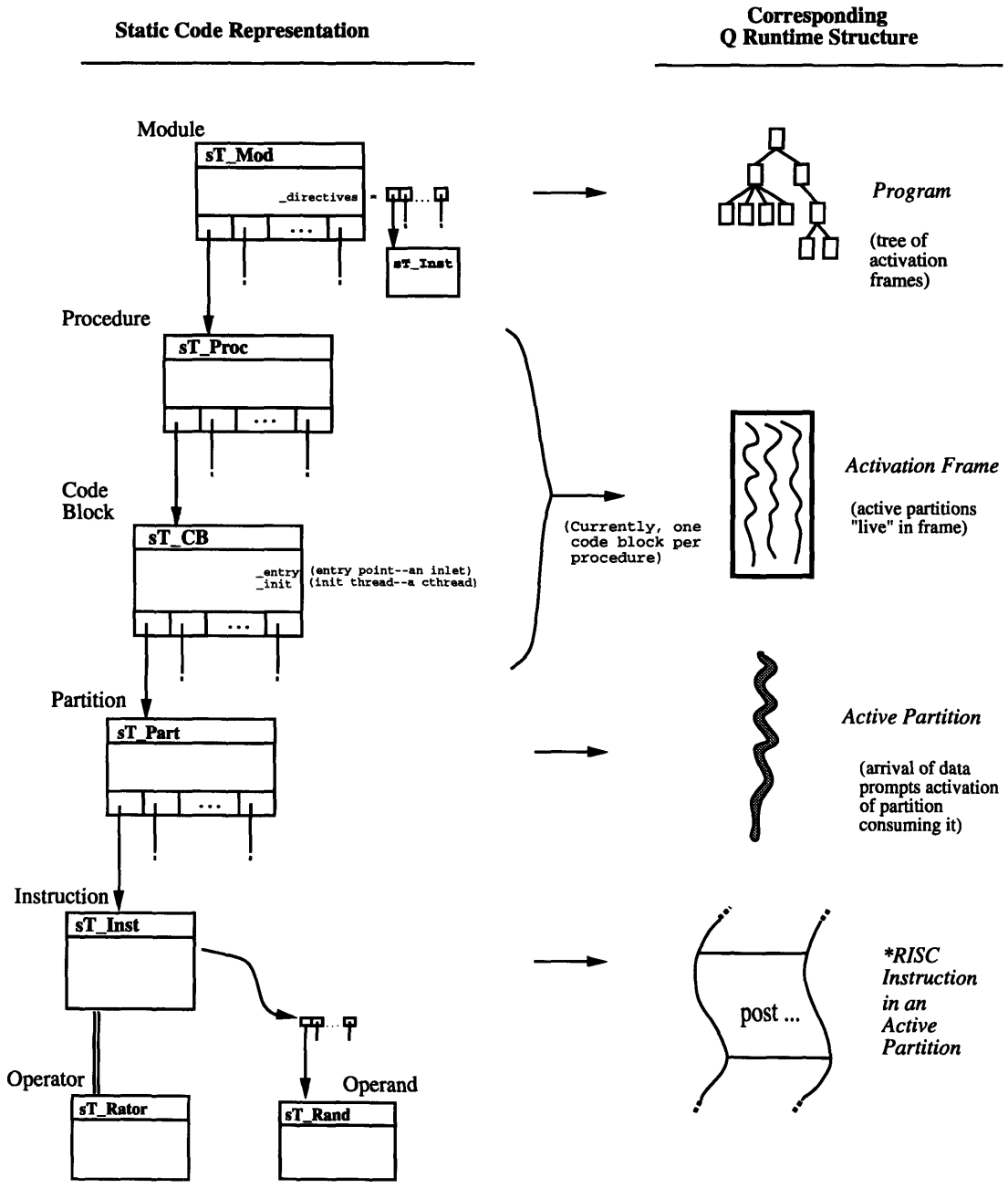

Figure 1-2: A Comparison of the Static Data Structures representing a Q *RISC module to the corresponding runtime structures.

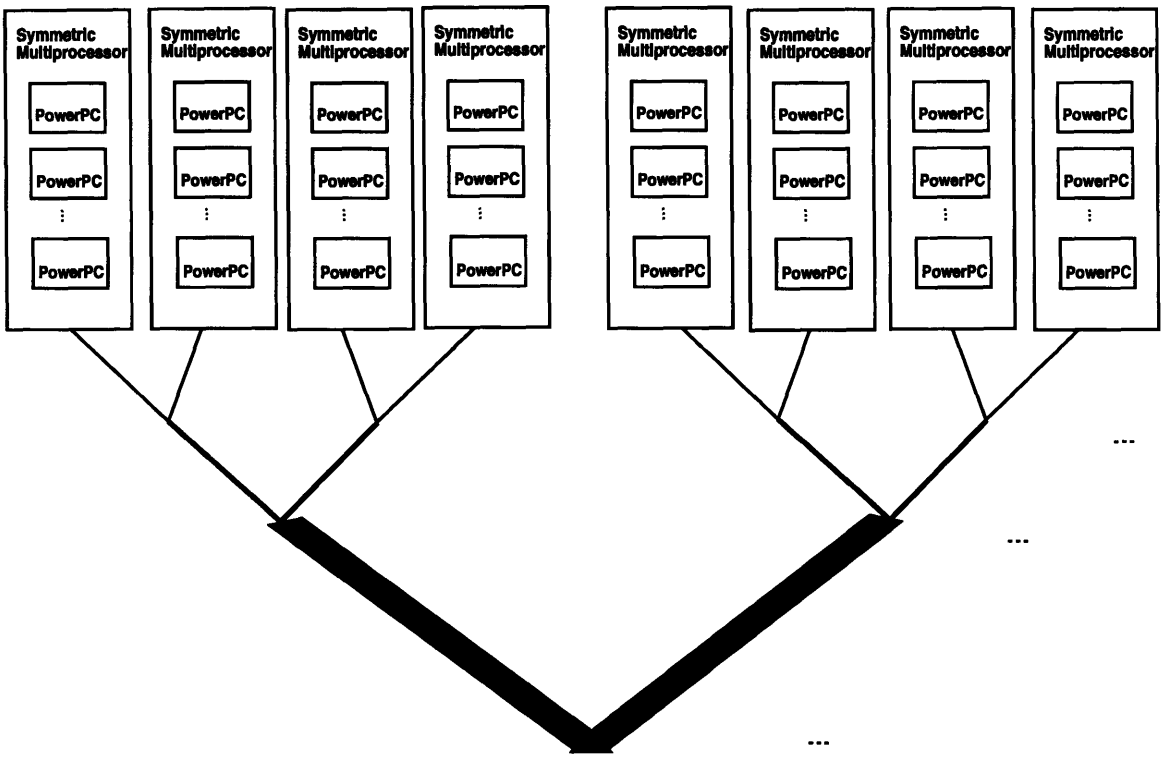

Figure 1-3: An Id program in Q will eventually be run on symmetric multiprocessors arranged in a fat tree. Each multiprocessor will be composed of PowerPC processors.

arbitrarily many registers—and then the last Q module performs register allocation on the "infinite register" code, emitting actual PowerPC assembly code.

1.2 A Profiler for the Q Compiler

Qprof, a profiling tool for the Q back end, was designed to help gauge the inefficiencies in an Id program and guide the programmer to the source of the problem. As the Q project is currently running only on a single-processor machine, the present implementation of Qprof is designed for a single processor. However, a straightforward extension to multiple processors would allow a per processor, per basic block measurement of both the time that the block's execution should require assuming no cache misses, and its actual execution time. Moreover, Qprof also determines how frequently each basic block and machine instruction type is executed, as well as the live time and invocation count of each C procedure and Id code block (split-phase) call.

This thesis is organized into seven chapters. In the following chapter, existing tools for performance measurement are surveyed, and the reasons why they are inadequate for the parallel environment of the Q project are discussed. Next, the proposed design for the initial version of Qprof is developed at

a high level and related to the existing Q compiler. With the groundwork for the profiler already laid, a discussion of Qprof's detailed design and implementation follows as chapter four. In chapter five, we evaluate Qprof on two *RISC programs, and note some shortcomings. Desired improvements are collected into chapter six, including both overhead-reducing optimizations and new features useful when more than a single processor comes into play. Such new features include the collection of idle times at particular processors to diagnose load imbalance, and the measurement of elapsed time between the completion of synchronizing partitions. A summary is included as the final chapter, followed by appendices containing Qprof's source code.

Originally, the idea of support in the Q back end for cache simulations was discussed—an option to generate data and instruction address traces. However, top-level analysis reveals that absent some form of special hardware to supply transparent data bandwidth, collecting an instruction trace would require enough bandwidth overhead to hopeless perturb the execution of a parallel, multiple-processor program. Programs running on a network of processors are dependent upon message arrival time relative to the local state of a given processor, and adding the required storage bandwidth by code augmentation would strongly skew any such timeline. The inherent problem is that unlike an execution *profile*, which to first order should occupy constant space regardless of the program's run time, the storage requirements of tracing vary directly with run time. Add to that the problem that, unlike ideal execution times, accurate address traces cannot be created statically (due to data dependencies), and any workable software solution is clearly going to perturb the original program significantly. Thus, address trace generation is not supported in Qprof.

Chapter 2

Existing Models of Performance Measurement

When considering what features should be supported in constructing an effective profiler for the Q project, it is helpful to consider existing Unix performance gauging tools developed for C programmers. Some of the more common profiling tools designed to interface with the C compiler include the real-time-gathering utilities **prof** [13] and **gprof** [12], which are available on most platforms, and the cycle-counting tool **pixie** [4], which is currently available only on MIPS-processor-based workstations. Though these utilities have shortcomings, they provide a baseline against which Qprof can be compared. In turn, each of the three approaches to performance measurement will be considered.

2.1 Prof

One of the simplest profiling tools available to the C programmer is **prof**, which takes advantage of the existing 60-100 Hz operating system interrupts to sample the processor's *program counter*. At run time, storage slots are allocated for the basic blocks in the target program's assembly code, and during an interrupt, the value of the program counter is binned by its position relative to the target program's assembly code labels, allowing the appropriate slot to be incremented. Although *program counter sampling*, as this process is called, has the advantage of generating timing information for every basic block in the program and not requiring that instrumentation code be added to the executable, it suffers from a lack of timer resolution. For example, given a processor clock rate of 42 MHz like that of the POWER RS/6000 Model 550, and an interrupt rate of 100 Hz, the optimal resolution is:

$$\tau = \frac{42 \times 10^6}{100} = 420,000 \text{ instructions}, \tag{2.1}$$

or perhaps half that for slower models of the POWER RS/6000. Given such coarse time resolution, programs which execute in less than a second cannot be accurately profiled. Although the user can vary the program's inputs to increase execution time, this may result in a very serious distortion of the original distribution of run time amongst the basic blocks.

While the use of existing interrupts makes prof non-intrusive, so that it can collect real time without inducing a significant *probe effect*, the coarseness of its interrupt-driven timing disconnects the accumulation of profiling statistics from the target program's fine-grained behavior. Reduced to state sampling by this defect, prof can acquire little runtime information other than elapsed real time. As a case in point, note that in prof there is no means to relate time spent in a callee procedure back to the caller. At the time of the interrupt, only the *instantaneous location* within user code is known—neither history information nor a detailed view of the state is available. As a result, given two procedures that each call a function **foo()**, the agent responsible for the majority of the calls cannot be determined.

The overhead of prof on a recursive Fibonacci function executing for sixteen seconds was found to be negligible. This looks reasonable since a 1000 cycle overhead (as an upper bound) occurring every 10 milliseconds would amount to a total overhead of only 0.2 percent. As the length of the target program increases, however, some paging delays may surface as a result of maintaining the accumulation bins for the timer.

2.2 Gprof

The development of **gprof** represents a significant improvement over prof. Using procedure-call counts collected at run time, **gprof** assembles a *weighted call-tree* linking the procedures of the target program. After the target program executes, the tree is used to allocate callee execution time, collected via program-counter sampling, to each caller. Since the call tree is an image of program execution, the picture presented to the user can lead to the detection of higher level inefficiencies, and even bugs, which could not have been uncovered using prof.

However, the call tree generated is *not* exact—it attributes a fraction of the total time spent in a given procedure to each caller by assuming that the time a given caller is responsible for is directly proportional to the number of calls made by that caller relative to the total number of callee invocations. Obviously, if one of the callers makes significantly more "difficult" calls to a given subprocedure than other callers, this heuristic would not yield accurate results. For example,

```
double fact(int n) {
   double product = 1.0;

  for( ; n > 0 ; n--)
    product = product*n;

  return(product);
};


void caller_one() {
   int i;

   for(i = 0; i < 100; i++)
      printf("Computing fact(%i) = %lf\n",i,fact(i));

};

void caller_two() {
   int i;

   for(i = 0; i < 100; i++)
      printf("Computing fact(4) = %lf\n",fact(4));

};

void main() {
   printf("Factorial tests.\n");
   caller_one();
   caller_two();
};
```

Figure 2-1: Gprof misrepresents the call-tree profile when two procedures such as **caller-one()** and **caller-two()** make a fixed number of requests of unequal difficulty to a callee.

suppose that as shown in Figure 2-1, the procedure **fact(n)** is called by **caller-one()**, which requests **fact(n)** for some very large values of n, and also by **caller-two()**, which issues much less difficult calls to **fact(n)**. Since **caller-one()** and **caller-two** each invoke **fact(n)** 100 times, gprof will allocate equal amounts of **fact**'s execution time to each, despite the fact that the majority of time spent in **fact** is almost certainly attributable to **caller-one()**.

Perhaps the most serious shortfall of either **prof** and **gprof** is simply that they are designed for use with C, a relatively low-level language. It should be clear that the most useful tool is one tailored for the conceptual model and programming structures of the target program's language. Only with such a tool can the bottlenecks be seen at the programmer's abstraction level. Most of the groundwork

for moving on to consider the design of Qprof has now been laid, but before leaving this topic, one more contemporary profiling tool should be considered to see an approach which avoids dealing with real time.

2.3 Pixie

Developed for early MIPS platforms, **pixie** is a performance gauging tool which instruments, or augments with performance gathering code, each basic block of a target program at the machine language level. As the instrumented program executes, basic block counters are incremented and the total *ideal time* required for program execution can be computed. In this context, *ideal time* represents the number of cycles needed for execution of the target program absent overhead such as cache misses and page faults. In the case of the early MIPS platform, pixie could easily transform basic block counts into ideal time by examining the object code files, since the assembler inserted pipelined delays and expanded macros before emitting machine code. The number of instructions in a basic block thus could be mapped easily into a cycle count, so that multiplying by the number of invocations of that basic block provided the total *ideal time* required for its execution.

Such low level code augmentation, without overhead-reducing heuristics, perturbs real time measurements of program execution by a significant amount. For example, a recursive Fibonacci function coded in C that took 40 seconds to run in its original state took *98* seconds as a **pixie**d executable, an increase of 145 percent! While longer basic blocks would help reduce the impact of pixie's instrumentation, any real time measurements (and time line interactions between the processors of a multicomputer), would certainly be skewed.

Within the context of pixie, real time is not even at issue, as the profile is based solely on invocation counts and basic block lengths. However, an approach such as pixie's that focuses solely on ideal time would be inadequate for the Q back end operating in multiple-processor mode, even if heuristics were used to reduce the number of instrumentation points, because the important issue of real time is not addressed. Without real time, one loses the ability to detect memory hierarchy effects and unpredicted pipeline stalls which are characteristic of the POWER RS/6000 processor, which is significantly more complicated than the MIPS. The basic problem stems from the network of processors. The introduction of the network into the model of performance measurement so complicates the situation that efficient solutions heavily demand emphasis on real time rather than calculated ideals, at least with respect to all statistics which must cross the abstraction barrier of a single processor.

Chapter 3

The Proposed Design

Now that the Q back end and its runtime storage structures have been described, and present profiling tools have been shown inadequate in efficiently gauging the performance of an Id program compiled through Q, the stage has been set for the design of a new profiling tool mated to Q and free of the shortcomings found in the standard Unix performance gauging routines. First, we concisely discuss incorporating ideal time calculation and code augmentation into the original Q model. Then, the design of the post processor, Postprof, is laid out and issues of data presentation are confronted.

3.1 Instrumentation and Pipeline Simulation

As illustrated in Figure 3-1 below, the initial design of Qprof divided the work of profiling an Id program into three steps. The goal of this process is to create an instrumented executable and a table of ideal execution times for the original basic blocks. First, infinite register PowerPC code taken from the Q compiler's fourth stage is routed into a *code instrumentor,* which augments the code at certain points with instructions to collect runtime statistics—such as invocation counts and actual pipeline delays—into the local activation frame. Next, the augmented code is routed back to the last stage of the Q compiler to allow register allocation to occur and fed into the *simulator* along with the original, uninstrumented code to generate a set of *ideal execution times* for each basic block. Inside the simulator, the uninstrumented code is needed to efficiently access the original assembly code *labels,* since they are altered during instrumentation.

Unlike cycle counting on the MIPS, ideal time calculation on the POWER RS/6000 architecture is a complicated task, and a *simulator* must be built to accomplish it. Given the multiple functional units of the POWER RS/6000 processor, it should be clear that no direct correlation exists between the number of instructions in a basic block and its ideal execution time. However, generating ideal

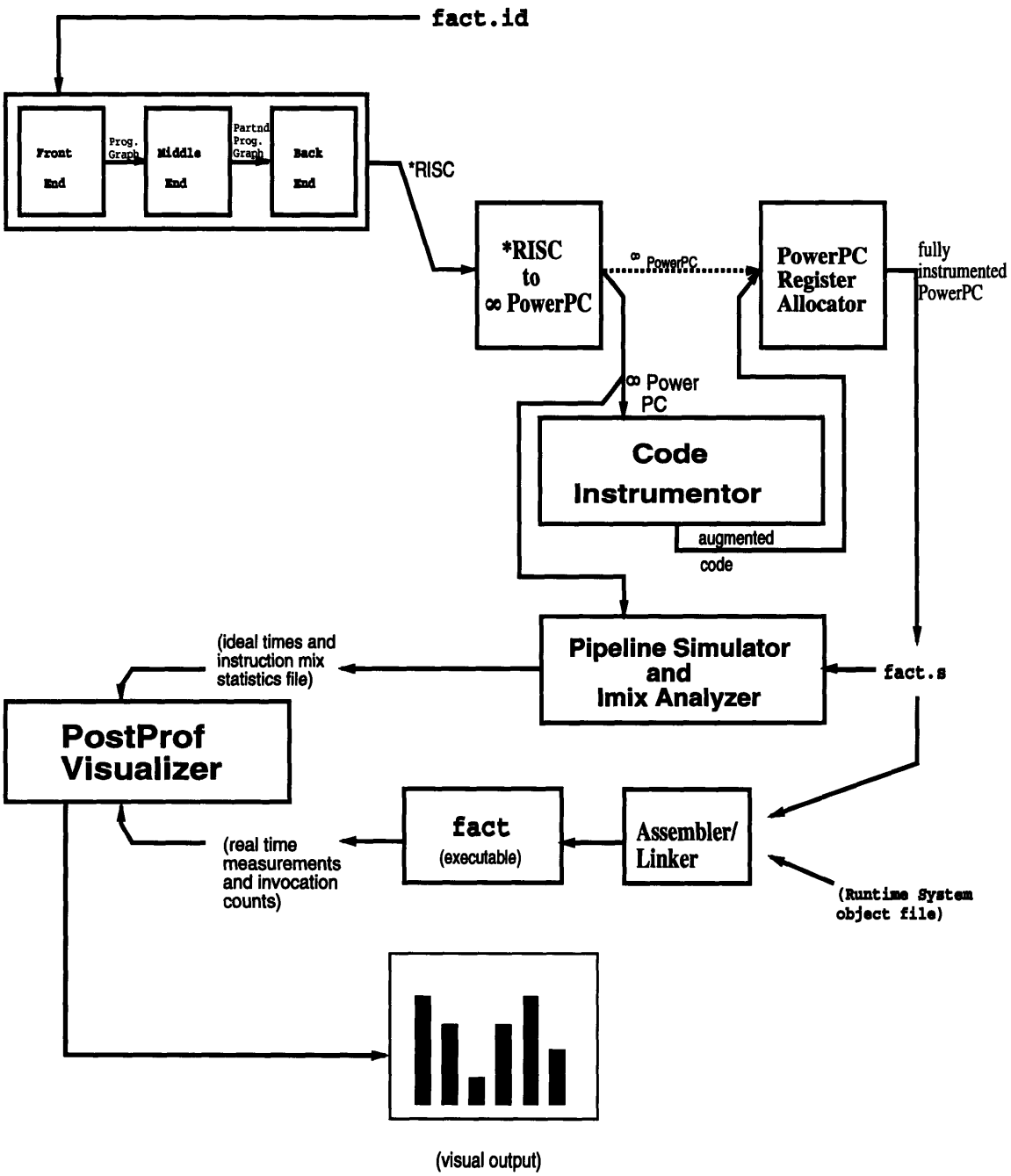

Figure 3-1: The conceptual outlay of Qprof. Shown are the five Q compiler stages and the additional stages required by Qprof.

times is of crucial importance to Qprof because, when united with the runtime statistics, ideal times can quantify and localize memory hierarchy effects.

As shown in the figure, after the first two phases of Qprof have run and the compiler has been invoked to generate an executable, all that remains is the data analysis phase fulfilled by the **PostProf** visualizer. Running the augmented executable (**fact** in the figure) generates a file of runtime statistics which can be used by PostProf together with the simulator's static *ideal times* and *instruction mixes* to give an accurate picture of a program's performance.

By splitting the performance data into a runtime and static file, it was hoped that runtime overhead could be kept to the minimum actually required to *collect* the statistics. All data analysis was to be done offline—either in the pipeline simulator before the executable was even generated, or in the post processor to reconstruct necessary information from a compacted minimal set. The decision to place instrumentation before register allocation was not arbitrary, but the result of analyzing a tradeoff between two evils. If the code were instrumented *after* register allocation, a number of registers—those required in the instrumentation code—would have to be *permanently allocated* to the profiler, which would mean either that significant performance would have to be sacrificed in the design of the Q compiler by allowing fewer real registers to be emitted by the allocator, or else a *switch* would have to be installed to squeeze the register allocator's output registers into a tighter set *when profiling is enabled.* The first choice is obviously poor, and the second may result in a significant probe effect; when fewer real registers are available to the allocator, the register allocation strategy may assign a radically different set of registers to the segments of original code. However, by placing the instrumentor *before* the register allocator, we suffer a significantly diminished probe effect *because a near totality of the registers used in the instrumentation code are only live inside the instrumentation.* Inside the original code segments, the register allocator should see the same number of live registers and the allocation map should be relatively unaltered compared to the unprofiled executable. Although one profiler register is live at all times, it has been reserved solely for Qprof and thus is not subject to allocation. The exact design of the instrumentation will be discussed in the next chapter.

Clearly, there is a fundamental assumption evident in augmenting code to collect real time intervals—the existence of a low-overhead timing facility. Fortunately, the PowerPC and RS/6000 architecture provide such a timer in the form of the Time Base register and Run Time Counter register, respectively. Clock accesses have a pipeline delay of a single cycle and a latency of two cycles in these architectures, although more complication ensues if long durations[1] are to be captured, since the time is stored across *two* registers which must be accessed independently. Much more will

[1] We refer here to time periods on the order of 1 second or more.

be said about the timer below.

3.2 PostProf

Early on it was decided after a review of Kesselman's dissertation [8] on parallel program performance measurement that a graphic approach should be taken in the post processor. When a single processor is at issue, a textual summary such as **prof** yields is readily analyzed, but as the number of processors increases, Kesselman has found only a *graphic format* is likely to present the vastly increased volume of information to the programmer in a manageable way. Given several processors, several basic blocks (or code blocks), and several types of measurements, it seems only natural that a visual approach might prove more useful than any other. The design conceived at this time was inspired by Kesselman's "three-dimensional histogram," in which code segment goes on the Y-axis, the processor goes on the X-axis, and grey-level intensity of each box (pixel) represents the magnitude of the measurement. In this scheme, only one type of measurement, such as basic block invocation count, can be viewed at a time. An example of a three-dimensional histogram is shown in Figure 3-2, in which the invocation count of each basic block in a particular code block is depicted. By scanning horizontally, the user can tell which processor has the highest count for a given basic block, and by scanning vertically he can tell which basic block has the highest count for a fixed processor.

Major operations that should be applicable to parallel execution data include *folding*, *sorting*, and *subsetting*, each brought about by highlighting a *selection set* via X-window interaction. Folding a selection set onto the Y-axis should generate a conventional histogram of magnitude vs. code segment, summed over all processors, whereas folding it onto the X-axis should generate one of magnitude vs. processor, summed over all code segments. Once a conventional histogram has been obtained, different types of data can be "stacked" onto the single histogram, each represented by a different shade of grey.

Although folding is perhaps the most powerful feature, the second two operations could also prove useful when applied directly to a "three dimensional histogram." Subsetting reduces the clutter presented and allows the user to focus on only those entries which are relevant to user's problem. Sorting of the selection set by a measurement magnitude could be carried out either on the processor (X) or code segment (Y) axes to simplify the task of spotting the most costly code blocks according to some performance metric.

While improvements and extensions to Qprof suggested by experience with the initial implementation are discussed later in the paper, the extended design of the post processor which has just been discussed was conceived early on in the project, but could not be implemented since only a

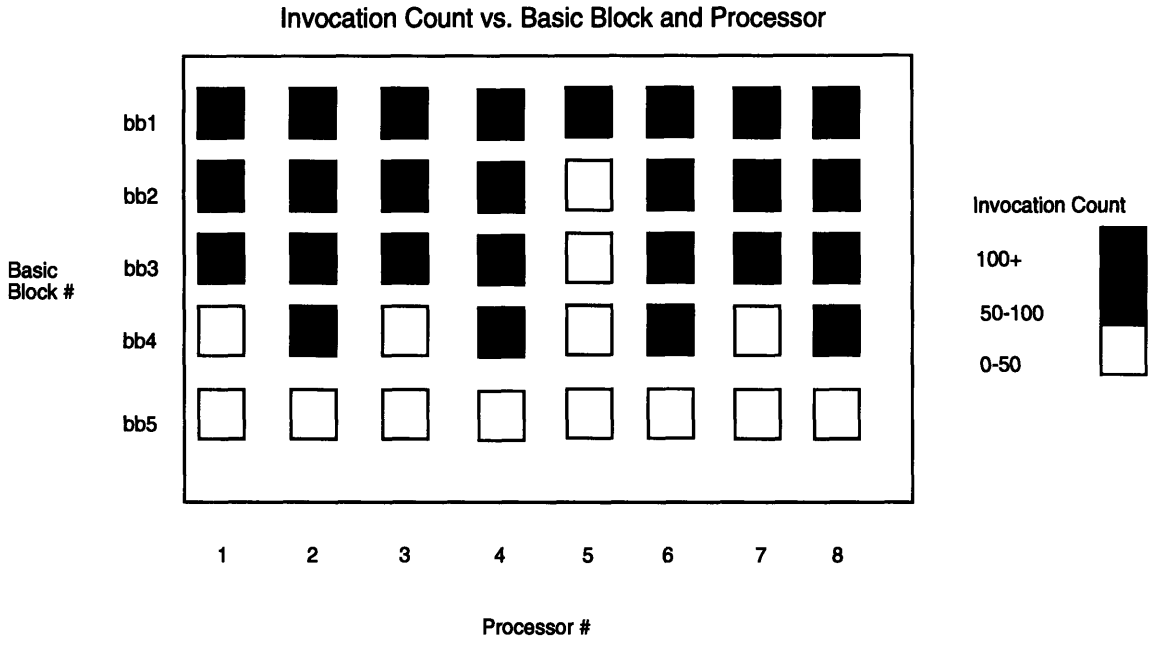

Figure 3-2: A 3-d histogram. Invocation count, represented by the intensity of the block, is plotted as a function of both basic block and processor number. There appears to be a scheduling problem starving the fifth processor in the first through fourth basic blocks, and another starving the odd processors in the third and fourth basic blocks.

single processor POWER RS/6000 system was available. The actual visualizer implemented will be discussed in the following chapter, which deals with the detailed design of Qprof.

Chapter 4

Detailed Design and Implementation

While the Q compiler project is hoped eventually to be implemented on a multicomputer with many PowerPC processors, the reality of the matter during the design phase of Qprof was that only a single processor RS/6000 system was available for the project. Given this reality, the post processor phase was greatly simplified relative to the original design, leaving *ideal time calculation* and *code augmentation* as the only real hurdles in accomplishing the implementation. Ideal time calculation was chosen as the first objective, since the problem could be analyzed and solved independently of the Q runtime system and compiler stages, which were under development while Qprof took shape.

4.1 Ideal Time: Building a Simulator for the RS/6000

The first major goal was the development of a *simulator* to compute the *ideal time* of a POWER instruction sequence. The *ideal time* of a sequence of Power/PowerPC ISA instructions is defined as the number of CPU cycles required to execute the sequence *assuming no cache*[1] *misses occur.* The significance of *ideal time* comes from the fact that given (1) the actual time spent executing a given basic block, (2) the number of times that basic block was called, and (3) the ideal time per invocation, we can compute the number of cycles lost to the memory hierarchy due to misses in the cache. Of course, this method assumes that every invocation of a basic block takes the same number of cycles given a 100 % cache hit ratio, which is clearly false since the immediate ancestor of a given basic block can leave various clutter in the function unit pipelines of the CPU that may

[1] Includes both I-cache and D-cache.

not be the same for each invocation. Still, our heuristic approach should prove effective. Note that as discussed above, the *register allocated* code is timed to ensure we analyze code as close as possible to that which will be executed. However, it is important to keep in mind that no matter how good the heuristics are, the ideal time can *never* be computed correctly in every case without running a complete simulation of the entire program, including register values, caches, main memory, etc. The reason is that some pipeline delays are *data dependent*, and the Halting Problem tells us that we cannot construct a algorithm to predict these delays without essentially *running a simulation of the original machine.* We must settle for an approximation to the ideal time.

4.1.1 Approximating Ideal Time with Simulated Pipelines

In this subsection, we give a brief overview of the components of the simulator, and then discuss some interesting problems which arise when one attempts to use such a model to compute accurate ideal time. Even an excellent simulator, if applied incorrectly to the instruction stream, can yet poor results. By appropriate heuristics, however, the error can be greatly reduced for most of the basic blocks in a partition after the first.

In computing the ideal time of a specific instruction sequence, we first place the simulated pipelines of the CPU in a particular state (assume empty). Next, we feed in the instructions one CPU cycle at a time, handing them off to the appropriate functional unit pipelines as they become available while scoreboarding registers not renamed[2] to preserve data dependencies. In the parlance of an object-oriented language, the functional units could be implemented as data types with manipulators to insert machine instructions and advance the global time, and predicates to determine whether the head of a pipeline is ready for an additional instruction. In such a design, the master process feeding the instruction sequence into the appropriate pipelines would have access to a global dispatch scoreboard to determine when and if instructions could be dispatched to a given pipeline. In the end, when the last instruction reaches a chosen point in its execution pipeline, both the total number of cycles required and the occupancy of the various pipelines must be stored. Generally speaking, the accuracy of raw ideal time will improve with the length of the code sequence due to the relative decrease in the "pipeline start up transient". By recording the state of the pipelines, we allow two accuracy-enhancing improvements to be easily made to the algorithm.

The first, and simplest, enhancement would be either to take a global average of the final relative occupancy of the pipelines and apply a formula to it to determine some number of extra cycles to add to each raw ideal time, or else to use the relative occupancy of the pipelines *for each possible*

[2]Power/PowerPC implementations rename floating-point registers.

preceding instruction sequence to determine a "fudge factor" in cycles to add to the raw ideal time of the immediately succeeding code sequence. The catch is that this method can only be used to add cycles to the raw ideal time of a code sequence if the preceding sequences are limited to a small set of user code sequences for which pipeline occupancy data has been stored. Obviously, various heuristics may be more successful than others, but we should be able to come close to the correct ideal time by using a heuristic sufficient to deal with the pipeline's startup transient.

In the above method, all pipelines were assumed empty upon running each code sequence. A more accurate calculation of ideal time can be made by simulating the ideal time of instruction sequences in the order in which the sequences appear in the code, so that the pipeline is occupied to a degree upon starting the ideal time calculation of each consecutive instruction sequence. This is in fact the technique chosen in the implementation below. Obviously, a problem occurs when a basic block has multiple ancestors, but if the ancestors are all user instruction sequences, each ancestor can be run starting with empty pipelines before running the target instruction sequence, and the average of the two (or more) ideal times can be used as the final ideal time. If accurate weighting information is desired—assuming each ancestor equally weighted sometimes yields a distorted profile—the control-graph weighting heuristics described in the "future improvements" section below may be used. In a sense, the only reason this approach should better than the previous paragraph's is that a heuristically-derived fudge factor is being replaced by a *recursive call* to the ideal time simulator.

Particular attention must be given to branch instructions, since they end basic blocks (and thus would not be found in the middle of an instruction sequence). Observe therefore that given two contiguous branch instructions (the first being conditional), the second would be placed in its own basic block and would encounter completely empty pipelines when run through the simulator, unless one of the two accuracy-enhancing optimizations above is included. (This would yield an ideal time of one cycle for the second branch, which would be erroneous on most CPU architectures.) Fortunately, if the second branch has only a single ancestor, the second optimization technique above would correctly capture the delay produced by the consecutive branch instructions. Indeed, the chosen implementation captures this behavior.

4.1.2 CPU Models

The CPU models considered in the design of the ideal time computation engine were the RS/6000 model shown in Figure 4-1, and the PowerPC 601 model shown in Figure 4-2. The RS/6000 CPU architecture is characterized by separate data and instruction caches, and three separate functional units—one for each of branch, floating point, and fixed-point operations. The branch unit can grab up to four instructions from the I-cache in a single cycle, and issue four instructions in a single cycle.

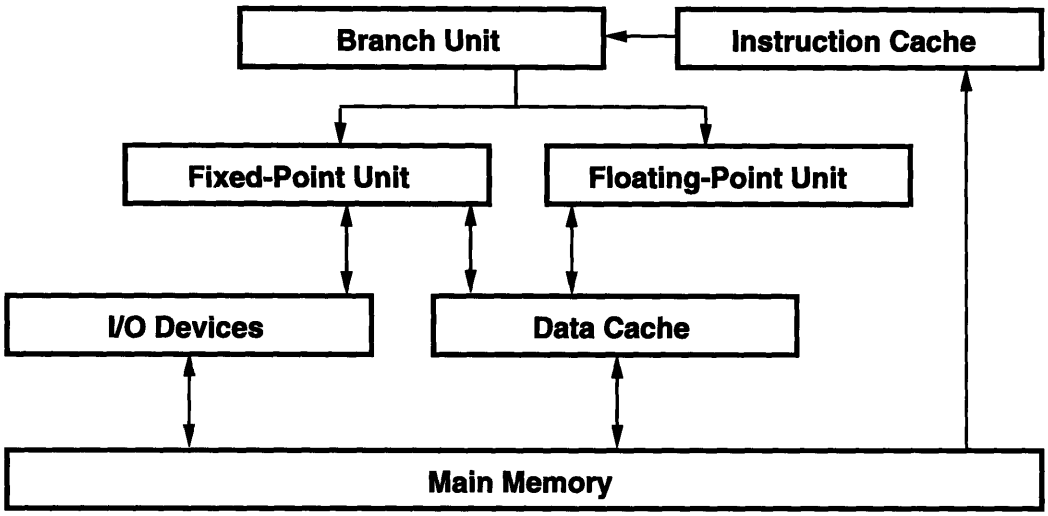

Figure 4-1: The Logical Organization of the RS/6000 CPU.

Supposing that a logical operation on CR (condition register) bits, a branch instruction, and fixed-point add, and a floating-point add were available to the branch unit, it could dispatch the floating and fixed-point operations to their respective functional units, and execute the branch and CR bit logical operation itself, all in a single clock cycle. This feature allows it to branch transparently, provided that in the case of conditional branches the branch is correctly predicted.

The main pipeline delays in the RS/6000 include a one cycle latency between a load from the cache and an instruction that uses the result register, a delay of one cycle between two dependent floating point instructions, a delay of three (eight) cycles between a fixed (floating) point unit compare setting CR bits and a successful branch using those condition bits, and a three cycle delay between two consecutive branch instructions, the first being conditional. All these except the last can be modeled by the simple scoreboarded set of pipeline abstractions, and the last can be handled by one of the accuracy-enhancing optimizations described above.

The PowerPC CPU architecture is somewhat more complicated, although a shared D-cache/I-cache is employed. Instead of being channeled through the branch unit, as in the RS/6000 architecture, the PowerPC 601 design incorporates an instruction queue (IQ) eight instructions deep that feeds all three functional pipelines (FPU, IU, and BPU). Unlike the RS/6000, the PowerPC 601 can only issue three instructions per clock cycle; however, it can fetch eight in a single clock cycle from the combined cache, if the cache is not in use. Floating-point and branch instructions may be grabbed by their respective functional units as soon as they fall into the bottom four IQ slots, but

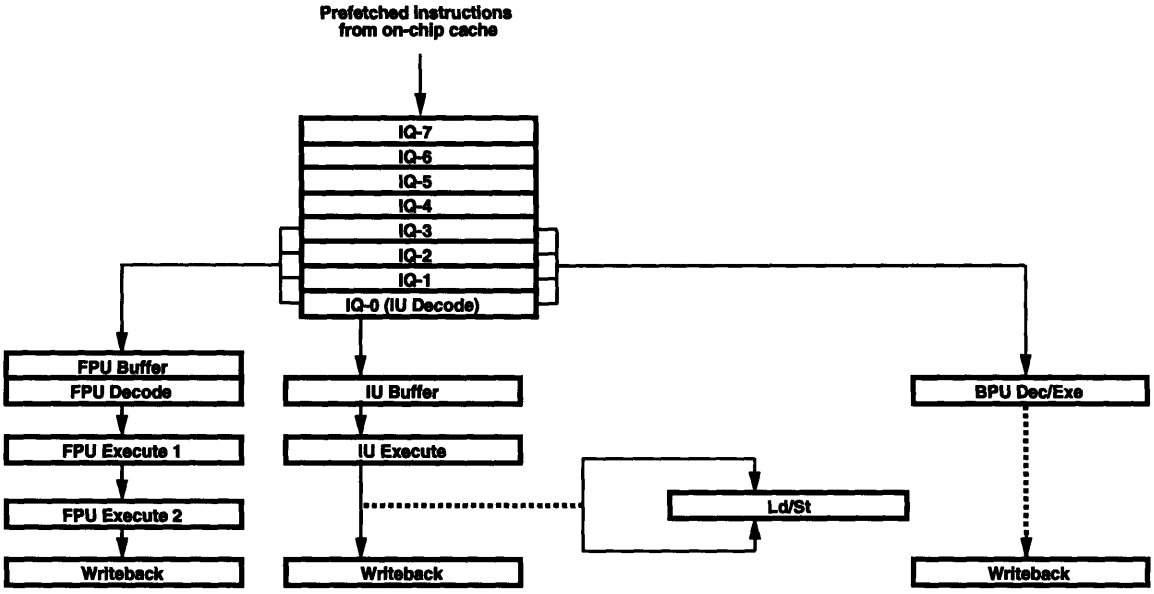

Figure 4-2: The Logical Organization of the PowerPC 601.

integer instructions and other logical operations assigned to the integer unit (IU), may only decode from the bottom IQ slot. Just as in the case of the POWER RS/6000 processor, the PowerPC 601 can be simulated in an object-oriented language by building a data type for each functional unit and new objects for each pipeline stage. The differences will lead to internal changes in the data types representing the functional units, and a new master process to hand incoming instructions off to the execution units from the IQ.

4.1.3 Timing Experiments and the Detailed Design of the Simulator

As soon as it was known that the RS/6000 processor had become the target of the first Qprof implementation, detailed experimental design began on the simulator. By studying an issue of IBM Journal of Research and Development dedicated to the RS/6000 architecture [7], much of the interior design of the branch, fixed, and floating point units came to be known immediately. As the design process progressed, the pipelines for each functional unit were laid out and interlocks or scoreboards were developed to hold back the execution of certain instructions. However, there were two cases in which the known layout inside a functional unit was not sufficient to predict actual execution time, and experiments had to be conducted to resolve the appropriate functionality. The first case was that of POWER ISA moves to and from special registers—the condition register, link register, counter register, etc. It had been clear from the beginning that lock bits must be introduced for writes to condition register (CR) fields to avoid having multiple outstanding writes to the same condition

register field, but it was not clear how special register operations involving special registers other than the CR, or even moves *from* the CR, interacted. In many, but not all cases, adjacent special register moves were found to create *pipeline delays* in the fixed-point unit where they executed. In order to probe the time required to execute a given piece of code in an experimental setting, the assembly code illustrated in Figure 4-3 below was developed.

The code takes advantage of the low overhead timer (Run Time Counter) on board the RS/6000 processor to time the execution of 5,000 repetitions of the target code segment. Given the RTC clock rate and that of the CPU, the calling C routine can compute the actual execution time of the target code segment. Knowing the actual pipeline delays produced by various combinations of move-to- and move-from-special-register instructions, a table was constructed, and—finally—a set of interlocks could be designed to mimic the correct pipeline stalls[3]. In the final implementation, many of the special registers were given both read and write lock bits, set when certain instructions dispatched and cleared when those instructions passed through the execute or post-execute stages of the fixed-point unit.

The second complication which couldn't be accounted for via the original functional unit stages were the pipeline *occupancy* delays. Several special instructions can occupy a single pipeline stage for varying lengths of time beyond the one cycle norm. Some, such as **stsi, lsi, stsx,** and **lscbx**, which are load/store string operations, take a *variable* amount of time depending on the contents of registers and even *memory*! Luckily, none of this set of instructions was included among the POWER instructions implemented in the Q back end. The load- and store-multiple instructions, **lm** and **stm**, have an occupancy statically determined by their target register, and so could have been simulated, but were also not included in the subset of POWER instructions implemented in Q. The only extended-occupancy instructions dealt with were thus **div/divs** (integer divide), which requires 19 to 20 clock cycles, **fd** (floating divide), which requires 19 cycles, and the fixed-point multiply instructions, which require 3-5 cycles depending upon the size (byte, halfword, word) of the last factor. Both divide operations were assigned 19 cycle occupancies, and the multiply was assigned a four cycle occupancy.

After having determined all interior delays in each of the functional units, we next considered the delays *between* units. Still to be resolved was the link between the store queues and data load inputs of the fixed and floating point units. For example, when a floating-point load, which passes through both the fixed and floating point units, executes in the real RS/6000, the FXU generates the address for the load request and the FPU unit activates its register renaming machinery and locks the target register until the data arrives there from memory. Our simulator must allow the FXU to send the

[3]Delays ranged from zero to three cycles for differing combinations.

```
.csect .rtcclk[PR]

mflr 0                          # load the link register into register 0
stm 20, -8*nfprs-4*ngprs(1)     # store several of the non-volatile registers
st 0, 8(1)                      # save reg. 0 (link reg. copy) on the stack
stu 1,-szdsa(1)                 # create new stack frame

lil 3, 5000                     # load register 3 with 5000...
mtctr 3                         # ... and store it it the count register

mfrtcl 4                        # place the lower word of the on-board clock
                                # in register 4


loop:                           # time the code, looping back repeatedly
                                # as the counter is decremented
 <code to be timed goes here>

bdn loop



mfrtcl 5                        # place the lower word of the on-board clock
                                # in register 5

l 0,szdsa+8(1)                  # retrieve link reg. from stk. and put in reg. 0
ai 1,1,szdsa                    # pop the stack frame
lm 20,-8*nfprs-4*ngprs(1)       # restore non-volatile registers

sf 3,4,5                        # compute the elapsed time, and place it
                                # in the result register

mtlr 0                          # restore the link register
blr                             # branch to it
```

Figure 4-3: POWER assembly code for probing actual execution time. The generic wrapper permits calls to be made inside the loop.

FPU a signal in the manner just described. In fact, the solution eventually accepted was to add a fourth functional unit, a *D-cache*, to the simulator design to establish the correct message-passing protocol between the FXU and FPU with regard to memory operations. Conceptual diagrams of each of the four functional units developed during implementation of the simulator are shown in Figures 4-4, 4-5, 4-6, 4-7

Once the diagrams for each functional unit had been prepared, developing most of the code for the simulator was straightforward in C++. Each functional unit became an object, as did each pipeline cell and buffer. An I-cache object was developed to feed the branch unit's input buffers with four instructions per cycle as dictated by the specification. Initialized with a PowerPC module representing the target program, the I-cache object can either return instructions consecutively until the end of a partition is reached, or fetch from a new target address given the *label* of the target. In all cases, simulated registers and memory are devoid of the actual data *values*; there is no dependency on actual values built into the simulator. This permits it to run at reasonable speed.

Branches

One issue still unresolved at this point in the design is the handling of branches. To what extent are branches to be simulated? How are they to be fed to the branch unit? Granted, the instructions themselves get there via the I-cache, but how do the *branch decisions* get to the branch unit? Remember, since no data values are kept in the simulator, the decision whether or not to take a conditional branch must be inserted by a controller external to the simulator. After experimenting to check feasibility, it was decided that given a partition and some target inside the partition, an external routine could generate the correct sequence of branch decisions to move from the beginning of the partition to the target and then exit the partition[4]. If this sequence of branch decisions were fed to the branch unit, the target basic block could be reached in such a manner as to prime the pipeline with the correct instructions, at least in many instances. (The first basic block is an exception, as is any basic block with more than one ancestor.) In fact, this is exactly how the branch unit is controlled in the current implementation; given a target, the branch unit and other pipelines are fed with a sequence of branch decisions known as a "branch chart" that leads to the target code segment.

A *very* tricky issue which at first seems almost invisible is *where* to begin and end timing of a given *basic block*. A naive answer might be, "as soon as the last instruction is dispatched from the branch unit". However, this is problematic for short instruction sequences since the dispatched instruction,

[4]The I-cache is equipped with an extra partition representing a generic C function that is entered whenever a branch to link register is taken.

Branch Unit (BRU) Design

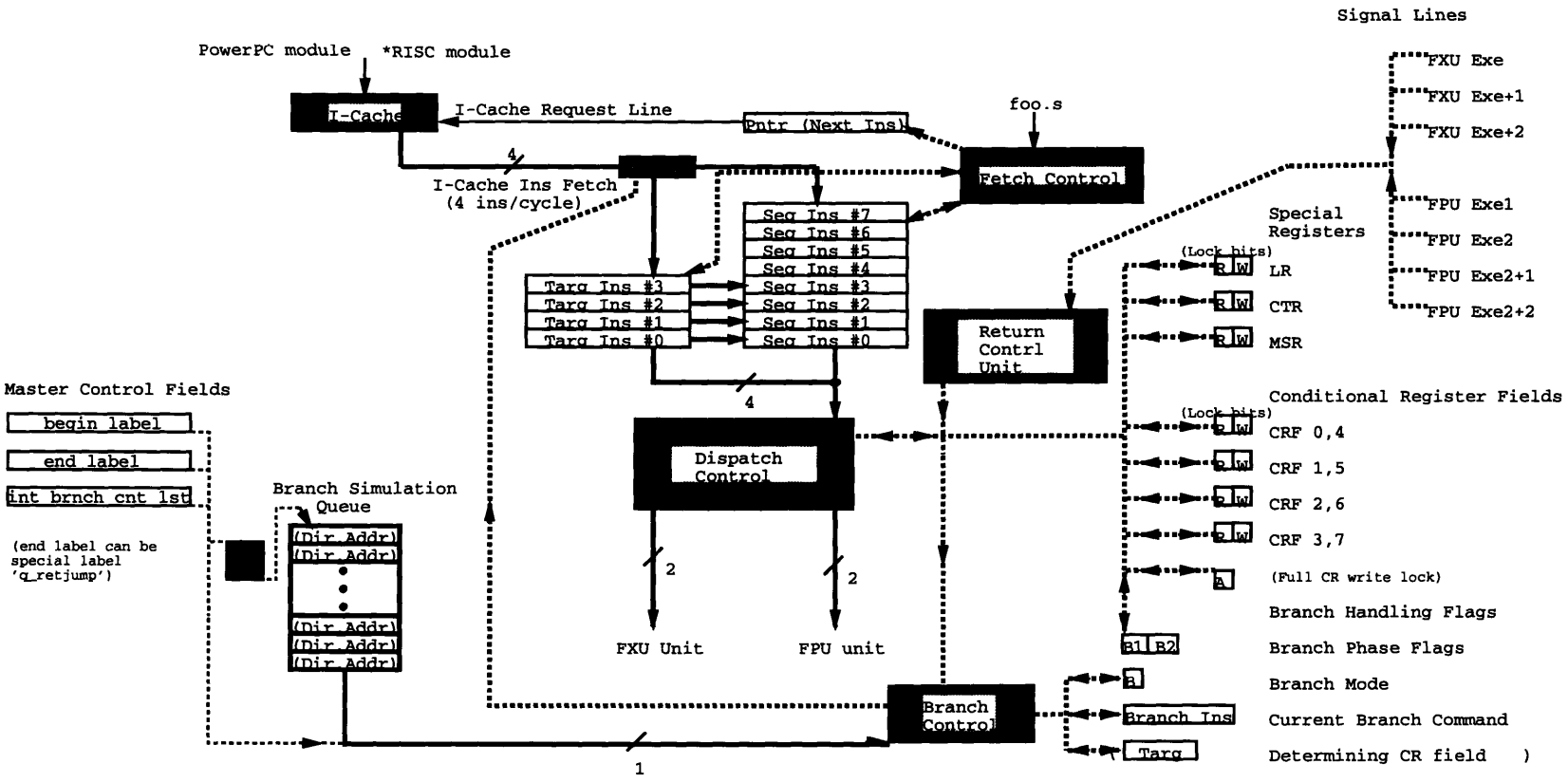

Figure 4-4: The RS/6000 Branch Unit.

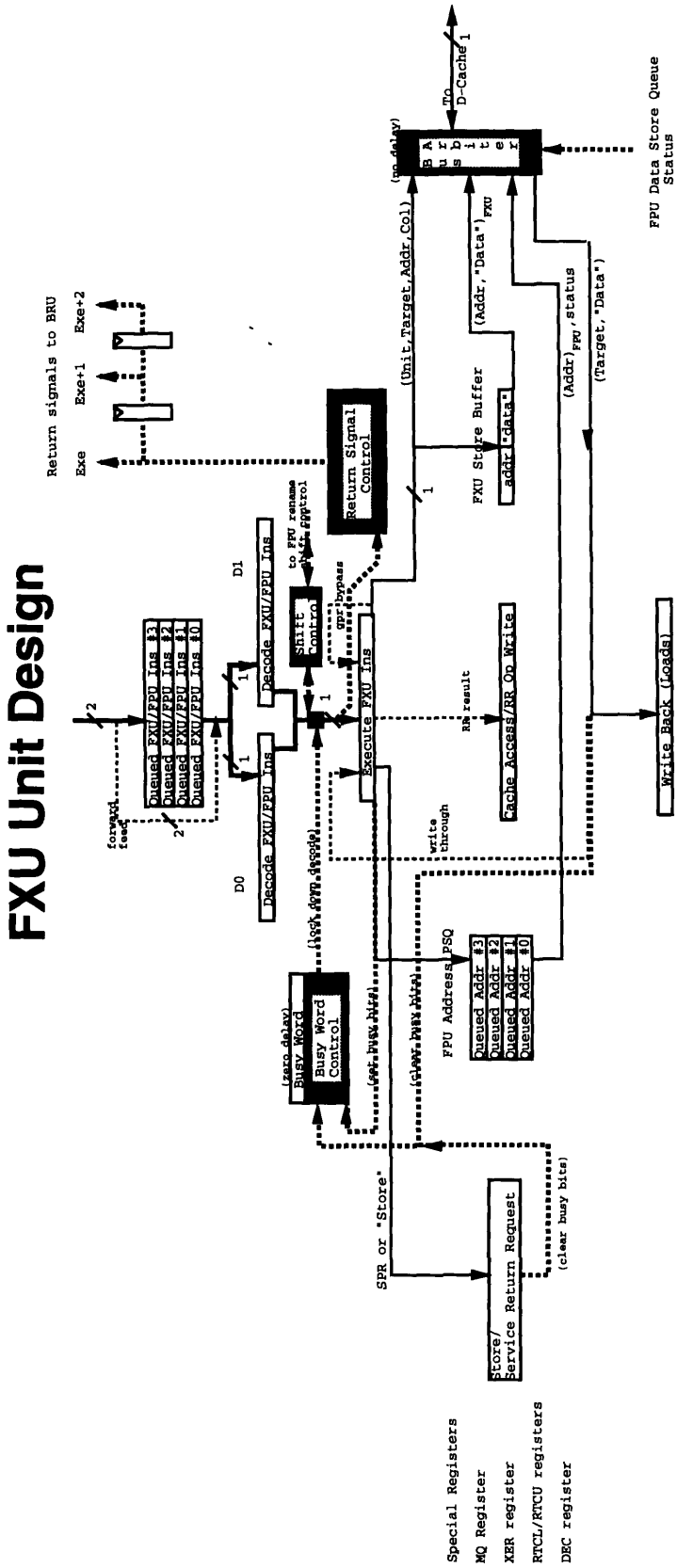

Figure 4-5: The RS/6000 Fixed-Point Unit.

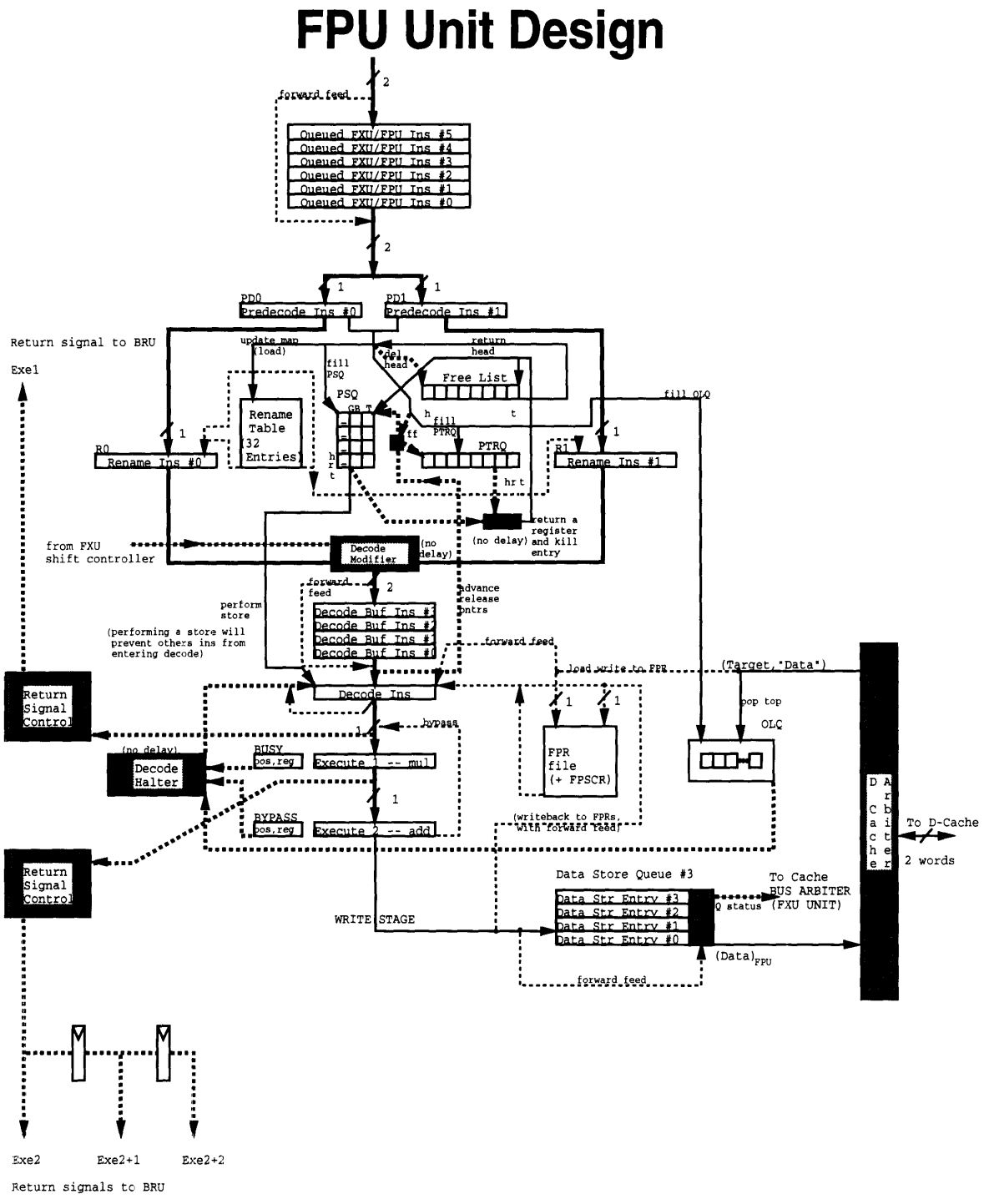

Figure 4-6: The RS/6000 Floating-Point Unit.

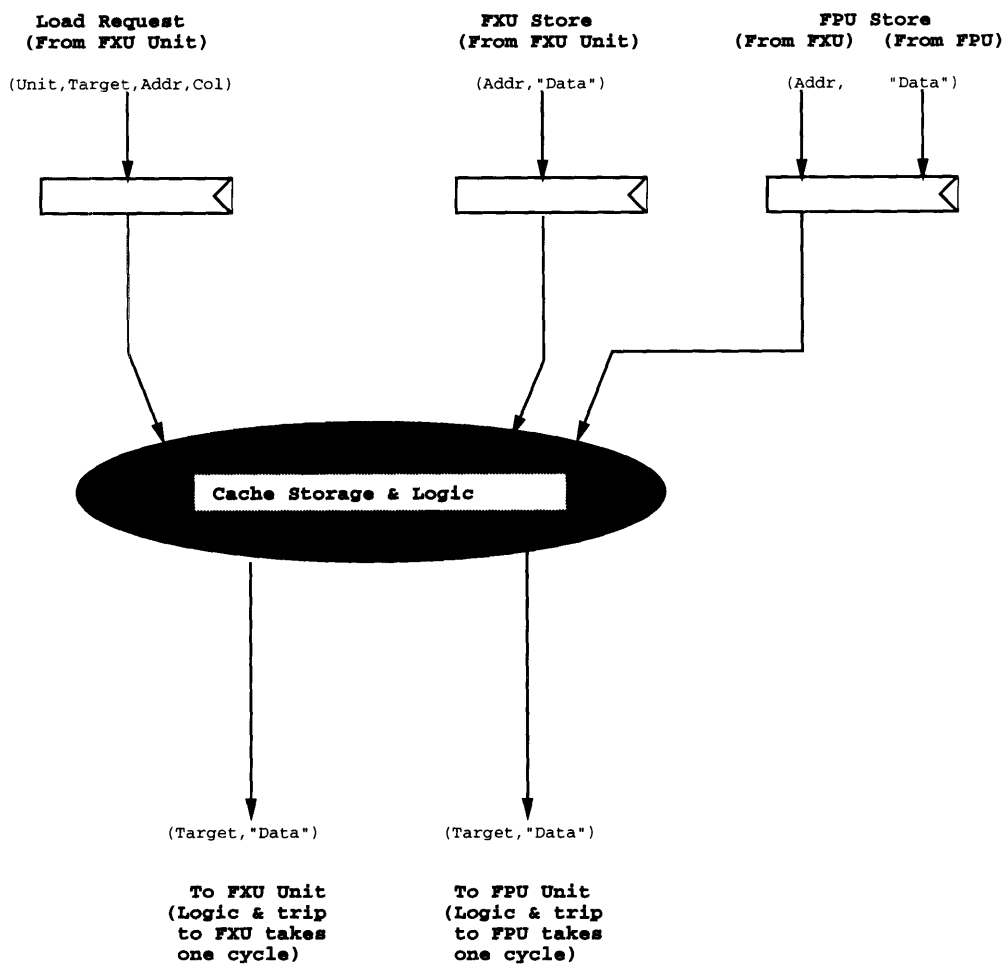

Figure 4-7: The RS/6000 D-Cache Unit.

although it will later cause a delay by sitting in the execution stage for several cycles, may well be able to enter a fixed-point instruction buffer as soon as it becomes eligible for dispatch. Stopping the time at the dispatch point would mean losing a significant portion of such an instruction's effect on the pipeline. The only bright point in this enigma is that by running the simulator through a partition from the beginning every time an interior basic block must be profiled, we can somewhat blur the accounting of which previous instruction caused which delay and still come up with a valid set of ideal times. The adopted solution to this problem was to cut off timing after the execute stage for the fixed-point unit, the first execute stage for the floating-point unit, and the dispatch stage for condition register operations and branch instructions. This strategy is not perfect but achieves the correct behavior as long as fixed point instructions begin new basic blocks, which occurs often.

Shown below in Figure 4-8 is a module-level view of the simulator code. Its object-oriented design is evident from the similarity to the RS/6000 CPU diagram seen previously. A test script was prepared to run the simulator on a suite of short code sequences so that the integrity of model could be quickly checked after making small corrections or additions. After much tuning of the various pipeline stages, the simulator was able to correctly simulate branch delays, moves to and from special registers, and large-occupancy instructions correctly. The C++ code is included in appendix A.

4.2 Instrumentation: A Detailed View

The major issues which must be resolved in actually instrumenting code are:

- What statistics are to be collected via profiling, and where are they to be stored during execution?
- What methods will be used to collect these statistics?
- How can the user direct the profiler as to which procedures should be profiled and which of the possible types of statistics should be collected?

We answer each of these questions in turn, starting with what information is to be collected and where it will reside during runtime. Note that runtime overhead—increased code size, data memory requirements, and execution time—all have deleterious effects on the truthfulness with which the profile data represents an execution of the unprofiled code.[5] This is especially true for increases in execution time. Thus, while analyzing the three guiding questions above to effect the

[5]Kesselman notes in his dissertation, that by keeping runtime overhead low, the probe effect can be eliminated leaving profiling code in even when statistics are not collected. Given the relatively higher overhead expected with QProf, however, this is not a viable option.

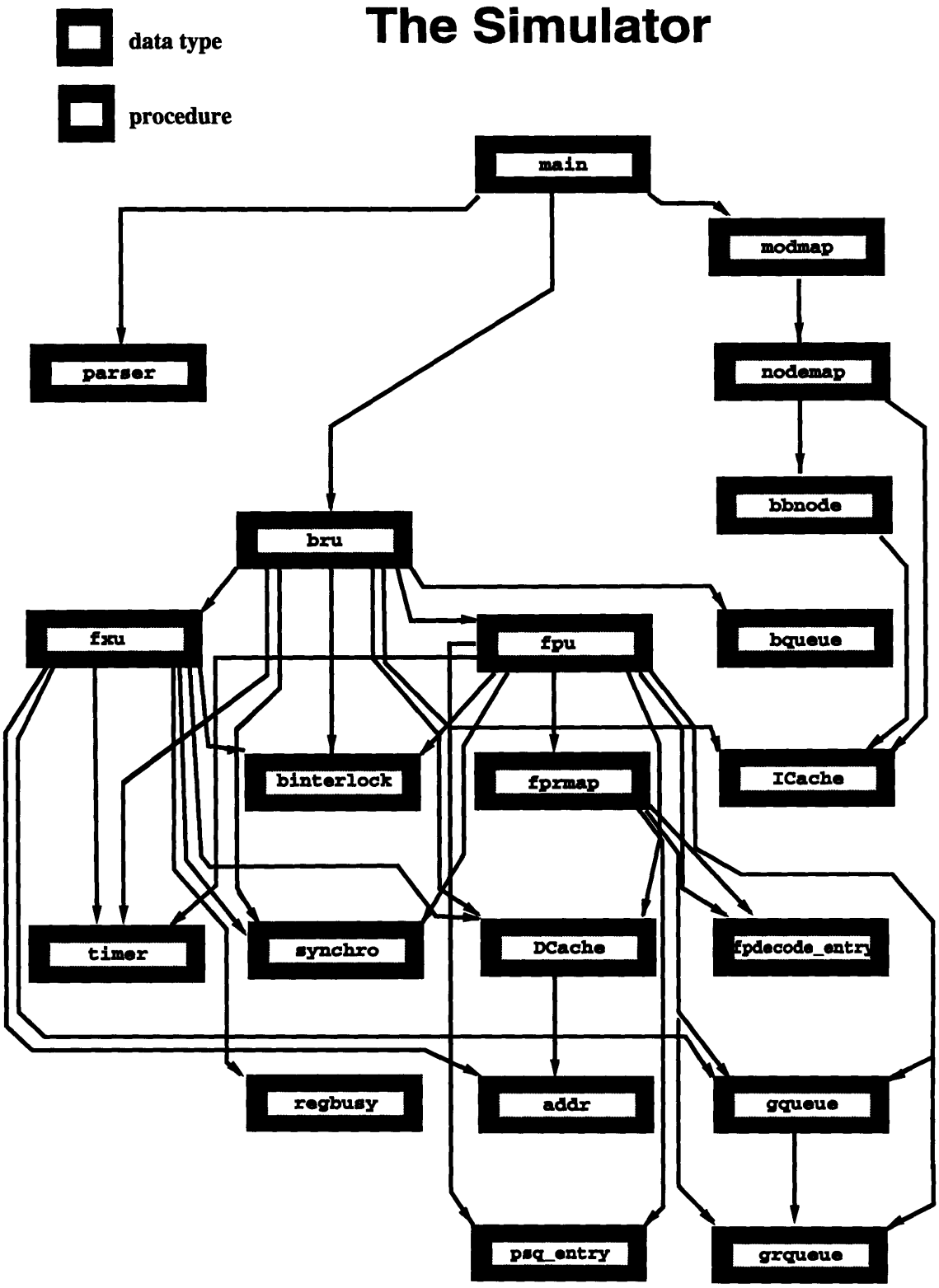

Figure 4-8: A module-level diagram of the data types and procedures used to implement a stand-alone version of the simulator.

implementation of the instrumentation phase, we also must keep in mind that after a working version of the instrumentation phase has been installed, we must work toward including time-saving optimizations.

In designing the original implementation of the instrumentor, we wanted to collect statistics, for each processor, on the number of times each basic block is executed and its total ideal and real execution time. It was thought at one point that even smaller sections of PowerPC code than basic blocks—such as the image of a single *RISC instruction under the *RISC $\rightarrow$ PowerPC translation function—could be instrumented with code for collecting elapsed real time. However, even before the coding phase, it became clear that such an object would almost certainly be too fine grain to profile. To see why, one must consider the RS/6000 on-board timer, the Run Time Counter (RTC) in relation to its CPU clock.

The RTC is accessed via a fixed point unit **mfspr** instruction which takes a single cycle in the FXU pipeline but has a latency of two cycles. For short intervals (<< one second), it suffices to sample only the lower word of the clock, which has a period of one billion nanoseconds and counts from 0 to 999,999,999 before rolling over. (Obviously the range of a 32 bit unsigned integer is larger than this, but it is not all used.) Although the upper 24 bits are implemented in all versions of the RS/6000, the lower 8 bits are not, with the result that "ticks" of the RTC in fact occur only every 256ns[6]. By contrast, the target RS/6000 model 550 used in this project had a CPU clock rate of *42 MHz.* Comparing the two, it becomes evident that *the RTC ticks only once every 10.8 CPU clock cycles.* Clearly, it's possible that problems might occur when targets must be measured which take fewer than 11 or so CPU cycles, or even of that order of magnitude. In mixed fixed and floating point code, superscalar effects imply that code segments of less than 20-30 instructions would fall in that range. We'll consider this effect in the interpretation of our results in later sections, but suffice it to say for now that attempting to profile below the basic block level would not be considered feasible.

In addition to real time intervals, it should be possible to collect statistics on the *types of instructions* executed, both at the PowerPC level and also at the *RISC level. For example, at the PowerPC level, we might ask how many fixed point operations were executed in relation to the number of floating point and memory operations. The crucial optimization here which saves greatly on runtime overhead, is that provided basic block counts are collected, this second set of statistics is *already determined.* To produce it, we merely read in the static instruction mix from simulator output file and scale each basic block's mix by the number of times it was actually executed. In order to have

[6]In the PowerPC 601, the RTC period is better—128ns; the PowerPC line beyond the 601 will include a Time Base (TB) rather than a RTC.

some sense of how many threads are forked or how many messages are sent in a given partition, the imix includes the distribution of instructions at the *RISC level as well, divided into categories such as *network, scheduling, and local memory operations.* As implemented, the instruction mix even reveals how many instructions were added by the register allocator so the user can gauge the local efficiency of the allocation algorithm on the target code.

Basic blocks are not the only unit of code which can be profiled; in particular, we would also like, for each Id procedure, on a per-processor basis, the number of times it was invoked, the total actual time that frames allocated it have been live, and the number of calls and elapsed time spent in each callee Id procedure. (For calls to C procedures rather than Id code, the actual execution time is collected rather than the live time of the callee's frame, since the call is not split-phase.) The overhead of these measurements is much less than that of the per-basic-block statistics, and indeed we find below that fewer optimizations are called for here than at the basic-block-level instrumentation points.

4.2.1 Profiling Data Structures

As shown in Figure 4-9, a two-level hierarchy is employed to store the statistics collected at runtime. When an Id procedure is called, the frame allocated contains slots for profiling the execution of each basic block and each called C function of that procedure. Using the activation frame to store this data takes advantage of locality, since the frame pointer is already in a register during the execution of any partition, and the page containing the frame is more likely to be in the cache than the page containing the code block descriptor. Storing the called-C-procedure statistics in the frame also makes sure that multiple invocations of the same Id procedure running on a given processor do not interfere with each other, as they would if they maintained non-atomically updated fields in the CBD. Each basic block is assigned four slots in the frame—one counting invocations that end in a fall-through, one counting invocations that end in a branch, and two that, together[7], store the total real time spent in the basic block. The distinction between fall-through and branching exits from a basic block is made because the ideal time required for a branching exit is longer than that needed for a fall through. The current implementation allows only a limited number of basic blocks per Id procedure, determined by a constant; to profile large routines, the constant might have to be increased.

Below the region used for basic blocks lies the called-C-function statistics area, where each called C routine is assigned five slots. The first slot counts the calls, the second and third, together, hold

[7]Real time is a 64-bit quantity.

the start time of a called routine during its execution, and the last two, together, hold the total real time spent inside the calls. At the very end of the frame we cache the time of the code block's allocation; the time can't be left in second-level, per procedure storage region since multiple live copies of a procedure would then each write into the same location.

The frame-level statistics are accumulated into the CBD at the end of each Id procedure call just before the frame is deallocated and the code block invocation count, located in the CBD, is incremented. In addition, the cached start time in the frame is used to accumulate the live time of each invocation into the total live time stored in the CBD. Finally, the CBD also stores callee invocation counts and live times, which can be used to generate an accurate call graph. As implemented, the accumulation of callee live time into the parent's CBD is the job of the *callee.* Looking at the end of its CBD to find its identification number, and using a pointer to its caller stored in the local frame, the callee can determine the correct parent CBD location to update.

4.2.2 Instrumenting a Partition

This initial implementation of Qprof applies a uniform instrumentation strategy to each partition. First, the partition is classified as either a cthread, inlet, or standard thread. Since the register allocator assumes the frame pointer (FP) always contains a valid storage base register, if the partition is a cthread—an *initialization thread* called from C—code must be added to save the nonvolatile registers, 13-31, as per C convention, and *move the frame pointer from the argument register (r3) where it lies upon partition entrance to the FP register.* Next, the basic block topology of the original partition is analyzed and a header and trailer are inserted before and after each basic block to increment the invocation count of that basic block and update its live time. The implementation distinguishes between fall-through and branching exits from a procedure, since they lead to different delays, but for illustrative purposes, a simplified header is shown in Figure 4-10 above which counts all invocations identically. The registers r100,r101,etc., are from the infinite register model since the instrumentation is added before register allocation. The exact header and trailer format used, showing how double trailers were added to each basic block, can be found in Figure 4-11.

The basic block ID is summed with the frame pointer before entering a basic block so that if the basic block exits via a branch and encounters a different trailer, its information still gets stored into the correct frame slots. If the base register were hard-coded as rFP, then we would have to place the branch ending a profiled basic block it after the trailer, outside the instrumentation code. Only one instruction of execution time overhead would be eliminated, but the I-cache overhead would be cut in significantly since the current scheme, requiring two copies the trailer for each basic block, one for fall throughs and one for branches to the succeeding basic block, could be abandoned in favor

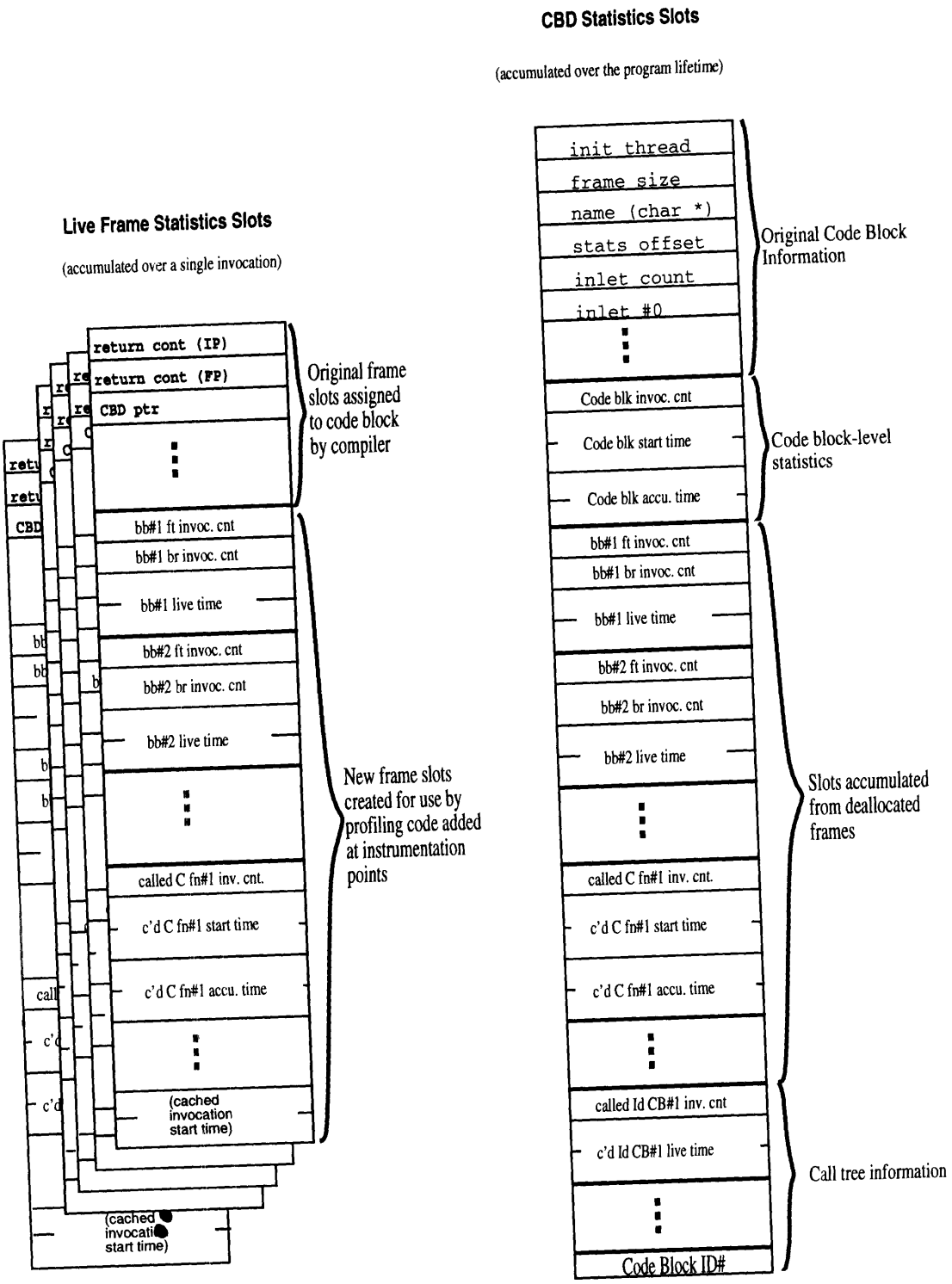

Figure 4-9: The QProf runtime data structures.

```
cal r100, 4*5 (rFP)   # save 4X the basic block's ID no., plus the FP, in r100
     mfrtcl rSTAT          # grab the lower word of the start time into a special reg.

     <original basic block, bb#n>


<bb#(n+1)'s label>:        # here's where jump entries to the next basic block enter

     mfrtcl r101           # grab the lower word of the end time and save in r101
     lwz r102, 0 (r100)    # grab the invocation count of the BB and save in r102
     lwz r103, 8 (r100)    # grab the low word of the BB's live time into r103
     addi r102,r102,1      # increment the invocation count
     stw r102, 0 (r100)    # store back the invocation count
     lwz r104, 12 (r100)   # grab the high word of the BB's live time into r104
     doz r102,rSTAT,r101   # put elapsed time in r102, 0 if the low word rolled over
     addc r103,r103,r102   # sum the elapsed time with the old live time
     addze r104,r104       # do the carry
     stw r103, 8 (r100)    # store back the low word of the live time
     stw r104, 12 (r100)   # store back the high word of the live time
```

Figure 4-10: A simplified view of the instrumentation header and trailer applied to each basic block. The doz instruction stands for "difference or zero," and prevents a rollover of the RTC's low word from throwing off the measurements.

```
cal r100, 4*5 (rFP)   # save 4X the basic block's ID no., plus the FP, into r100
     mfrtcl rSTAT          # grab the lower word of the start time into a special reg.

     <original user basic block, bb#n>

     # FALL THROUGH TRAILER -- put inv. count in first slot
     mfrtcl r101           # grab the lower word of the end time and save in r101
     lwz r102, 0 (r100)    # grab the invocation count of the BB and save in r102
     lwz r103, 8 (r100)    # grab the low word of the BB's live time into r103
     addi r102,r102,1      # increment the invocation count
     stw r102, 0 (r100)    # store back the invocation count
     b join_point

     # JUMP ENTRY TRAILER -- put inv. count in second slot
<bb#(n+1)'s label>:        # here's where jump entries to next user basic block enter
     mfrtcl r101           # grab the lower word of the end time and save in r101
     lwz r102, 4 (r100)    # grab the invocation count of the BB and save in r102
     lwz r103, 8 (r100)    # grab the low word of the BB's live time into r103
     addi r102,r102,1      # increment the invocation count
     stw r102, 4 (r100)    # store back the invocation count

join_point:                # The trailers share a large common section
     lwz r104, 12 (r100)   # grab the high word of the BB's live time into r104
     doz r102,rSTAT,r101   # save elapsed time in r102, or 0 if the low wd rolled over
     addc r103,r103,r102   # sum the elapsed time with the old live time
     addze r104,r104       # do the carry
     stw r103, 8 (r100)    # store back the low word of the live time
     stw r104, 12 (r100)   # store back the high word of the live time
```

Figure 4-11: The exact instrumentation header and trailers applied to each basic block. A fall through out of the original basic block takes the first trailer, and a jump entry to the following user basic block takes the second. The label of basic block #(n+1) is removed from its original location and placed as shown.

of a *single trailer approach.* However, since such an "optimization" would remove branches from measurable real time, the double trailer format was kept.

The only complication to this uniform instrumentation process occurs when a branch-and-link instruction is encountered. Since such an instruction signals a call to a C function, the instrumentor essentially rewrites the code to vector the C call through a special trailer which (1) updates the statistics information of the most recently executed basic block, and (2) records the current time to create a reference point. When the C call returns, the reference point is used to compute and store into the frame the real time spent in the call. To return to user code, the handler then makes a jump back to the header of the basic block following the original branch-and-link instruction. Relatively speaking, the overhead of this instrumentation point (a C call) is more acceptable than that of the previous set of instrumentation points (the basic blocks), since many C calls often take on the order of 1,000 or more cycles to complete.

The remaining runtime processing of the statistics data, such as accumulating frame information into the CBD before frame deallocation, and maintaining the invocation count and total live time of each Id procedure, is carried out by extensions to the Q runtime system. Specifically, the frame statistics slots are zeroed out, and the initial timestamp is cached in the frame, during *frame allocation*, and the accumulation steps and invocation count handling is done by a hook in the *frame deallocation routine.* Both of these modified Q files are listed in Appendix B. Before exiting, the instrumented program collects information from the CBDs and creates a file, **qresults**, representing the real time statistics gathered during execution.

Selective profiling, mentioned at the beginning of this section, has not yet been implemented in Qprof, although the basic block invocation counts and real execution delays could easily be made optional for each Id procedure according to user preference. It was felt that adding selective-profiling features would not be needed on the first version of Qprof, since they would likely provide little further insight into choosing the best instrumentation points and runtime storage strategies.

4.3 Integrating the Simulator and the Profiler

To implement the instrumentation algorithms discussed above, the **instrument-CB** and **instrument** procedures were designed and **ppc-testparse**, which converts *RISC to PowerPC assembly code, was modified to pass all code through the instrumentor as shown in Figure 3-1. Instrumenting a program generates a small file, **qstat-short** which contains basic block counts and code block names, etc.—items which would be hard for PostProf to reconstruct without help. Finally, **measure**, an extension to the simulator that records the *instruction mix* of each basic block in addition

to its ideal execution time, was incorporated into **ppc-testparse** so that the complete feed path of Figure 3-1, save for PostProf, was put into practice. Calling **measure** generates a long file of static measures such as ideal time, instruction mix, etc., named **qstat-long**. Both **measure**, **instrument**, and all the associated instrumentor and simulator extension data types and procedures are collected into Appendix C. A dependency diagram appears below as Figure 4-12.

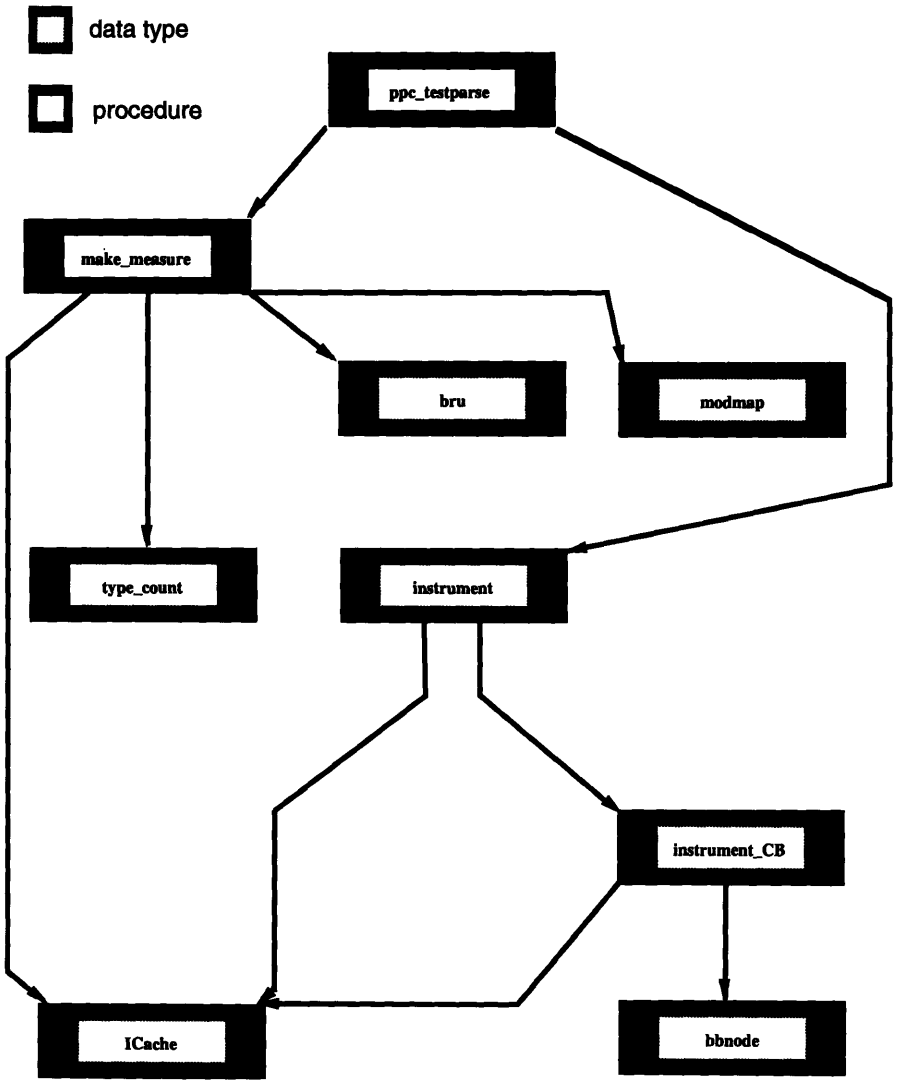

Figure 4-12: An overview of the data types and procedures used to extend ppc-testparse to comply with the diagram in Figure 3-1

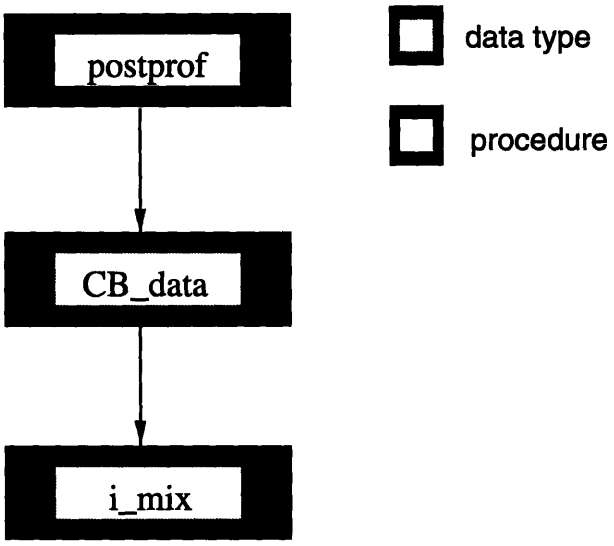

Figure 4-13: An overview of the data types and procedures of PostProf.

4.4 PostProf

Since the current implementation of Qprof runs on a single processor, the sophisticated graphics originally conceived for PostProf were not needed. Instead, a simple interactive graphic visualizer was written which pipes data through **gnuplot** in order to display it for the user. As currently written, PostProf can display the total live times and relative live times per invocation of each partition in an Id code block, or the live times and invocation counts of all C procedures called by an Id code block, and the live times and invocation counts of all *Id* procedures called by a given code block. Furthermore, each partition can be examined individually to see how the theoretical and actual execution delays compare, and to discover the PowerPC or *RISC instruction mix. PostProf responds to standard emacs keystrokes for "down" and "up" to advance through the code blocks, and to "left" and "right" keystrokes to advance through the display options available within a single code block. In order to see the instruction mix of a partition, one need merely to select the partition by moving to it using the "go right" command, and then invoke Control-S. A useful feature of PostProf is the *rerun* option, which is activated by Control-L; it runs the executable again to generate a new set of runtime statistics. The modules of PostProf are shown in Figure 4-13, while the code appears in Appendix D.

Chapter 5

Evaluating Qprof

Since the full Q compiler is not yet available, Qprof has been applied to two *RISC programs: **call-test-1.st**, representative of a simple Id program which returns a reference to a pair of values, and **fact.st** (see Appendix E), an iterative factorial function. To get the desired profile, we call the script **qprof** on the *RISC code, which invokes the new **ppc-testparse** routine to translate *RISC into PowerPC. By the time **ppc-testparse** is done, the two static measure files, **qstat-short** and **qstat-long**, have been written out, and **target.s** contains the instrumented assembly code. Next, **target.s** is assembled and linked with the Run Time System, and the resulting executable **target.qpf** is called. As **target.qpf** runs, statistics are collected and, before **target.qpf** is done, dumped to the file **qresults**. The profile can be viewed by calling **PostProf** as soon as the three statistics files are available.

5.1 PostProf as a Performance Visualizing Tool

Shown below in Figure 5-1 is a graph of *partition execution time* for **fact.st**'s first code block as generated by PostProf. Each partition is represented by two bars—the bar on the left represents the total amount of execution time accumulated by the given partition, and can be read against the Y-axis scale, and the bar on the right is a *dimensionless quantity* representing the *relative* execution time *per invocation* of the same partition. Whereas the left bar, representing total execution time, can be read off in CPU cycles against the Y-axis, the right bar cannot be since its height only has significance *relative to the per-invocation execution time shown for the other partitions.* The purpose of the right bar is to allow the per-invocation execution times of partitions to be quickly compared. For code blocks whose partitions are not all invoked the same number of times, it should be clear that total partition execution time, summed over all invocations, cannot be used for this purpose,

since it is weighted in favor of the more heavily-invoked partitions. In such a case, the right bar of each partition becomes useful. However, as total execution time was felt to be the most useful gauge of code block performance, it was given primary emphasis and control of the Y-axis.

The convention of pairing two bars with each partition, one of them a dimensionless quantity, was adopted to allow the two quantities to be displayed on the same screen with a single marked Y-axis. Above the left bar, the total number of invocations of that partition is listed in the form of a factor such as "100X", and at the top of the screen some statistics on the whole code block are listed, such as the number of times it has been invoked, its total *live time*, and its total *work time*. While live time measures how long an activation frame for the given code block remains allocated, work time measures the time spent in the code block executing *user code*. For a code block which calls C or Id procedures, the total live time will exceed the total work time. A similar plot for the second code block in **fact.st** is shown in Figure 5-2. Note that most time is spent inside FACT.part0 where the iteration occurs. The large discrepancy between live and work time is due mostly to the overhead associated with allocating and deallocating the activation frame, and scheduling—live time begins at some point during allocation, and ends at some point during deallocation. A less significant factor is the 80% instrumentation overhead present in FACT, which appears as live time but not work time

Postprof can also focus on performance data from a single partition. In Figure 5-3 the execution of the partition FACT.part0 is broken down into statistics for each basic block. Each basic block is represented in the plot by two vertical bars—the left bar representing actual execution time and the right bar indicating the *ideal* execution time in the absence of cache misses, interrupts, etc. Below the pair of bars is listed the total number of invocations of the particular basic block. From the plot, it's evident that most of the time spent in this partition is spent in the sixth basic block, the most complex basic block inside the iteration loop. The *RISC and PowerPC instruction mix graphs for FACT.part0 follow as Figures 5-4 and 5-5. From the *RISC instruction mix plot of FACT.part0, it's clear that the register allocator is responsible for generating more PowerPC instructions than all the *RISC instructions in that partition *combined*! The reason is that register allocator for the Q back end currently spills *every register*, storing each value into the frame after it gets produced, and loading it back right before it is needed again. Hopefully a better register allocator will be available for Q in the future. (The *RISC instruction mix is computed by mapping each PowerPC instruction present back to the *RISC instruction type which generated it, or to the register allocator.)

PostProf is also capable of displaying a performance snapshot of calls to Id procedures and C routines. After running Qprof on **call-test-1.st**, this feature of PostProf produced Figures 5-6 and 5-7. In the first of the two, we see that much more time is spent in C calls allocating and deallocating frames than in heap operation C calls. The difference between the total execution time

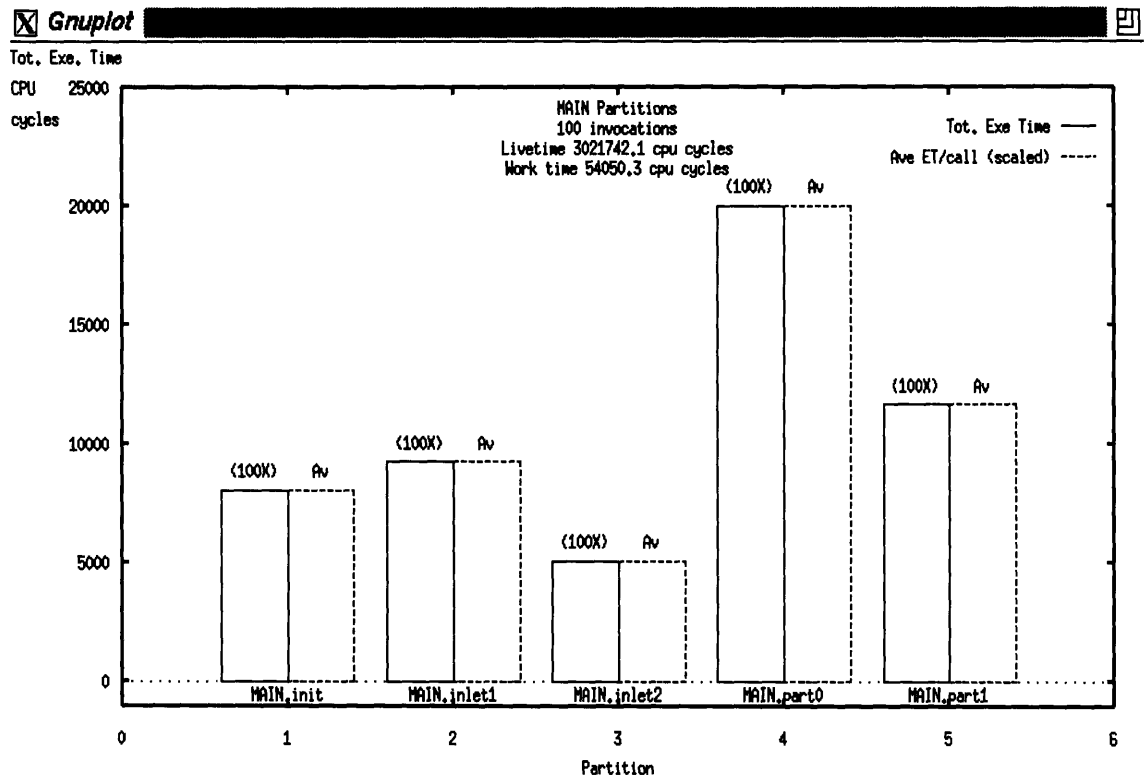

Figure 5-1: PostProf displays partition execution time for the partitions of fact.st's first code block.

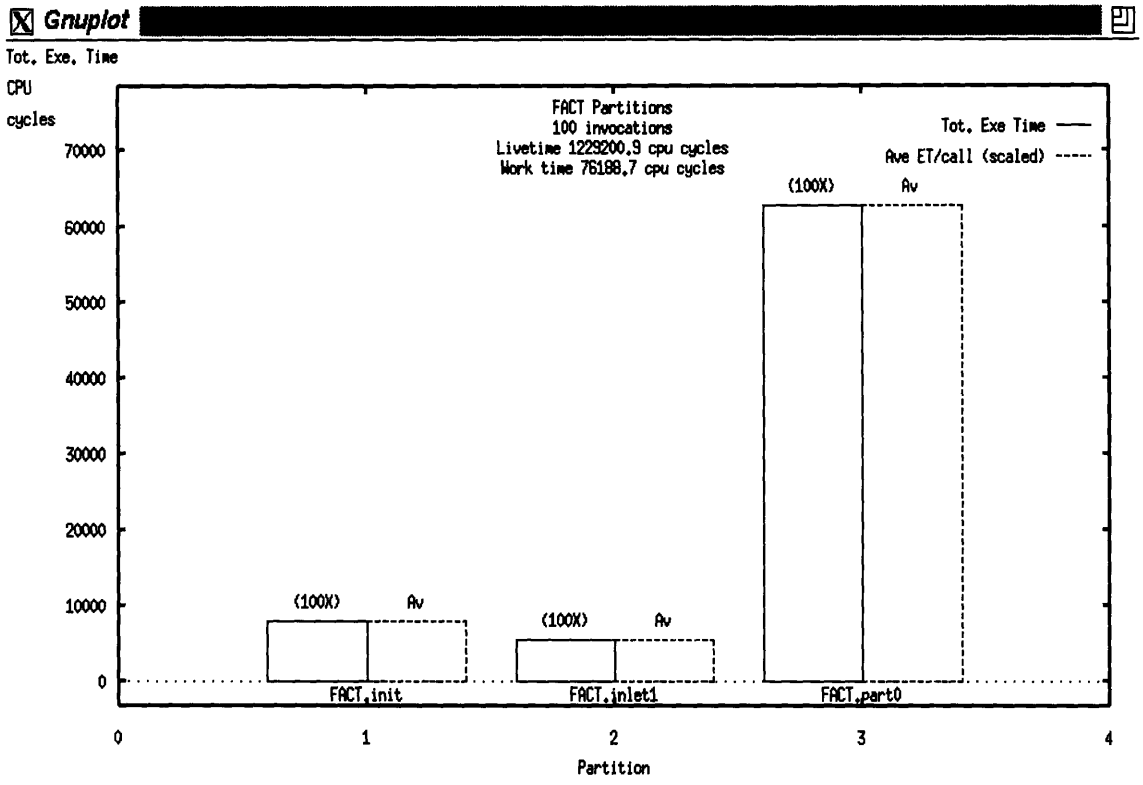

Figure 5-2: PostProf displays partition execution time for the partitions in fact.st's second code block.

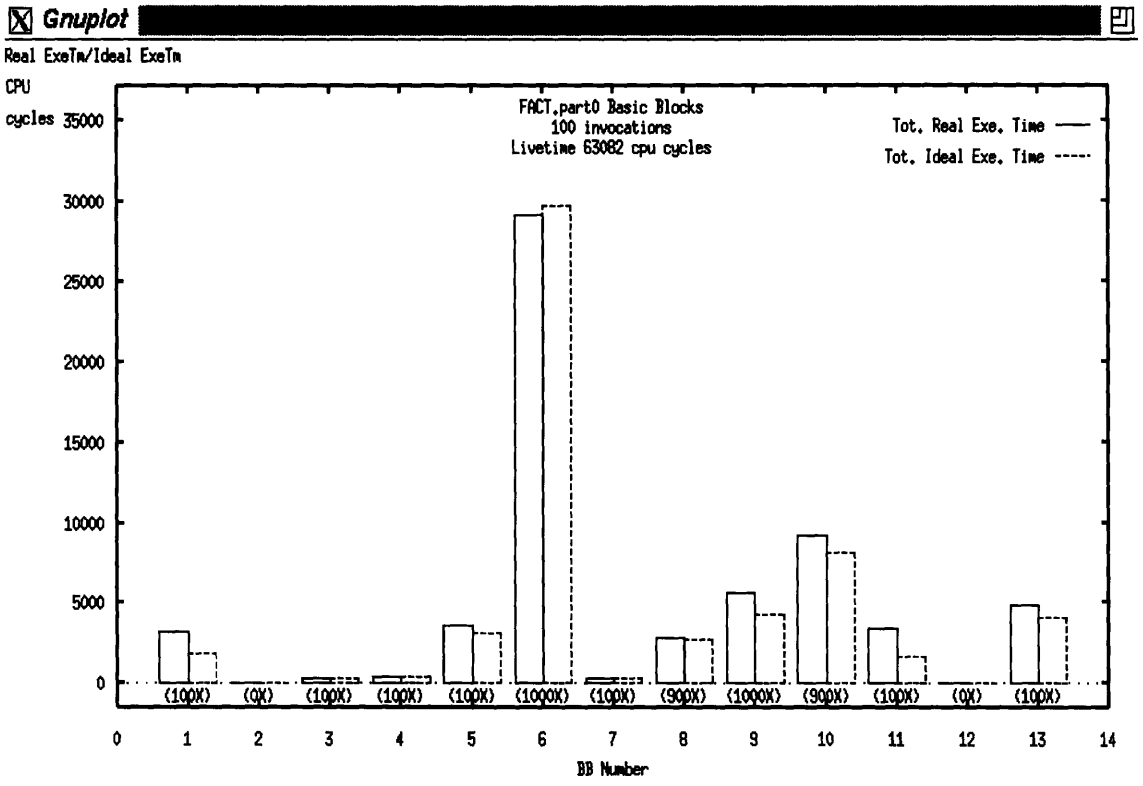

Figure 5-3: PostProf displays the actual and ideal execution times of the basic blocks in partition FACT.part0.

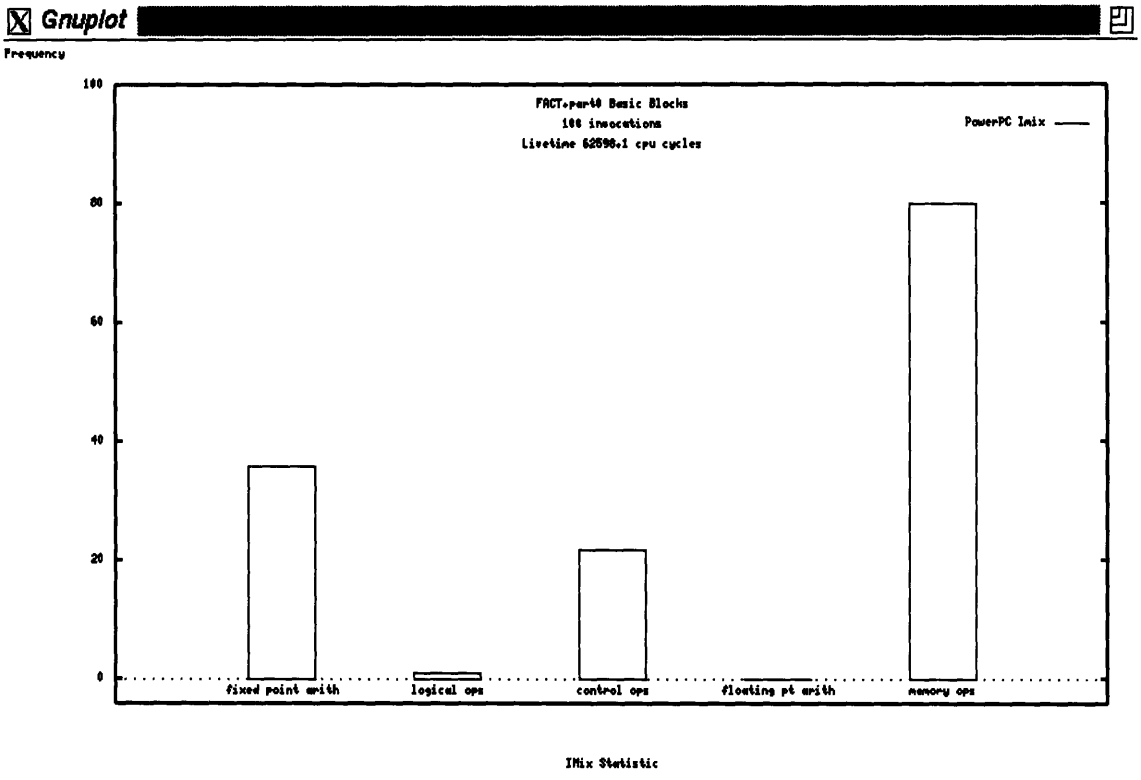

Figure 5-4: PostProf displays the PowerPC instruction mix of partition FACT.part0.

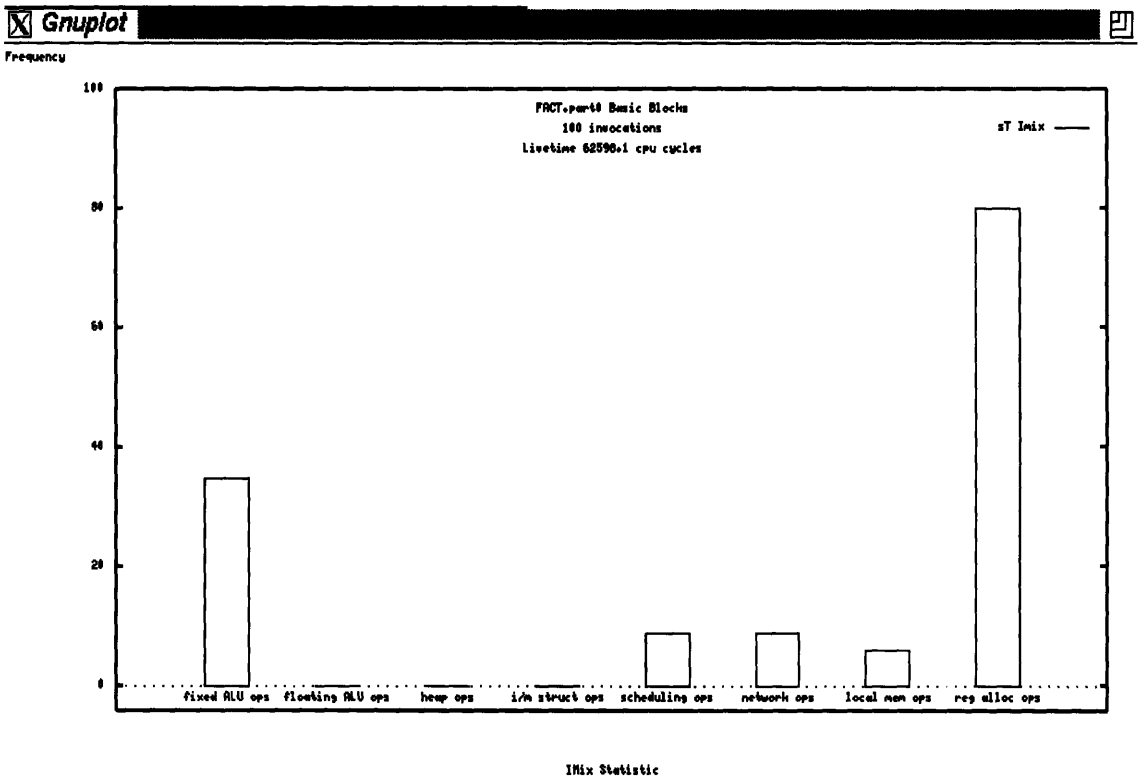

Figure 5-5: PostProf displays the *RISC instruction mix of partition FACT.part0.

and relative execution time per invocation can be seen in the case of RTS-istore-handler, which was called twice as often as the other three C procedures and which thus has a *relative execution time per invocation* which is shorter compared to its total execution time than is the case with other three procedures. Looking at the second figure, we see that no calls to MAIN or TEST01 were made from procedure TEST01, but that one call is made to TEST02 each time TEST01 is run. In the event several code blocks were called from within a single Id procedure, this call-graph snapshot provided by PostProf would allow the user to determine which callee produced the longest delay.

In summary, **Qprof** can measure the real execution time of Id procedures, much as **prof** can calculate the real time spent in C procedures. Moreover, the callee invocation counts and callee live times which Qprof maintains for each Id procedure can be used to construct a call graph similar to, but more accurate than, the one produced for C routines by **gprof**. The additional accuracy comes from the fact that gprof propagates execution time up the call graph by using invocation counts, whereas Qprof collects actual callee live time *at run time*. Finally, Qprof also keeps track of the *ideal time* required by basic blocks, much as **pixie** does. However, unlike pixie, Qprof can use ideal time to present the user with *memory hierarchy overhead time*, since it also has real time measurements of execution available. In short, Qprof incorporates the features of prof, gprof, and pixie into a cohesive whole that can accomplish more than any tool designed around a single approach to performance measurement. Best of all, Qprof is *scalable* to a network of symmetric multiprocessors.

5.2 Shortcomings of QProf

Although an effective performance visualizing tool even in its first version, QProf suffers from some difficulties. The most obvious is that Qprof generates a large instrumentation overhead. For available test programs, the overhead has in some cases approached 100 percent. In other words, the instrumented executable takes twice as long to run as the original user code, despite the effort made to keep the instructions added during basic block instrumentation to a bare minimum. Now on a *single processor*, profiling overhead for schemes such as the one we employ only perturbs cache hit rates since no live instrumentation registers are maintained inside user code. While altered cache behavior can indeed create a probe effect, there can be no doubt that the probe effect of such a large overhead would be much more significant in a network of processors in which timing patterns could be thrown off. A remedy to this problem is discussed in the next chapter.

Aside from question of overhead, the most puzzling result obtained is perhaps the large variation between observed and ideal time for some basic blocks profiled. Because of the inaccuracies present in ideal time calculation when a basic block in a partition can be preceded in execution by more

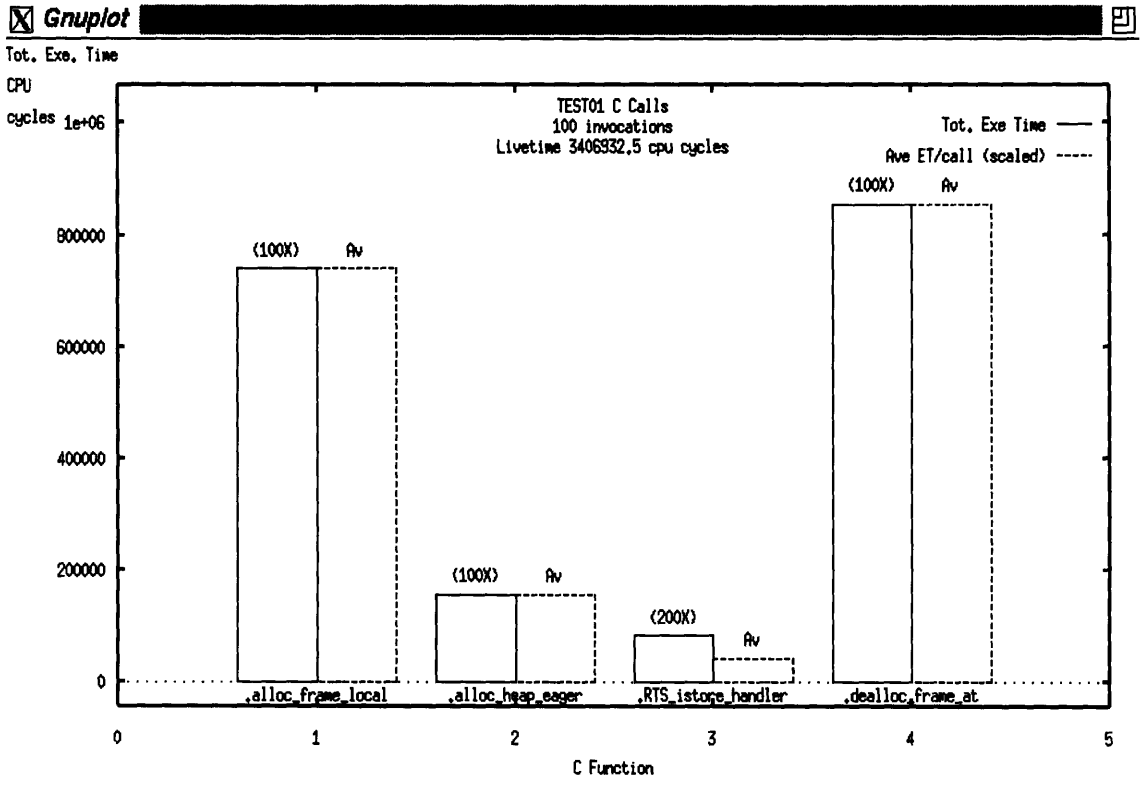

Figure 5-6: PostProf displays the time spent in C calls inside the code block TEST01.

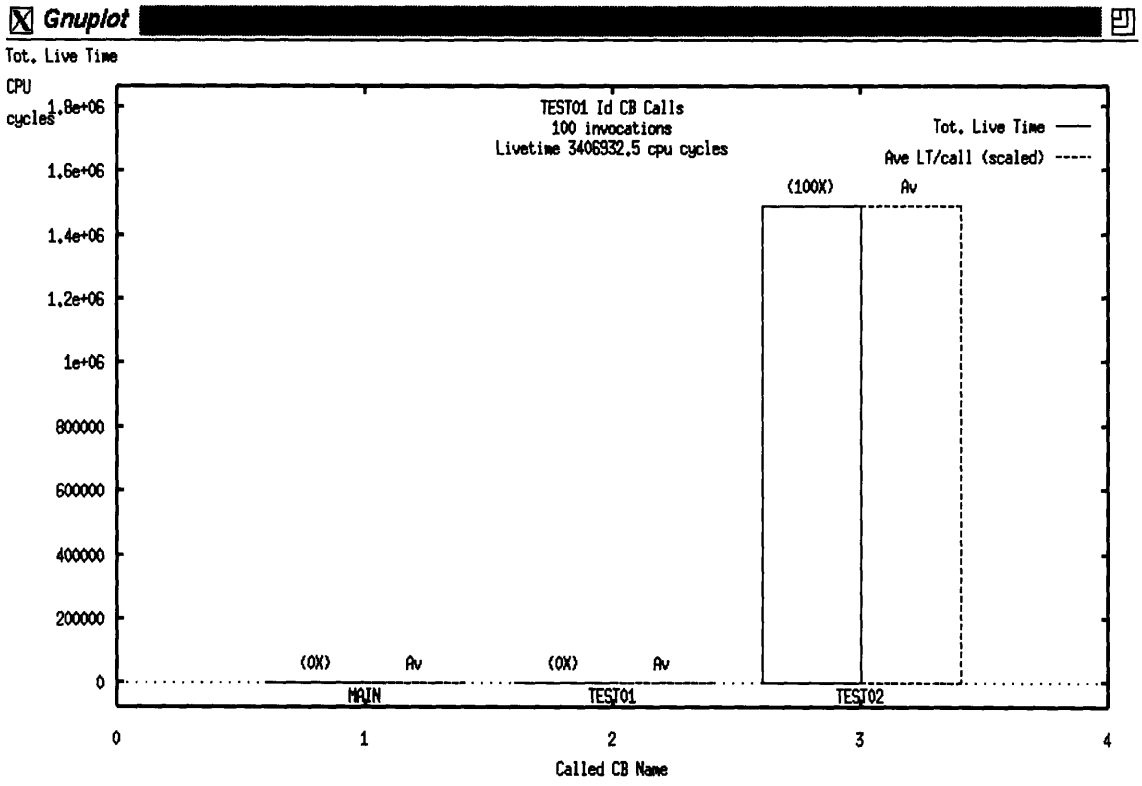

Figure 5-7: PostProf displays the live time and invocation count of each Id procedure (code block) called from within TEST01.

than a single ancestor, the calculated ideal time Qprof displays is sometimes one or two percent off the true ideal value. This explains why the real execution time is sometimes shown marginally below the the ideal time in PostProf plots such as Figure 5-3, where such a difference can be seen for the sixth basic block. Even when present, this phenomenon has never been significant enough to be considered a a flaw in Qprof's design.

By contrast, during the testing of Qprof it was often found that the real time required to execute a segment of code was significantly greater than the ideal time, even in cases where no memory hierarchy effects should be present. A typical example of this can been seen in Figure 5-8, in which the third basic block's actual execution time is about 16 cycles, or 89%, longer than the predicted ideal. A number of factors may be responsible for the inflation of real time recorded.

The first possibility is that the simulator is deficient in a way which causes it to underreport the total number of cycles required in some instances. To eliminate the simulator from consideration, a simple experiment was performed. Given a basic block with nearly equal ideal and actual times according to PostProf, a fixed number of arithmetic instructions were inserted into the block. When PostProf was run, it was found that the actual time per invocation of the basic block in question had jumped by a significantly higher amount than the number of arithmetic instructions inserted. This indicates that the simulator is not the source of the overreported actual time, since the simple code sequence inserted had a trivially-easy-to-compute ideal time, and still the actual time expanded by a significantly larger amount.

Interrupts need not be considered as the possible source, since the real time had a tendency to jump up in steps during the arithmetic instruction insertion experiment discussed above. In other words, the real time remains *nearly constant* as 1 through n arithmetic instructions are inserted into a basic block, but suddenly jumps to a higher value when the $(n + 1)$st instruction is added. An interrupted instruction stream would have a higher measured real time delay, since interrupts take a while to service, and would not have the particular step-like behavior described for such short sequences ($<<$ an interrupt interval) of inserted instructions.

The next potential culprit, the graininess of the Run Time Counter relative to the CPU clock, was known and feared before Qprof had even been written. In the case of the 42 MHz RS/6000 550 used in the testing of Qprof, the RTC ticks only once every 10.8 CPU clock cycles. It follows immediately that unless several iterations are performed through a basic block with the RTC phase randomly aligned with the first instruction, small basic blocks cannot be accurately timed using the RTC. To attempt to prove this factor responsible, the executables originally compiled for the 550 were recompiled with a few modified constants and run on an RS/6000 320, whose CPU clock runs at 20 MHz but whose RTC clock has the same frequency as the 550's. If the clocking graininess

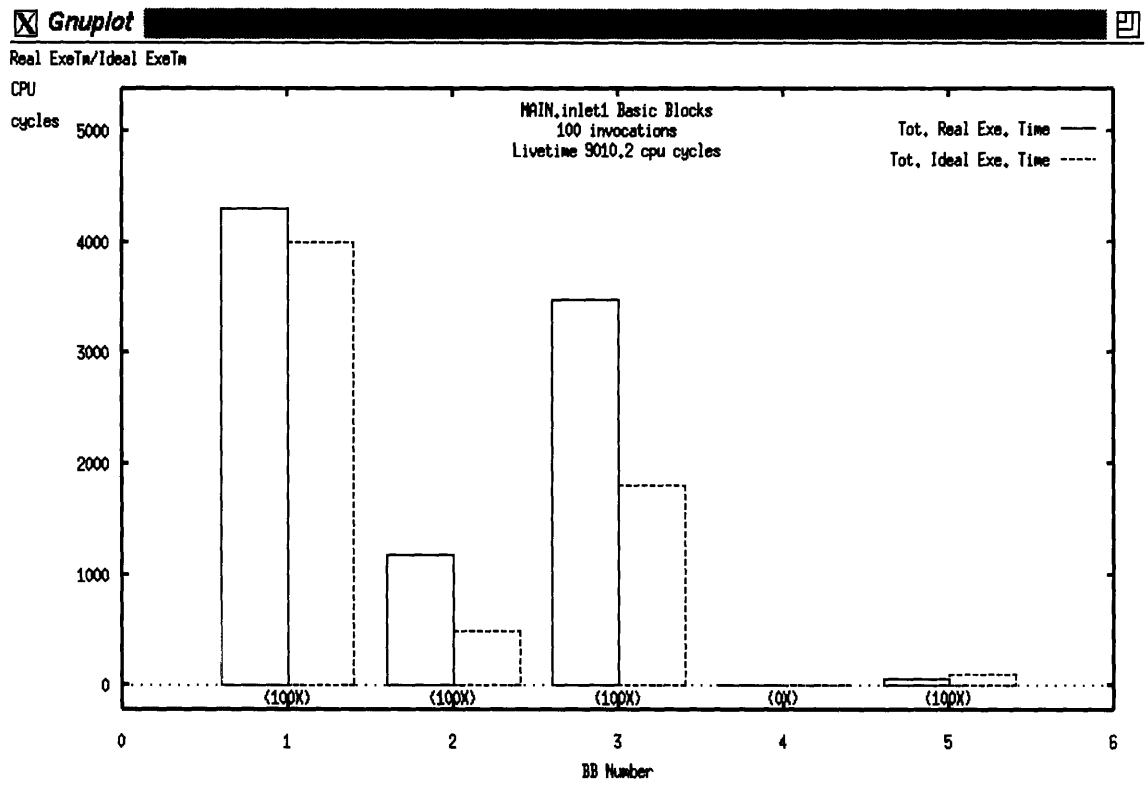

Figure 5-8: An illustration of the discrepancies between ideal and actual time in one of PostProf's performance snapshots of a partition. It happens to be a the second partition in fact.st's first code block.

were responsible for the elevated real execution time, the discrepancies on the 320 should be nearly half what they were on the 550.

Surprisely, *no substantial change* to the real time inflation was observed when running executables on the 320. When the third basic block shown Figure 5-8 is examined by Qprof on the 320, the overreported actual time for the third basic block is still present. Further investigation running a large number of iterations of fact.st revealed that the Qprof runtime system subroutine designed to randomize the phase of the RTC with respect to the timed edges of each basic block, or some other factor such as interrupts, was effective at making the granularity of the RTC disappear when many iterations were run. When the Qprof executable for fact.st is run 1000 times in succession to accumulate data for 1000 trials, the results produced by the 320 and the 550 are nearly identical. Unfortunately, this disqualifies timer resolution as a candidate for the observed anomaly.

Certainly the actual time *will* reflect memory hierarchy overhead as it was designed to do, but there is still a legitimate question as to whether the *particular anomaly observed* is also due to memory access operations. However, the previous experiment also sheds light on the current question. Namely, since adding a sequence of *arithmetic* operations to a basic block produced a large differential between actual and ideal time where no significant difference had existed before, data memory operations don't seem to be involved. Here we ignore the possibility that the remaining instructions from the original basic block, perhaps containing some memory operations, are made to take D-cache misses by the newly inserted arithmetic instructions. Our justification is that register operations inserted directly into the assembly code should not change D-cache behavior.

Only one possibility remains—I-cache misses. In order to check whether the I-cache could be responsible for the anomaly, a more detailed study of real time inflation in the presence of added instructions was conducted. The target basic block was the one discussed in conjunction with Figure 5-8, and the added instructions were **addi** operations. First on the 550, and then on the 320, the real time inflation was measured as a function of added instructions for 1000 iterations of fact.st. The resultant plots appear as Figures 5-9 and 5-10. As can be seen, instead of the expected linear relationship, the plots are *quasiperiodic* with a period of 16 instructions. This means the period cannot be related to the RTC or CPU clock. In fact, it turns out that since the I-cache has a line size of 64 bytes, this is exactly the number of instructions in an I-cache line! The conclusion seems clear that the I-cache is responsible for the observed behavior. As a check, note that the plateaus are separated vertically by roughly 20 cpu cycles on each plot, exactly the time required to execute a row of instructions from the I-cache *and load a second row from main memory.* To see this, note that when an I-cache miss occurs, a delay of eight cycles ensues, after which the 16 instructions in the desired line return two per cycle. Since the fixed point unit has a four instruction input buffer,

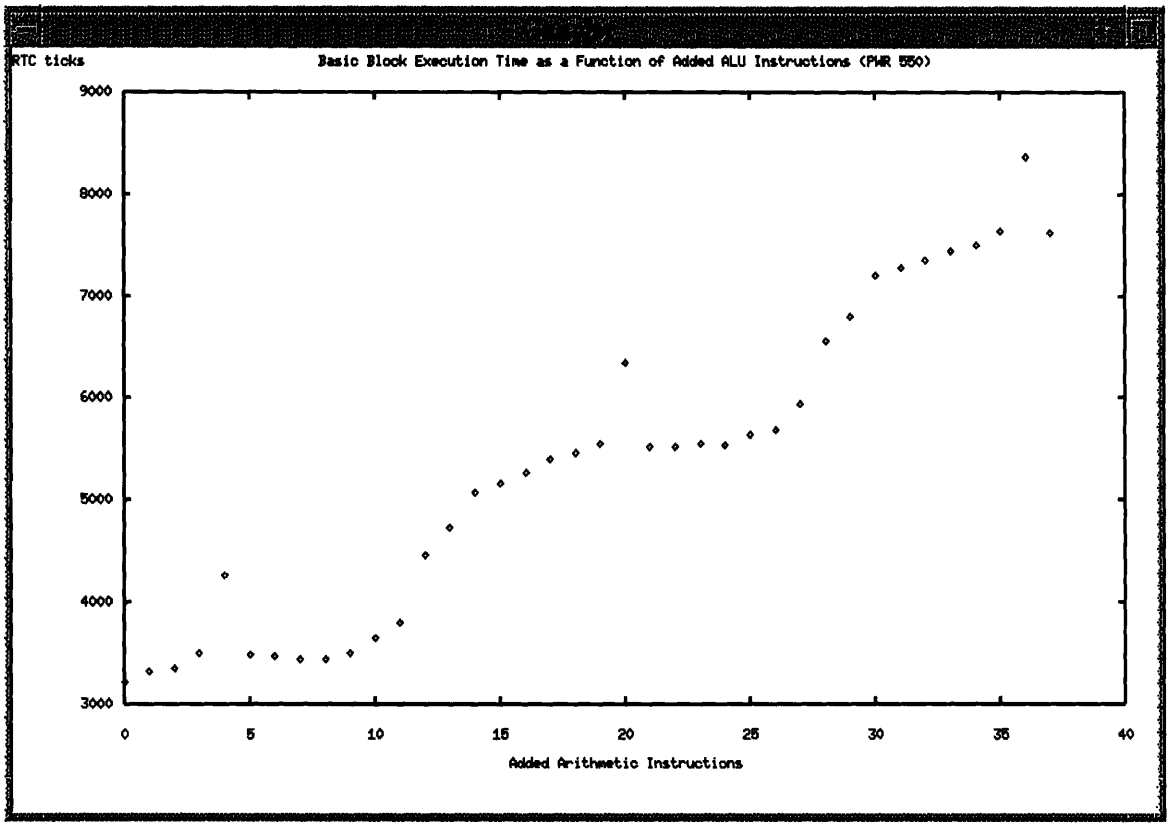

Figure 5-9: A plot of total real execution time as a function of the number of instructions added to a basic block. This plot represents 1000 runs of fact.st on the RS/6000 550.

this means that every 16 instructions a delay should be seen of exactly $8-4=4$ cycles. When these four cycles of delay are added to the 16 required to execute the instructions in a line, we obtain the 20 cycle figure as just mentioned.

Although the exact shape of the graph cannot be explained, the sharp rises probably occur when the last instruction in the basic block, a branch, advances from being the last instruction in a particular I-cache line to being the first instruction in the next line loaded. Since the branch automatically causes a new I-cache line to be loaded, pushing it off the end of the line causes an addition 8 cycle delay to arise which cannot be covered. The flat portion before this sharp rise occurs when the branch is near the end of the I-cache line. Since the branch target line must be fetched from main memory anyway, several instructions may be executed "for free" during the fetch latency.

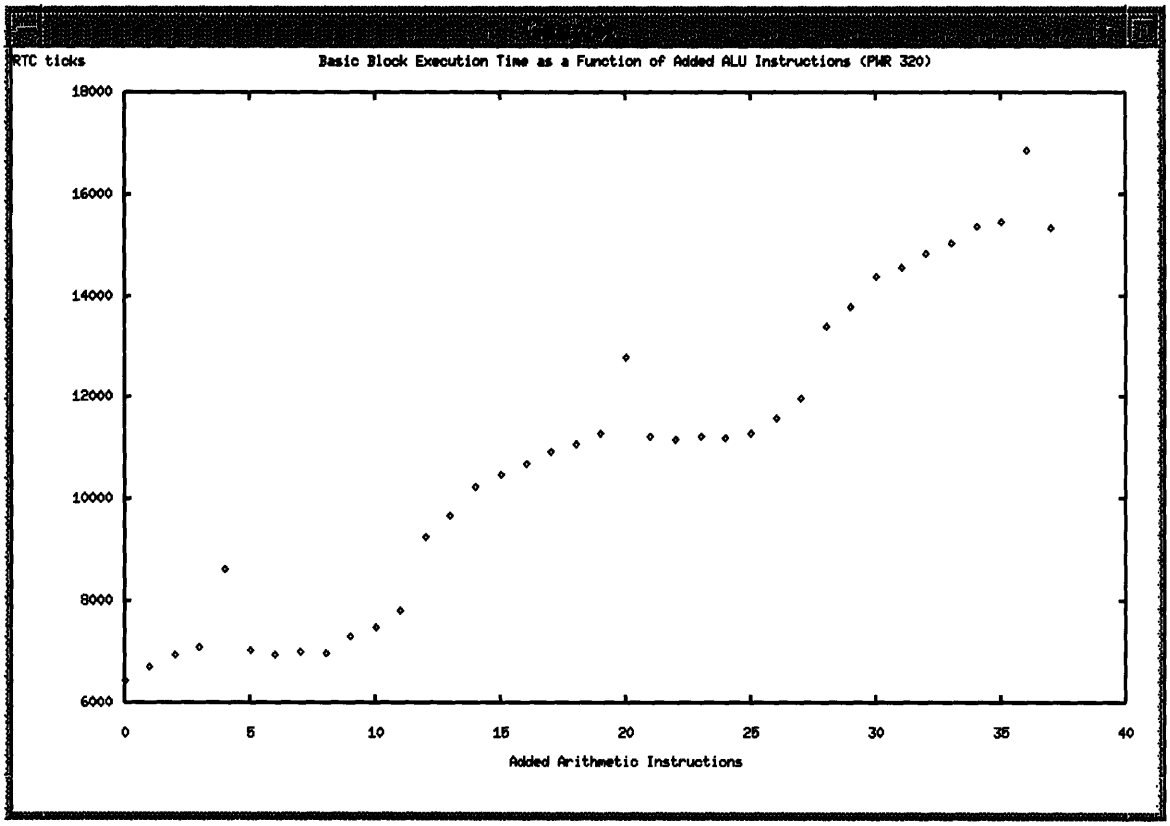

Figure 5-10: A similar plot, again showing the total real execution time as a function of the number of added instructions. This plot represents 1000 runs of fact.st on the RS/6000 320.

Chapter 6

Desired Improvements

After having implemented an initial version of Qprof, it has become clear that although it is an effective profiling tool on a single-processor system, its present overhead will generate a heavy probe effect in parallel environments. Four key optimizations must be made to the current version of Qprof before it can become an effective tool on a multiple-processor network. Runtime statistics must be stored more efficiently, basic block counter placement must be optimized, instrumentation points must be moved to larger objects, and the instrumentation code must be made to use both functional units.

6.1 Storing the Runtime Statistics More Efficiently

First, the runtime system must be modified to allow a variable number of statistics slots to be stored in each CBD. The number of storage slots available in the CBD should be a *function of the code block* rather than a global constant. Such an improvement would both save space when few statistics need be collected in a given code block, and at the same time allow for a code block requiring an arbitrarily large number of statistics slots. Instead of actually making the CBD of variable size, the goal can be accomplished as shown in Figure 6-1, where only two dedicated slots in the CBD are required to accumulate basic block and called procedure statistics. When a CBD is created, the top location is set to the number of slots that will be required to store runtime statistics on all the basic blocks in that code block together with all the procedures called by that code block. In the bottom location, we store a pointer that is originally *null*. If a given code block is eventually invoked during the execution of a particular program, space to hold the precalculated number of slots is malloc'ed, and a pointer to that space is stored back into the bottom location. This approach has an advantage over simply making the CBD's of variable length. Namely, memory is conserved because the space

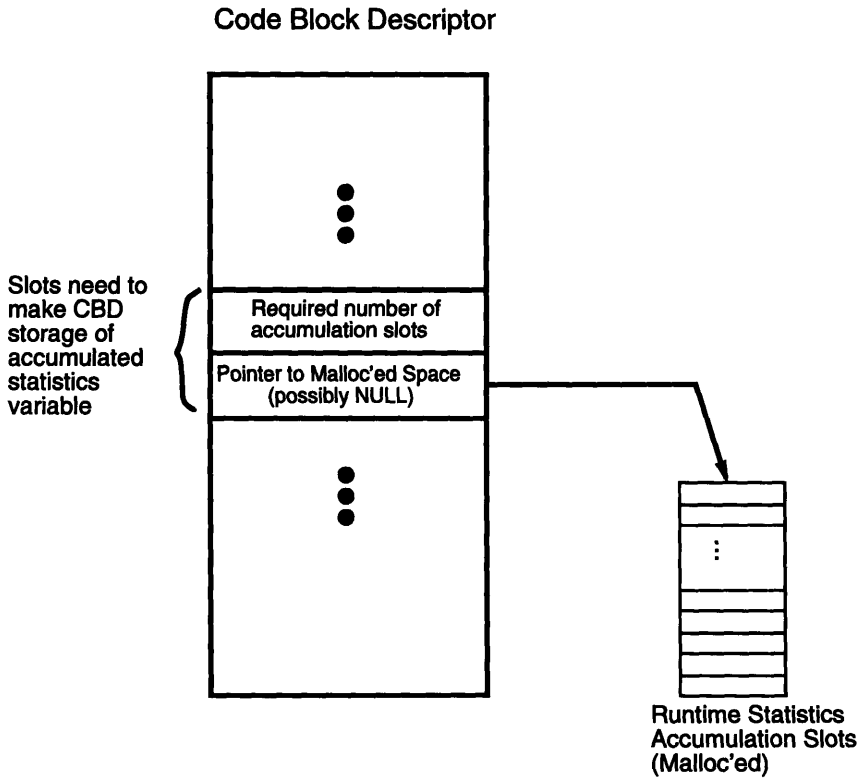

Figure 6-1: A scheme for making variable the amount of profiler storage associated with CBDs. The top slot contains a count of how many words should be malloc'ed by the runtime system, and lower slot contains a pointer, possibly null, to the allocated space.

is not allocated unless the associated code block is called at least once.

6.2 Separating Real Time Collection and Invocation Counting

Next, the maintenance of basic block counts must be divorced from collection of elapsed real time to allow graph-theory optimizations to be performed on basic block counter placement. As discussed in-depth in the last section below, a minimal set of counters placed at the correct locations in each code block can be used to reconstruct a full set of invocation counts for the basic blocks of every partition. Unless real time collection and invocation counting are separated, therefore, the real time measurements at many basic blocks will be lost. A slight complication that must be dealt with is that *two levels* of control graphs must be studied in working with the set of basic blocks in a code block—the *RISC control flow between partitions, and the assembly language control flow between basic blocks in a partition. Even a naive application of the techniques of Section 6.6 should result

in an overhead reduction of 50% or more.

6.3 Collecting Real Time at Coarser Intervals

Unlike invocation counts, real time must be collected at every point where it is desired—it can't be reconstructed from a small set of measurements. Since it cannot be so optimized, the instrumentation to collect it must be moved to *coarser objects* to alleviate the heavy overhead currently experienced at each basic block. To maintain an acceptable level of overhead, the instrumentation points for real time must be changed from basic blocks to *partitions*. As it turns out, 8 out of the 13 cycles in the current Qprof header/trailer are required to accumulate the real time, and this is too great a price to pay at each basic block. However, most partitions are long enough to make such a one-time overhead acceptable. In addition, if the programmer knew a given application would run for strictly less than one second, those eight cycles could be optimized down to five by ignoring the upper word of the time. Since real time must be used to find memory hierarchy delays, our change will mean that such effects can only be localized to a partition rather than a basic block, but this is not that bad of a situation since partitions represent a relatively fine-grained division of an Id procedure.

6.4 Using Both the Fixed and Floating Point Units

Finally, the invocation count headers and trailers should in some cases be made to use *floating-point registers*. Since independent pipelines exist for floating-point and fixed-point operations, such a scheme could save at least one cycle at each instrumentation point. No workable means exists to perform the real time collection with the floating point unit, since the real time must be brought through the fixed-point unit during retrieval from the RTC. However, the invocation-counting code can easily be made to use the floating-point unit by permanently assigning one of the floating-point registers to the profiler and keeping the value 1.0 in it at all times. While dependent floating-point operations do create a pipeline bubble because of the latency of the two-stage multiply-add logic, this delay only occurs *if the second instruction is a floating-point register-to-register operation*. If the second instruction is a store, no delay ensues because of the floating-point unit's *pending store queue* and *data store queue*. Fortunately, the code sequence for updating an invocation count follows that pattern exactly—a calculation followed by a store. What's more, since the fixed-point unit often runs ahead of the floating-point unit by a few instructions, and the fixed-point unit is the one which computes the address of a floating-point load and makes the D-cache request, the one cycle load-use delay found in fixed-point code is usually not present in floating-point code.

6.5 Improvements for the Multiple-Processor Environment

Other optional improvements might also be considered. For example, we could collect the *idle time* on a per-processor basis. The idle time is defined as the time spent unsuccessfully in the scheduling loop looking for enabled threads in allocated frames belonging to user code blocks. By modifying the runtime system's low-level assembly kernel, the scheduling loop could be appropriately instrumented to collect elapsed time. A possible technique for more accurately gauging the cause of idle time would involve dividing the idle time evenly among the code blocks associated with live frames (on that processor); this would allow the idle time to be a per-code-block rather than a per-processor statistic. The thinking is that the live code blocks are waiting for some result to arrive, and that if it was simply the case that all computation at that processor were done, all frames would have been deallocated.

As a tentative proposal, it might also be possible to collect total synchronization times—total actual time between arrival of the first and last thread at a join control point, summed over all invocations—for each *RISC join in a code block. In order to facilitate such a scheme, room would have to be made in the statistics slots of the *frame* for a timestamp, and additional overhead would have to be taken at each *RISC join. Whether or not the feedback is worth paying the overhead is a matter for further research.

6.6 Basic Block Invocation Count Optimizations

In order to run the algorithm allocating a minimal set of basic block counters in the most efficient configuration, a complete map of control flow through the basic blocks of a code block must be obtained. This entails a two-step process. First, working at the partitioned program graph or *RISC level, a heuristic must be applied to each code block to build a directed graph of a special type, T. What makes graphs of type T special is that each node can either represent an ordinary graph node, or contain a *list* of graphs, themselves of type T. In such a scheme, partitions in *RISC become ordinary nodes, and control flow such as a conditional branch can be represented by multiple edges leaving a single node. The special feature of type T is used to represent synchronization. Sequences of partitions which must synchronize with respect to each other are placed within a single node, and can themselves contain control-flow substructure. A sample type T graph is shown in Figure 6-2; it happens to be the graph of type T corresponding to the **Fib** code block described in Culler's TAM paper [3].

Second, after a graph of type T is obtained for each *RISC-level code block, a simple basic block detection algorithm like that implemented in the simulator's **nodemap** module can be used to

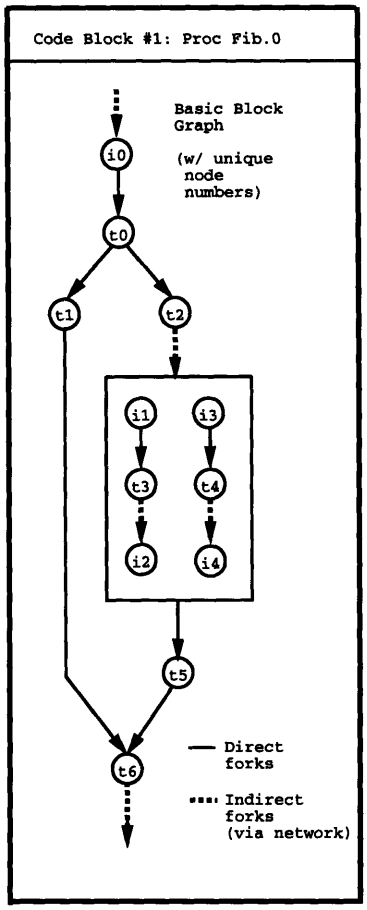

Figure 6-2: An instance of a graph of type T. This one represents the *T code block implementing Fibonacci in Culler's TAM paper.

construct a directed graph of the basic blocks in each partition. After flattening[1] the compound nodes by recursively replacing each one with its interior control graphs linked in series, and inserting each partition's basic block map into its node on the graph, a single directed graph of basic blocks is obtained for the entire code block. The reason this strategy yields a valid graph for the purpose of invocation counting is that treating the parallel, synchronizing threads as sequentially executed threads enforces the constraint that if one thread in the set is executed n times, every member of the set must be.

6.6.1 Adding Optimal Counters to a Basic Block Graph

Our clever idea is the following: consider the control flow graph as an undirected graph G, and remove a spanning tree to generate G'; then, the flow of control in G (in terms of number of traversals of each edge) is uniquely determined by the flow of control in G'. Thus, by removing a *maximum* spanning tree (where the edges have been weighted by expected number of traversals), the remaining edges contain all necessary information to reconstruct an invocation count of the BB's, and have minimal overhead[2]. To recover the "lost" edge data later in PostProf, we follow Goldberg's algorithm [5]:

1. Set the labels of all edges in G to 0.

2. For each edge e that is not in S = G-G', there is a unique cycle in G, called a fundamental cycle, comprised of e and edges from S. Traverse the cycle clockwise, edge by edge. If an edge is in S and the direction of traversal matches the direction of the edge, add the label of e to the label of the edge, otherwise subtract e's label.

A major remaining problem is how to select a *maximal* spanning tree. One possibility is that multiple passes could be made with Qprof, so that execution statistics from the first pass could be used to reduce the cost of profiling the second run, perhaps leading to more accurate measurements. However, a cleaner approach would be to use a heuristic to artificially weight the control graph before removing the maximal spanning tree. Goldberg suggests the a method which relies on a depth-first search to weight control-flow nodes heavier at the "bottom" of the graph. When integrated with the original idea of removing a spanning tree, we get the following algorithm:

[1] To flatten a compound node in a graph of type T, the interior graphs are removed and placed in sequence. In other words, when control flow enters the compound node, it goes first to the initial node of one of the interior graphs, travels down that graph until it reaches a node with no successor, and then enters the first node of one of the remaining interior graphs. After all interior graphs are exhausted, control flow leaves the compound node.

[2] A slight complication is that self-loops and loops between a pair of nodes in the original control-flow graph must be expanded to three node loops before subtracting the spanning tree. Another complication is that if some user calls (e.g. `exit`) do not return to the user code, yet are not in basic blocks represented by leaves in the control flow diagram, additional edges must be added to the graph

1. Perform a depth-first search and assign post-order numbers to nodes.

2. Label each edge with the post-order number of its source node. (These numbers will range from 1 to N.)

3. Assign edges corresponding to induction variables the label 0 (highest priority for instrumentation).

4. Add N to the label of loop return edges that do not correspond to induction variables (lowest priority).

5. Assign pseudo-edges from the leaves to the root (added to satisfy Kirchoff's laws) the priority N+1

6. Eliminate the maximum cost spanning tree (which contains the lowest priority edges) and select the remaining high priority edges to measure[3].

In Goldberg's tests, this heuristic scored better than random selection of a spanning tree on most benchmarks, although far from ideal. Larus & Ball [9] suggest another heuristic in their paper on optimal profiling; the heuristic assumes (1) each loop executes 10 times, (2) if a loop is entered N times and has E exit edges then each exit edge gets weight N/E, and (3) predicates are equally likely to take any of their non-exit branches. The evidence in favor of the Larus & Ball heuristic is slight but it may be the better of the two.

Thus, by first creating a control-flow graph of the basic blocks in a code block, and then removing the maximum spanning tree, we are left with the set of edges that should be instrumented to collect basic block counts with the least overhead. In cases where the user code already maintains a loop counter, counting along the loop return path can be done for free. Together with the other optimizations, minimizing the number of instrumentation points as just discussed should lower the overhead of Qprof enough to permit successful parallel profiling in a multi-computer environment.

[3]Dikjistra's Algorithm will efficient find a maximal spanning tree.

Chapter 7

Summary

Since the commonly available Unix tools for profiling the performance of a program lack the resolution to make fine-grain time measurements and aren't designed to be effective in a multicomputer environment, Qprof was developed, and implemented as an *integral part* of the Q runtime system and the IBM POWER RS/6000. Taking advantage of the data structures of the runtime system to store statistics near where they are updated, and relying on the high-resolution on-board timer of the RS/6000 processor set, Qprof can successfully generate real time measurements of basic block and C subroutine execution as well as Id procedure live time. By combining statically collected measures such as basic block ideal execution time and instruction mix with real time measurements and invocation counts, Qprof can present the user with an accurate picture of memory hierarchy hot spots and dynamic instruction mix. Qprof even collects *call-tree* information (to a single level) so a user can at a glance determine which Id procedures are called by a given code block, how often they are called, and how much time they remain active. Although the current implementation of Qprof suffers from a heavy overhead (100% in some test cases), methods have been outlined to optimize overhead down to an acceptable level by measuring real time intervals at the partition level rather than the basic block level, and by including only the *minimal set* of basic block counters at strategic locations rather than blindly counting invocations at each basic block. When Qprof is eventually reimplemented in a multicomputer setting, the post processor will be written using graphic presentation techniques such as the three-dimensional histogram described by Kesselman in order to communicate the performance statistics cleanly and effectively.

Appendix A

Appendix: The Simulator Code

```
// QPROF main.C file -- sybok@athena

// Generate the Ideal Time of a basic block specified by a label

#include <iostream.h>
#include <string.h>

#include "ppc_common.h"
#include "bru.h"
#include "modmap.h"
#include "err.h"

#include "TOC_Entry.h"

#include "Qparams.h"

#include "sT_Inst.h"
#include "sT_Rator.h"
#include "sT_Rand.h"
#include "sT_Reg.h"

#include "sT_to_PPC.h"
#include "ICache.C"
#include "parser.C"

void main(int argc, char* argv[])
{ String t_label = *(new String("begin"));
  bool exit_direction = TRUE;
  bool repeat = FALSE;
  bool print_module = FALSE;
  char dir_prefix[100] = "";
  char* test_module;

  switch(argc) {
  case 5:
    print_module = (strcmp("-p",argv[4]) == 0);
  case 4:
    repeat = (strcmp("1",argv[3]) == 0);
    print_module = print_module || (strcmp("-p",argv[3]) == 0);
  case 3:
    exit_direction = (strcmp("1",argv[2]) == 0);
    print_module = print_module || (strcmp("-p",argv[2]) == 0);
  case 2:
    test_module = strcat(dir_prefix,argv[1]);
    break;

  default:
    sigerr("main: Improper number of arguments passed to qprof.");
  };

  ppc_Mod& mod = *file_to_ppc_Mod(test_module);

  ICache* ic1 = new ICache(mod);        // generate 1st ICache
  ICache* ic2 = new ICache(mod);        // generate 2nd ICache

  modmap* mmap = new modmap(ic1,mod);

  if (print_module)
    mod.print();

  // generate the list of all possible branch charts

  branch_chart_list& bcl =
    mmap->make_branch_map(t_label,exit_direction,repeat);

  // This section can be enabled to print out the branch chart that
  // will be used by the simulator.

  /*

  for(DListIter<String*> iter(*bcl[0]); !iter.end_p(); iter++)
    cout << *(iter.value()) << endl;

  */

  // create a trial branch unit using the head of this list

  bru rs6000(*ic1,*ic2,*bcl[0],t_label);

  // report the results -- the Ideal Time

  cout << argv[1] << "," << exit_direction << "," << repeat << " = ";
  cout << rs6000.time_code() << endl;

};
```

```
// QPROF bru ADT -- sybok@athena

// This ADT is designed to simulate the high level BRU operation.

#ifndef QPROF_BRU
#define QPROF_BRU

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"

#include "timer.h"
#include "synchro.h"
#include "binterlock.h"
#include "bqueue.h"

#include "regbusy.h"
#include "DCache.h"
#include "ICache.h"

#include "fxu.h"
#include "fpu.h"

#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include <iostream.h>
#include <String.h>

// some useful enums

enum bfstatus {OFF, WAIT_ON_ADDR, CALC_ADDR, WAIT_ON_FETCH, LOADED};

void swap_icaches(ICache*& ic1, ICache*& ic2);      // forward declarations

// the class 'bru'

class bru
{
 public:

  // the constructor for a BRU; it's used by the high-level caller

  bru(ICache& ic1, ICache& ic2, DList<String*>& branch_chart,
      const String& start, const String& end = end_default);

  int time_code();                    // simulate RS6000

  void bru_tick();                    // advance the BRU one tick

 private:

  fxu* fxu_unit;              // pointers to various subordinate fuctional units
  fpu* fpu_unit;
  DCache* dcache_unit;
  ICache* icache_unit1;
  ICache* icache_unit2;
  synchro* sc_unit;
  binterlock* ilock_unit;
  timer* tmr;
  bqueue* ifeeder;

  // status of current branch, if any, and some branch info slots

  bfstatus fetch_stat;
  String* bra_addr;
  ppc_Inst* b_ins;

  // the iterator containing the synthesized branch map

  DListIter<String*> iIter;
  bool bc_mode;

  // internal operations

  void fetch_instructions();     // get instructions from ICache
  void dispatch_instructions();  // send them to FXU and FPU
  void tick_fxfpdc();            // advance the FXU, FPU, and DCache one tick

  void detect_and_setup_branch();// begin to simulate branch
  void advance_branch_status();  // continue stages of branch
  void complete_branch();


};


#endif
```

```
// QPROF bru ADT -- sybok@athena

// This ADT is designed to simulate the high level BRU operation.

#include "bru.h"
#include "bbnode.h"

// the constructor

bru::bru(ICache& ic1, ICache& ic2, DList<String*>& branch_chart,
 const String& target, const String& end)

{ tmr = new timer(target,end);
  dcache_unit = new DCache;
  sc_unit = new synchro;
  ilock_unit = new binterlock(dcache_unit);

  fxu_unit = new fxu(ilock_unit,dcache_unit,sc_unit,tmr);
  fpu_unit = new fpu(ilock_unit,dcache_unit,sc_unit,tmr);

  ilock_unit->link(fxu_unit,fpu_unit);

  icache_unit1 = &ic1;
  icache_unit2 = &ic2;

  iIter = *(branch_chart.iter());
  bc_mode = TRUE;
  fetch_stat = OFF;
  bra_addr = 0;

  String initial_fetch_target =
icache_unit1->partition(target)->instIter().value()->label();

  ifeeder = new bqueue;
  icache_unit1->fetch(initial_fetch_target);
  ifeeder->load_seq(icache_unit1->load());

};

// time the code

int bru::time_code()
{
  while (!tmr->done_p())
    bru_tick();

  return(tmr->get_count());
};


// advance time one tick

void bru::bru_tick()
{
  detect_and_setup_branch();     // look for branches and handle any seen

  advance_branch_status();       // advance status of current branch

  dispatch_instructions();       // dispatch the instructions to FXU and FPU


/*

ifeeder->print();
cout << "FETCHSTAT = " << fetch_stat << endl;

*/

  fetch_instructions();          // fetch ins. into BRU dispatch buffers

  ++(*tmr);                      // advance timer

  complete_branch();             // complete current branch if proper



};

// fetch instructions from the ICache

void bru::fetch_instructions()
{
  switch(fetch_stat) {

  case WAIT_ON_FETCH:
    if (!bra_addr->empty())
      {icache_unit2->fetch(*bra_addr);
       ifeeder->load_jmp(icache_unit2->load());};
    break;

  case LOADED:
    if (!bra_addr->empty())
      {icache_unit2->fetch(ifeeder->jmp_room());
       ifeeder->load_jmp(icache_unit2->load());};
    break;

  default:
    icache_unit1->fetch(ifeeder->seq_room());
    ifeeder->load_seq(icache_unit1->load());
    break;
  };

};

// finish the current branch if its time to do so

void bru::complete_branch()
{
  if (ilock_unit->bra_mode() && ilock_unit->branch_resolved_p() )

    if (bra_addr->empty())

      {ifeeder->purge_jmp();
       fetch_stat = OFF;
       ilock_unit->branch_done();}

    else if ((fetch_stat == LOADED) || (fetch_stat == WAIT_ON_FETCH))

      {ifeeder->sideload();
       swap_icaches(icache_unit1,icache_unit2);
       fetch_stat = OFF;
       ilock_unit->branch_done();};
};


// dispatch instructions from the BRU through the binterlock object
// to the FXU and FPU

void bru::dispatch_instructions()
{
  tick_fxfpdc();

  int fxufpu_ins = 0;

  bool can_dispatch = TRUE;
  bool saw_cr_op = FALSE;
  bool saw_branch = FALSE;

  ppc_Inst* curr_ins;

  while ((ifeeder->seqsize() > 0) && can_dispatch)

    {curr_ins =  ifeeder->seq_peek();
     can_dispatch = ilock_unit->dispatch_ready_p(*curr_ins,*bra_addr);

     switch(curr_ins->op().opt()) {
     case BRNCH:
       can_dispatch = can_dispatch && (!saw_branch);
       break;
     case CR_OP:
       can_dispatch = can_dispatch && (!saw_cr_op);
       break;
     default:
       can_dispatch = can_dispatch && (fxufpu_ins < 2);
       break;
      };

      if (can_dispatch)
{ilock_unit->dispatch(*curr_ins,tmr,*bra_addr);
 ifeeder->seq_pop();};

      switch(curr_ins->op().opt()) {
      case BRNCH:
saw_branch = TRUE;
break;
      case CR_OP:
saw_branch = FALSE;
break;
      default:
++fxufpu_ins;
break;
      };
   };

  ilock_unit->signal_dispatch_done();
};


// Advance the FXU, FPU, and DCache units one tick.  Checkpointed
// copies are advanced as well.  The check followed by the alternate
// orderings of calls to the FXU and FPU is to enforce the constraints
// concerning how far the FXU and FPU can be out of sync.

void bru::tick_fxfpdc()
{
  dcache_unit->tick();

  if (sc_unit->fxu_ahead())
    {fpu_unit->tick();
     fxu_unit->tick();}
  else
    {fxu_unit->tick();
     fpu_unit->tick();};

};

// look for a branch in the BRU instruction buffers and handle any seen

void bru::detect_and_setup_branch()
{
  if ((fetch_stat == OFF) && (ifeeder->see_branch_p()))
```

```
    {b_ins = ifeeder->extract_branch();
     fetch_stat = WAIT_ON_ADDR;
     if (bc_mode)
       {bra_addr = ilter.value();
iIter++;
if (iIter.end_p()) bc_mode = FALSE;}
     else
       bra_addr = &(get_branch_target(*b_ins));};
};

// advance the status of any ongoing branch

void bru::advance_branch_status()
{
  switch (fetch_stat) {

  case WAIT_ON_ADDR:
    if (b_ins->op().reg_br_p())
      {if ((b_ins->op().uses_lr_p()) && (ilock_unit->bra_mode()))
 {if (ilock_unit->link_reg_ready_p())
    fetch_stat = WAIT_ON_FETCH;}
      else if (b_ins->op().uses_ctr_p() && (ilock_unit->bra_mode()))
{if (ilock_unit->count_reg_ready_p())
   fetch_stat = WAIT_ON_FETCH;};}
    else if (!(b_ins->op().inert_p()))
      if (b_ins->op().absolute_p())
fetch_stat = WAIT_ON_FETCH;
      else
fetch_stat = CALC_ADDR;
    break;

  case CALC_ADDR:
    fetch_stat = WAIT_ON_FETCH;
    break;

  case WAIT_ON_FETCH:
    fetch_stat = LOADED;
    break;
  };

};

// Swap the sequential and branch target ICaches

void swap_icaches(ICache*& ic1, ICache*& ic2)
{
  ICache* temp;

  temp = ic1;
  ic1 = ic2;
  ic2 = temp;
};
```

```
// QPROF fxu ADT -- sybok@athena

// The fxu ADT represents the FXU unit in the RS/6000.

#ifndef QPROF_FXU
#define QPROF_FXU

#include <String.h>
#include <iostream.h>

#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"

#include "addr.h"
#include "synchro.h"
#include "timer.h"
#include "binterlock.h"
#include "regbusy.h"
#include "DCache.h"
#include "ICache.h"
#include "gqueue.h"

// some useful constants

const IBUF_SIZE = 4;
const PSQ_FOR_FP_LOADS_SIZE = 3;
const SB_FOR_FX_LOADS_SIZE = 1;

const NO_SPR_REQ = -1;

// the class fxu

class fxu
{

 public:

  // the constuctor

  fxu(binterlock* bi_lock, DCache* dc, synchro* sc, timer* tmr);

  void tick();                                  // advance FXU one clock tick

  void dispatch(ppc_Inst& ins);                 // dispatch an FXU instruction

  bool full_p() {return (ibuf->room() <= 0);}; // Is the FXU full?

  void print();

 private:

  int exec_delay;                               // FXU pipeline occupancy

  // begin subordinate functional unit pointers

  synchro* sc_unit;
  regbusy* busyregs;
  timer* t_unit;

  gqueue<ppc_Inst*>* ibuf;
  gqueue<ppc_Inst*> *d0, *d1, *exe, *exe_plus_1, *exe_plus_2;
  gqueue<addr> *fpu_psq, *fpu_psq2, *fpu_psq3, *fpu_psq4, *fxu_sb;

  binterlock* bra_lock;
  DCache* dcache_unit;

  int overload_count;

  // end subordinate functional unit pointers

  int spr_req_buf;                              // handle special register loads

  bool dcache_busy;                             // is the DCache busy?

  bool l_advanced;                              // have the exe. buffers been
                                                // advanced

  void handle_decode();
  void handle_exe();                    // handle exe. of non-load instructions
  void handle_exe_load();               // handle exe. of load instructions
  void perform_store();                 // handle exe. of store instruction

  void clear_dispatch_flags();          // clear flags in binterlock unit
  void handle_fxu_load();               // treat load returning from DCache
  void handle_fxu_stbuf();              // push FX store buf. entry to DCache
  void handle_fpu_stbuf();              // push FP store buf. entry to DCache

  void advance_line(ppc_Inst* ins);     // advance exe. buffers
  void advance_line();

};
```

```
// these two procedures advance the line of execution history buffers;
// these buffers are used to clear flags in the dispatch unit

inline void fxu::advance_line(ppc_Inst* ins)
{
  if (!exe_plus_2->empty_p())
    exe_plus_2->pop();
  if (!exe_plus_1->empty_p())
    exe_plus_2->load_element(exe_plus_1->pop());
  exe_plus_1->load_element(ins);
};

inline void fxu::advance_line()
{
  if (!exe_plus_2->empty_p())
    exe_plus_2->pop();
  if (!exe_plus_1->empty_p())
    exe_plus_2->load_element(exe_plus_1->pop());
};


#endif
```

```
// QPROF fxu ADT -- sybok@athena

// The fxu ADT represents the FXU unit in the RS/6000.

#include "fxu.h"

// construct an FXU

fxu::fxu(binterlock* bi_lock, DCache* dc, synchro* sc, timer* tmr)
{
  exec_delay = 0;
  overload_count = 0;

  sc_unit = sc;
  t_unit = tmr;

  busyrege = new regbusy;

  ibuf = new gqueue<ppc_Inst*>(IBUF_SIZE);

  d0 = new gqueue<ppc_Inst*>(1);
  d1 = new gqueue<ppc_Inst*>(1);
  exe = new gqueue<ppc_Inst*>(1);

  exe_plus_1 = new gqueue<ppc_Inst*>(1);
  exe_plus_2 = new gqueue<ppc_Inst*>(1);

  spr_req_buf = NO_SPR_REQ;

  fpu_psq = new gqueue<addr>(1);
  fpu_psq2 = new gqueue<addr>(1);
  fpu_psq3 = new gqueue<addr>(1);
  fpu_psq4 = new gqueue<addr>(1);
  fxu_sb = new gqueue<addr>(SB_FOR_FX_LOADS_SIZE);

  bra_lock = bi_lock;
  dcache_unit = dc;

  dcache_busy = FALSE;
  l_advanced = FALSE;

};

// dispatch an instruction to the FXU

void fxu::dispatch( ppc_Inst& ins )
{
  if (ibuf->size() <= 0)
    {if (d0->room() > 0)
       d0->load_element(&ins);
     else if (d1->room() > 0)
       d1->load_element(&ins);
     else if (ibuf->room() > 0)
       ibuf->load_element(&ins);
     else
       sigerr("fxu::dispatch: No room to dispatch.");}

  else if (ibuf->room() > 0)
    ibuf->load_element(&ins);
  else
    sigerr("fxu::dispatch: No room to dispatch.");

};

void fxu::tick()
{
// push the instruction buffer entries, if any, down into the d0 and d1 stages

  if (d0->room() > 0)
    {if (ibuf->size() > 0)
       d0->load_element(ibuf->pop());
     if (ibuf->size() > 0)
       d1->load_element(ibuf->pop());};

  clear_dispatch_flags();                    // clear flags in binterlock unit

  dcache_busy = FALSE;          // has cache bandwith been used?
  l_advanced = FALSE;           // have exe. history bufs. been advanced

  // handle loads returning from the DCache unit

  handle_fxu_load();

  // handle clearing interlocks set during loads from special registers

  if (spr_req_buf != NO_SPR_REQ)
    {busyrege->unlock(spr_req_buf);
     spr_req_buf = NO_SPR_REQ;};

  // decrement pipeline occupancy count

  if ((exe->size() > 0) && (exec_delay > 0))
    exec_delay--;

  // push the pipeline through

  handle_exe_load();                 // handle a load in the exe. stage
```

```
  handle_fxu_stbuf();                  // handle the FXU store buffer
  handle_fpu_stbuf();                  // handle the FPU store buffer
  handle_exe();                        // handle the FXU execution stage


  handle_decode();                     // handle the movement of instructions
                                       // from decode into the execute stage
};

// handle the execution stage of the FXU pipeline, except loads

void fxu::handle_exe()
{
  if ((exec_delay <= 0) && (exe->size() > 0) &&
      !(exe->peek()->op().load_p()))                 // don't handle loads here
    {if (exe->peek()->op().store_p())                // handle stores
       perform_store();
     else if (exe->peek()->op().opt() == FXU_MFSPR)          // handle loads from
       {spr_req_buf = exe->peek()->rand(1)->reg().actual(); // special registers
        busyregs->lock(spr_req_buf);
        advance_line(exe->pop());}
     else advance_line(exe->pop());}                 // handle all other FXU
                                                     // instructions
  else if (!l_advanced)
    advance_line();

};

// push loads through the execution stage of the FXU pipeline

void fxu::handle_exe_load()
{
  if ((exec_delay <= 0) && (exe->size() > 0))
    if (exe->peek()->op().load_p())
      {ppc_Inst* ins = exe->peek();
       addr* new_addr = new addr(*ins);

        if (!(dcache_busy) &&                        // make sure the DCache is
    dcache_unit->request_load_ready_p())   // not be used for other
                                                     // purposes this clk cycle
 {dcache_busy = TRUE;
  dcache_unit ->
    request_load(ins->op().fx_load_p(),
 ins->rand(1)->reg().actual(),
 *new_addr,
 fpu_psq->contains(*new_addr) +
 fpu_psq2->contains(*new_addr) +
                    fpu_psq3->contains(*new_addr) +
 fpu_psq4->contains(*new_addr) +
 fxu_sb->contains(*new_addr));

  busyregs->lock(ins->rand(1)->reg().actual());

  l_advanced = TRUE;
  advance_line(exe->pop());};
     };
};

// handle a load from the DCache to the FXU becoming ready

void fxu::handle_fxu_load()
{
  if (dcache_unit->fxu_load_val_ready_p())
    {dcache_busy = TRUE;
     busyregs->unlock(dcache_unit->fxu_load_val());};
};


// Send signals to the binterlock unit to clear dispatch flags when
// instructions pass through certain FXU pipeline stages

void fxu::clear_dispatch_flags()
{
  if (exe->size() > 0)
    bra_lock->signal_FXU_Exe(*exe->peek());

  if (exe_plus_1->size() > 0)
    bra_lock->signal_FXU_Exe_plus_1(*exe_plus_1->peek());

  if (exe_plus_2->size() > 0)
    bra_lock->signal_FXU_Exe_plus_2(*exe_plus_2->peek());

};

void fxu::perform_store()
{
 ppc_Inst* ins = exe->peek();
 addr* new_addr = new addr(*ins);

 if ((ins->op().fp_store_p()) &&
     (fpu_psq->room() > 0) && (overload_count < 4))
   {fpu_psq->load_element(*(new addr(*exe->peek())));
    overload_count++;
    advance_line(exe->pop());}
 else
   overload_count = 0;

 if ((ins->op().fx_store_p()) &&
     (fxu_sb->room() > 0))
   {fxu_sb->load_element(*(new addr(*exe->peek())));
    advance_line(exe->pop());};

};


// advance decode region of FXU pipeline

void fxu::handle_decode()
{
  for(int i = 1; i <= 2; i++)   // shift two instructions out, if possible,
    {                           // per clock cycle
      if ((d0->size() > 0) && sc_unit->fxu_shift_ok_p())
switch(d0->peek()->op().opt()) {

case FPU: case FPU_A: case FPU_CMP:      // discard FPU operations
  sc_unit->fxu_shift();
  d0->pop();
  break;

default:                                 // process FXU operations
  if ((exe->room() > 0) &&
      !(busyregs->clash(d0->peek()->live_GPR_rands())))
    {sc_unit->fxu_shift();
     exe->load_element(d0->pop());
     exec_delay = exe->peek()->occupancy();
     t_unit->alert_timer(exe->peek()->label());};
  break;

};

      if ((d1->size() > 0) && (d0->room() > 0))  // advance decode buffers
d0->load_element(d1->pop());
    };
};

// pop and process FXU store buffer entries if possible

void fxu::handle_fxu_stbuf()
{
  if ((fxu_sb->size() > 0) && !dcache_busy &&
      dcache_unit->sb_store_ready_p())
    {
      dcache_busy = TRUE;
      dcache_unit->sb_store(fxu_sb->pop());
    };

};

// pop and process FPU store buffer entries if possible

void fxu::handle_fpu_stbuf()
{  if ((fpu_psq4->size() > 0) && !dcache_busy &&
       dcache_unit->psq_store_ready_p())
     {
       dcache_busy = TRUE;
       dcache_unit->psq_store(fpu_psq4->pop());
     };

   if ((fpu_psq4->room() > 0) && (fpu_psq3->size() > 0))
     fpu_psq4->load_element(fpu_psq3->pop());

   if ((fpu_psq3->room() > 0) && (fpu_psq2->size() > 0))
     fpu_psq3->load_element(fpu_psq2->pop());

   if ((fpu_psq2->room() > 0) && (fpu_psq->size() > 0))
     fpu_psq2->load_element(fpu_psq->pop());

};

// print out the current state of the FXU

void fxu::print()
{ cout << "COUNT = " << overload_count << endl;
  cout << "************************** FXU STATE **************************";

  cout << endl;

  cout << "IBUF:" << endl; ibuf->print();

  cout << "D1: " << endl; d1->print();
  cout << "D0: " << endl; d0->print();

  cout << "EXE(" << exec_delay << ") :" << endl;
  exe->print();

  cout << "EXE+1: " << endl; exe_plus_1->print();
  cout << "EXE+2: " << endl; exe_plus_2->print();

  cout << "FPU_PSQ: " << endl; fpu_psq->print();
  cout << "FPU_PSQ: " << endl; fpu_psq2->print();
  cout << "FPU_PSQ: " << endl; fpu_psq3->print();
  cout << "FPU_PSQ: " << endl; fpu_psq4->print();
  cout << "FXU_SB: " << endl; fxu_sb->print();

  cout << "***************************************************************";
  cout << endl;

};
```

```
// QPROF fpu ADT -- sybok@athena

// The fpu ADT is used to simulate the operation of the RS/6000 FPU unti


#ifndef QPROF_FPU
#define QPROF_FPU

#include <iostream.h>
#include <String.h>

#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"

#include "synchro.h"
#include "timer.h"
#include "binterlock.h"
#include "fprmap.h"
#include "grqueue.h"
#include "gqueue.h"
#include "fpdecode_entry.h"
#include "psq_entry.h"
#include "DCache.h"
#include "ICache.h"

// This ADT simulates the RS/6000 FPU unit.

const FPU_IBUF_SIZE = 6;
const FPU_DBUF_SIZE = 3;
const FPU_DSQ_SIZE = 3;
const FPU_PSQ_SIZE = 3;
const FPU_PTRQ_SIZE = 8;
const FPU_FLIST_SIZE = 8;

const FPU_OLQ_SIZE = 6; // Chosen for simulation purposes--arbitrary

// the class fpu

class fpu
{

 public:

 // constructor, used by BRU unti

  fpu(binterlock* bi_lock, DCache* dc, synchro* sc, timer* tmr);

  void dispatch( ppc_Inst& ins);               // dispatch an FPU instruction

  void tick();                                 // advance FPU one tick

  bool full_p() {return (ibuf->room() <= 0);}; // test whether FPU is full

  void print();                                // print FPU state

 private:

  fpu* saved_state;                            // saved state

  int exec_delay;                              // instruction pipeline occupancy

  // begin subordinate functional unit pointers

  synchro* sc_unit;
  fprmap* fprmtable;
  timer* t_unit;

  gqueue<ppc_Inst*>* ibuf;
  gqueue<ppc_Inst*> *pd0, *pd1, *rn0, *rn1;
  gqueue<fpdecode_entry> *dbuf, *decode;
  grqueue<psq_entry>* psq;
  gqueue<ppc_Inst*> *exe1, *exe2;
  gqueue<ppc_Inst*> *exe2_plus_1, *exe2_plus_2, *exe2_plus_3, *exe2_plus_4;

  gqueue<fpu_data>* dsq;
  gqueue<int>* olq;


  gqueue<int>* free_list;
  grqueue<int>* ptrq;

  binterlock* bra_lock;

  DCache* dcache_unit;

  // end subordinate functional unit pointers

  bool dcache_busy;                            // is the DCache busy?

  bool can_decode, can_pop;

  // does instruction use locked registers?

  bool clash(ppc_Inst* ins) const;

  // pipeline advancing operations

  void clear_dispatch_flags();          // clear flags in binterlock unit

  void advance_ibufs();                 // advance input buffers
  void handle_fpu_load();               // treat returning loads

  void load_rename_stage();
  void rename_and_push_ins();           // do rename phase
  void rename_into_psq();               // rename stores and fill PSQ
  void rename_into_decode_stage();      // rename arith. ins. and fill dbufs
  void prepare_for_fpu_load();          // rename loads and modify reg. map

  void pop_psq();                       // remove psq entry and process
  void advance_dbufs();                 // advance decode buffers
  void handle_decode();

  void handle_execute();
  void empty_PTRQ();

  void advance_dsq();                   // send out DSQ entry if possible

  void advance_line(ppc_Inst* ins);
  void advance_line();

};

// handle the emptying of the pending target return queue

inline void fpu::empty_PTRQ()
{
  if ((ptrq->size() > 0) && (ptrq->can_pop()))
    if (psq->contains(*(new psq_entry(0,ptrq->peek()))))
      psq->apply_to_some(*(new psq_entry(0,ptrq->pop())),&set_gb);
    else if (free_list->room() > 0)
      free_list->load_element(ptrq->pop());
};


// move instructions in the decode buffer into the decode stage

inline void fpu::advance_dbufs()
{
  if ((decode->room() > 0) && (dbuf->size() > 0))
    decode->load_element(dbuf->pop());
};


// If the data storage queue is not empty, try to send some data to
// the DCache

inline void fpu::advance_dsq()
{
  if ((dsq->size() > 0) && (!dcache_busy) &&
      dcache_unit->dsq_store_ready_p())
    {
      dcache_busy = TRUE;
      dcache_unit->dsq_store(dsq->pop());
    };
};


// handle an incoming load from the DCache to the FPU

inline void fpu::handle_fpu_load()
{
  if ((dcache_unit->fpu_load_val_ready_p()) && (olq->size() > 0))
    {
      dcache_busy = TRUE;
      olq->pop();                   // actual register
      dcache_unit->fpu_load_val(); // virtual register
     };
};

// fill the rename stages if empty

inline void fpu::load_rename_stage()
{
  if ((rn0->room() > 0) && (rn1->room() > 0))
    {if (pd0->size() > 0)
       rn0->load_element(pd0->pop());
     if (pd1->size() > 0)
       rn1->load_element(pd1->pop());};
};


#endif
```

```
// QPROF fpu ADT -- sybok@athena

// The fpu ADT is used to simulate the operation of the RS/6000 FPU unit

#include "fpu.h"

// construct an FPU

fpu::fpu(binterlock* bi_lock, DCache* dc, synchro* sc, timer* tmr)
{

  exec_delay = 0;

  sc_unit = sc;
  t_unit = tmr;

  fprmtable = new fprmap;

  ibuf = new gqueue<ppc_Inst*>(FPU_IBUF_SIZE);

  pd0 = new gqueue<ppc_Inst*>(1);
  pd1 = new gqueue<ppc_Inst*>(1);
  rn0 = new gqueue<ppc_Inst*>(1);
  rn1 = new gqueue<ppc_Inst*>(1);

  dbuf = new gqueue<fpdecode_entry>(FPU_DBUF_SIZE);

  decode = new gqueue<fpdecode_entry>(1);

  exe1 = new gqueue<ppc_Inst*>(1);
  exe2 = new gqueue<ppc_Inst*>(1);
  exe2_plus_1 = new gqueue<ppc_Inst*>(1);
  exe2_plus_2 = new gqueue<ppc_Inst*>(1);
  exe2_plus_3 = new gqueue<ppc_Inst*>(1);
  exe2_plus_4 = new gqueue<ppc_Inst*>(1);

  dsq = new gqueue<fpu_data>(FPU_DSQ_SIZE);

  olq = new gqueue<int>(FPU_OLQ_SIZE);

  psq = new grqueue<psq_entry>(FPU_PSQ_SIZE);

  free_list = new gqueue<int>(FPU_FLIST_SIZE);

  for(int i= 32; i < 40; i++)
    free_list->load_element(i);

  ptrq = new grqueue<int>(FPU_PTRQ_SIZE);

  bra_lock = bi_lock;
  dcache_unit = dc;

  dcache_busy = FALSE;
  can_decode = TRUE;
  can_pop = TRUE;
};

// dispatch an FPU instruction

void fpu::dispatch( ppc_Inst& ins)
{
  if (ibuf->size() <= 0)
    {
      if (pd0->room() > 0)
pd0->load_element(&ins);
      else if (pd1->room() > 0)
pd1->load_element(&ins);
      else if (ibuf->room() > 0)
ibuf->load_element(&ins);
      else
sigerr("fpu::dispatch: No room to dispatch.");}

  else if (ibuf->room() > 0)
    ibuf->load_element(&ins);
  else
    sigerr("fpu::dispatch: No room to dispatch.");
};

// advance the FPU by one clock tick

void fpu::tick()
{
  advance_ibufs();          // move instructions down from the ins. buffers

  clear_dispatch_flags();   // clear flags in the binterlock unit

  dcache_busy = FALSE;

  if ((exe1->size() > 0) && (exec_delay > 0))
    exec_delay--;

  // setup flags for decoding and popping a psq entry, respectively

  can_decode = (decode->size() > 0) ? !clash(decode->peek().get_ins()) : TRUE;
  can_pop = (psq->size() > 0) ? !clash(psq->peek().get_ins()) : TRUE;

  handle_fpu_load();
  advance_dsq();

  handle_execute();

  pop_psq();

  empty_PTRQ();

  handle_decode();

  rename_and_push_ins();

  advance_dbufs();

};

// Send signals to the binterlock unit to clear dispatch flags when
// instructions pass through certain FPU pipeline stages

void fpu::clear_dispatch_flags()
{
  if (exe1->size() > 0)
    bra_lock->signal_FPU_Exe1(*exe1->peek());

  if (exe2_plus_1->size() > 0)
    bra_lock->signal_FPU_Exe2_plus_1(*exe2_plus_1->peek());

  if (exe2_plus_3->size() > 0)
    bra_lock->signal_FPU_Exe2_plus_3(*exe2_plus_3->peek());

};


// advance instructions in the buffers upstream from the predecode buffers
// into the predecode registers

void fpu::advance_ibufs()
{
  if ((pd0->room() > 0) && (pd1->room() > 0))
    {if (ibuf->size() > 0)
      pd0->load_element(ibuf->pop());
     if (ibuf->size() > 0)
       pd1->load_element(ibuf->pop());};
};


// Twice, do the following:
//
// Rename the instruction in rn0, whether an arithmetic FPU instruction
// or an FPU load or store, and then dispatch it to the appropriate pipeline.
// Finally, advance the preceding processor buffers one notch.

void fpu::rename_and_push_ins()
{
 for(int disp_loop = 1; disp_loop <= 2; disp_loop++)
    {
      if ((rn0->size() > 0) && sc_unit->fpu_shift_ok_p())
switch(rn0->peek()->op().opt()) {

case FPU: case FPU_A: case FPU_CMP:
  rename_into_decode_stage();
  break;

default:
  if (rn0->peek()->op().fp_store_p())
    rename_into_psq();
  else if (rn0->peek()->op().fp_load_p())
    prepare_for_fpu_load();
  else
    {rn0->pop();
     sc_unit->fpu_shift();};
  break;
};

      if ((rn1->size() > 0) && (rn0->room() > 0))
rn0->load_element(rn1->pop());
    };

   load_rename_stage();

};

// rename the floating point store in rn0 and if possible push it onto
// the psq

void fpu::rename_into_psq()
{
  if (psq->room() > 0)
    {psq->load_element(fprmtable->store_map(rn0->pop()));
     if (dbuf->size() > 0)
       dbuf->apply_to_last(&add_store);
     else if (decode->size() > 0)
       decode->apply_to_last(&add_store);
     else
       psq->release();
     sc_unit->fpu_shift();}
};

// update the virtual->real register map and various register queues
// to reflect an incoming load to an FPU register

void fpu::prepare_for_fpu_load()
{
  if (fprmtable->load_map_possible_p(free_list, ptrq, olq))
```

```
    {fprmtable->load_map(rn0->pop(),free_list,ptrq,olq);
     if (dbuf->size() > 0)
       dbuf->apply_to_last(&add_load);
     else if (decode->size() > 0)
       decode->apply_to_last(&add_load);
     else
       ptrq->release();
     sc_unit->fpu_shift();};
};


// rename the instruction in rn0 and stick it into the decode stage, or
// the decode buffers, as room permits

void fpu::rename_into_decode_stage()
{
  if ((dbuf->size() <= 0) && (decode->room() > 0))
    {decode->load_element(fprmtable->arith_map(rn0->pop()));
     sc_unit->fpu_shift();}
  else if (dbuf->room() > 0)
    {dbuf->load_element(fprmtable->arith_map(rn0->pop()));
     sc_unit->fpu_shift();};
};


// Attempt to move the instruction in decode down to the first execute
// stage.

void fpu::handle_decode()

{
  if ((exe1->room() > 0) && (decode->size() > 0) && can_decode)

    {exe1->load_element(decode->peek().get_ins());
     exec_delay = decode->peek().get_ins()->occupancy();
     t_unit->alert_timer(decode->peek().get_ins()->label());
     for(int i = decode->peek().get_lcount(); i > 0; i--)
       ptrq->release();
     for(int j = decode->peek().get_scount(); j > 0; j--)
       psq->release();

     decode->pop();};
};

// handle the removal of entries from the FPU pending store queue

void fpu::pop_psq()
{
  if ((exe1->room() > 0) && (psq->size() > 0) && (psq->can_pop()) &&
      can_pop)
    {if (psq->peek().give_back_p())
       {if (free_list->room() > 0)
  {exe1->load_element(psq->peek().get_ins());
   free_list->load_element(psq->peek().get_target());};}
      else
        exe1->load_element(psq->peek().get_ins());

     psq->pop();};
};

// handle the execute stages of the pipeline

void fpu::handle_execute()
{
  if (exe2->size() > 0)
    {if ((exe2->peek()->op().fp_store_p()) && (dsq->room() > 0))
       {dsq->load_element(FDATA);
advance_line(exe2->pop());}
    else if (!exe2->peek()->op().fp_store_p())
      advance_line(exe2->pop());}
  else
    advance_line();

  if ((exec_delay <= 0) && (exe1->size() > 0) &&
      (exe2->room() > 0))
    exe2->load_element(exe1->pop());
};

void fpu::advance_line(ppc_Inst* ins)
{
  if (!exe2_plus_4->empty_p())
    exe2_plus_4->pop();
  if (!exe2_plus_3->empty_p())
    exe2_plus_4->load_element(exe2_plus_3->pop());
  if (!exe2_plus_2->empty_p())
    exe2_plus_3->load_element(exe2_plus_2->pop());
  if (!exe2_plus_1->empty_p())
    exe2_plus_2->load_element(exe2_plus_1->pop());
  exe2_plus_1->load_element(ins);
};

void fpu::advance_line()
{
  if (!exe2_plus_4->empty_p())
    exe2_plus_4->pop();
  if (!exe2_plus_3->empty_p())
    exe2_plus_4->load_element(exe2_plus_3->pop());
  if (!exe2_plus_2->empty_p())
    exe2_plus_3->load_element(exe2_plus_2->pop());
  if (!exe2_plus_1->empty_p())
    exe2_plus_2->load_element(exe2_plus_1->pop());
};


// look for a "clash" which prevents an instruction for being executed

bool fpu::clash( ppc_Inst* ins) const
{ // check first for pipeline dependencies involving adjacent instructions

  if (exe2->size() > 0)     // check for FRA-dependent loads
    for(DListIter<int>
j1(exe2->peek()->target_FPR_rands()); !j1.end_p(); j1++)
      for(DListIter<int> i1(ins->source_FRA_rands()); !i1.end_p(); i1++)
if (i1.value() == j1.value())
  return TRUE;

  if (exe1->size() > 0)    // check for all other dependent loads
    for(DListIter<int>
j2(exe1->peek()->target_FPR_rands()); !j2.end_p(); j2++)
      for(DListIter<int> i2 (ins->source_FPR_rands()); !i2.end_p(); i2++)
 if (i2.value() == j2.value())
   return TRUE;

  // check for source or destination registers to which an outstanding load
  // is assigned

  for(DListIter<int> i3(ins->source_FPR_rands()); !i3.end_p(); i3++)
    if (olq->contains(i3.value()))
      return TRUE;

  for(DListIter<int> i4(ins->target_FPR_rands()); !i4.end_p(); i4++)
    if (olq->contains(i4.value()))
      return TRUE;

  return FALSE;

};

// print out the current state of the FPU

void fpu::print()
{
  cout << "************************** FPU STATE **************************";
  cout << endl;

  cout << "IBUF:" << endl; ibuf->print();

  cout << "PD1: " << endl; pd1->print();
  cout << "PD0: " << endl; pd0->print();

  cout << "RN1: " << endl; rn1->print();
  cout << "RN0: " << endl; rn0->print();

  cout << "FREELIST:" << endl; free_list->print();
  cout << "PTRQ:" << endl; ptrq->print();

  cout << "DBUF:" << endl; dbuf->print();
  cout << "DECODE:" << endl; decode->print();

  cout << "PSQ:" << endl; psq->print();

  cout << "EXE1(" << exec_delay << ") :" << endl; exe1->print();
  cout << "EXE2:" << endl; exe2->print();
  cout << "EXE2+1: " << endl; exe2_plus_1->print();
  cout << "EXE2+2: " << endl; exe2_plus_2->print();
  cout << "EXE2+3: " << endl; exe2_plus_3->print();
  cout << "EXE2+4: " << endl; exe2_plus_4->print();

  cout << "OLQ: " << endl; olq->print();

  cout << "DSQ: " << endl; dsq->print();

  cout << "***********************************************************";

  cout << endl;

};
```

```
// QPROF parser code -- sybok@athena

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

#include "ppc_Mod.h"
#include "ppc_Proc.h"
#include "ppc_CB.h"
#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"
#include "err.h"

enum parse_pos {AT_LABEL, AT_OP, AT_RANDS};

inline void addCR(char* l, ppc_Inst* inst);
void addGPR(char* l, ppc_Inst* inst);
void addFPR(char* l, ppc_Inst* inst);
void addNUM(char* l, ppc_Inst* inst);
void addLABEL(char* l, ppc_Inst* inst);
bool is_w_space(char c);


// The parser is used to convert an RS/6000 assembly file into
// a PowerPC module in order to test the ideal time module of QPROF

ppc_Mod* file_to_ppc_Mod(char* filename)
{
  // first create a stream bound to the input file

  ifstream f_in(filename);

  if (f_in.fail())
sigerr("parser: Given file name not found.");

  const char cond_reg[3] = "cr";
  const char fp_reg[4] = "fpr";
  const char fx_reg = 'r';

  char label[100];
  int index;

  char curr_char;
  parse_pos pos = AT_LABEL;

  String* l_name = 0;
  String* op_name = 0;

  ppc_Rator* rat;
  ppc_Opcode op;

  int num_rand;

  ppc_Mod* mod = new ppc_Mod("test mod");
  ppc_Proc* proc = new ppc_Proc("test proc",NORMAL_PROC);
  ppc_CB* cb = new ppc_CB("test cb");
  ppc_Part* part = new ppc_Part("test part",THREAD_PART); part->regidx(0);
  ppc_Inst* ins;


  while (!f_in.eof())
    {f_in.get(curr_char);

     while ((is_w_space(curr_char)) && (!f_in.eof()))
       {if (curr_char == '\n')
{if ((pos == AT_OP) && !l_name->empty())
   part->append(new ppc_Inst(ppc_none,*l_name));
 pos = AT_LABEL;};
        f_in.get(curr_char);};

     index = 0;

     while ((!is_w_space(curr_char)) && (!f_in.eof())) {
       if (curr_char != ':')
         label[index++] = curr_char;
       f_in.get(curr_char);};

     label[index] = '\0';

     switch(pos) {
      case AT_LABEL:
        l_name = new String(label);
        rat = new ppc_Rator(*l_name);
        op = rat->op();
        delete rat;

        if (op == ppc_none)
          pos = AT_OP;
        else
          {ins = new ppc_Inst(op,"");
           part->append(ins);
           pos = AT_RANDS;};
        break;

      case AT_OP:
        op_name = new String(label);
        rat = new ppc_Rator(*op_name);
        op = rat->op();
        delete rat;
        ins = new ppc_Inst(op,*l_name);
        part->append(ins);
        pos = AT_RANDS;
        break;

      default:
        if ((label == strstr(label,cond_reg)) && (strlen(label) == 3))
          addCR(label,ins);
        else if ((label == strchr(label,fx_reg)) && (strlen(label) <= 3))
          addGPR(label,ins);
        else if ((label == strstr(label,fp_reg)) && (strlen(label) <= 5))
          addFPR(label,ins);
        else if (isdigit(label[0]))
          addNUM(label,ins);
        else
          addLABEL(label,ins);
        break;
      };

     if (curr_char == '\n')
       {if ((pos == AT_OP) && !l_name->empty())
  part->append(new ppc_Inst(ppc_none,*l_name));
 pos = AT_LABEL;};
  };
  part->append(new ppc_Inst(ppc_none));
  cb->append(part); cb->append(part);
  proc->append(cb);
  mod->append(proc);
  return(mod);
};

// add a condition register operand to an instruction

inline void addCR(char* l, ppc_Inst* inst)
{
  int r = (int) l[2]-'0';
  inst->addRand(new ppc_Rand(*(new ppc_Reg(r,CR_RTYPE,r))));
};

// add a general purpose register operand to an instruction

void addGPR(char* l, ppc_Inst* inst)
{
  int r;

  if (l[2] == '\0')
    r = (int) l[1] - '0';
  else
    r = (int) ((l[1] - '0')*10 + (l[2]-'0'));

  inst->addRand(new ppc_Rand(*(new ppc_Reg(r,GPR_RTYPE,r))));
};

// add a floating-point register operand to an instruction

void addFPR(char* l, ppc_Inst* inst)
{
  int r;

  if (l[4] == '\0')
    r = (int) l[3] - '0';
  else
    r = (int) ((l[3] - '0')*10 + (l[4]-'0'));

  inst->addRand(new ppc_Rand(*(new ppc_Reg(r,FPR_RTYPE,r))));

};

// add an integer operand to an instruction

void addNUM(char* l, ppc_Inst* inst)
{
  int num = 0;
  int index = 0;

  while (l[index] != '\0')
    num = 10*num+((int) l[index++]-'0');

  inst->addRand(new ppc_Rand(num));

};

// add a label operand to an instruction

void addLABEL(char* l, ppc_Inst* inst)
{
  inst->addRand(new ppc_Rand(*(new String(l))));
};

// define a "white space" character as a conventional C white space
// character, or a comma.

bool is_w_space(char c) {return (isspace(c) || (c == ','));};
```

```
// QPROF ADT binterlock -- sybok@athena

// This ADT represents the instruction dispatch interlocks that limit
// the dispatching of instructions in the BRU unit.


#ifndef QPROF_BINTERLOCK
#define QPROF_BINTERLOCK

#include <iostream.h>
#include <String.h>

#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"

#include "timer.h"

// forward definitions

class fxu;
class fpu;
class DCache;

bool check_intersection(int mask, int crfield);

// useful enums

enum status {CLEAR, SET};               // Status of flags

enum bdstatus {RESOLVED, UNRESOLVED};  // Status of current branch

enum settype {NO_STYPE, FXU_STYPE, FPU_STYPE};

const NO_CR_FIELD = -1;

// structure for representing a read and write flag pair

struct flagpair {
  status rflag;
  status wflag;
};

// The class binterlock

class binterlock
{
 public:
  binterlock(DCache* dc);                   // constructor

  void link(fxu* fixed, fpu* floating);  // wire in FXU and FPU units
                                         // to already constructed binterlock

  // These are the main operations called by high order users--they
  // check whether a new (general) instruction can be dispatched, and
  // dispatch it, respectively.

  bool dispatch_ready_p(const ppc_Inst& test_ins, String& targ) const;

  void dispatch(ppc_Inst& ins, timer* tmr, String& targ);

  // These procedures are called by tha (subordinate) FXU and FPU units,
  // to clear certain interlock flags.

  void signal_FXU_Exe(ppc_Inst& ins);
  void signal_FXU_Exe_plus_1(ppc_Inst& ins);
  void signal_FXU_Exe_plus_2(ppc_Inst& ins);

  void signal_FPU_Exe1(ppc_Inst& ins);
  void signal_FPU_Exe2_plus_3(ppc_Inst& ins);
  void signal_FPU_Exe2_plus_1(ppc_Inst& ins);

  // these are signalers used by the caller to end a dispatch or branch

  void signal_dispatch_done() {cr_op_exists = FALSE;};
  void branch_done();

  // here are some predicates useful for examining the binterlock's state

  bool bra_mode() {return bmode;};
  bool branch_resolved_p();

  bool link_reg_ready_p() const {return (!lr.wflag);};
  bool count_reg_ready_p() const {return (!ctr.wflag);};

private:

  int cond_disp_ins;          // the number of conditionally dispatched
                              // instructions

  bool fpu_not_ready;         // prevent instructions from being conditionally
                              // dispatched during a branch dependent
                              // upon an FPU computation?

  settype stype[8];           // record whether a CR-setting instruction
                              // is an FXU or FPU operation

  // begin lock flags

  flagpair lr;
  flagpair ctr;

  flagpair cr0_4;
  flagpair cr1_5;
  flagpair cr2_6;
  flagpair cr3_7;

  status cr_A;

  status bi;
  status bmode;

  // end lock flags

  bdstatus resolve_stat;         // conditional branch status
  bdstatus cntr_rstat;

  ppc_Inst* binst;          // pointer to the branch instruction
  int bcrfield;                  // and the branch condition field
  bool sucjmp;

  int cr_op_lock;                // interlocks for CR operations
  bool cr_op_exists;


  fxu* fxu_unit;                 // pointers to functional units
  fpu* fpu_unit;
  DCache* dcache_unit;

  // flag manipulation operations

  bool check_cr_field(int field) const;
  bool check_cr_field_r(int field) const;

  void set_cr_field(int bf, settype st);
  void clear_cr_field(int bf);

  bool check_all_CR_r() const;

  void set_cr_mask(int mask);
  void set_cr_mask(int mask, settype st);
  void clear_cr_mask(int mask);
  bool check_cr_mask(int mask) const;

  // useful macros

  void handle_illegal_dispatches(optype op);
  void handle_mtspr_dispatch(ppc_Inst& ins);
  void handle_mfspr_dispatch(ppc_Inst& ins);
  bool can_dispatch_branch_p(const ppc_Inst& test_ins) const;
  bool can_dispatch_mtspr_p(const ppc_Inst& test_ins) const;
  bool can_dispatch_cr_op_p(const ppc_Inst& test_ins) const;

};


// link in FXU and FPU units after creating a binterlock object

inline void binterlock::link(fxu* fixed, fpu* floating)
{
  fxu_unit = fixed;
  fpu_unit = floating;
};

// check whether the current branch has been resolved

inline bool binterlock::branch_resolved_p()
{
  if (!bmode) sigerr("binterlock::branch_resolved_p: No branch.");

  return ((resolve_stat == RESOLVED) && (cntr_rstat == RESOLVED));

};

// some useful macros for classifying a PowerPC instruction

inline bool ins_is_mtlr(const ppc_Inst& ins);
inline bool ins_is_mtctr(const ppc_Inst& ins);
inline bool ins_is_mflr(const ppc_Inst& ins);
inline bool ins_is_mfctr(const ppc_Inst& ins);
inline bool ins_is_mfxer(const ppc_Inst& ins);

// some subprocedures for handling the dispatch of PowerPC instructions
// and testing dispatch readiness

// can a branch instruction be dispatched?

inline bool binterlock::can_dispatch_branch_p(const ppc_Inst& test_ins) const
{
  if ((test_ins.op().linked_p()) && lr.wflag)
    return FALSE;
  else if (test_ins.op().reg_br_p())
    {
      if (test_ins.op().uses_lr_p())
return (link_reg_ready_p());
```

```
      else if (test_ins.op().uses_ctr_p())
return (count_reg_ready_p());
      else
return TRUE;
    }
  else
    return TRUE;

};

// check for illegal dispatches and flag them

inline void binterlock::handle_illegal_dispatches(optype op)
{
  if (op == ILLEGAL)
    sigerr("binterlock::dispatch: Illegal dispatch.");

  if (bmode)                                  // if branch mode is in effect
    if ((op == CR_OP) || (op == BRNCH))
      sigerr("binterlock::dispatch: Illegal dispatch.");
    else if (cntr_rstat != RESOLVED)
      sigerr("binterlock::dispatch: Illegal dispatch.");
    else if (resolve_stat != RESOLVED)
      cond_disp_ins++;

  if (cond_disp_ins > 4) sigerr("binterlock::dispatch: Illegal dispatch.");

};

// internal flag (CR fields and LR/CTR special register fields) operations
//

// check write flag of branch field

inline bool binterlock::check_cr_field(int bf) const
{
  if (cr_op_exists && (((cr_op_lock-bf) % 4) == 0))
    return FALSE;

  if (((bf == 0) || (bf == 4)) && cr0_4.wflag) return FALSE;
  else if (((bf == 1) || (bf == 5)) && cr1_5.wflag) return FALSE;
  else if (((bf == 2) || (bf == 6)) && cr2_6.wflag) return FALSE;
  else if (((bf == 3) || (bf == 7)) && cr3_7.wflag) return FALSE;
  else return TRUE;
};

// check read flag of branch field

inline bool binterlock::check_cr_field_r(int bf) const
{
  if (cr_op_exists && (((cr_op_lock-bf) % 4) == 0))
    return FALSE;

  if (((bf == 0) || (bf == 4)) && cr0_4.rflag) return FALSE;
  else if (((bf == 1) || (bf == 5)) && cr1_5.rflag) return FALSE;
  else if (((bf == 2) || (bf == 6)) && cr2_6.rflag) return FALSE;
  else if (((bf == 3) || (bf == 7)) && cr3_7.rflag) return FALSE;
  else return TRUE;
};


// set both read and write flags of a field

inline void binterlock::set_cr_field(int bf, settype st)
{ stype[bf] = st;

  if ((bf == 0) || (bf == 4))     {cr0_4.rflag = SET; cr0_4.wflag = SET;}
  else if ((bf == 1) || (bf == 5)){cr1_5.rflag = SET; cr1_5.wflag = SET;}
  else if ((bf == 2) || (bf == 6)){cr2_6.rflag = SET; cr2_6.wflag = SET;}
  else if ((bf == 3) || (bf == 7)){cr3_7.rflag = SET; cr3_7.wflag = SET;}
  else
    sigerr("binterlock::set_cr_field: Invalid field.");
};

// clear both read and write flags of a field

inline void binterlock::clear_cr_field(int bf)
{
  if ((bf == 0) || (bf == 4))     {cr0_4.rflag = CLEAR; cr0_4.wflag = CLEAR;}
  else if ((bf == 1) || (bf == 5)){cr1_5.rflag = CLEAR; cr1_5.wflag = CLEAR;}
  else if ((bf == 2) || (bf == 6)){cr2_6.rflag = CLEAR; cr2_6.wflag = CLEAR;}
  else if ((bf == 3) || (bf == 7)){cr3_7.rflag = CLEAR; cr3_7.wflag = CLEAR;}
  else
    sigerr("binterlock::clear_cr_field: Invalid field.");
};

// check all read flags, including the CR operation lock field

inline bool binterlock::check_all_CR_r() const
{
  return((!cr0_4.rflag) && (!cr1_5.rflag) && (!cr_op_exists) &&
 (!cr2_6.rflag) && (!cr3_7.rflag));
};

// set read and write flags based on a mask

inline void binterlock::set_cr_mask(int mask, settype st)
{
  if (mask & 0x88)
    {cr0_4.rflag = SET; cr0_4.wflag = SET;
     if (mask & 0x80) stype[0] = st;
     if (mask & 0x08) stype[4] = st;};
  if (mask & 0x44)
    {cr1_5.rflag = SET; cr1_5.wflag = SET;
     if (mask & 0x40) stype[1] = st;
     if (mask & 0x04) stype[5] = st;};
  if (mask & 0x22)
    {cr2_6.rflag = SET; cr2_6.wflag = SET;
     if (mask & 0x20) stype[2] = st;
     if (mask & 0x02) stype[6] = st;};
  if (mask & 0x11)
    {cr3_7.rflag = SET; cr3_7.wflag = SET;
     if (mask & 0x10) stype[3] = st;
     if (mask & 0x01) stype[7] = st;};
};

inline void binterlock::set_cr_mask(int mask)
{ if (mask & 0x88) {cr0_4.rflag = SET; cr0_4.wflag = SET;};
  if (mask & 0x44) {cr1_5.rflag = SET; cr1_5.wflag = SET;};
  if (mask & 0x22) {cr2_6.rflag = SET; cr2_6.wflag = SET;};
  if (mask & 0x11) {cr3_7.rflag = SET; cr3_7.wflag = SET;};
};

// clear read and write flags based on a mask

inline void binterlock::clear_cr_mask(int mask)
{ if (mask & 0x88) {cr0_4.rflag = CLEAR; cr0_4.wflag = CLEAR;};
  if (mask & 0x44) {cr1_5.rflag = CLEAR; cr1_5.wflag = CLEAR;};
  if (mask & 0x22) {cr2_6.rflag = CLEAR; cr2_6.wflag = CLEAR;};
  if (mask & 0x11) {cr3_7.rflag = CLEAR; cr3_7.wflag = CLEAR;};
};


#endif
```

```
// QPROF ADT binterlock -- sybok@athena

// This ADT represents the instruction dispatch interlocks that limit
// the dispatching of instructions in the BRU unit.

#include "binterlock.h"
#include "fxu.h"
#include "fpu.h"
#include "DCache.h"

binterlock::binterlock(DCache* dc)
{
  cond_disp_ins = 0;

  fpu_not_ready = TRUE;

  for(int i = 0; i < 8; i++)
    stype[i] = NO_STYPE;


  lr.wflag = CLEAR; lr.rflag = CLEAR;

  ctr.wflag = CLEAR; ctr.rflag = CLEAR;

  cr0_4.wflag = CLEAR; cr0_4.rflag = CLEAR;
  cr1_5.wflag = CLEAR; cr1_5.rflag = CLEAR;
  cr2_6.wflag = CLEAR; cr2_6.rflag = CLEAR;
  cr3_7.wflag = CLEAR; cr3_7.rflag = CLEAR;

  cr_A = CLEAR;

  bi = CLEAR; bmode = CLEAR;

  resolve_stat = RESOLVED;
  cntr_rstat = RESOLVED;

  binst = 0;

  cr_op_exists = FALSE;

  fxu_unit = 0;
  fpu_unit = 0;
  dcache_unit = dc;

};
// hooks called by subordinate FXU unit

void binterlock::signal_FXU_Exe(ppc_Inst& ins)
{  ppc_Opcode opcode = ins.op().op();
   optype op = ins.op().opt();

   switch(op) {

   case FXU_MTSPR:
     if (opcode == ppc_mtcrf)
       {int mask = ins.rand(1)->snum();
lr.rflag = CLEAR;
ctr.rflag = CLEAR;
if (bmode && check_intersection(mask,bcrfield))
  bi = CLEAR;}
     else if (ins_is_mtlr(ins) || ins_is_mtctr(ins))
       {cr0_4.rflag = CLEAR; cr1_5.rflag = CLEAR;
cr2_6.rflag = CLEAR; cr3_7.rflag = CLEAR;
lr.rflag = CLEAR; ctr.rflag = CLEAR;}
     else if (opcode == ppc_mcrxr)
       if (bmode && ((bcrfield == ins.rand(1)->reg().actual())))
 bi = CLEAR;
     break;

   case FXU_CMP:
     if (bmode && ((bcrfield == ins.rand(1)->reg().actual())))
       bi = CLEAR;
     break;

   case FXU_LOG:
     if ((opcode == ppc_andi_) || (opcode == ppc_andis_))
       if (bmode && ((bcrfield  == 0)))
 bi = CLEAR;
     break;
   };

 };

void binterlock::signal_FXU_Exe_plus_1(ppc_Inst& ins)
{
   ppc_Opcode opcode = ins.op().op();
   optype op = ins.op().opt();

   switch(op) {

   case FXU_MTSPR:
     if (opcode == ppc_mtcrf)
       {int mask = ins.rand(1)->snum();
        cr_A = CLEAR;
        clear_cr_mask(mask);
        if (bmode && check_intersection(mask,bcrfield))
          if (resolve_stat == UNRESOLVED)
            resolve_stat = RESOLVED;}
     else if (ins_is_mtlr(ins)) lr.wflag = CLEAR;
     else if (ins_is_mtctr(ins))
       {ctr.wflag = CLEAR;
if (bmode)
  cntr_rstat = RESOLVED;}
     else if (opcode == ppc_mcrxr)
       {clear_cr_field(ins.rand(1)->reg().actual());
        if (bmode && (bcrfield == ins.rand(1)->reg().actual()))
          if (resolve_stat == UNRESOLVED)
            resolve_stat = RESOLVED;}
     break;

   case FXU_MFSPR:
     if (ins_is_mfxer(ins))
       {cr_A = CLEAR;
        lr.wflag = CLEAR;
        ctr.wflag = CLEAR;};
     break;

   case FXU_CMP:
     clear_cr_field(ins.rand(1)->reg().actual());
     if (bmode && (bcrfield == ins.rand(1)->reg().actual()))
       if (resolve_stat == UNRESOLVED)
         resolve_stat = RESOLVED;
     break;

   case FXU_LOG:
     if ((opcode == ppc_andi_) || (opcode == ppc_andis_))
       {clear_cr_field(0);
        if (bmode && ((bcrfield == 0)))
          if (resolve_stat == UNRESOLVED)
            resolve_stat = RESOLVED;};
     break;
   };
 };

void binterlock::signal_FXU_Exe_plus_2(ppc_Inst& ins)
{
  ppc_Opcode opcode = ins.op().op();
  optype op = ins.op().opt();

  if (op == FXU_MFSPR)
    if (opcode == ppc_mfcr)
      {lr.wflag = CLEAR;
       ctr.wflag = CLEAR;
       clear_cr_mask(0xFF);}
    else if (ins_is_mflr(ins))
      {cr_A = CLEAR;
       lr.rflag = CLEAR;
       lr.wflag = CLEAR; ctr.wflag = CLEAR;}
    else if (ins_is_mfctr(ins))
      {cr_A = CLEAR;
       ctr.rflag = CLEAR;
       ctr.wflag = CLEAR; lr.wflag = CLEAR;};

};

// hooks called by subordinate FPU unit

void binterlock::signal_FPU_Exe1(ppc_Inst& ins)
{
   optype op = ins.op().opt();

   if (op == FPU_CMP)
     if (bmode && ((bcrfield == ins.rand(1)->reg().actual())))
       bi = CLEAR;
 };

void binterlock::signal_FPU_Exe2_plus_1(ppc_Inst& ins)
{
   optype op = ins.op().opt();

   if (op == FPU_CMP)
     if (bmode && ((bcrfield == ins.rand(1)->reg().actual())))
       fpu_not_ready = FALSE;
 };

void binterlock::signal_FPU_Exe2_plus_3(ppc_Inst& ins)
{
   ppc_Opcode opcode = ins.op().op();
   optype op = ins.op().opt();

   if (op == FPU_CMP)
     {clear_cr_field(ins.rand(1)->reg().actual());
       if (bmode && (bcrfield == ins.rand(1)->reg().actual()))
         if (resolve_stat = UNRESOLVED)
           resolve_stat = RESOLVED;};

};

// terminate branch mode.

void binterlock::branch_done()
{
  bmode = CLEAR;
  cond_disp_ins = 0;
  fpu_not_ready = TRUE;
};

// can a particular instruction be dispatched?

bool
binterlock::dispatch_ready_p(const ppc_Inst& test_ins, String& targ) const
```

```
{ ppc_Opcode opcode = test_ins.op().op();
  optype op = test_ins.op().opt();

  if (bmode)                                    // if branch mode is in effect
    if ((op == CR_OP) || (op == BRNCH))
      return FALSE;
    else if ((!targ.empty()) || (cntr_rstat != RESOLVED))
      return FALSE;
    else
      switch(stype[bcrfield]){

      case FXU_STYPE:
        if (cond_disp_ins >= 4)
          return FALSE;
        break;

      case FPU_STYPE:
if ((cond_disp_ins >= 4) || fpu_not_ready)
          return FALSE;
        break;

     };

  if ((op != CR_OP) && (op != BRNCH))               // make sure input queues
    if (fxu_unit->full_p() || fpu_unit->full_p())   // aren't full
      return FALSE;

                          // in any event, switch on optype
  switch (op) {

  case FXU_MFSPR:
    if (opcode == ppc_mfcr)
      return(check_all_CR_r());
    else if (ins_is_mflr(test_ins))
      return((!lr.rflag) && (!bi));
    else if (ins_is_mfctr(test_ins))
      return((!ctr.rflag) && (!bi));
    else return TRUE;

  case FXU_MTSPR:
    return(can_dispatch_mtspr_p(test_ins));

  case FXU_CMP: case FPU_CMP:
    return ((!bi) && (check_cr_field(test_ins.rand(1)->reg().actual())));

  case FXU_LOG:
    if ((opcode == ppc_andi_) || (opcode == ppc_andis_))
      return ((!bi) && (check_cr_field(0)));
    else return TRUE;

  case CR_OP:
    return(can_dispatch_cr_op_p(test_ins));

  case BRNCH:
    return(can_dispatch_branch_p(test_ins));

  default: return TRUE;
 };
};

// dispatch an instruction

void binterlock::dispatch( ppc_Inst& ins, timer* tmr, String& targ)
{
  ppc_Opcode opcode = ins.op().op();
  optype op = ins.op().opt();

  handle_illegal_dispatches(op);
  tmr->mark_end(ins.label());

  // switch on op type

  switch (op) {

  case FXU_MFSPR:
    handle_mfspr_dispatch(ins);
    break;

  case FXU_MTSPR:
    handle_mtspr_dispatch(ins);
    break;

  case FXU_CMP:
    set_cr_field(ins.rand(1)->reg().actual(),FXU_STYPE);
    fxu_unit->dispatch(ins);
    fpu_unit->dispatch(ins);
    break;

  case FPU_CMP:
    set_cr_field(ins.rand(1)->reg().actual(),FPU_STYPE);
    fxu_unit->dispatch(ins);
    fpu_unit->dispatch(ins);
    break;

  case FXU_LOG:
    if ((opcode == ppc_andi_) || (opcode == ppc_andis_))
      set_cr_field(0,FXU_STYPE);
    fxu_unit->dispatch(ins);
    fpu_unit->dispatch(ins);
    break;

  case CR_OP:
    tmr->alert_timer(ins.label());
    if (opcode == ppc_mcrf)
        {cr_op_exists = TRUE;
         cr_op_lock = ins.rand(1)->reg().actual() % 4;}
      else
        {cr_op_exists = TRUE;
         cr_op_lock = (ins.rand(1)->snum()/4) % 4;};
    break;

  case BRNCH:
    tmr->alert_timer(ins.label());
    bmode = SET;
    binst = &ins;
    sucjmp = !targ.empty();

    if (binst->op().brc_p(binst->rands(),binst->nrands()))
      {if (binst->rand(1)->reg_p())
bcrfield = binst->rand(1)->reg().actual();
       else
bcrfield = (binst->rand(2)->snum() / 4);}
    else
      bcrfield = NO_CR_FIELD;

    if ((ins.op().brc_p(ins.rands(),ins.nrands())) &&
!check_cr_field(bcrfield))
      {resolve_stat = UNRESOLVED; bi = SET;}
    else
      {resolve_stat = RESOLVED; bi = CLEAR;}

    if (ins.op().cdep(binst->rands(),binst->nrands()) && ctr.wflag)
      {cntr_rstat = UNRESOLVED; bi = SET;}
    else
      {cntr_rstat = RESOLVED;};
    break;

  default:
    fxu_unit->dispatch(ins);
    fpu_unit->dispatch(ins);
    break;
  };

};

// subprocedures for handling dispatch and dispatch readiness

// handle the dispatch of a move to special register instruction

void binterlock::handle_mtspr_dispatch(ppc_Inst& ins)
{
  if (ins.op().op() == ppc_mtcrf)
    {set_cr_mask(ins.rand(1)->snum(),FXU_STYPE);
     lr.rflag = SET;
     ctr.rflag = SET;
     cr_A = SET;}
  else if (ins.op().op() == ppc_mcrxr)
    set_cr_field(ins.rand(1)->reg().actual(),FXU_STYPE);
  else if (ins_is_mtlr(ins))
    {cr0_4.rflag = SET; cr1_5.rflag = SET;
     cr2_6.rflag = SET; cr3_7.rflag = SET;
     lr.rflag = SET;
     ctr.rflag = SET;
     lr.wflag = SET;}
  else if (ins_is_mtctr(ins))
    {cr0_4.rflag = SET; cr1_5.rflag = SET;
     cr2_6.rflag = SET; cr3_7.rflag = SET;
     lr.rflag = SET;
     ctr.rflag = SET;
     ctr.wflag = SET;};
  fxu_unit->dispatch(ins);
  fpu_unit->dispatch(ins);

};

// handle the dispatch of a move from special register instruction

void binterlock::handle_mfspr_dispatch(ppc_Inst& ins)
{
  if (ins.op().op() == ppc_mfcr)
    {set_cr_mask(0xFF);
     lr.wflag = SET;
     ctr.wflag = SET;}
  else if (ins_is_mflr(ins))
    {cr_A = SET;
     lr.rflag = SET;
     lr.wflag = SET;
     ctr.wflag = SET;}
  else if (ins_is_mfctr(ins))
    {cr_A = SET;
     ctr.rflag = SET;
     lr.wflag = SET;
     ctr.wflag = SET;}
  else if (ins_is_mfxer(ins))
    {cr_A = SET;
     lr.wflag = SET;
     ctr.wflag = SET;};
  fxu_unit->dispatch(ins);
  fpu_unit->dispatch(ins);
};

// can a move to special register instruction be dispatched?
```

```
bool binterlock::can_dispatch_mtspr_p(const ppc_Inst& test_ins) const
{
  if (test_ins.op().op() == ppc_mtcrf)
    if (bmode || cr_A)
      return FALSE;
    else
      return(check_cr_mask(test_ins.rand(1)->snum()));
  else if (test_ins.op().op() == ppc_mcrxr)
    return ((!b1) && (check_cr_field(test_ins.rand(1)->reg().actual())));
  else if (ins_is_mtlr(test_ins))
    return((!lr.wflag) && (!b1));
  else if (ins_is_mtctr(test_ins))
    return((!ctr.wflag) && (!b1));
  else return TRUE;

};

// can a CR operation instruction be dispatched?

bool binterlock::can_dispatch_cr_op_p(const ppc_Inst& test_ins) const
{
  if (test_ins.op().op() == ppc_mcrf)
    return ((check_cr_field(test_ins.rand(1)->reg().actual())) &&
    (check_cr_field_r(test_ins.rand(2)->reg().actual())));
  else if (test_ins.nrands() == 3)
    return ((check_cr_field(test_ins.rand(1)->snum()/4)) &&
    (check_cr_field_r(test_ins.rand(2)->snum()/4)) &&
    (check_cr_field_r(test_ins.rand(3)->snum()/4)));
  else if (test_ins.nrands() == 2)
    return ((check_cr_field(test_ins.rand(1)->snum()/4)) &&
    (check_cr_field_r(test_ins.rand(2)->snum()/4)));
  else
    return ((check_cr_field(test_ins.rand(1)->snum()/4)) &&
    (check_cr_field_r(test_ins.rand(1)->snum()/4)));
};

// some useful macros for classifying PowerPC instructions

inline bool ins_is_mtlr(const ppc_Inst& ins)
{return
   ((ins.op().op() == ppc_mtlr) ||
    ((ins.op().op() == ppc_mtspr) &&
     (ins.rand(1)->reg().actual() == 8)));};

inline bool ins_is_mtctr(const ppc_Inst& ins)
{return
   ((ins.op().op() == ppc_mtctr) ||
    ((ins.op().op() == ppc_mtspr) &&
     (ins.rand(1)->reg().actual() == 9)));};

inline bool ins_is_mflr(const ppc_Inst& ins)
{return
   ((ins.op().op() == ppc_mflr) ||
    ((ins.op().op() == ppc_mfspr) &&
     (ins.rand(2)->reg().actual() == 8)));};

inline bool ins_is_mfctr(const ppc_Inst& ins)
{return
   ((ins.op().op() == ppc_mfctr) ||
    ((ins.op().op() == ppc_mfspr) &&
     (ins.rand(2)->reg().actual() == 9)));};

inline bool ins_is_mfxer(const ppc_Inst& ins)
{return
   ((ins.op().op() == ppc_mfxer) ||
    ((ins.op().op() == ppc_mfspr) &&
     (ins.rand(2)->reg().actual() == 1)));};


// CR flag checking and setting/clearing procedures

// check whether a given mask interferes with set CR flags

bool binterlock::check_cr_mask(int mask) const
{ int local_lock =  cr_op_exists ? cr_op_lock : -1;

  if ((mask & 0x88) && (cr0_4.wflag || (local_lock == 0)))
    return FALSE;
  else if ((mask & 0x44) && (cr1_5.wflag || (local_lock == 1)))
    return FALSE;
  else if ((mask & 0x22) && (cr2_6.wflag || (local_lock == 2)))
    return FALSE;
  else if ((mask & 0x11) && (cr3_7.wflag || (local_lock == 3)))
    return FALSE;
  else
    return TRUE;}

// check whether a mask intersects a CR field--this function is not a member
// of binterlock--same as above except using an explicit crfield rather than
// the current binterlock object's flags

bool check_intersection(int mask, int crfield)
{
  return
    (((mask & 0x80) && (crfield  == 0)) ||
     ((mask & 0x40) && (crfield  == 1)) ||
     ((mask & 0x20) && (crfield  == 2)) ||
     ((mask & 0x10) && (crfield  == 3)) ||
     ((mask & 0x08) && (crfield  == 4)) ||
     ((mask & 0x04) && (crfield  == 5)) ||
     ((mask & 0x02) && (crfield  == 6)) ||
     ((mask & 0x01) && (crfield  == 7)));
};
```

```
// QPROF modmap ADT -- sybok@athena

// The modmap ADT represents a collection of nodemaps, one for each
// partition in the moduel


#ifndef QPROF_MODMAP
#define QPROF_MODMAP

#include <iostream.h>
#include <String.h>

#include "err.h"
#include "nodemap.h"


class modmap
{
 public:

  modmap(ICache* ic, ppc_Mod& mod);  // constructor

  branch_chart_list& // build a branch chart list for a given BB
    make_branch_map(String& target, bool fall_though_p, bool repeat_p);

  branch_chart_list& // build a branch chart list for a string of connected BBs
    make_branch_map(String& target1, String& target2,
    bool fall_though_p, bool repeat_p);

void  print(ppc_Part* ptr)
    {(*partmap)[ptr]->print();};

 private:

  ICache* icache_unit;

  QVHMap<ppc_Part*,nodemap*>* partmap;

};

// generate a branch map to a given BB

inline branch_chart_list&
modmap::make_branch_map(String& target, bool fall_through_p, bool repeat_p)
{
  return((*partmap)[icache_unit->partition(target)]->
 make_branch_map(target,fall_through_p,repeat_p));
};

// generate a branch map to a string of topologically connected BBs

inline branch_chart_list&
modmap::make_branch_map(String& target1, String& target2,
bool fall_through_p, bool repeat_p)
{
  return((*partmap)[icache_unit->partition(target1)]->
 make_branch_map(target1,target2,fall_through_p, repeat_p));
};

#endif
```

```
// QPROF modmap ADT -- sybok@athena

// The modmap ADT represents a collection of nodemaps, one for each
// partition in the module

#include "modmap.h"

// construct a modmap

modmap::modmap(ICache* ic, ppc_Mod& mod): icache_unit(ic)
{
  partmap = new QVHMap<ppc_Part*,nodemap*>(0,20);

  for (ppc_ProcIter iProc(mod.procIter()); !iProc.end_p(); iProc++)
    for (ppc_CBIter iCB(iProc.value() -> CBIter()); !iCB.end_p() ; iCB++)
      for (ppc_PartIter iPart(iCB.value() -> partIter());
   !iPart.end_p(); iPart++)
(*partmap)[iPart.value()] = new nodemap(iPart.value(),ic);

};
```

```
// QPROF nodemap ADT -- sybok@athena

// THe nodemap ADT is the top-level data type for "branch map" synthesis--
// generating a list of labels to guide the simulator through the target
// basic block.  To each partition in the current module, there will
// correspond a nodemap data object.

#ifndef QPROF_NODEMAP
#define QPROF_NODEMAP

#include <iostream.h>
#include <String.h>

#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ICache.h"
#include "bbnode.h"

// make some shorthand notations

typedef DList<String*> branch_chart;
typedef DList<DList<String*>*> branch_chart_list;

// some forward declarations

branch_chart_list&
  operator *(branch_chart_list& s1, branch_chart_list& s2);

branch_chart_list&
  multi_append_lists(branch_chart_list& ml, branch_chart& tail);

// the class nodemap

class nodemap
{
 public:

  nodemap(ppc_Part* partition, ICache* icache_p);  // constructor

  bbnode& grab_root() const                       // extract the root bbnode
    {return *root;};                              // of a nodemap

  branch_chart_list&   // build a branch chart for a given BB
    make_branch_map(String& target, bool fall_through_p, bool repeat_p);

  branch_chart_list&   // build a branch chart for a string of connected BBs
    make_branch_map(String& target1, String& target2,
    bool fall_through_p, bool repeat_p);

void  print()
  {root->print(); cout << endl;};

 private:

  ICache* icp;              // store the ICache built from the nodemap's module

  ppc_Part* part;           // store a pointer to the nodemap's partition

  bbnode* root;             // store the root bbnode of the nodemap


  void find_path(bbnode* curr_node,                // find a path from
 String& target,              // a node of the nodemap
 branch_chart& working_path,  // to a node labeled with
 QVHMap<bbnode*,bool>& nr,          // a target string
 bbnode*& target_node,
 branch_chart_list& ml);


  void create_node(ppc_InstIter& iIter,  // recursively build a node in the
   bbnode*& bbnref);     // nodemap given an iterator
                                       // pointing at the beginning of the
                                       // appropriate BB


  void find_exit_path(branch_chart& exit_path,      // find a path out of
      bbnode* t_node,               // a nodemap from a
      QVHMap<bbnode*,bool>& nr,     // a starting node
      bool fall_through_p,
      bool repeat_p,
      String& btarget);


  QVHMap<ppc_Inst*,bbnode*> node_registry;  // this holds the node registry
                                            // used to avoid duplicating
                                            // nodes unnecessarily or
                                            // entering infinite loops


  void build_init_exit_path(branch_chart& exit_path,   // treat exit from
    bbnode*& t_node,            // target BB
    QVHMap<bbnode*,bool>& nr,
    bool& fall_through_p,
    bool& repeat_p,
    String& btarget, bool& done_with_main_BB);
};

#endif
```

```
// QPROF nodemap ADT -- sybok@athena

// THe nodemap ADT is the top-level data type for "branch map" synthesis--
// generating a list of labels to guide the simulator through the target
// basic block.  To each partition in the current module, there will
// correspond a nodemap data object.

#include "nodemap.h"

// This is the constructor for the nodemap ADT.  It takes as arguments
// the partition to be represented by the nodemap and the ICache object of
// the containing module.  A call to create_node generates the nodal
// structure pointed to by the root node of a nodemap.

nodemap::nodemap(ppc_Part* partition, ICache* icache_p):
  part(partition), icp(icache_p), node_registry(0,10)
{bbnode* root_node;

 ppc_InstIter i(part->instIter());   // set up an iterator at the first
 i.first();                          // instruction of a partition

 if (!i.end_p())                     // build the nodes
   {create_node(i,root_node);
    root = root_node;}
 else
   root = 0;
};

// This member, create_node, recursively builds the nodemap structure.  It
// takes an iterator over instructions and a reference to a pointer to a node.
// It first builds a node corresponding to the BB beginning at the instruction
// marked by the iterator's read position, then smashes the reference-passed
// pointer to point to the new node, and finally calls itself to generate
// the descendant nodes of the current node.  It uses the node_registry
// variable to keep track of the nodes it's seen in a given partition.

void nodemap::create_node(ppc_InstIter& iIter, bbnode*& bbnref)
{bool found_left = FALSE;
bool found_right = FALSE;

if (node_registry.contains(iIter.value()))
   bbnref = node_registry[iIter.value()];
 else
   {DList<ppc_Inst*>* bb = new DList<ppc_Inst*>; // this will point to the list
                                                 // of instructions,
                                                 // comprising a BB, that
                                                 // will be passed to the
                                                 // bbnode constructor

    ppc_Inst* curr_inst;   // points at most current instruction

    bbnode* new_node;      // will be passed referentially to callees to
                           // permit them to set it to the created bbnode

    ppc_Part* part;        // hold current partition
    ppc_Inst* ins;         // hold an instruction

    ppc_Inst* first_i = iIter.value();  // grab first instruction in BB

   // the reason all variables had to be declared above is that none
   // can be declared inside a loop.

    while (!iIter.end_p()) {
      curr_inst = iIter.value();  // grab pointer to current instruction and
      bb->append(curr_inst);      // append it to the object pointed at by bb

      iIter++;

      if ((!curr_inst->op().inert_p()) ||     // check for end of BB
  ((!iIter.end_p()) &&
   (!(iIter.value()->label().empty()))))
{ bbnref = new bbnode(*bb,icp); // create the new bbnode

  node_registry[first_i] = bbnref;         // record in registry
                                           // that this bbnode has now
                                           // been already seen

 if (bbnref->has_direct_target_p()) // if BB has a direct-target BB...
   {
    part = icp->partition(bbnref->target_label()); // grab partition
    ins = icp->inst(bbnref->target_label());       // and inst of
           found_right = TRUE;                             // top of target BB

    for(ppc_InstIter inner_Iter(part->instIter()); // construct an
!(inner_Iter.end_p()); inner_Iter++)       // iterator pointing
      if (inner_Iter.value() == ins)               // at the beginning
{                                          // of the target BB

 create_node(inner_Iter,new_node);    // create the
 break;};                             // new bbnode

    bbnref->set_direct_target(new_node);

    if (new_node->get_guide() !=
NO_GUIDE)
      bbnref->set_guide(GO_RIGHT);

                                          };    // set the right child
                                                // of the original node
                                                // *bbnref to point to
                                                // newly created new_node

if (bbnref->has_indirect_target_p() || // if BB has a fall-thru BB
    bbnref->has_fall_thru_p())         // or an indirect-target BB...
  {
   create_node(iIter,new_node);        // create a new bbnode, using
                                                 // the current iterator pos.

   if (bbnref->has_indirect_target_p())      // for indirect-targets,
     {bbnref->set_indirect_target(new_node);
      found_right = TRUE;
      if (new_node->get_guide() !=
  NO_GUIDE)
bbnref->set_guide(GO_RIGHT);};

                                                 // set right child of
                                                 // *bbnref to new bbnode

   if (bbnref->has_fall_thru_p())            // for fall-thrus, set
     {bbnref->set_fall_thru(new_node);
      found_left = TRUE;
      if (new_node->get_guide() !=
  NO_GUIDE)
bbnref->set_guide(GO_LEFT);};};      // left child of *bbnref
                                                 // to new bbnode

 if (!found_right && !found_left)
   bbnref->set_guide(END_POINT);

 break;};
   };};
};

// The find_path member uses the connectivity of the nodemap to compute
// all unique paths from a given node to a second node (BB) with a given
// target label (which touch each node at most once).  These paths are
// specified as lists of actions required at branches.  A node registry
// is used to ensure that infinite loops do not occur during processing.

// The result is returned via the reference variable ml, a list of branch
// charts (action lists), and the pointer passed to target_node is set to
// the target BB's bbnode when it is found.

void nodemap::find_path(bbnode* curr_node, String& target,
branch_chart& working_path,
QVHMap<bbnode*,bool>& nr, bbnode*& target_node,
branch_chart_list& ml)
{
  if (curr_node->contains_label(target)) // if node has target label, we are
    {ml.append(&working_path);           // done; append current branch chart
     target_node = curr_node;}           // and return to caller
  else
    if (!(nr.contains(curr_node)))       // otherwise...
      {
QVHMap<bbnode*,bool>* registry               // duplicate registry,
                 = new QVHMap<bbnode*,bool>(nr);     // since it is passed
                                                     // referentially
(*registry)[curr_node] = TRUE;   // add current bbnode to registry

//  In the case of a fall through label, simply recurse on descendent

       if ((curr_node->get_exit_type()) == FALL_THRU_2_LABEL)
 find_path(curr_node->get_fall_thru(), target, working_path,
   *registry, target_node, ml);

// Otherwise, for a branch-terminated BB...

       else
 {if (curr_node->has_fall_thru_p())

// If the branch has a fall through, add a "" to the branch chart and recurse

    {branch_chart* left_path = new branch_chart(working_path);
     left_path->append(new String(""));
     find_path(curr_node->get_fall_thru(), target, *left_path,
       *registry, target_node, ml);};

// If the branch has a non-empty direct target, add target label to branch
// chart and recurse,

  if ((curr_node->has_direct_target_p()) &&
      (!(curr_node->target_label().empty())))
    {branch_chart* right_path = new branch_chart(working_path);
     right_path->append(new String(curr_node->target_label()));
     find_path(curr_node->get_direct_target(), target, *right_path,
       *registry, target_node, ml);}

// or if instead it has a non-empty indirect target, add the target label
// of the simulated RTS stub and a return label to the branch chart--then
// recurse

  else if ((curr_node->has_indirect_target_p()) &&
   (!(curr_node->get_indirect_target()->get_label().empty())))
    {branch_chart* right_path = new branch_chart(working_path);
     right_path->append(new String("_simulated_RTS_stub"));
     right_path->append(new
String(curr_node->get_indirect_target()->get_label()));
     find_path(curr_node->get_indirect_target(), target, *right_path,
       *registry,target_node,ml);};};
      };
}
```

```
// The find_exit_path member, when given an initial node and two
// boolean parameters, finds a unique short path to the end of the
// partition, favoring fall-throughs over branches when the fall-through
// node has not been visited before.

// The exit_path reference variable is used to return the exit chart
// (which will become the tail end of the branch chart), and a node
// registry, nr, is again used to prevent infinite loops from occurring
// during processing.

// The flag repeat_p causes the BB to be repeated once before taking
// the exit path, and the flag fall_through_p determines whether the
// exit from the chosen BB occurs via fall-through or branch, where the
// chosen option is possible.

void nodemap::find_exit_path(branch_chart& exit_path,bbnode* t_node,
     QVHMap<bbnode*,bool>& nr,
     bool fall_through_p, bool repeat_p,
     String& btarget)

{bool done_with_main_BB = FALSE;

// while the current node still has children and is untouched...



 while ((t_node->has_direct_target_p())   ||
 (t_node->has_indirect_target_p()) ||
 (t_node->has_fall_thru_p()))

   {done_with_main_BB =
      ((t_node->target_label() != btarget) && done_with_main_BB);



  /*  if (done_with_main_BB && fall_through_p && !repeat_p)
      nr[t_node]=TRUE;   // mark node as reached
*/
// If the current node is a fall-through type, we can't loop back
// or determine the exit method to satisfy the 2 boolean flags,
// so turn off the initial mode flag and begin to work on the
// fall-through node.

     if (t_node->get_exit_type() == FALL_THRU_2_LABEL)
       {
t_node = t_node->get_fall_thru();
       done_with_main_BB = TRUE;
repeat_p = FALSE;
fall_through_p = TRUE;
continue;}

// If the current node is a branch-type node (terminated by a branch),
// then clear the initial mode flag (for the benefit of future loop
// iterations) and check whether the boolean parameters can be satisfied.

     else if (!done_with_main_BB)
       {
done_with_main_BB = TRUE;
       build_init_exit_path(exit_path, t_node, nr,
                            fall_through_p, repeat_p,
                            btarget,done_with_main_BB);
       continue;}

// If the initial stage is over,

     else

// For the default case (other than the initial stage) ...

// If the current bbnode has a fall-through node which has not been touched,
// then add a "" to the exit chart and recurse on the fall-through node.

      if ((t_node->has_fall_thru_p()) && (t_node->get_guide() != GD_RIGHT))
  {
exit_path.append(new String (""));
   t_node = t_node->get_fall_thru();
   continue;}

// If the current bbnode has a direct target which has not been touched,
// then add the target lable to the exit chart and recurse on the direct
// target node.

      else if (t_node->has_direct_target_p())
   {
exit_path.append(new String(t_node->target_label()));
    t_node = t_node->get_direct_target();
    continue;}

// If the current bbnode has an indirect target which has not been touched,
// then add a jump to "_simulated_RTS_stub" to the exit chart, followed
// by a jump back to the user bbnode pointed at as the right child of
// of the original node (indirect target bbnode).

      else if (t_node->has_indirect_target_p())
   {
exit_path.append(new String("_simulated_RTS_stub"));
    t_node = t_node->get_indirect_target();
    exit_path.append(new String(t_node->get_label()));
    continue;}
 else
   sigerr("nodemap::find_exit_path: Partitioning error--Inf. loop.");
   };

// At the end of the exit chart, add a jump into the simulated RTS stub
// to simulate the return to the runtime system code (scheduler) following
// the completion of a partition.

 exit_path.append(new String("_simulated_RTS_stub"));

};



// The member make_branch_map is one of the two user-callable routines
// for creating a complete branch chart through a partition.  Given
// a single target, and the two boolean parameters fall_through_p and
// repeat_p, it computes all paths from the root node to the target BB
// (which touch each bbnode at most once), and then adds an exit chart
// to each route to the target bbnode to create a complete branch chart
// for that partition

branch_chart_list& nodemap::make_branch_map(String& target,
bool fall_through_p, bool repeat_p)

{
  branch_chart_list* master_list = new branch_chart_list;
  QVHMap<bbnode*,bool>* nr = new QVHMap<bbnode*,bool>(FALSE,10);
  bbnode* t_node;

// find all paths to the target bbnode

  find_path(root,target,*(new branch_chart),*nr,t_node, *master_list);

// clear the node registry

  nr->~QVHMap();
  nr = new QVHMap<bbnode*,bool>(FALSE,10);

// compute the exit chart from the target bbnode

  branch_chart exit_path = *(new branch_chart);
  find_exit_path(exit_path,t_node,*nr,fall_through_p,repeat_p,target);

// append the exit chart to each path from the root bbnode to the target
// bbnode

  multi_append_lists(*master_list,exit_path);

  return *master_list;

};

// Although bearing the same name as the preceding member, this version
// of make_branch_map allows branch charts to be generated which, instead
// of merely passing through a single target bbnode, can be chosen to
// pass through a series of connected bbnodes.  To call this branch charting
// procedure, therefore, you must supply *two* targets, which should
// correspond to the labels of the first and last bbnodes in the connected
// series you want to travel down.  The parameters are the same as in
// the first version of make_branch_map, and determine exit chart
// characteristics.

branch_chart_list&
nodemap::make_branch_map(String& target1, String& target2,
 bool fall_through_p, bool repeat_p)

{ branch_chart_list* master_list1 = new branch_chart_list;
  QVHMap<bbnode*,bool>* nr = new QVHMap<bbnode*,bool>(FALSE,10);
  bbnode* t_node;

  // compute branch charts from the root bbnode to the first target bbnode

  find_path(root,target1,*(new branch_chart),*nr,t_node, *master_list1);

  // clear the node registry

  nr->~QVHMap();
  nr = new QVHMap<bbnode*,bool>(FALSE,10);

  // compute branch charts from the first target bbnode to the second

  branch_chart_list* master_list2 = new branch_chart_list;
  find_path(t_node,target2,*(new branch_chart),*nr,t_node, *master_list2);

  // take the direct product of the two preceding branch charts

  branch_chart_list& ml = (*master_list1) * (*master_list2);

  // clear the node registry

  nr->~QVHMap();
  nr = new QVHMap<bbnode*,bool>(FALSE,10);

  // compute exit chart from the second target bbnode out of the partition

  branch_chart exit_path = *(new branch_chart);
  find_exit_path(exit_path,t_node,*nr,fall_through_p,repeat_p,target1);

  // append the exit chart onto each branch chart in ml running from the
  // root bbnode to the second target bbnode.
```

```
  multi_append_lists(ml,exit_path);

  return ml;
};


// The following procedure appends one branch chart to each element of
// a list of branch charts.  It's used to append the exit chart to the
// list of branch charts from the root bbnode to the target bbnode.

branch_chart_list&
multi_append_lists(branch_chart_list& ml,
    branch_chart& tail)
{for(DListIter<branch_chart*> mlIter = *(ml.iter()); !mlIter.end_p(); mlIter++)
   mlIter.value()->append(tail);

 return ml;};


// The following procedure acting on branch_chart_lists was found
// useful in the implementation of the nodemap ADT.  It takes a direct
// product of two branch_chart_lists.

branch_chart_list& operator *(branch_chart_list& s1, branch_chart_list& s2)
{
  branch_chart_list* product = new branch_chart_list;  // make the output list

  DListIter<branch_chart*> iIter1 = *s1.iter();      // build the iterators
  DListIter<branch_chart*> iIter2 = *s2.iter();      // over the two input
                                                     // lists
  branch_chart* temp;

  for ( ; !iIter1.end_p(); iIter1++, iIter2.first())  // loop over the two
    for ( ; !iIter2.end_p(); iIter2++)                // lists
      {temp = new branch_chart(*iIter1.value());
       temp->append(*iIter2.value());
       product->append(temp);};              // appending each product of two
                                             // terms from separate lists to
                                             // the output list

  return *product;                           // return the output list
};


// handle the initial portion of the exit path in a branch chart

void
nodemap::build_init_exit_path(branch_chart& exit_path, bbnode*& t_node,
      QVHMap<bbnode*,bool>& nr,
      bool& fall_through_p, bool& repeat_p,
      String& btarget, bool& done_with_main_BB)

{

// If the repeat flag is set, and the loop can jump to the beginning
// of the current node, add such a loop back to the exit chart.

if (repeat_p && (t_node->target_label() == btarget))
        {exit_path.append(&btarget);
   t_node = t_node->get_direct_target();
   repeat_p = FALSE;}
else
  {repeat_p = FALSE;
// If the fall-through flag is set, and the node has a fall-through
// descendent, then take the fall-through and add a "" to the exit chart.

if ((fall_through_p) && t_node->has_fall_thru_p())
  {exit_path.append(new String(""));
   t_node = t_node->get_fall_thru();}

// If the fall-through flag is clear, and the node has a direct
// branch target, add that branch label to the exit chart, and then take
// the branch to the target BB node.

else if (!(fall_through_p))
  { fall_through_p = TRUE;
    if (t_node->has_direct_target_p())
      {exit_path.append(new String(t_node->target_label()));
       t_node = t_node->get_direct_target();}

// If the fall-through flag is clear and the node has an indirect target,
// then jump to the indirectly linked node, and add a "_simulated_RTS_stub"
// label and the label of the indirectly linked node to the exit chart.

    else if (t_node->has_indirect_target_p())
      {exit_path.append(new String("_simulated_RTS_stub"));
       t_node = t_node->get_indirect_target();
       exit_path.append(new String(t_node->get_label()));};


  };
 };
};
```

```
// QPROF bbnode ADT implementation -- sybok@athena


// This data type is used to represent basic blocks of PowerPC instructions,
// to allow the branch path generator to generate a correct sequence of
// branches through a partition.

#ifndef QPROF_BBNODE
#define QPROF_BBNODE

#include <iostream.h>
#include <String.h>
#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"
#include "ICache.h"


const NOT_FOUND = -1;

enum ntype {NODE, INDIRECT_NODE, EMPTY};                    // type of child

enum nexit_type {BRANCH, FALL_THRU_2_LABEL, UNINITIALIZED};  // type of bbnode

enum guide_elt {NO_GUIDE,GO_LEFT,GO_RIGHT,END_POINT};


String& get_branch_target( ppc_Inst& i);


class bbnode
{
 public:

  // constructors

  bbnode(): _type(UNINITIALIZED)
    {lbl = ""; left_node = 0; right_node = 0;
     left_type = EMPTY; right_type = EMPTY; guide = NO_GUIDE;};

  bbnode( DList<ppc_Inst*>& dl, ICache* icache_p);

  guide_elt get_guide() const
    {return guide;};

  void set_guide(guide_elt guide_selection)
    {guide = guide_selection;};

  // left child predicates and modifiers

  bool has_fall_thru_p() const
    {return (left_type == NODE);};

  void set_fall_thru(bbnode* bbp)
    {left_node = bbp;};

  bbnode* get_fall_thru() const
    {return left_node;};

  // right child direct-target predicates and modifiers

  bool has_direct_target_p() const
    {return (right_type == NODE);};

  void set_direct_target(bbnode* bbp)
    {right_node = bbp;};

  bbnode* get_direct_target() const
    {return right_node;};

  // right child indirect-target predicates and modifiers

  bool has_indirect_target_p() const
    {return (right_type == INDIRECT_NODE);};

  void set_indirect_target(bbnode* bbp)
    {right_node = bbp;};

  bbnode* get_indirect_target() const
    {return right_node;};

  // get branch bbnode's target

  String target_label() const
    {return t_label;};

  // get type of bbnode

  const nexit_type get_exit_type()
    {return _type;};

  // check and get label of bbnode

  bool contains_label(const String& s)
```

```
    {return (lbl == s);};

  String get_label()
    {return lbl;};

void print(int level = 0) {

if (level < 10)
{  cout << "[ " << lbl << "(";
   if (has_fall_thru_p())
     get_fall_thru()->print(level+1);
   else cout << "EMPTY";

   cout << ") (";

   if (has_direct_target_p())
     get_direct_target()->print(level+1);
   else if (has_indirect_target_p())
     get_indirect_target()->print(level+1);
   else cout << "EMPTY";

   cout << ")]";}
else
  cout << "..."; };

 private:

  bool is_lbr_rts_jump( ppc_Inst& i) const;

  // local pointer to the ICache

  ICache* icp;

  // bbnode represenation data structures

  String t_label;

  String lbl;

  nexit_type _type;

  bbnode* left_node;
  ntype left_type;

  bbnode* right_node;
  ntype right_type;

  guide_elt guide;

};


#endif
```

```
// QPROF bbnode ADT implementation -- sybok@athena


// This data type is used to represent basic blocks of PowerPC instructions,
// to allow the branch path generator to generate a correct sequence of
// branches through a partition.

#include "bbnode.h"

// This the constructor for bbnodes which works given a DList of
// the instructions in the bbnode and the ICache object for the containing
// module.

bbnode::bbnode( DList<ppc_Inst*>& dl, ICache* icache_p)
{
  t_label = *(new String(""));

  guide = NO_GUIDE;

  icp = icache_p;

  ppc_InstIter& iIter = *dl.iter();
  lbl = iIter.value()->label();

  iIter.last();
  ppc_Inst* lasti = iIter.value();

  if (lasti->op().inert_p())

    {left_type = NODE;
     right_type = EMPTY;
     _type = FALL_THRU_2_LABEL;}
  else
    {_type = BRANCH;
     if ((!lasti->op().reg_br_p()) && (!is_lbr_rts_jump(*lasti)))
       {right_type = NODE;
t_label = get_branch_target(*lasti);}
     else if (lasti->op().linked_p())
       right_type = INDIRECT_NODE;
     else
       right_type = EMPTY;

     if (lasti->op().brc_p(lasti->rands(),lasti->nrands()) ||
         lasti->op().cdep(lasti->rands(),lasti->nrands()))
       left_type = NODE;
     else
       left_type = EMPTY;};
};


// The is_lbr_rts_jump function looks at the target name of a direct
// jump to a label and determines whether the label is in user code or
// the RTS.  It returns true iff the jump is into the RTS.

bool bbnode::is_lbr_rts_jump( ppc_Inst& i) const
{
  ppc_Rand** rarray= i.rands();
  int location = NOT_FOUND;

  if (i.op().inert_p() || i.op().reg_br_p())
    sigerr("bbnode::is_lbr_rts_jump: Argument not a branch to label.");

  for (int j = 0 ; j < i.op().nrands(); j++)
    if (rarray[j]->label_p()) location = j;

  if (location == NOT_FOUND)
    sigerr("bbnode::is_lbr_rts_jump: Branch to label missing label.");

  return !(icp->exists_p(rarray[location]->label()));

};

// The get_branch_target function returns the exit target label of a bbnode
// supposing that the bbnode is terminated by a branch to label instruction
// (rather than a branch to register).

String& get_branch_target( ppc_Inst& i)
{

ppc_Rand** rarray= i.rands();
  int location = NOT_FOUND;

  if (i.op().inert_p() || i.op().reg_br_p())
    sigerr("bbnode::get_branch_target: Argument not a branch to label.");

  for (int j = 0; j < i.op().nrands(); j++)
       if (rarray[j]->label_p()) location = j;

  if (location == NOT_FOUND)
    sigerr("bbnode::get_branch_target: Branch to label missing label.");

  String* local_str = new String(rarray[location]->label());

  return *local_str;

};
```

```
// QPROF bqueue ADT -- sybok@athena

// The bqueue ADT is designed to simulate the instruction dispatch buffers
// of the branch processor.  There are 8 buffers for dispatching sequential
// instructions, and 4 for collecting branch target instructions before
// the branch has been resolved.

#ifndef NULL
#define NULL = 0
#endif

// define an improved modulus function

#ifndef NEWMOD
#define NEWMOD

inline int mod(int n, int m)
{ return ((n % m) + m) % m; };

#endif

#ifndef QPROF_BQUEUE
#define QPROF_BQUEUE

#include <iostream.h>
#include <String.h>

#include "DList.h"
#include "ppc_Inst.h"
#include "err.h"

const SEQSIZE = 8;
const JMPSIZE = 4;


// some important primitives

inline int seq_succ(int j) {return mod(j + 1,SEQSIZE+1);};
inline int seq_pred(int j) {return mod(j - 1,SEQSIZE+1);};

inline int jmp_succ(int j) {return mod(j + 1,JMPSIZE+1);};
inline int jmp_pred(int j) {return mod(j - 1,JMPSIZE+1);};


class bqueue
{
 public:
  bqueue() {seqhead = 0; seqtail = 0;
    jmphead = 0; jmptail = 0;};

  int seq_empty_p() {return (seqtail == seqhead);};
  int jmp_empty_p() {return (jmptail == jmphead);};

  void load_seq( DList<ppc_Inst*>& fetched_list );
  void load_jmp( DList<ppc_Inst*>& fetched_list );

  void sideload();       // slam the seq. buffers with target buf. contents

  void purge_jmp();      // reset jmp. buffers to empty state (branch failed)

  bool see_branch_p();          // can you see a branch downstream?

  ppc_Inst* extract_branch();   // extract it

  int seq_room() {return (SEQSIZE - seqsize());};
  int jmp_room() {return (JMPSIZE - jmpsize());};

  ppc_Inst* seq_peek();
  ppc_Inst* seq_pop();
  ppc_Inst* jmp_pop();

  int seqsize() {return mod(seqtail-seqhead,SEQSIZE+1);};
  int jmpsize() {return mod(jmptail-jmphead,JMPSIZE+1);};

  void print();

 private:

  // arrays are used to hold the instruction buffers

  ppc_Inst* seqbuf[SEQSIZE+1];
  ppc_Inst* jmpbuf[JMPSIZE+1];

  int seqhead, seqtail;
  int jmphead,jmptail;

  void internal_load_seq(ppc_Inst* ins);
  void internal_load_jmp(ppc_Inst* ins);
};


// load seq. buffer with 1 instruction

inline void bqueue::internal_load_seq(ppc_Inst* ins)
{
  if (seq_room() <= 0) sigerr("bqueue::internal_load_seq: No room left.");

  seqbuf[seqtail] = ins;
  seqtail = seq_succ(seqtail);

};


// load branch target buffer with 1 instruction

inline void bqueue::internal_load_jmp(ppc_Inst* ins)
{
  if (jmp_room() <= 0) sigerr("bqueue::internal_load_jmp: No room left.");

  jmpbuf[jmptail] = ins;
  jmptail = jmp_succ(jmptail);

};


// load seq. buffer with list of instructions

inline void bqueue::load_seq( DList<ppc_Inst*>& fetched_list )
{
  for(ppc_InstIter& iIter = *(fetched_list.iter());
      !iIter.end_p() ; iIter++)
    internal_load_seq(iIter.value());
};

// load branch target buffer with list of instructions

inline void bqueue::load_jmp( DList<ppc_Inst*>& fetched_list )
{
  for(ppc_InstIter& iIter = *(fetched_list.iter());
      !iIter.end_p() ; iIter++)
    internal_load_jmp(iIter.value());
};

// pop seq. buffer entry

inline ppc_Inst* bqueue::seq_pop()
{

  if (seqsize() <= 0) sigerr("bqueue::seq_pop: Can't pop empty queue.");

  ppc_Inst* ip = seqbuf[seqhead];
  seqhead = seq_succ(seqhead);
  return ip;

};

// peek at head of seq. buffer

inline ppc_Inst* bqueue::seq_peek()
{
  if (seqsize() <= 0) sigerr("bqueue::seq_pop: Can't pop empty queue.");

  return (seqbuf[seqhead]);

};

// pop branch target buffer entry

inline ppc_Inst* bqueue::jmp_pop()
{
  if (jmpsize() <= 0)  sigerr("bqueue::jmp_pop: Can't pop empty queue.");

  ppc_Inst* ip = jmpbuf[jmphead];
  jmphead = jmp_succ(jmphead);
  return ip;

};

// smash seq. buf with branch target buffer entries

inline void bqueue::sideload()
{ seqhead = 0; seqtail = 0;
  while (jmpsize() > 0)
    internal_load_seq(jmp_pop());
  jmphead = 0; jmptail = 0;
};


// reset jmp. buf (fail branch)

inline void bqueue::purge_jmp()
{jmphead = 0; jmptail = 0;}


// look downstream for a branch

inline bool bqueue::see_branch_p()
{
  for (int i = seqhead;
       (i !=  seqtail) && (i != (mod(seqhead+5,SEQSIZE+1)));
i = seq_succ(i))

    if (!seqbuf[i]->op().inert_p())
      return TRUE;

  return FALSE;
};

// extract branch from sequential buffer

inline ppc_Inst* bqueue::extract_branch()
```

```
{
  for (int i = seqhead;
       (seqhead !=  seqtail) && (i != mod(seqhead+5,SEQSIZE+1));
       i = seq_succ(i))
    if (!seqbuf[i]->op().inert_p())
      return seqbuf[i];

  sigerr("bqueue::extract_branch: No Branch.");
  return NULL;
};

inline void bqueue::print()
{
  cout << "BQUEUE OBJECT:" << endl;
  cout << "------------------------------------------------------" << endl;

  cout << "[SEQ QUEUE: size = " << seqsize();
  cout << ", room = " << seq_room() << "]" << endl;
  for(int i = seqhead; i != seqtail; i = seq_succ(i))
    seqbuf[i]->print();

  cout << "[JMP QUEUE: size = " << jmpsize();
  cout << ", room = " << jmp_room() << "]" << endl;
  for(i = jmphead; i != jmptail; i = jmp_succ(i))
    jmpbuf[i]->print();

  cout << "------------------------------------------------------" << endl;

};

#endif
```

```
// QPROF ICache abstraction -- sybok@athena

// This ADT simulates the ICache by fetching up to 4 instructions at a time
// from a partition of a PowerPC module.  A new ICache object must be
// generated for each PowerPC module.  The ICache has its own internal
// instruction pointer which is used as a default when instructions are
// fetched and loaded from it.  A label target can be specified instead,
// which would typically be used to satisfy branches.

#ifndef QPROF_ICACHE
#define QPROF_ICACHE

#include <iostream.h>
#include <String.h>
#include "DList.h"
#include "QVHMap.h"
#include "err.h"

#include "ppc_Mod.h"
#include "ppc_Proc.h"
#include "ppc_CB.h"
#include "ppc_Part.h"
#include "ppc_Inst.h"

#include "sT_to_PPC.h"

// The class ins_pointer represents an ICache ip.  It must contain
// a pointer to the partition to allow the instructions following the
// one currently point at to be obtained.

class ins_pointer
{
 public:
  ins_pointer()
    {i_pntr = 0;
     p_pntr = 0;}

  ins_pointer(ppc_Inst* i, ppc_Part* p)
    {i_pntr = i; p_pntr = p;}

  ppc_Inst* ins() const {return i_pntr;};
  ppc_Part* part() const {return p_pntr;};

 private:
  ppc_Inst* i_pntr;
  ppc_Part* p_pntr;
};


// some important definitions and constants

typedef QVHMap<String,ins_pointer*> labelMap; // map targets to instructions

typedef DList<ppc_Inst*> ins_list;       // The ICache will return a list
                                         // pointers to PPC instructions
                                         // representing the fetched
                                         // instructions.

const String header("_simulated_RTS_stub");

const FETCH_LIMIT = 4;

// the class ICache

class ICache
{
 public:

  ICache(const ppc_Mod& in_mod): _map(0)
                                                  // construct from a ppc module
    {_mod = in_mod; _curr_inst = 0; _outlist = 0; init_labelMap();};

  void fetch(const String& target);               // fetch at target
  void fetch(int lim = FETCH_LIMIT);              // fetch at current ICache ip

  ins_list& load() const {return *_outlist;}      // load fetched instructions

  bool exists_p(const String& target) const       // checks whether label exists
    {return _map.contains(target);}               // in current module

  ppc_Part* partition(const String& target);      // returns partition containing
                                                  // target label--assumes label
                                                  // exists in current module

  ppc_Inst* inst(const String& target);

 private:

  void init_labelMap();                  // the initialization function
  ppc_Mod _mod;                          // the original PPC module

  labelMap _map;                         // the target -> instruction map

  ins_pointer* _curr_inst;               // the ICache ip

  ins_list* _outlist;                    // the fetched instructions

  ppc_Part* generate_RTS_stub();
```

```
};


// This function returns the partition associated with a label.

inline ppc_Part* ICache::partition(const String& target)
{
  if (!_map.contains(target)) sigerr("ICache::partition: Target not found.");

  return _map[target]->part();

};

inline ppc_Inst* ICache::inst(const String& target)
{
  if (!_map.contains(target)) sigerr("ICache::inst: Target not found.");

  return _map[target]->ins();

};


#endif
```

```
// QPROF ICache abstraction -- sybok@athena

// This ADT simulates the ICache by fetching up to 4 instructions at a time
// from a partition of a PowerPC module.  A new ICache object must be
// generated for each PowerPC module.  The ICache has its own internal
// instruction pointer which is used as a default when instructions are
// fetched and loaded from it.  A label target can be specified instead,
// which would typically be used to satisfy branches.

#include "ICache.h"

// To perform the fetch at the current ICache ip, traverse the
// partition corresponding to that ICache ip, and when the instruction
// marked by that ip is found, start building the list of fetched
// instructions.  Collect exactly lim instructions, where lim is
// between 0 and 4 inclusive. The list becomes _outlist.

void ICache::fetch(int lim)
{lim = (lim < FETCH_LIMIT) ? lim : FETCH_LIMIT;

 if (_outlist != 0) _outlist->~DList();

 ins_list* temp = new ins_list;

 String cached_label = "";

 int size_cnt = 0;
 int k;

 ppc_Inst* ppci;
 ppc_Rand** pr;

  if ((_curr_inst != 0) && (lim > 0))
    {
      for(ppc_InstIter iInst(_curr_inst->part()->instIter()) ;
  !iInst.end_p() && (size_cnt < lim); iInst++)
{
  if ((iInst.value()->op().op() == ppc_none) ||
      ((iInst.value()->op().op() >= ppc__proc) &&
       (iInst.value()->op().op() <= ppc__extern)))

   {if (iInst.value() == _curr_inst->ins()) {lim++; size_cnt++;};
    if ((!iInst.value()->label().empty()) && (cached_label.empty()))
      cached_label = iInst.value()->label();
    continue;};

  if ((iInst.value() == _curr_inst->ins()) || (size_cnt > 0))
   if ((cached_label.empty()) ||
       ((iInst.value() == _curr_inst->ins())
&& (!iInst.value()->label().empty())))
     {size_cnt++; temp->append(iInst.value());}
   else
     {size_cnt++;
      ppci = new ppc_Inst(cached_label,iInst.value()->op(),
  iInst.value()->source(),
  iInst.value()->line());
      ppci->codeGenProps(iInst.value()->codeGenProps());
      pr = iInst.value()->rands();
             for(k = 0; k < iInst.value()->nrands(); k++)
ppci->addRand(pr[k]);
      temp->append(ppci);};

  cached_label = "";
       };

      while ((!iInst.end_p()) &&
     ((iInst.value()->op().op() == ppc_none) ||
      ((iInst.value()->op().op() >= ppc__proc) &&
       (iInst.value()->op().op() <= ppc__extern))))
iInst++;

      if (!iInst.end_p())
_curr_inst = new ins_pointer(iInst.value(),_curr_inst->part());
      else
_curr_inst = 0;

    };

 _outlist = temp;

};

// To fetch from a target, smash the ICache ip with the value of the map
// function, then revert to the simple fetch function.

void ICache::fetch(const String& target)
{
  if (!_map.contains(target)) sigerr("ICache::fetch: Target not found.");

  _curr_inst = _map[target];

  fetch(FETCH_LIMIT);
};


// To initialize the map, we traverse each Procedure, Code Block, Partition,
// and Instruction, remembering the labelled instructions.

void ICache::init_labelMap()
{
```

```
  for(ppc_ProcIter iProc(_mod.procIter()); !iProc.end_p(); iProc++)
    for(ppc_CBIter iCB(iProc.value() -> CBIter()); !iCB.end_p() ; iCB++)
      for(ppc_PartIter iPart(iCB.value()->partIter());!iPart.end_p();iPart++)
for(ppc_InstIter
    iInst(iPart.value()->instIter());!iInst.end_p();iInst++)
  {String s = iInst.value()->label();
   if (!s.empty())
     _map[s] = new ins_pointer(iInst.value(),iPart.value());};

  ppc_Part* part = generate_RTS_stub();
  _map[header] = new ins_pointer((*part)[0],part);

};


// We generate an artificial partition simulating instructions that
// might be found in the runtime system code to simulate jumps into
// the RTS.

ppc_Part* ICache::generate_RTS_stub()
{
  ppc_Part* part = new ppc_Part("RTS stub",THREAD_PART);


 ppc_Inst** i = new ppc_Inst*[11];

 i[1] = new ppc_Inst(ppc_mflr,header);
 i[1]->rands(new ppc_Rand(r0));

 i[2] = new ppc_Inst(ppc_mfcr);
 i[2]->rands(new ppc_Rand(r10));

 i[3] = new ppc_Inst(ppc_stw);
 i[3]->rands(new ppc_Rand(r0),new ppc_Rand(rSP),new ppc_Rand(8));

 i[4] = new ppc_Inst(ppc_stw);
 i[4]->rands(new ppc_Rand(r10),new ppc_Rand(rSP),new ppc_Rand(4));

 i[5] = new ppc_Inst(ppc_stwu);
 i[5]->rands(new ppc_Rand(rSP),new ppc_Rand(rSP),new ppc_Rand("neg_szdsa"));

 i[6] = new ppc_Inst(ppc_addi);
 i[6]->rands(new ppc_Rand(rSP),new ppc_Rand(rSP),new ppc_Rand("szdsa"));

 i[7] = new ppc_Inst(ppc_lwz);
 i[7]->rands(new ppc_Rand(r0),new ppc_Rand(rSP),new ppc_Rand(8));

 i[8] = new ppc_Inst(ppc_lwz);
 i[8]->rands(new ppc_Rand(r10),new ppc_Rand(rSP),new ppc_Rand(4));

 i[9] = new ppc_Inst(ppc_mtlr);
 i[9]->rands(new ppc_Rand(r0));

 i[10] = new ppc_Inst(ppc_mtcrf);
 i[10]->rands(new ppc_Rand(0x38),new ppc_Rand(r10));

 i[11] = new ppc_Inst(ppc_b);
 i[11]->rands(new ppc_Rand(header));

 for(int j = 1; j <= 11; j++)
   part->append(i[j]);

 return part;

};
```

```
// QPROF DCache ADT -- sybok@athena

// This ADT simulates the behavior of the data cache in accepting load
// and store requests from the FXU and FPU units and returning "data"
// (simulated) to the requesting functional unit.


#ifndef QPROF_DCACHE
#define QPROF_DCACHE

#include "addr.h"
#include "err.h"

enum fpu_data { FDATA }; // singleton data type

const NO_TARGET = -1;

class DCache
{
 public:

  DCache();         // the constructor

  void tick();      // advance the state of the DCache by one clock cycle

  // return the DCache's saved state

  // process the "return value" phase of an FXU load request

  bool fxu_load_val_ready_p() const {return fxu_load_return;};
  int fxu_load_val();

  // process the "return value" phase of an FPU load request

  bool fpu_load_val_ready_p() const {return fpu_load_return;};
  int fpu_load_val();

  // process the "request" phase of a load request (always made by FXU)

  bool request_load_ready_p() const {return !load_request_present;};
  void request_load(bool fxu_is_target_p,
    int target,
    addr s_addr,
    int collision_cnt);

  // process an FX "store buffer" store (FXU supplies DATA and ADDRESS)

  bool sb_store_ready_p() const {return !fxu_store_present;};
  void sb_store(addr target_addr);

  // process a FP "pending store queue" store (FXU supplies ADDRESS only)

  bool psq_store_ready_p() const
    {return (!fpu_store_addr_present);};
  void psq_store(addr target_addr);

  // process a FP "data store queue" store (FPU supplies DATA only)

  bool dsq_store_ready_p() const {return !fpu_store_data_present;};
  void dsq_store(fpu_data data);

 private:

  bool load_request_present;       // slots for receiving load request
  bool fxu_load;
  int load_target;
  int load_collisions;
  addr source_addr;

  bool fxu_load_return;            // slots for load returning to FXU
  int fxu_load_ret_target;

  bool fpu_load_return;            // slots for load returning to FPU
  int fpu_load_ret_target;

  bool fxu_store_present;          // slots for accepting FX store from FXU
  addr fxu_st_target_addr;

  bool fxu_performing_store;       // delay slot for FXU store
  addr fxu_perf_st_target_addr;


  bool fpu_store_addr_present;     // slots for accepting FP store from FXU
  bool fpu_store_data_present;     // and FPU
  addr fpu_st_target_addr;

  bool fpu_performing_store;       // delay slot for FPU store
  addr fpu_perf_st_target_addr;

  addr* make_generic_addr()
    {return new addr(*(new ppc_Inst(ppc_lwzx)));};

};


inline int DCache::fxu_load_val()
{
  if (!fxu_load_return) sigerr("DCache::fxu_load_val: No Fxu Load Ready.");

  fxu_load_return = FALSE;
```

```
  return fxu_load_ret_target;

};

inline void DCache::sb_store(addr target_addr)
{
  if (fxu_store_present)
    sigerr("DCache::sb_store: Can't Send FXU Store Yet.");

  fxu_st_target_addr = target_addr;
  fxu_store_present = TRUE;

};


inline void DCache::psq_store(addr target_addr)
{
  if (fpu_store_addr_present)
    sigerr("DCache::psq_store: Can't Send FPU Store Yet.");

  fpu_st_target_addr = target_addr;
  fpu_store_addr_present = TRUE;

};

inline int DCache::fpu_load_val()
{
  if (!fpu_load_return) sigerr("DCache::fpu_load_val: No FPU Load Ready.");

  fpu_load_return = FALSE;
  return fpu_load_ret_target;

};

inline void DCache::dsq_store(fpu_data data)
{
  if (fpu_store_data_present)
    sigerr("DCache::dsq_store: No FPU Load Ready.");

  fpu_store_data_present = TRUE;

};

#endif
```

```
// QPROF DCache ADT -- sybok@athena

// This ADT simulates the behavior of the data cache in accepting load
// and store requests from the FXU and FPU units and returning "data"
// (simulated) to the requesting functional unit.

#include "DCache.h"

void DCache::request_load(bool fxu_is_target_p,
  int target,
  addr s_addr,
  int collision_cnt)
{
  if (load_request_present)
    sigerr("DCache::request_load: Can't Send Load Request Yet.");

  fxu_load = fxu_is_target_p;
  load_target = target;
  load_collisions = collision_cnt;
  source_addr = s_addr;
  load_request_present = TRUE;

};

void DCache::tick()
{
  // check whether a store has just occurred with an address that matches
  // that of the load waiting due to a detected collision.  If so, decrement
  // the collision counter.

  if ((load_request_present) && (load_collisions > 0))
    {if ((fxu_performing_store) &&
 (fxu_perf_st_target_addr == source_addr))
       load_collisions--;

     if ((fpu_performing_store) &&
 (fpu_perf_st_target_addr == source_addr))
       load_collisions--;
   };

  // if all collisions have been dealt with, and the load return slots
  // for the appropriate unit are free, fill them with the return data
  // for this load and free up the load request slot.

  if ((load_request_present) && (load_collisions == 0))
    if ((fxu_load) && (!fxu_load_return))
      {fxu_load_return = TRUE;
       load_request_present = FALSE;
       fxu_load_ret_target = load_target;}
    else if ((!fxu_load) && (!fpu_load_return))
      {fpu_load_return = TRUE;
       load_request_present = FALSE;
       fpu_load_ret_target = load_target;};

  // process an fxu store.

  fxu_performing_store = FALSE;
  if (fxu_store_present)
    {fxu_store_present = FALSE;
     fxu_performing_store = TRUE;
     fxu_perf_st_target_addr = fxu_st_target_addr;};

  // process an fpu store.

  fpu_performing_store = FALSE;
  if ((fpu_store_addr_present) && (fpu_store_data_present))
    {fpu_store_addr_present = FALSE;
     fpu_store_data_present = FALSE;
     fpu_performing_store = TRUE;
     fpu_perf_st_target_addr = fpu_st_target_addr;};

};


DCache::DCache()
{
  load_request_present = FALSE;
  fxu_load = FALSE;
  load_target = NO_TARGET;
  load_collisions = 0;
  source_addr = *make_generic_addr();

  fxu_load_return = FALSE;
  fpu_load_return = FALSE;

  fxu_store_present = FALSE;
  fxu_st_target_addr = *make_generic_addr();
  fxu_performing_store = FALSE;
  fxu_perf_st_target_addr = *make_generic_addr();

  fpu_store_addr_present = FALSE;
  fpu_store_data_present = FALSE;
  fpu_st_target_addr = *make_generic_addr();
  fpu_performing_store = FALSE;
  fpu_perf_st_target_addr = *make_generic_addr();

};
```

```
// QPROF fprmap ADT -- sybok@athena

// The fprmap ADT is designed to simulate the register remapping strategy
// used by the RS/6000 FPU.

#ifndef QPROF_FPRMAP
#define QPROF_FPRMAP

#include <iostream.h>
#include <String.h>
#include "err.h"

#include "fpdecode_entry.h"
#include "psq_entry.h"
#include "gqueue.h"
#include "grqueue.h"

const FPR_COUNT = 32;

class fprmap
{
 public:
  fprmap();

  // handle arithmetic FPU instructions

  fpdecode_entry arith_map(ppc_Inst* ins) const;


  // handle FP stores

  psq_entry store_map(ppc_Inst* ins) const;

  // handle FP loads

  bool load_map_possible_p(gqueue<int>* flist,
   grqueue<int>* ptrq, gqueue<int>* olq);

  void load_map(ppc_Inst* ins, gqueue<int>* flist,
grqueue<int>* ptrq, gqueue<int>* olq);

  // allow direct access to map

  int checkmap(int orig_target) const {return maptable[orig_target];};

  // copy the maptable

  fprmap* copy();

 private:

  int maptable[FPR_COUNT];

};


// the constructor

inline fprmap::fprmap()
{
  for(int i = 0; i < FPR_COUNT; i++)
    maptable[i] = i;
};


// check whether doing a load is possible

inline bool fprmap::load_map_possible_p(gqueue<int>* flist,
 grqueue<int>* ptrq,
 gqueue<int>* olq)
{
  return((flist->size() > 0) && (ptrq->room() > 0) &&
 (olq->room() > 0));
};

// copy the fprmap ADT object

inline fprmap* fprmap::copy()
{
  fprmap* new_fprmap = new fprmap;

  for(int i = 0; i < FPR_COUNT; i++)
    new_fprmap->maptable[i] = maptable[i];

  return new_fprmap;
};


#endif
```

```
// QPROF fprmap ADT -- sybok@athena

// The fprmap ADT is designed to simulate the register remapping strategy
// used by the RS/6000 FPU.

#include "fprmap.h"

// Arithmetic instructions are handled simply by remapping their registers
// according to the table.

fpdecode_entry fprmap::arith_map(ppc_Inst* ins) const
{
  optype op = ins->op().opt();
  if ((op != FPU) && (op != FPU_A) && (op != FPU_CMP))
    sigerr("fprmap::arith_map: Not an FPU arithmetic instruction.");

  ppc_Inst* new_ins =
    new ppc_Inst(ins->label(),ins->op(),ins->source(),ins->line());

  for(int i = 1; i <= ins->nrands(); i++)
    {ppc_Rand* rd = ins->rand(i);
     if ((rd->reg_p()) && (rd->reg().type() == FPR_RTYPE))
       {ppc_Reg* rg = new ppc_Reg(rd->reg().id(),
  FPR_RTYPE,
  maptable[rd->reg().actual()]);
new_ins->addRand(new ppc_Rand(*rg));}
      else
        new_ins->addRand(rd);};

  return(*(new fpdecode_entry(new_ins)));

};

// Stores also have their FP registers remapped, but the result is
// a psq_entry with a "GB" bit of zero rather than a fpdecode_entry object

psq_entry fprmap::store_map(ppc_Inst* ins) const
{
  if (!(ins->op().fp_store_p()))
    sigerr("fprmap::store_map: Not an FPU store.");

  ppc_Inst* new_ins =
    new ppc_Inst(ins->label(),ins->op(),ins->source(),ins->line());

  for(int i = 1; i <= ins->nrands(); i++)
    {ppc_Rand* rd = ins->rand(i);
     if ((rd->reg_p()) && (rd->reg().type() == FPR_RTYPE))
       {ppc_Reg* rg = new ppc_Reg(rd->reg().id(),
  FPR_RTYPE,
  maptable[rd->reg().actual()]);
new_ins->addRand(new ppc_Rand(*rg));}
      else
        new_ins->addRand(rd);};

  return(*(new psq_entry(new_ins)));

};
// Do the load by manipulating the various register flag buffers and
// altering the table.

void fprmap::load_map(ppc_Inst* ins, gqueue<int>* flist,
      grqueue<int>* ptrq, gqueue<int>* olq)
{
  if ((flist->size() <= 0) || (ptrq->room() <= 0) ||
      (olq->room() <= 0))
    sigerr("fprmap::load_map: Can't load now.");

  ppc_Reg rg = ins->rand(1)->reg();

  ptrq->load_element(maptable[rg.actual()]);

  maptable[rg.actual()] = flist->pop();

  olq->load_element(maptable[rg.actual()]);

};
```

```
// QPROF timer ADT -- sybok@athena

// The timer ADT is used to keep track of the cycles required to
// execute a series of PowerPC instructions

#ifndef QPROF_TIMER
#define QPROF_TIMER

enum timer_cond {TIMER_DONE, TIMER_READY, TIMER_ON};

const String end_default = "_qprof_single_BB_default";

class timer
{
 public:

  timer(const String& s_label, const String& e_label): // constructor
    start(s_label), stop(e_label), orig_stop(e_label)
      {tc = TIMER_READY; count = 0; seen_start = FALSE;};


  bool done_p() {return (tc == TIMER_DONE);};          // is timing done?

  int get_count() const {return(count);};              // get the count

  void operator++() {if (tc == TIMER_ON) count++;};    // advance timer

  void alert_timer(const String& label);               // signal timer of
                                                       // event

  void mark_end(const String& label);                  // found generic end

 private:

  timer_cond tc;
  int count;
  String start, stop, orig_stop;
  bool seen_start;
};

inline void timer::mark_end(const String& label)
{
  if (start == label)
    seen_start = TRUE;
  else if (seen_start && !label.empty())
    {if (stop == end_default)
       stop = label;
     seen_start = FALSE;};
};


inline void timer::alert_timer(const String& label)
{

  switch(tc) {
  case TIMER_READY:
    if (label == start)
      tc = TIMER_ON;
    break;
  case TIMER_ON:
    if (label == stop)
      tc = TIMER_DONE;
    break;
  };

  if ((tc == TIMER_DONE) && (count > 0) && (label == start))
    {tc = TIMER_ON;
     count = 0;
     stop = orig_stop;};

};

#endif
```

```
// QPROF synchro ADT -- sybok@athena

// The synchro ADT is designed to enforce the constraint that the FPU and
// FXU remain a certain number of cycles, at most, out of sync.  By noting
// when each use dispatches an instruction into the pipeline, it can
// determine which unit is ahead and enforce this constraint.

#ifndef QPROF_SYNCHRO
#define QPROF_SYNCHRO

#include "err.h"

const FPU_lead_max = 6;
const FXU_lead_max = 2;

class synchro
{
 public:

  synchro() {displacement = 0;};

  bool fxu_shift_ok_p();   // is it OK to shift an FXU ins into FXU pipeline
  bool fpu_shift_ok_p();   // is it OK to shift an FPU ins into FPU pipeline

  void fxu_shift();        // shift an FXU ins into FXU pipeline
  void fpu_shift();        // shift an FPU ins into FPU pipeline

  bool fxu_ahead() const {return (displacement <= 0);}; // which unit is ahead?
  bool fpu_ahead() const {return (displacement > 0);};

  // state checkpointing operations

  void print();

 private:

  int displacement;

};

// ok to shift out FXU ins?

inline bool synchro::fxu_shift_ok_p()
{
  return(displacement > -FXU_lead_max);
};

// ok to shift out fpu ins?

inline bool synchro::fpu_shift_ok_p()
{
  return(displacement < FPU_lead_max);
};

// shift out an FXU ins

inline void synchro::fxu_shift()
{
  if (displacement > -FXU_lead_max)
    displacement--;
  else
    sigerr("synchro::fxu_shift: FXU unit too far ahead.");
};

inline void synchro::fpu_shift()
{
  if (displacement < FPU_lead_max)
    displacement++;
  else
    sigerr("synchro::fpu_shift: FPU unit too far ahead.");
};

inline void synchro::print()
{
  cout << "[SYNCHRO: fxu/fpu balance = " << displacement << "]" << endl;
};


#endif
```

```
// QPROF fpdecode_entry ADT -- sybok@athena

// The fpdecode_entry ADT is used to store the representation of a PowerPC
// instruction in the decode stage or decode buffers--it includes a pointer
// to the instruction as well as a store and load count field to accommodate
// the architecture.

#ifndef QPROF_FPDECODE_ENTRY
#define QPROF_FPDECODE_ENTRY

#include <iostream.h>
#include <String.h>

#include "ppc_Inst.h"
#include "ppc_Rator.h"

class fpdecode_entry
{
 public:
  fpdecode_entry(ppc_Inst* ins = 0);

  const ppc_Rator& get_rator() const {return *rat;};
  ppc_Inst* get_ins() const {return inst;};

  int get_lcount() const {return load_cnt;};
  int get_scount() const {return store_cnt;};

  friend fpdecode_entry add_load(fpdecode_entry e);
  friend fpdecode_entry add_store(fpdecode_entry e);

  bool operator ==(fpdecode_entry e) const {return FALSE;};

  friend ostream& operator << (ostream &s, const fpdecode_entry& fpdec_obj);

  void print();

 private:

  ppc_Inst* inst;
  const ppc_Rator* rat;

  int load_cnt;
  int store_cnt;
};


// create new object with incremented LC

inline fpdecode_entry add_load(fpdecode_entry e)
{
  fpdecode_entry* new_entry = new fpdecode_entry(e);
  new_entry->load_cnt = e.load_cnt+1;
  return(*new_entry);
};

// create new object with incremented SC

inline fpdecode_entry add_store(fpdecode_entry e)
{
  fpdecode_entry* new_entry = new fpdecode_entry(e);
  new_entry->store_cnt = e.store_cnt+1;
  return(*new_entry);
};

inline ostream& operator << (ostream &s, const fpdecode_entry& fpdec_obj)
{
  cout << "[FPDECODE_ENTRY: LC = " << fpdec_obj.load_cnt;
  cout << ", SC = " << fpdec_obj.store_cnt << ", ";
  fpdec_obj.inst->print();
  cout << "]" << endl;

  return s;
};

inline void fpdecode_entry::print()
{
  cout << "[FPDECODE_ENTRY: LC = " << load_cnt;
  cout << ", SC = " << store_cnt << ", ";
  inst->print();
  cout << "]" << endl;

};


#endif
```

```
// QPROF fpdecode_entry ADT -- sybok@athena

// The fpdecode_entry ADT is used to store the representation of a PowerPC
// instruction in the decode stage or decode buffers--it includes a pointer
// to the instruction as well as a store and load count field to accommodate
// the architecture.

#include "fpdecode_entry.h"
#include "err.h"

// construct an fpdecode_entry

fpdecode_entry::fpdecode_entry(ppc_Inst* ins):
      inst(ins), rat(&(ins->op()))
{
  if (ins != 0)
    {optype op = ins->op().opt();

     if ((op != FPU) && (op != FPU_A) && (op != FPU_CMP))
       sigerr("fpdecode_entry::constructor: Not FPU instruction.");

     load_cnt = 0;
     store_cnt = 0;
   };

};
```

```
// QPROF regbusy ADT -- sybok@athena

// The regbusy ADT is used to hold the lock status of the FXU registers
// in the FXU unit.

#ifndef QPROF_REGBUSY
#define QPROF_REGBUSY

#include "err.h"
#include "DList.h"

const REGCOUNT = 32;

class regbusy
{

 public:

  regbusy();                                    // construct the register lock

  bool busy_p(int n) const                      // determine whether a given
    {return lockset[n];};                      // lock is or is not busy


  void lock(int n);                             // lock a register
  void unlock(int n);                           // unlock a register

  bool clash(DList<int>& i) const;              // does a register list
                                                // contain any locked regs.?

 private:

  bool* lockset;

};

// construct a regbusy object

inline regbusy::regbusy()
{
  lockset = new bool[REGCOUNT];
  for(int i = 0; i < REGCOUNT; i++)
    lockset[i] = FALSE;

};

// lock a register

inline void regbusy::lock(int n)
{
  if (lockset[n])
    sigerr("regbusy::lock: Bit already locked.");
  else
    lockset[n] = TRUE;
};

// unlock a register

inline void regbusy::unlock(int n)
{
  if (!lockset[n])
    sigerr("regbusy::unlock: Bit already clear.");
  else
    lockset[n] = FALSE;
}

// test whether any registers in an iterator are locked

inline bool regbusy::clash(DList<int>& i) const
{
  for (DListIter<int> j(i); !j.end_p() ; j++)
    if (busy_p(j.value()))
      return TRUE;

  return FALSE;
};

#endif
```

```
// QPROF addr ADT -- sybok@athena

// This ADT was designed to hold the addresses stored in the SB and PSQ
// of the FXU and in certain internal DCache slots.

#ifndef QPROF_ADDR
#define QPROF_ADDR

#include <String.h>
#include "ppc_Inst.h"

class addr
{
 public:

  addr(const ppc_Inst& ins);              // standard constructor

  addr() {label = ""; generic = TRUE;};  // default constructor (generic addr)

  bool operator ==(const addr& address) const;     // equality testor

  friend ostream& operator << (ostream &s, const addr& addr_obj);

  void print();

 private:

  String label;   // target label of addr (if target is label)
  int n_label;    // numerical label of addr (if target is number)
  int reg;        // base register

  bool generic;   // whether or not the address is generic; above info
                  // only applies if it's not
};

inline ostream& operator << (ostream &s, const addr& addr_obj)
{
  cout << "[ADDR object: label = " << addr_obj.label;
  cout << ", n_label = " << addr_obj.n_label;
  cout << ", register = " << addr_obj.reg;
  cout << ", generic = " << addr_obj.generic << "]";

  return s;
};

inline void addr::print()
{
  cout << "[ADDR object: label = " << label << ", n_label = " << n_label;
  cout << ", register = " << reg << ", generic = " << generic << "]" << endl;

};

#endif
```

```
// QPROF addr ADT -- sybok@athena

// This ADT was designed to hold the addresses stored in the SB and PSQ
// of the FXU and in certain internal DCache slots.

#include "addr.h"
#include "err.h"

// Construct an 'addr' object from a store.  To be equal to objects must
// not be "generic" addresses (two register addressing), and must share the
// same base register and offset.

addr::addr(const ppc_Inst& ins)
{
  ppc_Opcode opcode = ins.op().op();
  optype op = ins.op().opt();

  if ((ins.op().load_p()) ||
      (ins.op().store_p()))
    if (ins.op().indexed_p())
      {label = "";
       generic = TRUE;}
    else
      {ppc_Rand* base = ins.rand(2);
       ppc_Rand* offset = ins.rand(3);

       label = "";
       generic = FALSE;

       if (offset->label_p())
 label = offset->label();
       else if (offset->snum_p())
 n_label = offset->snum();
       else if (offset->unum_p())
         n_label = offset->unum();
       else
 sigerr("addr::constructor: Illegal Offset.");

       reg = base->reg().actual();

     }
  else
     sigerr("addr::constructor: Not a Storage Command.");
};

// test objects are equality based on previous definition.

bool addr::operator==(const addr& address) const
{
// This is a best-case scenario; generic addresses are assumed
// to miss.

   if (generic || address.generic)
    return FALSE;
  else if (reg != address.reg)
    return FALSE;
  else if ((label.empty()) && (address.label.empty()))
    return ((n_label == address.n_label));
  else
    return((label == address.label));
};
```

```
// QPROF gqueue ADT -- sybok@athena

// The gqueue ADT is an abstraction representing a generalized queue of
// fixed length with head and tail pointers.  The gqueue is a container
// class which can be instatiated so as to contain an arbitrary class as
// queue elements.

// an improved modulus function

#ifndef NEWMOD
#define NEWMOD

inline int mod(int n, int m)
{ return ((n % m) + m) % m; };

#endif

#ifndef QPROF_GQUEUE
#define QPROF_GQUEUE

#include <iostream.h>
#include <String.h>

#include "DList.h"
#include "ppc_Inst.h"
#include "addr.h"
#include "fpdecode_entry.h"
#include "psq_entry.h"
#include "err.h"

// forward declarations for primitive display operations

void outshow(ppc_Inst* i);
void outshow(addr a);
void outshow(int i);
void outshow(String s);
void outshow(fpdecode_entry e);
void outshow(psq_entry e);

// the class gqueue

template <class T>
class gqueue
{
 public:

  gqueue(const int size = 1): sz(size)            // our constructor; the
    {head = 0; tail = 0; buf = new T[sz+1];};     // size is specified as a
                                                  // parameter
  int empty_p() {return (tail == head);};

  void load(DList<T>& fetched_list);        // load the queue
  void load_element(T ins);

  int room() {return (sz - size());};             // check how many vacancies
                                                  // it currently contains

  int size() const {return mod(tail-head,sz+1);}; // check how many entries
                                                  // it currentlyh contains

  T pop();                                        // pop an entry and return it
  T peek() const;                                 // just peek at it


  // gqueue higher-order operations and tests

  int contains(T ins) const;           // search for some elt.

  void apply_to_all(T (*remap)(T));   // apply a function to each elt.


  void apply_to_some(T target,        // apply a function to each
     T (*remap)(T));  // elt. matching "target"

  void apply_to_first(T (*remap)(T)); // apply function to first elt.


  void apply_to_last(T (*remap)(T));  // apply function to last elt.


  void print();

 protected:    // protected so as to allow subclasses to be defined

  T* buf;                  // the array storing the queue entry pointers

  int sz;                  // gqueue size
  int head, tail;          // gqueue head and tail pointers

  int succ(int j) const {return mod(j + 1,sz+1);};  // useful primitives
  int pred(int j) const {return mod(j - 1,sz+1);};

};

// load the gqueue from a DList of the contained class

template <class T>
inline void gqueue<T>::load(DList<T>& fetched_list)
{
  for(DListIter<T>& iter = *(fetched_list.iter());
```

```
    !iIter.end_p() ; iIter++)
    load_element(iIter.value());
};

// load the gqueue using a single pointer to an instance of the contained
// class

template <class T>
inline void gqueue<T>::load_element(T ins)
{
  if (room() > 0)
    {
      buf[tail] = ins;
      tail = succ(tail);
    }
  else
    sigerr("gqueue::load_element: No room left.");
};

// pop an elt. of the gqueue and return it

template <class T>
inline T gqueue<T>::pop()
{
  if (size() <= 0) sigerr("gqueue::pop: Can't pop empty queue.");

  T ip = buf[head];
  head = succ(head);
  return ip;

};

// just peek at the top elt. of the gqueue

template <class T>
inline T gqueue<T>::peek() const
{
  if (size() <= 0) sigerr("gqueue::peek: Can't peek at empty queue.");

  return(buf[head]);

};

// check whether the gqueue contains the specified target "ins".  Equally
// is determined using the == operator of the contained class.

template <class T>
inline bool gqueue<T>::contains(T ins) const
{int count = 0;

 for(int i = head; i != tail; i = succ(i))
   if (buf[i] == ins)
     count++;
  return count;
};

// apply a specified function to each element of the gqueue

template <class T>
inline void gqueue<T>::apply_to_all(T (*remap)(T))
{
 for(int i = head; i != tail; i = succ(i))
   buf[i] = (*remap)(buf[i]);
};

// apply a specified function to each element of the gqueue which matches
// the target "target".  Equally is determined using the == operator of the
// contained class.

template <class T>
inline void gqueue<T>::apply_to_some(T target, T (*remap)(T))
{
 for(int i = head; i != tail; i = succ(i))
   if (target == buf[i])
     buf[i] = (*remap)(buf[i]);
};

// apply a specified function to the first elt. of the gqueue

template <class T>
inline void gqueue<T>::apply_to_first(T (*remap)(T))
{
  if (head != tail)
    buf[head] = (*remap)(buf[head]);

};

// apply a specified function the last elt. of the gqueue

template <class T>
inline void gqueue<T>::apply_to_last(T (*remap)(T))
{
  if (head != tail)
    buf[pred(tail)] = (*remap)(buf[pred(tail)]);

};

// display the gqueue

template<class T>
inline void gqueue<T>::print()
{
  cout << "[GQUEUE: size = " << size() << ", room = " << room() << "]" << endl;

  for(int i = head; i != tail; i = succ(i))
    outshow(buf[i]);

  cout << endl;

};

#endif
```

```
// QPROF gqueue ADT -- sybok@athena

// The gqueue ADT is an abstraction representing a generalized queue of
// fixed length with head and tail pointers.  The gqueue is a container
// class which can be instatiated so as to contain an arbitrary class as
// queue elements.

#include "gqueue.h"

// overload the output primitive 'outshow'

void outshow(ppc_Inst* i) {i->print();};
void outshow(addr a) {a.print();};
void outshow(int i) {cout << i;};
void outshow(String s) {cout << s;};
void outshow(fpdecode_entry e) {cout << e;};
void outshow(psq_entry e) {cout << e;};
```

```
// QPROF grqueue ADT -- sybok@athena

// The grqueue ADT is a subclass of the grqueue data type, and extends it
// by including a release pointer which prevents the head of the queue
// from being popped until it (the release pointer) has been advanced by
// an appropriate amount.

// an improved modulus function

#ifndef NEWMOD
#define NEWMOD

inline int mod(int n, int m)
{ return ((n % m) + m) % m; };

#endif

#ifndef QPROF_GRQUEUE
#define QPROF_GRQUEUE

#include "gqueue.h"

// the class grqueue

template <class T>
class grqueue: public gqueue<T>
{

 public:
  grqueue(const int size = 1): gqueue<T>(size)  // construct a grqueue
    {rel = 0;};

  bool can_pop() const {return (head != rel);}; // can we pop the head?
  T pop();                                      // pop the head

  void release();                               // advance the release pointer
                                                // once click

  void print();


private:

  // here's the additional state of a grqueue above and beyond a gqueue

  int rel;                      // its release pointer
};

// pop the head of a grqueue

template <class T>
inline T grqueue<T>::pop()
{
  if (size() <= 0) sigerr("grqueue::pop: Can't pop empty queue.");

  if (rel == head) sigerr("grqueue::pop: Can't pop unreleased item.");

  T ip = buf[head];
  head = succ(head);
  return ip;

};

// advance the release pointer of the grqueue

template <class T>
inline void grqueue<T>::release()
{
  if (rel == tail)
    sigerr("grqueue::release: Too many releases.");
  else
    rel = succ(rel);
};

// display the grqueue

template<class T>
inline void grqueue<T>::print()
{ int r_elt = 0;

  for(int i = head; i != rel; i = succ(i))   // count number of released elts.
    r_elt++;

  cout << "[GRQUEUE: size = " << size() << ", room = " << room();
  cout << ", rel. elts. = " << r_elt << "]" << endl;

  for(i = head; i != tail; i = succ(i))
    outshow(buf[i]);

  cout << endl;
};


#endif
```

```
// QPROF psq_entry ADT -- sybok@athena

// The psq_entry ADT represents entries in the FPU pending store queue--a
// target register and a "give back" bit.  (It actually stores a pointer
// to the original store instruction as well.)

#ifndef QPROF_PSQ_ENTRY
#define QPROF_PSQ_ENTRY

#include <iostream.h>
#include <String.h>

#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"
#include "ppc_Rand.h"
#include "ppc_Reg.h"

#include "err.h"

const ILLEGAL_TARGET = -1;

class psq_entry
{
 public:
  psq_entry(ppc_Inst* st = 0, int t = ILLEGAL_TARGET);  // constructor

  int get_target() const {return target;};      // return target reg.

  ppc_Inst* get_ins() const {return store;};    // return ptr. to store ins.

  bool give_back_p() const {return give_back;}; // return "give back" bit

  bool operator==(psq_entry e)                  // equality definition
    {return (target == e.target);};

  friend psq_entry set_gb(psq_entry e);         // define an operation to
                                                // yield a new psq_entry
                                                // with a modified "give back"
                                                // bit

  friend ostream& operator << (ostream &s, const psq_entry& psq_obj);

  void print();

 private:

  int target;
  ppc_Inst* store;
  bool give_back;

};

// construct the initial psq_entry (with a zero "give back" bit)

inline psq_entry::psq_entry(ppc_Inst* st, int t)
{
  store = st;

  if (st != 0)
    target = st->rand(1)->reg().actual();
  else
    target = t;

  give_back = 0;
};

// yield a new psq_entry with a set "give back" bit

inline psq_entry set_gb(psq_entry e)
{
  if (e.give_back)
    sigerr("psq_entgry::set_gb: Already set.");

  psq_entry* new_entry = new psq_entry(e);
  new_entry->give_back = TRUE;
  return(*new_entry);
};


// handle output of a psq_entry object

inline ostream& operator << (ostream &s, const psq_entry& psq_obj)
{
  cout << "[PSQ_ENTRY: " << *psq_obj.store;
  cout << ", GB = " << psq_obj.give_back << "]" << endl;

  return s;
};

// handle printing a psq_entry object

inline void psq_entry::print()
{
  cout << "[PSQ_ENTRY: " << *store << ", GB = " << give_back << "]" << endl;

};

#endif
```

Appendix B

Appendix: Runtime System Profiling Support

```
/* Alejandro Caro
 *
 * RTS for Q (PowerPC): statistics support
 *
 */


#ifndef RTSSTATS_H
#define RTSSTATS_H 1

#include "Qparams.h"

/* Number of slots used for statistics in a frames and CBDs. */

#define MAX_BB_PER_CB           30
#define MAX_CALLED_FN_PER_CB    6
#define MAX_CALLED_CB_PER_CB    6

#define CBD_ACCUMULATORS         5+3*MAX_CALLED_CB_PER_CB

#define FRAME_STATS_SLOTS  (4*MAX_BB_PER_CB+ 5*MAX_CALLED_FN_PER_CB+2)
#define CBD_STATS_SLOTS    (FRAME_STATS_SLOTS+CBD_ACCUMULATORS+1)

void getFrameStats( int pe, IdWord* fp );

void getProgStats(void);

void printProgStats(int verbose);

void settime(IdWord* cbd_ptr, long offset);
void accumtime(IdWord* cbd_ptr, long offset);

double expt(double x,double y);
double convert_time(unsigned long highticks, unsigned long lowtick);

#endif

/* N.B.  Since the CBD space for stats is fixed and is a superset of the
   local frame stats, room must be allocated for a worst case.  Thus, some
   maximum number of BBs per CB must be assumed.

   This is not too bad in reality, since the number of BBs per CB is
   a static feature of a program, and thus if a particular program
   requires more than the currently allowed number, that number can
   be raised to allow that program to be profiled.
*/
```

```
/* Alejandro Caro
 *
 * RTS for Q (PowerPC): statistics support
 *
 */

#include <stdio.h>
#include <math.h>

#include "rtsstats.h"

const long CPU_CLOCK_RATE = 42e6;                       /* hertz */

const long RTC_TICKS_PER_SECOND = 1e9;                  /* hertz */

const double IDWORD_SCALE_FACTOR = 4294967296.0;        /* 2^32  */

/* Pointer to start of CBD area.  Defined in assembly routines. */

extern IdWord *CBD_Area;



/*************************************************************************
 * getFrameStats
 *
 * This routine is invoked every time a frame is deallocated.  It
 * traverses the frame and gathers information from the stats slots
 * in the frame into the CBD for the code block.
 */

void
getFrameStats( int pe, IdWord *fp )
{ IdWord* cbd_ptr;
  IdWord* parent_cbd_ptr;
  int no_of_inlets, cbd_stat_start, frame_stat_start;
  int parent_no_of_inlets;

  int i, id, base;
  long high_word, low_word;


  cbd_ptr = (IdWord*) fp[2];
  parent_cbd_ptr = (IdWord*) ((IdWord*) fp[1])[2];

  if ((int) cbd_ptr[CBD_STATS] > 0)
    {no_of_inlets = cbd_ptr[CBD_INLET_COUNT];
     cbd_stat_start = no_of_inlets+5;

     frame_stat_start = cbd_ptr[CBD_STATS];

     low_word =  cbd_ptr[cbd_stat_start+3];
     high_word = cbd_ptr[cbd_stat_start+4];

     cbd_ptr[cbd_stat_start+1] = fp[frame_stat_start+FRAME_STATS_SLOTS-2];
     cbd_ptr[cbd_stat_start+2] = fp[frame_stat_start+FRAME_STATS_SLOTS-1];

     accumtime((IdWord*)cbd_ptr,(cbd_stat_start+1)*4);

     id = cbd_ptr[cbd_stat_start+CBD_STATS_SLOTS-1];

     (cbd_ptr[cbd_stat_start])++;

     for(i = 0; i < FRAME_STATS_SLOTS-2; i++)
       cbd_ptr[cbd_stat_start+5+i] += fp[i+frame_stat_start];

     if ((int) parent_cbd_ptr[CBD_STATS] > 0)
       {parent_no_of_inlets = parent_cbd_ptr[CBD_INLET_COUNT];
        base = parent_no_of_inlets+5+5+FRAME_STATS_SLOTS+3*id;

(parent_cbd_ptr[base])++;

parent_cbd_ptr[base+1] += cbd_ptr[cbd_stat_start+3]-low_word;
parent_cbd_ptr[base+2] += cbd_ptr[cbd_stat_start+4]-high_word;
      };

   };

};



/*************************************************************************
 * getProgStats
 *
 * This routine is invoked when a program terminates. It gathers stats
 * from all code blocks on all processors.
 */

void
getProgStats(void)
{int i;
 int no_of_inlets, cbd_stat_start, frame_stat_start, fn_data_start;
 int cb_data_start;
 double cb_live_time, bb_live_time, fn_live_time;
 int cb_count = 0;
 char codeblockname[20];

 IdWord *currCBD = CBD_Area;
```

```
FILE *in_file, *out_file;

char begin_token[10];

int bb_count, fn_count;

in_file =
  fopen("/home/jj/sybok/Profiler/Static_measures/002/qstat_short","r");

out_file =
  fopen("/home/jj/sybok/Profiler/Static_measures/002/qresults","w");

fscanf(in_file,"%s",begin_token);

 while ( CBD_p( currCBD ) )
   if ((int) currCBD[CBD_STATS] > 0)
     {currCBD += 5 + currCBD[CBD_INLET_COUNT] + CBD_STATS_SLOTS;
      cb_count++;}
   else
     currCBD += 5 + currCBD[CBD_INLET_COUNT];

currCBD = CBD_Area;

 while ( CBD_p( currCBD ) )
   if ((int) currCBD[CBD_STATS] > 0)
     {/* Init thread function descriptor. */
IdWord* FD = (IdWord* ) currCBD[CBD_INIT_CODE];

fscanf(in_file,"%d %d",&bb_count,&fn_count);

for(i = 0; i < cb_count; i++)
  fscanf(in_file,"%s",codeblockname);

fprintf(out_file,"%s\n", currCBD[CBD_NAME]);

fprintf(out_file,"%d\n", currCBD[CBD_STATS]);

no_of_inlets = currCBD[CBD_INLET_COUNT];
cbd_stat_start = no_of_inlets+5;

fprintf(out_file,"%d\n",currCBD[cbd_stat_start]);

cb_live_time = convert_time(currCBD[cbd_stat_start+4],
    currCBD[cbd_stat_start+3]);

fprintf(out_file,"%.1f\n",cb_live_time);

for(i = 0; i < bb_count; i++)
  {fprintf(out_file,"%d\n",(int) currCBD[cbd_stat_start+5+i*4]);
   fprintf(out_file,"%d\n",(int) currCBD[cbd_stat_start+6+i*4]);

   bb_live_time = convert_time(currCBD[cbd_stat_start+8+i*4],
       currCBD[cbd_stat_start+7+i*4]);

   fprintf(out_file,"%.1f\n",bb_live_time);
    };

fn_data_start = cbd_stat_start+5+MAX_BB_PER_CB*4;

for(i = 0; i < fn_count; i ++)
  {
    fprintf(out_file,"%d\n",currCBD[fn_data_start+5*i]);
          fn_live_time = convert_time(currCBD[fn_data_start+5*i+4],
currCBD[fn_data_start+5*i+3]);
    fprintf(out_file,"%.1f\n",fn_live_time);
    };


cb_data_start = fn_data_start+MAX_CALLED_FN_PER_CB*5+2;

  for(i = 0; i < cb_count; i++)
    {
      fprintf(out_file,"%d\n",currCBD[cb_data_start+3*i]);
      cb_live_time = convert_time(currCBD[cb_data_start+3*i+2],
  currCBD[cb_data_start+3*i+1]);
      fprintf(out_file,"%.1f\n",cb_live_time);
    };

  currCBD += 5 + currCBD[CBD_INLET_COUNT] + CBD_STATS_SLOTS;
      }
    else
      currCBD += 5 + currCBD[CBD_INLET_COUNT];


 fclose(out_file);
 fclose(in_file);

};



/*****************************************************************************
 * CBD_p     @@@ This is temporary code.
 *
 * Tests if a pointer is likely to be a pointer to a CBD.
 * The first word should point to something like a function descriptor.
 * The second word should be an integer that is not too large.
 */

#define CODE_SEG 0x10000000
#define DATA_SEG 0x20000000
#define HEAP_SEG 0x30000000

int CBD_p( IdWord *CBDptr )
{
  unsigned long *FD = (unsigned long *) CBDptr[0];
  int FSize = (int) CBDptr[1];
  int Stats = (int) CBDptr[3];
  int Ninlets = (int) CBDptr[4];


  return  (   ( FD[0] > CODE_SEG ) && ( FD[0] < DATA_SEG )
   && ( FD[1] > DATA_SEG ) && ( FD[1] < HEAP_SEG )
   && ( FSize > 0 ) && ( FSize < 1000 )
   && ( Stats < FSize )
   && ( Ninlets > 0 ) && ( Ninlets < 1000 ) );

}

/*****************************************************************************
 * printProgStats
 *
 * This routine is used to print statistics about a program.
 */

void
printProgStats(int verbose)
{ int i;
  int no_of_inlets, cbd_stat_start, frame_stat_start, fn_data_start;
  int cb_data_start;
  double cb_live_time, bb_live_time, fn_live_time;

  IdWord *currCBD = CBD_Area;

  if (verbose)
    {

  printf("\n\n");
  printf("CBD Report\n");
  printf("----------\n\n");


  while ( CBD_p( currCBD ) )
    {/* Init thread function descriptor. */
      IdWord* FD = (IdWord* ) currCBD[CBD_INIT_CODE];

      printf("%s\n", currCBD[CBD_NAME] );

      /* Name of init thread. */
      printf("\tInit Thread:      %s\n", (abs(FD[2])));

      /* Frame Size */
      printf("\tFrame Size:       %d\n", currCBD[CBD_FRAME_SIZE] );

      /* Stats location in frame */
      printf("\tStats Frame Loc.: %d\n", currCBD[CBD_STATS] );

      /* Number of inlets */
      printf("\tInlets:           %d\n\n", currCBD[CBD_INLET_COUNT] );

      no_of_inlets = currCBD[CBD_INLET_COUNT];
      cbd_stat_start = no_of_inlets+5;

      if ((int) currCBD[CBD_STATS] > 0)
{
  printf("\tInvocations:   %d\n",currCBD[cbd_stat_start]);

  cb_live_time = convert_time(currCBD[cbd_stat_start+4],
      currCBD[cbd_stat_start+3]);

  printf("\tLive time:     %.0f\n",cb_live_time);

  printf("\tBASIC BLOCK STATISTICS\n");

  for(i = 0; i < MAX_BB_PER_CB; i++)
    { if ((currCBD[cbd_stat_start+5+i*4] == 0) &&
  (currCBD[cbd_stat_start+6+i*4] == 0))
continue;
      printf("\n\t\tBasic Block #%d\n",i+1);
      printf("\t\tInvocations:\n");
      printf("\t\t\t (fall thru) %d\n",
     (int) currCBD[cbd_stat_start+5+i*4]);
      printf("\t\t\t (branch)    %d\n",
     (int) currCBD[cbd_stat_start+6+i*4]);

      bb_live_time = convert_time(currCBD[cbd_stat_start+8+i*4],
  currCBD[cbd_stat_start+7+i*4]);

      printf("\t\tLive time:     %.0f\n",bb_live_time);
    };

  fn_data_start = cbd_stat_start+5+MAX_BB_PER_CB*4;

  for(i = 0; i < MAX_CALLED_FN_PER_CB; i++)
    {  if (currCBD[fn_data_start+5*i] == 0)
continue;

      printf("\n\t\t Called Function #%d\n",i+1);
```

```
      printf("\t\tInvocations: %d\n",currCBD[fn_data_start+5*i]);

      fn_live_time = convert_time(currCBD[fn_data_start+5*i+4],
  currCBD[fn_data_start+5*i+3]);

      printf("\t\tLive time:     %.0f\n",fn_live_time);
    };

  cb_data_start = fn_data_start+MAX_CALLED_FN_PER_CB*5+2;

  for(i = 0; i < MAX_CALLED_CB_PER_CB; i++)
    {  if (currCBD[cb_data_start+3*i] == 0)
continue;

      printf("\n\t\t Called Id Procedure #%d\n",i+1);
      printf("\t\tInvocations: %d\n",currCBD[cb_data_start+3*i]);

      cb_live_time = convert_time(currCBD[cb_data_start+3*i+2],
  currCBD[cb_data_start+3*i+1]);

      printf("\t\tLive time:     %.0f\n",cb_live_time);
    };

  currCBD += 5 + currCBD[CBD_INLET_COUNT] + CBD_STATS_SLOTS;
}
      else
currCBD += 5 + currCBD[CBD_INLET_COUNT];


    printf("--\n");

    };
  };
};

double expt(double x,double y)  {return exp(log(x)*y);};

double convert_time(unsigned long highticks, unsigned long lowticks)
{
  double totalticks = highticks*IDWORD_SCALE_FACTOR+lowticks;
  return((totalticks/RTC_TICKS_PER_SECOND)*CPU_CLOCK_RATE);
};
```

```
/* Alejandro Caro
 *
 * RTS for Q (PowerPC): frame support
 *
 */

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


#include "Qparams.h"

#include "rtscommon.h"
#include "rtsmsgstack.h"
#include "rtsstats.h"
#include "rtsframe.h"


/*****************************************************************************
 * Internal module definitions.
 */

/* Number of extra slots used in a frame by the RTS.  They are actuaaly
 * allocted from the preceding frame!
 */
#define FRAME_INFO_IDWORDS 1
#define FRAME_INFO_SIZE_OFF -1

/* Pointer to the next available frame. */
static IdWord *frame_available;

/* Start of frame area. */
static IdWord *frame_area_start;

/* End of frame area. */
static IdWord *frame_area_end;

/* The external interfaces simply send a message to one of these
 * handlers.
 */
void alloc_frame_eager_handler(void);

/* The actual allocator of frames. */
IdWord *alloc_frame_internal(int nidwords);


void frame_report(void);

static IdWord msg_buffer[10];

static void init_frame_handlers(void);

/*****************************************************************************
 * Simple Frame Allocator
 *
 * This frame allocator simply allocates. It never deallocates, so
 * frame usage will keep increasing as the program runs.  Each frame
 * in the heap has several header words, which are "invisible" to the
 * user of the frame.
 */


/*****************************************************************************
 * init_frame_area
 *
 * The RTS allocated the range [start, end] for the heap. These pointers
 * are passed to this routine in case the frame manager needs to initialize.
 * start:  guaranteed aligned on page boundary
 * end:    guaranteed to point to end of a page
 */


void init_frame_area(void *start, void *end)
{

  /* The following aligns the start pointer on a frame alignment
   * boundary, leaving enough space for the FRAME_INFO_IDWORDS before the
   * first frame.
   */

  void *frameBase = start + (FRAME_INFO_IDWORDS * sizeof(IdWord));

  frame_area_start =
    (IdWord *) align_ptr_next(frameBase, FRAME_ALIGNMENT_BYTES);
  frame_available = frame_area_start;
  frame_area_end =
    (IdWord *) align_ptr_prev(end - sizeof(IdWord), sizeof(IdWord));

  /* Check for bad bounds on frame area. */
  if (frame_area_start >= frame_area_end)
    rts_error_msg( "(init_frame_area): bad frame area bounds." );

  /* @@@ This is a huge hack.  I'm just trying it out! */
  init_frame_handlers();

}
```

```
/*****************************************************************************
 * alloc_frame_local
 *
 * Allocate a local frame.
 */

IdWord *alloc_frame_local(IdWord *cbd_ptr, IdWord retcont1, IdWord retcont2)
{ long frame_stat_start = cbd_ptr[CBD_STATS];
  int no_of_inlets = cbd_ptr[CBD_INLET_COUNT];
  int cbd_stat_start = no_of_inlets+5;

  int size = (int) cbd_ptr[CBD_FRAME_SIZE];
  FrameInitializer fi = (FrameInitializer) cbd_ptr[CBD_INIT_CODE];

  int i;
  IdWord *frame;
  frame = alloc_frame_internal(size);

  if (frame_stat_start > 0)
    settime((IdWord*) frame, (frame_stat_start+FRAME_STATS_SLOTS-2)*4);

  /* RTS initialization of the frame:
   * 0: retcont.ip
   * 1: retcont.fp
   * 2: cbd_ptr
   */
  frame[0] = retcont1; frame[1] = retcont2;
  frame[2] = (IdWord) cbd_ptr;

  /* initial frame stat slots to 0 */
    if (frame_stat_start > 0)
    for(i = frame_stat_start+FRAME_STATS_SLOTS-3; i >= frame_stat_start; i--)
      frame[i] = 0;

  /* Initialize the frame to the proper values. */
  (*fi)(frame, retcont1, retcont2);

  /* Return the frame pointer to the caller. */
  return frame;
}


/*****************************************************************************
 * alloc_frame_eager
 *
 * Sends a message to alloc_frame_eager_handler.  This "two-stage"
 * approach allows the code to return very quickly to the calling thread.
 *
 * @@@NOTE: This has been optimized for uniprocessors.
 */

void alloc_frame_eager(IdWord *CBD, IdWord retcontip, IdWord retcontfp,
       IdWord reqcontip, IdWord reqcontfp)
{
  /* Send a message to the routine that actually handles the
   * heap allocation.
   * The message parameters are:
   * - 5 pieces of data
   * - pe zero (this is actually ignored)
   * - target IP is alloc_frame_eager
   * - target FP (ignored in this case)
   * - data...
   */
  msg_send( 5, 0, (IdWord) alloc_frame_eager_handler, 0,
   CBD, retcontip, retcontfp, reqcontip, reqcontfp );

}




/*****************************************************************************
 * alloc_frame_eager_handler
 *
 * Split-phase, eager allocation of frames on any processor.
 * Expects a message with the following format:
 * 0: cbd_ptr
 * 1: retcont_ip
 * 2: retcont_fp
 * 3: reqcont_ip
 * 4: reqcont_fp
 *
 * returns a message with the following data:
 * 0: frame pe
 * 1: frame ptr
 */

void alloc_frame_eager_handler()
{ long frame_stat_start;
  int  no_of_inlets;
  int cbd_stat_start;

  int i;

/* Get message payload */
  IdWord *cbd_ptr;
  IdWord retcont_ip;
  IdWord retcont_fp;
  IdWord reqcont_ip;
  IdWord reqcont_fp;
  int size;
  FrameInitializer fi;
  IdWord *frame;

  msg_recv( 5, msg_buffer );

  cbd_ptr = (IdWord *) msg_buffer[0];
  retcont_ip = msg_buffer[1];
  retcont_fp = msg_buffer[2];
  reqcont_ip = msg_buffer[3];
  reqcont_fp = msg_buffer[4];

  frame_stat_start = cbd_ptr[CBD_STATS];

  no_of_inlets = cbd_ptr[CBD_INLET_COUNT];
  cbd_stat_start = no_of_inlets+5;


  size = (int) cbd_ptr[CBD_FRAME_SIZE];
  fi = (FrameInitializer) cbd_ptr[CBD_INIT_CODE];
  frame = alloc_frame_internal(size);

  if (frame_stat_start > 0)
    settime((IdWord*) frame, (frame_stat_start+FRAME_STATS_SLOTS-2)*4);

  /* RTS initialization of the frame:
   * 0: retcont.ip
   * 1: retcont.fp
   * 2: cbd_ptr
   */
  frame[0] = retcont_ip; frame[1] = retcont_fp;
  frame[2] = (IdWord) cbd_ptr;

  /* initial frame stat slots to 0 */

  if (frame_stat_start > 0)
    for(i = frame_stat_start+FRAME_STATS_SLOTS-3; i >= frame_stat_start; i--)
      frame[i] = 0;

  /* Initialize the frame */
  (*fi)(frame, retcont_ip, retcont_fp);


  /* Construct message to request continuation.
   *
   * @@@ NOTE: This code is optimized for uniprocessors
   * since it ignores the PE field.
   */

  { IdWord ip = CONT_IP( reqcont_ip, reqcont_fp );
    IdWord fp = CONT_FP( reqcont_ip, reqcont_fp );

    /* For now, everything is on PE0. */
    msg_send(2, 0, ip, fp, 0, (IdWord) frame);
  }

}


/*****************************************************************************
 * dealloc_frame_at
 *
 * Deallocates a remote frame.  The implementation of this procedure is
 * currently a stub.  It simply negates the value of the frame size to
 * indicate that it has been deallocated.
 *
 * @@@NOTE: This procedure is only for uniprocessors.
 */

void dealloc_frame_at(int pe, IdWord *fp)
{
  int frameSize = *(((int *) fp) - 1);

  /* Check if the frame has been deallocated. */

  if (frameSize < 0)
    rts_error_msg( "(dealloc_frame_at): frame at [%p] already deallocated.",
  (void *) fp );



  /* Gather statistics from the frame. */

  getFrameStats( pe, fp );



  /* "Deallocate" the frame. */

  *(((int *) fp) - 1) = -frameSize;

}


/*****************************************************************************
 * Internal Procedures
 *****************************************************************************/

/*****************************************************************************
 * alloc_frame_internal
```

```
 *
 * This functions allocates a frame from the frame area and returns a
 * pointer to the beginning of the frame.
 */


IdWord *alloc_frame_internal(int nidwords)
{
  div_t frameInfo = div(nidwords, FRAME_ALIGNMENT_IDWORDS );
  int frameSize;
  IdWord *frame;

  /* The following frame (in memory) writes its internal information
   * at the end of this frame.  Therefore, we have to leave enough room
   * for that information.  The following code checks that we have enough
   * room, or otherwise, it increases the size of the frame by one allocation
   * unit (64 bytes or FRAME_ALIGNMENT_BYTES).
   */
  if (frameInfo.rem < (FRAME_ALIGNMENT_IDWORDS - FRAME_INFO_IDWORDS))
    frameSize = (frameInfo.quot + 1) * FRAME_ALIGNMENT_IDWORDS;
  else
    frameSize = (frameInfo.quot + 2) * FRAME_ALIGNMENT_IDWORDS;

  /* Save the current frame_available pointer as the return value, and
   * increment it to point to the next available frame.
   */
  frame = frame_available;
  frame_available = frame_available + frameSize;

  if (frame_available >= frame_area_end)
    rts_error_msg( "(alloc_frame_internal): ran out of frames." );

  /* Write the frameSize info into the frame.
   */
  *(((int *) frame) + FRAME_INFO_SIZE_OFF) = frameSize;

  return frame;

}


/*****************************************************************************
 * frame_report
 *
 * Prints a report about all the frames in the system, beginning at the
 * start of the frame area.
 */

void frame_report(void)
{

  IdWord *currentFrame = frame_area_start;
  int frameSize = (int) *(currentFrame-1);

  if (currentFrame < frame_available)
    { printf("Frame Report\n");
      printf("Address\t\tSize (idwords)\tStatus\n");
      printf("---------------------------------------\n");
    }
  else
    printf("No frame allocations/deallocations have occurred.\n");

  while ( currentFrame < frame_available)
    {
      if (frameSize < 0)
{ frameSize = -frameSize;
  printf("0x%p\t\t%d\t\tD\n", currentFrame, frameSize);
}
      else
printf("0x%p\t\t%d\t\tA\n", currentFrame, frameSize);

      currentFrame = currentFrame + frameSize ;
      frameSize = (int) *(currentFrame-1);

    }

}



/*****************************************************************************
 *@@@ init_frame_handlers
 *
 * This is a huge hack.  It writes a string into the last position
 * of the function descriptor for each frame handler.  This will
 * allow the RTS to print the name of the handler when it is invoked.
 */

static char alloc_frame_eager_handler_name[] = "alloc_frame_eager_handler";
static void init_frame_handler( int*, char* );

static void
init_frame_handlers(void)
{

  init_frame_handler( (int*) alloc_frame_eager_handler,
      alloc_frame_eager_handler_name );


}

/* @@@ huge hack continued! */
void
init_frame_handler( int* desc, char* name )
{
  *((char**)(desc+2)) = name ;
}


/*****************************************************************************
 * Unimplemented.
 */


/* Returns fp in a message to reqcont. */
void alloc_frame_lazy_local(IdWord *cbd_ptr, IdWord retcont1, IdWord retcont2,
    IdWord reqcont1, IdWord reqcont2)
{
  rts_error_msg( "(alloc_frame_lazy_local): not implemented." );
}


/* Returns fp in message to reqcont. */
void alloc_frame_lazy_at(IdWord *cbd_ptr, IdWord retcont1, IdWord retcont2,
 int pe, IdWord reqcont1, IdWord reqcont2)
{
  rts_error_msg( "(alloc_frame_lazy_at): not implemented." );
}


/* Returns fp in message to reqcont. */
void alloc_frame_eager_at(IdWord *cbd_ptr, IdWord retcont1, IdWord retcont2,
  int pe, IdWord reqcont1, IdWord reqcont2)
{
  rts_error_msg( "(alloc_frame_eager_at): not implemented." );
}

/* Returns fp and pe in message to reqcont. */
void alloc_frame_lazy(IdWord *cbd_ptr, IdWord retcont1, IdWord retcont2,
      IdWord reqcont1, IdWord reqcont2)
{
  rts_error_msg( "(alloc_frame_lazy): not implemented." );
}
```

```
/* Alejandro Caro
 *
 *
 * RTS for Q (PowerPC): main function.
 *
 *
 */


#include <stdio.h>
#include <errno.h>

#include "rtsinit.h"
#include "rtsdispatch.h"
#include "rtssighandlers.h"
#include "rtsmsgstack.h"
#include "rtsschedstack.h"
#include "rtsstats.h"

/*****************************************************************************
 * Declarations
 */


/* The return value of the Id program is stored here by the boot code. */
IdWord valueId;
IdWord mainArg;



/*****************************************************************************
 * main
 *
 * main is the standard function called to start a C program.
 * In the RTS, main performs initialization, and then calls the
 * rts_main_dispatch_loop to begin execution of the Id program.
 *
 * By default, main returns the value returned by the Id version of
 * "main", as an integer.
 *
 */

inline void ran_pause(void);
int str_to_num(char* str);
long delay(long);

const SEED = 2384901;

int main (int argc, char *argv[])
{ int i, j, count;
  int verbose;

  srandom(SEED);

  if (argc > 1)
    count = str_to_num(argv[1]);
  else
    count = 100;

  if (argc > 2)
    verbose = TRUE;
  else
    verbose = FALSE;

  /* Initialize the RTS.  This sets up the different memory areas and
   * internal data structures used by the Run Time System and the Id program.
   */
  rts_init();

  /* Establish longjmp target for exceptions, and jump
   * to exception handler if necessary.
   */
  { int statCode = setjmp(signalExitContext);

    if ( statCode )
      rts_generic_signal_handler(statCode);
  }

  /* Create input vector for Id program.  When the Id program
   * is booted by the dispatch loop, this value will be
   * passed as argument to the Id version of "main"
   */
  mainArg = (IdWord) rts_create_Id_argv(argc, argv);

  /* Start the dispatch loop, which cause the Id version of "main"
   * to be invoked.
   */
  { int statCode;

  /* Returns non-zero if there is an error. */
    for(i = 0; i < count; i++)
      {statCode = rts_main_dispatch_loop();
       ran_pause();};

    if (statCode)
      rts_main_dispatch_loop_error(statCode);
    else
      printf("Normal Program Exit.  Value: %d (0x%x).\n", valueId, valueId);
  }

  printProgStats(verbose);

  /* This procedure gathers program-wide statistics information. */
  getProgStats();

  return valueId;

}


inline void ran_pause(void) {

  int num = random() % 11;

  delay(num);

};


int str_to_num(char* str)
{
  int num = 0;
  int index = 0;

  while (str[index] != '\0')
    num = 10*num+((int) str[index++]-'0');

  return num;

};
```

Appendix C

Appendix: The Instrumentor, the Extended Simulator, and ppc-testparse.

```
//QPROF instrument module

#include "ppc_Mod.h"

#ifndef INSTRUMENT_H
#define INSTRUMENT_H

ppc_Mod& instrument(ppc_Mod& mod);

#endif
```

```
// QPROF instrument module
#include <iostream.h>
#include <fstream.h>


#include "instrument.h"
#include "instrument_CB.h"
#include "ICache.h"

#include "i_mix.h"

ppc_Mod& instrument(ppc_Mod& mod)

{ICache* icp = new ICache(mod);

 ppc_Mod* new_mod = new ppc_Mod(mod.name());
 for(DListIter<sT_Inst*> iter(mod.directives()); !iter.end_p(); iter++)
   new_mod->appendDirective(iter.value());

 int i = 1;


 int count = 0;

 int cb_count = 0;

 ofstream f_out("/home/jj/sybok/Profiler/Static_measures/002/qstat_short");

 if (f_out.fail())
   sigerr("make_measure: Error in writing statics measure file.");


 for(ppc_ProcIter iPr(mod.procIter()); !iPr.end_p();
     iPr++)
   for(ppc_CBIter iCdB(iPr.value()->CBIter());
       !iCdB.end_p() ; iCdB++)
     cb_count++;

 f_out << cb_count << endl;

 f_out.close();

 for(ppc_ProcIter iProc(mod.procIter()); !iProc.end_p(); iProc++, i++)
   {int j = 1;

    ppc_Proc* new_proc =
      new ppc_Proc(iProc.value()->name(),iProc.value()->type());

    for(ppc_CBIter iCB(iProc.value()->CBIter()); !iCB.end_p() ; iCB++, j++)
{instrument_CB instrCB(i,j,*iCB.value(),icp);
instrCB.new_CB().id(count++);
new_proc->append(&instrCB.new_CB());

ofstream
  f_out("/home/jj/sybok/Profiler/Static_measures/002/qstat_short",
      ios::app);

 for(ppc_ProcIter iPr2(mod.procIter()); !iPr2.end_p();
     iPr2++)
   for(ppc_CBIter iCdB2(iPr2.value()->CBIter());
       !iCdB2.end_p() ; iCdB2++)
     f_out << iCdB2.value()->name() << " ";

 f_out << endl;
 f_out.close();};

    new_mod->append(new_proc);
  };

 return(*new_mod);
};
```

```
// QPROF instrumentor ADT

#ifndef INSTRUMENT_CB_H
#define INSTRUMENT_CB_H 1

#include "ICache.h"
#include "ppc_Reg.h"
#include "QVHMap.h"

ppc_Reg RTCL = *(new ppc_Reg(5,SPR_RTYPE,5));
ppc_Reg RTCU = *(new ppc_Reg(4,SPR_RTYPE,4));

const INSTR_REGS = 6;

class instrument_CB{

 public:

  instrument_CB(int i, int j, ppc_CB& cb, ICache* ic);

  ppc_CB& new_CB() {return *new_cb;};

 private:

  void build_header(DList<ppc_Inst*>& bb, int k, int l,
    const String& bb_label,
    int bb_id, int reg_idx_start, PartType ptype);

  DList<ppc_Inst*>& make_fall_thr_ins(int first_reg, String& join_pt);
  DList<ppc_Inst*>& make_jump_entry_ins(int first_reg, const String& labl);
  DList<ppc_Inst*>& make_common_ins(int first_reg, const String& join_pt);
  DList<ppc_Inst*>& make_cold_entry(int first_reg, String& c_entry,
    String& new_label, int id, PartType ptype);

  DList<ppc_Inst*>& generate_bootstrap(const String& init_label, int k,
       int start_reg, bool cthread_p);

  ppc_Inst* filter(ppc_Inst* orig_ins, const String& target,
   DList<ppc_Inst*>* suffix, int k, int l,
   int start_reg, int& ccount, PartType ptype);


  String& generate_next_anon();

  int i,j;     // procedure and code block numbers

  ppc_CB* new_cb;

  ICache* i_cache;

  int anon_id_count;

  QVHMap<String,int>* registry;


};



#endif
#include <iostream.h>
#include <fstream.h>

#include "instrument_CB.h"
#include "bbnode.h"
#include "rtsstats.h"


 instrument_CB::instrument_CB(int inum, int jnum, ppc_CB& old_cb, ICache* ic):
  i(inum), j(jnum)
{ anon_id_count = 0;

  new_cb = new ppc_CB(old_cb.name(),old_cb.frameSize());
  new_cb->statsOffset(old_cb.statsOffset());
  new_cb->init(old_cb.init());
  new_cb->inlets(old_cb.inlets());

  int count = 0;
  int ccount = 0;

  registry = new QVHMap<String,int>(-1,MAX_CALLED_FN_PER_CB);

  int k = 1;

  int l;

  i_cache = ic;

  const String* first_BB = 0;

  for(ppc_PartIter iPart(old_cb.partIter()); !iPart.end_p();iPart++, k++)
    {ppc_Part& old_part = *iPart.value();
     int start_reg = old_part.regidx();
     int l = 0;

     String test("MAIN.part0");

     if (iPart.value()->data_p())
       {new_cb->append(&old_part);
continue;};

     ppc_Part* new_part = new ppc_Part(old_part,6);
     DList<ppc_Inst*>* new_part_suffix = new DList<ppc_Inst*>;

     bool start_BB = TRUE;
     const String* last_label = 0;
     DList<ppc_Inst*>* new_bb = new DList<ppc_Inst*>;

     String* first_label
       = new String("first."+num_to_str(i)+"."+num_to_str(j)+"."+
                    num_to_str(k)+".1");

     for(ppc_InstIter iInst(old_part.instIter());!iInst.end_p(); )
       {ppc_Inst* ins= new ppc_Inst(*iInst.value());

        if (ins->label().empty() && (ins->op().op() == ppc_none))
          {iInst++; delete ins; continue;};

        if (start_BB)
          {start_BB = FALSE;
           if (ins->label().empty())
    {String* lbl = new String(generate_next_anon());
     iInst.value()->label(*lbl);
     ins->label(*lbl);};

    if (l > 1)
      {build_header(*new_bb,k,l,*last_label,
    count++,start_reg,new_part->type());
       new_part->append(*new_bb);}
    else if (l == 1)
      {first_BB = last_label;
       build_header(*new_bb,k,l,*first_label,
    count++,start_reg,new_part->type());
       new_part->append(*new_bb);};

    ins->label("");
    last_label = &iInst.value()->label();

    new_bb = new DList<ppc_Inst*>;


          if (l > 0)
            new_bb->append(filter(ins,*first_BB,new_part_suffix,
k,l,start_reg, ccount, new_part->type()));
          else
            new_bb->append(filter(ins,*last_label,new_part_suffix,
k,l,start_reg, ccount, new_part->type()));

          if (!ins->op().inert_p())
            {start_BB = TRUE; l++;};

          iInst++;

          while (!iInst.end_p() && iInst.value()->label().empty() &&
                 (iInst.value()->op().op() == ppc_none))
             {iInst++;};

          if (iInst.end_p() ||
              ((!iInst.end_p()) && (!iInst.value()->label().empty())))
            if (!start_BB)
              {start_BB = TRUE; l++;};

          if (iInst.end_p())
            if (l > 1)
              {build_header(*new_bb,k,l,*last_label,
    count++,start_reg, new_part->type());
               new_part->append(*new_bb);}
            else if (l == 1)
              {first_BB = last_label;
               build_header(*new_bb,k,l,*first_label,
    count++,start_reg, new_part->type());
               new_part->append(*new_bb);};

       };

     new_part->append(*new_part_suffix);
     new_part->prepend(generate_bootstrap(*first_BB,k,start_reg,
    new_part->cthread_p()));
     new_cb->append(new_part);

    };

  if (count > MAX_BB_PER_CB)
    sigerr("intrument_CB::constructor: Too many BBs for a single code block.");

  ofstream f_out("/home/jj/sybok/Profiler/Static_measures/002/qstat_short",
       ios::app);

  if (f_out.fail())
    sigerr("instrument_CB:constructor: Error in writing short stat file.");

  f_out << count << " " << ccount << endl;

  f_out.flush();
  f_out.close();
```

```
  if (f_out.fail())
    sigerr("instrument_CB: Error in writing statics measure file.");

};

void
instrument_CB::build_header(DList<ppc_Inst*>& bb, int k, int l,
    const String& bb_label, int bb_id,
    int reg_idx_start, PartType ptype)
{ String suffix =
    *(new String(num_to_str(i) + "." + num_to_str(j) + "."+
                 num_to_str(k)+"."+num_to_str(l)));

  String new_label = *(new String("beg." + suffix));
  String join_point = *(new String("join." + suffix));
  String end_point = *(new String("end." + suffix));
  String cold_entry = *(new String("ce." + suffix));

  DList<ppc_Inst*>* header = new DList<ppc_Inst*>;

  header->append(make_fall_thr_ins(reg_idx_start,join_point));
  header->append(make_jump_entry_ins(reg_idx_start,bb_label));
  header->append(make_common_ins(reg_idx_start,join_point));
  header->append(make_cold_entry(reg_idx_start,
 cold_entry, new_label,
 bb_id, ptype));

//  header->append(new ppc_Inst(ppc_none,new_label));

  bb.prepend(*header);

  bb.append(new ppc_Inst(ppc_none,end_point));

};


DList<ppc_Inst*>&
instrument_CB::make_fall_thr_ins(int first_reg, String& join_pt)
{
  ppc_Reg r1 = *(new ppc_Reg(first_reg+1));
  ppc_Reg r2 = *(new ppc_Reg(first_reg+2));
  ppc_Reg r4 = *(new ppc_Reg(first_reg+4));
  ppc_Reg r5 = *(new ppc_Reg(first_reg+5));

  DList<ppc_Inst*>* code_seg = new DList<ppc_Inst*>;

  ppc_Inst* ft_1 = new ppc_Inst(ppc_mfspr);
  ft_1->rands(new ppc_Rand(r2), new ppc_Rand(RTCL));
  code_seg->append(ft_1);

  ppc_Inst* ft_2 = new ppc_Inst(ppc_lwz);
  ft_2->rands(new ppc_Rand(r4), new ppc_Rand(r1), new ppc_Rand(0));
  code_seg->append(ft_2);

  ppc_Inst* ft_3 = new ppc_Inst(ppc_lwz);
  ft_3->rands(new ppc_Rand(r5), new ppc_Rand(r1), new ppc_Rand(8));
  code_seg->append(ft_3);

  ppc_Inst* ft_4 = new ppc_Inst(ppc_addi);
  ft_4->rands(new ppc_Rand(r4), new ppc_Rand(r4), new ppc_Rand(1));
  code_seg->append(ft_4);

  ppc_Inst* ft_5 = new ppc_Inst(ppc_stw);
  ft_5->rands(new ppc_Rand(r4), new ppc_Rand(r1), new ppc_Rand(0));
  code_seg->append(ft_5);

  ppc_Inst* ft_branch = new ppc_Inst(ppc_b);
  ft_branch->rands(new ppc_Rand(join_pt));
  code_seg->append(ft_branch);

  return(*code_seg);

};

DList<ppc_Inst*>&
instrument_CB::make_jump_entry_ins(int first_reg, const String& labl = "")
{
  ppc_Reg r1 = *(new ppc_Reg(first_reg+1));
  ppc_Reg r2 = *(new ppc_Reg(first_reg+2));
  ppc_Reg r4 = *(new ppc_Reg(first_reg+4));
  ppc_Reg r5 = *(new ppc_Reg(first_reg+5));

  DList<ppc_Inst*>* code_seg = new DList<ppc_Inst*>;

  ppc_Inst* je_1 = new ppc_Inst(ppc_mfspr, labl);
  je_1->rands(new ppc_Rand(r2), new ppc_Rand(RTCL));
  code_seg->append(je_1);

  ppc_Inst* je_2 = new ppc_Inst(ppc_lwz);
  je_2->rands(new ppc_Rand(r4), new ppc_Rand(r1), new ppc_Rand(4));
  code_seg->append(je_2);

  ppc_Inst* je_3 = new ppc_Inst(ppc_lwz);
  je_3->rands(new ppc_Rand(r5), new ppc_Rand(r1), new ppc_Rand(8));
  code_seg->append(je_3);

  ppc_Inst* je_4 = new ppc_Inst(ppc_addi);
  je_4->rands(new ppc_Rand(r4), new ppc_Rand(r4), new ppc_Rand(1));
  code_seg->append(je_4);

  ppc_Inst* je_5 = new ppc_Inst(ppc_stw);
  je_5->rands(new ppc_Rand(r4), new ppc_Rand(r1), new ppc_Rand(4));
  code_seg->append(je_5);

  return(*code_seg);

};


DList<ppc_Inst*>&
instrument_CB::make_common_ins(int first_reg, const String& join_pt = "")
{
  ppc_Reg r1 = *(new ppc_Reg(first_reg+1));
  ppc_Reg r2 = *(new ppc_Reg(first_reg+2));
  ppc_Reg r3 = *(new ppc_Reg(first_reg+3));
  ppc_Reg r4 = *(new ppc_Reg(first_reg+4));
  ppc_Reg r5 = *(new ppc_Reg(first_reg+5));

  DList<ppc_Inst*>* code_seg = new DList<ppc_Inst*>;

  ppc_Inst* jn_1 = new ppc_Inst(ppc_lwz,join_pt);
  jn_1->rands(new ppc_Rand(r4), new ppc_Rand(r1), new ppc_Rand(12));
  code_seg->append(jn_1);

  ppc_Inst* jn_2 = new ppc_Inst(ppc_doz);
  jn_2->rands(new ppc_Rand(r3), new ppc_Rand(rSTAT), new ppc_Rand(r2));
  code_seg->append(jn_2);

  ppc_Inst* jn_3 = new ppc_Inst(ppc_addc);
  jn_3->rands(new ppc_Rand(r5), new ppc_Rand(r5), new ppc_Rand(r3));
  code_seg->append(jn_3);

  ppc_Inst* jn_4 = new ppc_Inst(ppc_addze);
  jn_4->rands(new ppc_Rand(r4), new ppc_Rand(r4));
  code_seg->append(jn_4);

  ppc_Inst* jn_5 = new ppc_Inst(ppc_stw);
  jn_5->rands(new ppc_Rand(r5), new ppc_Rand(r1), new ppc_Rand(8));
  code_seg->append(jn_5);

  ppc_Inst* jn_6 = new ppc_Inst(ppc_stw);
  jn_6->rands(new ppc_Rand(r4), new ppc_Rand(r1), new ppc_Rand(12));
  code_seg->append(jn_6);

  return(*code_seg);

};

DList<ppc_Inst*>&
instrument_CB::make_cold_entry(int first_reg,
       String& c_entry,
       String& new_label,
       int id, PartType ptype)
{ String* id_str =
    new String(new_cb->name()+"...fstatsbase+" + num_to_str(16*id));


  ppc_Reg r1 = *(new ppc_Reg(first_reg+1));
  ppc_Reg r2 = *(new ppc_Reg(first_reg+2));
  ppc_Reg r3 = *(new ppc_Reg(first_reg+3));

  ppc_Reg base_Reg;

  switch(ptype) {
  case INLET_PART:
    base_Reg = rIFP;
    break;
  case CTHREAD_PART:
    base_Reg = rFP;
    break;
  default:
    base_Reg = rFP;
    break;
  };

  DList<ppc_Inst*>* code_seg = new DList<ppc_Inst*>;

  ppc_Inst* cs_1 = new ppc_Inst(ppc_addi,c_entry);
  cs_1->rands(new ppc_Rand(r1), new ppc_Rand(base_Reg),
      new ppc_Rand(*id_str));
  code_seg->append(cs_1);

  ppc_Inst* cs_2 = new ppc_Inst(ppc_mfspr,new_label);
  cs_2->rands(new ppc_Rand(rSTAT), new ppc_Rand(RTCL));
  code_seg->append(cs_2);

  return(*code_seg);
};

ppc_Inst*
instrument_CB::filter(ppc_Inst* orig_ins, const String& target,
      DList<ppc_Inst*>* suffix, int k, int l,
      int start_reg, int& ccount, PartType ptype)
{
  ppc_Inst* new_ins = new ppc_Inst(*orig_ins);

  ppc_Rand** rlist = orig_ins->rands();

  ppc_Rand* targ_address = 0;

  String* linker_label;
```

```
  ppc_Reg rCARG1 = *(new ppc_Reg(3,GPR_RTYPE,3));
  ppc_Reg rCARG2 = *(new ppc_Reg(4,GPR_RTYPE,4));
  ppc_Reg cr0 = *(new ppc_Reg(0,CR_RTYPE,0));
  ppc_Reg r1 = *(new ppc_Reg(start_reg+1));
  ppc_Reg r2 = *(new ppc_Reg(start_reg+2));
  ppc_Reg r3 = *(new ppc_Reg(start_reg+3));
  ppc_Reg r4 = *(new ppc_Reg(start_reg+4));
  ppc_Reg r5 = *(new ppc_Reg(start_reg+5));

  int call_count;

  for(int loop = 0; loop < orig_ins->nrands(); loop++)
    if ((rlist[loop]->label_p()) && (rlist[loop]->label() == target))
      new_ins->rands()[loop] =
new ppc_Rand("first."+num_to_str(i)+"."+num_to_str(j)+"."+
     num_to_str(k)+".1");

  if (orig_ins->op().linked_p() &&
      !i_cache->exists_p(get_branch_target(*orig_ins)))
    {
      for(loop = 0; loop < orig_ins->nrands(); loop++)
        if (rlist[loop]->label_p())

          {targ_address = rlist[loop];

   if (registry->contains(targ_address->label()))
     call_count = (*registry)[targ_address->label()];
   else
     {call_count = ccount++;
      (*registry)[targ_address->label()] = call_count;};

   if (call_count > MAX_CALLED_FN_PER_CB)
     sigerr("instrument_CB::filter: too many called procs.");

          linker_label =
     new String(targ_address->label() +".lnk."+num_to_str(i)+"."+
num_to_str(j)+"."+ num_to_str(k)+"."+num_to_str(l+1));

          new_ins->rands()[loop] = new ppc_Rand(*linker_label);

          suffix->append(make_jump_entry_ins(start_reg,*linker_label));
          suffix->append(make_common_ins(start_reg));

   ppc_Reg base_Reg;

   switch(ptype) {
   case INLET_PART:
     base_Reg = rIPP;
     break;
   case CTHREAD_PART:
     base_Reg = rFP;
     break;
   default:
     base_Reg = rFP;
     break;
   };

   String* cid_str =
     new String(new_cb->name()+"...fstatsbase+" +
num_to_str(16*MAX_BB_PER_CB+20*call_count));

   ppc_Inst* grab_t_1 = new ppc_Inst(ppc_mfspr);
   grab_t_1->rands(new ppc_Rand(r1), new ppc_Rand(RTCU));
   suffix->append(grab_t_1);

   ppc_Inst* grab_t_2 = new ppc_Inst(ppc_mfspr);
   grab_t_2->rands(new ppc_Rand(r2), new ppc_Rand(RTCL));
   suffix->append(grab_t_2);

   ppc_Inst* grab_t_3 = new ppc_Inst(ppc_mfspr);
   grab_t_3->rands(new ppc_Rand(r3), new ppc_Rand(RTCU));
   suffix->append(grab_t_3);

   ppc_Inst* grab_t_4 = new ppc_Inst(ppc_cmpw);
   grab_t_4->rands(new ppc_Rand(cr0), new ppc_Rand(r1),
   new ppc_Rand(r3));
   suffix->append(grab_t_4);

   ppc_Inst* s_time_1 = new ppc_Inst(ppc_addi);
   s_time_1->rands(new ppc_Rand(r4), new ppc_Rand(base_Reg),
   new ppc_Rand(*cid_str));
   suffix->append(s_time_1);

   ppc_Inst* count_call_1 = new ppc_Inst(ppc_lwz);
   count_call_1->rands(new ppc_Rand(r5), new ppc_Rand(r4),
       new ppc_Rand(0));
   suffix->append(count_call_1);

          String local_targ = generate_next_anon();

   ppc_Inst* s_time_2 = new ppc_Inst(ppc_beq);
   s_time_2->rands(new ppc_Rand(cr0),new ppc_Rand(local_targ));
   suffix->append(s_time_2);

   ppc_Inst* s_time_3 = new ppc_Inst(ppc_mfspr,generate_next_anon());
   s_time_3->rands(new ppc_Rand(r2), new ppc_Rand(RTCL));
   suffix->append(s_time_3);

   ppc_Inst* s_time_4 = new ppc_Inst(ppc_stw,local_targ);
   s_time_4->rands(new ppc_Rand(r2), new ppc_Rand(r4),
   new ppc_Rand(4));
   suffix->append(s_time_4);

   ppc_Inst* s_time_5 = new ppc_Inst(ppc_stw);
   s_time_5->rands(new ppc_Rand(r3), new ppc_Rand(r4),
   new ppc_Rand(8));
   suffix->append(s_time_5);

   ppc_Inst* count_call_2 = new ppc_Inst(ppc_addi);
   count_call_2->rands(new ppc_Rand(r5), new ppc_Rand(r5),
       new ppc_Rand(1));
   suffix->append(count_call_2);

   ppc_Inst* count_call_3 = new ppc_Inst(ppc_stw);
   count_call_3->rands(new ppc_Rand(r5), new ppc_Rand(r4),
       new ppc_Rand(0));
   suffix->append(count_call_3);

          ppc_Inst* ext_brnch = new ppc_Inst(ppc_bl);
          ext_brnch->rands(targ_address);
          suffix->append(ext_brnch);
/*
          ppc_Inst* nop = new ppc_Inst(ppc_cror,generate_next_anon());
          nop->rands(new ppc_Rand(31),new ppc_Rand(31), new ppc_Rand(31));
          suffix->append(nop);

 */
          ppc_Inst* save = new ppc_Inst(ppc_mr,generate_next_anon());
   save->rands(new ppc_Rand(r1), new ppc_Rand(rCARG1));
   suffix->append(save);

   ppc_Inst* accum_1 = new ppc_Inst(ppc_addi);
   accum_1->rands(new ppc_Rand(rCARG1), new ppc_Rand(base_Reg),
   new ppc_Rand(*cid_str));
   suffix->append(accum_1);

   ppc_Inst* accum_2 = new ppc_Inst(ppc_li);
   accum_2->rands(new ppc_Rand(rCARG2), new ppc_Rand(4));
   suffix->append(accum_2);

   ppc_Inst* accum_3 = new ppc_Inst(ppc_bl);
          accum_3->rands(new ppc_Rand(".accumtime"));
          suffix->append(accum_3);

          ppc_Inst* restore = new ppc_Inst(ppc_mr, generate_next_anon());
   restore->rands(new ppc_Rand(rCARG1), new ppc_Rand(r1));
   suffix->append(restore);

          ppc_Inst* ret_brnch = new ppc_Inst(ppc_b);
          ret_brnch->rands(new ppc_Rand("ce."+num_to_str(i)+"."+
 num_to_str(j)+"."+num_to_str(k)+"."+
                                           num_to_str(l+2)));
          suffix->append(ret_brnch);

          break;
        };
    };

  return new_ins;

};

DList<ppc_Inst*>&
instrument_CB::generate_bootstrap(const String& init_label,
  int k, int start_reg, bool cthread_p)
{ ppc_Reg r0 = *(new ppc_Reg(start_reg));
  ppc_Reg rCARG1 = *(new ppc_Reg(3,GPR_RTYPE,3));

  DList<ppc_Inst*>* header = new DList<ppc_Inst*>;

  header->append(new ppc_Inst(ppc_none,init_label));

  ppc_Inst* boot_setup;

  if (cthread_p)
    {for(int i = 0; i <= INSTR_REGS; i++)
       {ppc_Reg store_reg = (i < INSTR_REGS) ?
  *(new ppc_Reg(13+i,GPR_RTYPE,13+i)):
    *(new ppc_Reg(31,GPR_RTYPE,31));
ppc_Inst* st_inst = new ppc_Inst(ppc_stw);
st_inst->rands(new ppc_Rand(store_reg),new ppc_Rand(rSP),
       new ppc_Rand(-80-i*4));
header->append(st_inst);}

     boot_setup = new ppc_Inst(ppc_mr);
     boot_setup->rands(new ppc_Rand(rFP), new ppc_Rand(rCARG1));
     header->append(boot_setup);};

  ppc_Inst* boot_1 = new ppc_Inst(ppc_mflr);
  boot_1->rands(new ppc_Rand(r0));
  header->append(boot_1);

  ppc_Inst* boot_2 = new ppc_Inst(ppc_bl);
  boot_2->rands(new ppc_Rand("ce."+num_to_str(i)+"."+num_to_str(j)+
    "."+num_to_str(k)+".1"));
  header->append(boot_2);
  /*
  ppc_Inst* boot_3 = new ppc_Inst(ppc_cror,generate_next_anon());
  boot_3->rands(new ppc_Rand(31),new ppc_Rand(31),new ppc_Rand(31));
  header->append(boot_3);
  */
```

```
  header->append(make_jump_entry_ins(start_reg,generate_next_anon()));
  header->append(make_common_ins(start_reg));

  ppc_Inst* boot_4 = new ppc_Inst(ppc_mtlr);
  boot_4->rands(new ppc_Rand(r0));
  header->append(boot_4);

  if (cthread_p)
    for(int i = 0; i <= INSTR_REGS; i++)
      {ppc_Reg load_reg = (i < INSTR_REGS) ?
*(new ppc_Reg(13+i,GPR_RTYPE,13+i)) : *(new ppc_Reg(31,GPR_RTYPE,31));
       ppc_Inst* ld_inst = new ppc_Inst(ppc_lwz);
       ld_inst->rands(new ppc_Rand(load_reg),new ppc_Rand(rSP),
      new ppc_Rand(-80-i*4));
       header->append(ld_inst);};

  header->append(new ppc_Inst(ppc_blr));

  return(*header);
};


String& instrument_CB::generate_next_anon()
{return *(new String("anon."+num_to_str(i)+"."+num_to_str(j)+"."+
   num_to_str(anon_id_count++)));}
```

```
//QPROF ppc_type ADT

// The ppc_type ADT is used to collect statistics about the types of
// PowerPC instructions which occur in each BB of a program.

#ifndef PPC_TYPE_H
#define PPC_TYPE_H 1

#include <String.h>

#include "err.h"
#include "bbnode.h"
#include "nodemap.h"
#include "ppc_Rator.h"
#include "sT_Rator.h"
#include "DList.h"

#include "ICache.h"

class type_count
{
 public:

  type_count(ICache& ic1, DList<String*>& branch_chart, const String& targ);

  int arith_cnt() {return arith;};
  int logl_cnt() {return logl;};
  int contrl_cnt() {return contrl;};
  int floatp_cnt() {return floatp;};
  int mem_cnt() {return mem;};

  int alu_int_cnt() {return alu_int;};
  int alu_flt_cnt() {return alu_flt;};
  int heap_cnt() {return heap;};
  int i_m_struct_cnt() {return i_m_struct;};
  int sched_cnt() {return sched;};
  int network_cnt() {return network;};
  int lmem_cnt() {return lmem;};
  int reg_alloc_cnt() {return reg_alloc;};

  DList<String*> get_calls() {return call_list;};

 private:

  int arith, logl, contrl, floatp, mem;

  int alu_int, alu_flt, heap, i_m_struct, sched, network, lmem, reg_alloc;

  DList<String*> call_list;

  String target;

  ICache* ic;

  void accum_ppc_type(optype op);
  void accum_sT_type(ppc_Inst& ins);
  void accum_call(ppc_Inst& ins);

};

inline void type_count::accum_call(ppc_Inst& ins)
{ ppc_Rand** rlist = ins.rands();

  if (ins.op().linked_p())
    for(int loop = 0; loop < ins.nrands(); loop++)
      if (rlist[loop]->label_p())
if ((rlist[loop]->label().freq('.') >= 5) &&
    (rlist[loop]->label().contains(".lnk.")))
  {int pos = rlist[loop]->label().index(".lnk.");
   String* call_name = new String(rlist[loop]->label());
   call_list.append(new String(call_name->before(pos)));
   break;};
};

#endif
```

```
//QPROF type_count ADT

#include "type_count.h"
#include "bbnode.h"

type_count::type_count(ICache& ic1, DList<String*>& branch_chart,
   const String& targ): target(targ)
{
  ic = &ic1;
  bool collecting = FALSE;
  bool done = FALSE;

  arith = 0; logl = 0; contrl = 0; floatp = 0; mem = 0;

  alu_int = 0; alu_flt = 0; heap = 0; i_m_struct = 0;
  sched = 0; network = 0; lmem = 0; reg_alloc = 0;

  DListIter<String*> path(branch_chart), call_list;

  String initial_fetch_target =
    ic->partition(target)->instIter().value()->label();

  ic->fetch(initial_fetch_target);

  DList<ppc_Inst*> working_i_list = ic->load();

  DListIter<ppc_Inst*>* i;
  optype op;
  sT_Opcode sT_op;

  while (!working_i_list.empty() && !done) {
    for(i = working_i_list.iter(); !i->end_p(); (*i)++)
      {

if ((collecting) && (!i->value()->label().empty()))
  {collecting = FALSE; done = TRUE;}
else if ((!collecting) && (i->value()->label() == target))
  collecting = TRUE;

op = i->value()->op().opt();
sT_op = i->value()->sT_source();
       ppc_Inst* curr_ins = i->value();

if (collecting)
  {accum_ppc_type(op);
   accum_sT_type(*curr_ins);
          accum_call(*curr_ins);};

if (!done && (op == BRNCH))
  if (!path.value()->empty())
    {ic->fetch(*path.value());
     working_i_list = ic->load();
     path++;
     break;}
  else
    path++;

if (!done && i->next_end_p())
  {ic->fetch(4);
   working_i_list = ic->load();
   break;};
      };

    delete i;

  };
};


void type_count::accum_ppc_type(optype op) {

  switch(op) {
  case FXU: case FXU_CMP:
    arith++;
    break;
  case FXU_LOG: case CR_OP:
    logl++;
    break;
  case FXU_LXX: case FXU_LXI: case FXU_LPX: case FXU_LPI:
  case FXU_SXX: case FXU_SXI: case FXU_SPX: case FXU_SPI:
    mem++;
    break;
  case FXU_MFSPR: case FXU_MTSPR: case BRNCH:
    contrl++;
    break;
  case FPU: case FPU_A: case FPU_CMP:
    floatp++;
    break;
  };

};

void type_count::accum_sT_type(ppc_Inst& ins) {

  sT_Opcode sT_op = ins.sT_source();

  switch(sT_op) {

  case sT_add:     case sT_sub:     case sT_mul:      case sT_div:
  case sT_neg:     case sT_and:     case sT_or:       case sT_xor:
  case sT_not:     case sT_shiftl:  case sT_shiftr:   case sT_ashiftr:
  case sT_rotl:    case sT_rotr:    case sT_cmp:      case sT_cmpl:
  case sT_padd:    case sT_mov:
    alu_int++;
    break;

  case sT_fadd:  case sT_fsub:  case sT_fmul:   case sT_fdiv:
  case sT_fneg:  case sT_fcmp:  case sT_fabs:
  case sT_tofloat: case sT_toint:
    alu_flt++;
    break;

  case sT_load: case sT_store: case sT_prefetch:
    lmem++;
    break;

  case sT_hload: case sT_hstore: case sT_hgetpb: case sT_hsetpb:
    heap++;
    break;

  case sT_iload:  case sT_istore:   case sT_mload:
  case sT_mstore: case sT_mexamine: case sT_mupdate:
    i_m_struct++;
    break;

  case sT_mkcont: case sT_contpe:    case sT_contip:
  case sT_contfp: case sT_mkargcont: case sT_mkvalcont:
  case sT_send:   case sT_recv:      case sT_recvdone: case sT_netpoll:
    network++;
    break;

  case sT_merge: case sT_jump: case sT_blt:   case sT_ble:    case sT_beq:
  case sT_bne:   case sT_bge:  case sT_bgt:   case sT_bjoin:  case sT_bnjoin:
  case sT_fork:  case sT_post: case sT_halt:
  case sT_call:  case sT_carg: case sT_cret:
    sched++;
    break;

  default:
    if (ins.op().op() != ppc_none)
      reg_alloc++;
  };

};
```

```
//QPROF make_measure module

// This module generates static measures from a PowerPC module,
// including ideal execution time, instruction mix, "flavor" mix,
// and procedure call counts

#include <String.h>
#include <fstream.h>

#include "ppc_Mod.h"
#include "ICache.h"
#include "modmap.h"


#ifndef MAKE_MEASURE_H
#define MAKE_MEASURE_H 1

void make_measure(ppc_Mod& old_mod, ppc_Mod& new_mod);
void measure_BB(const String str, ostream& f_out,
ICache* ic1, ICache* ic2, modmap* mmap);
String& num_to_str(int i);

#endif
#include <fstream.h>
#include <String.h>

#include "ppc_Mod.h"
#include "ppc_Proc.h"
#include "ppc_CB.h"
#include "ppc_Part.h"
#include "ppc_Inst.h"
#include "ppc_Rator.h"

#include "ICache.h"
#include "modmap.h"
#include "bru.h"
#include "type_count.h"

#include "make_measure.h"


void make_measure(ppc_Mod& old_mod, ppc_Mod& new_mod) {

  ICache* ic1 = new ICache(new_mod);        // generate 1st ICache
  ICache* ic2 = new ICache(new_mod);        // generate 2nd ICache

  modmap* mmap = new modmap(ic1,new_mod);

  String srch_target;

  ofstream f_out("/home/jj/sybok/Profiler/Static_measures/002/qstat_long");

  if (f_out.fail())
    sigerr("make_measure: Error in writing statics measure file.");

  int i = 1;


  for(ppc_ProcIter iProc(old_mod.procIter()); !iProc.end_p(); iProc++, i++)
    {int j = 1;

     for(ppc_CBIter iCB(iProc.value()->CBIter()); !iCB.end_p() ; iCB++, j++)
       {int k = 1;

for(ppc_PartIter iPart(iCB.value()->partIter());
    !iPart.end_p();iPart++, k++)
  {
    int l = 1;
    bool start_BB = TRUE;

    if (iPart.value()->data_p())
      continue;

    for(ppc_InstIter
iInst(iPart.value()->instIter());!iInst.end_p(); )
      {ppc_Inst& ins= *iInst.value();

       if (ins.label().empty() && (ins.op().op() == ppc_none))
 {iInst++; continue;};

      if (start_BB)
 {start_BB = FALSE;
  if (!ins.label().empty())
    {f_out << i << " " << j << " " << k << " " << l << " ";
     f_out << iPart.value()->name() << endl;
     srch_target = "beg."+
       num_to_str(i)+"."+num_to_str(j)+"."+
       num_to_str(k)+"." + num_to_str(l);
     measure_BB(srch_target,f_out,ic1,ic2,mmap);}
  else
    sigerr("make_measure: BB detected without valid label.");};

      if (!ins.op().inert_p())
{start_BB = TRUE; l++;};

      iInst++;

      while (!iInst.end_p() && iInst.value()->label().empty() &&
  (iInst.value()->op().op() == ppc_none))
{iInst++;};

      while (ins.label().empty() && (ins.op().op() == ppc_none))
{iInst++;};

      if ((!iInst.end_p()) && (!iInst.value()->label().empty()))
if (!start_BB)
  {start_BB = TRUE;
   l++;};};};};

 f_out.flush();
 f_out.close();

};

void measure_BB(const String targ_str, ostream& f_out,
ICache* ic1, ICache* ic2, modmap* mmap)
{ String str(targ_str);

  // time "fall through" path

  branch_chart_list& bcl1 = mmap->make_branch_map(str,TRUE,FALSE);

  if (bcl1.empty())
    sigerr("measure_BB: Labeling error.");




  /*

  mmap->print(ic1->partition(str));

  cout <<"$ ---------" << endl;
  for(DListIter<String*> iter(*bcl1[0]); !iter.end_p(); iter++)
    cout << "$" << *(iter.value()) << endl;
  cout <<"$ ---------" << endl;

  */

  bru rs6000a(*ic1,*ic2,*bcl1[0],str);

  f_out << rs6000a.time_code() << endl;

  // time "jump exit" path

  branch_chart_list& bcl2 = mmap->make_branch_map(str,FALSE,FALSE);

  if (bcl2.empty())
    sigerr("measure_BB: Labeling error.");

   /*

  cout <<" $+++++++++--------" << endl;
  for(DListIter<String*> itr(*bcl2[0]); !itr.end_p(); itr++)
    cout << "$" << *(itr.value()) << endl;
  cout <<" $+++++++++--------" << endl;

    */

  bru rs6000b(*ic1,*ic2,*bcl2[0],str);

  f_out << rs6000b.time_code() << endl;

  // collect other statistics

  type_count itypes(*ic1, *bcl1[0], str);

  f_out << itypes.arith_cnt() << " " << itypes.logl_cnt() << " ";
  f_out << itypes.contrl_cnt() << " " << itypes.floatp_cnt() << " ";
  f_out << itypes.mem_cnt() << endl;

  f_out << itypes.alu_int_cnt() << " " << itypes.alu_flt_cnt() << " ";
  f_out << itypes.heap_cnt() << " " << itypes.i_m_struct_cnt() << " ";
  f_out << itypes.sched_cnt() << " " << itypes.network_cnt() << " ";
  f_out << itypes.lmem_cnt() << " " << itypes.reg_alloc_cnt() << endl;

  DList<String*> c_list = itypes.get_calls();

  f_out << c_list.length() << endl;

  for(DListIter<String*> strIter(c_list); !strIter.end_p(); strIter++)
    f_out << *strIter.value() << endl;

};

String& num_to_str(int i)
{
  String* str = new String;

  do
    {(*str) += (char) (48+(i % 10));
     i = i /10;}
  while (i != 0);

  str->reverse();

  return(*str);
```

};

Appendix D

Appendix: PostProf

```
#include "CB_data.h"
#include "err.h"

const CNTRL_P = 16;
const CNTRL_N = 14;
const CNTRL_F = 6;
const CNTRL_B = 2;
const CNTRL_S = 19;
const CNTRL_L = 12;

int plot_and_capture(int stat, DList<CB_data*>& cb_list);
DList<CB_data*>& install();

extern "C" {
int grab_keyboard_char();
};

void main()
{
  ifstream f1("/home/jj/sybok/Profiler/Static_measures/002/qstat_short");
  if (f1.fail())
    sigerr("f1 failed.");


  ifstream f2("/home/jj/sybok/Profiler/Static_measures/002/qstat_long");
  if (f2.fail())
    sigerr("f2 failed.");


  ifstream f3("/home/jj/sybok/Profiler/Static_measures/002/qresults");
  if (f3.fail())
    sigerr("f3 failed.");


  int cb_count;
  f1 >> cb_count;

  DList<CB_data*> cb_list;

  for (int i = 0; i < cb_count; i++)
    cb_list.append(new CB_data(f1,f2,f3,cb_count));

  f1.close();
  f2.close();
  f3.close();

  int cb_pos = 0;
  int part_pos = PARTITIONS;
  int inner_pos = BBS;
  int command = EOF;

  cout << "Entering PostProf visualizer." << endl;

  while ((command != ' ') && (command != '\n'))
    {cb_list[cb_pos]->generate_plot(part_pos,inner_pos);
     command = plot_and_capture(inner_pos,cb_list);

     switch(command) {
     case CNTRL_P:
       if (cb_pos > 0)
{cb_pos--;
 part_pos = PARTITIONS;
 inner_pos = BBS;
 cout << "Moving to previous Code Block." << endl;};
       break;
     case CNTRL_N:
       if (cb_pos < cb_count-1)
{cb_pos++;
 part_pos = PARTITIONS;
 inner_pos = BBS;
 cout << "Moving to next Code Block." << endl;};
       break;
     case CNTRL_F:
       if (part_pos < cb_list[cb_pos]->get_p_count()-1)
{part_pos++;
 inner_pos = BBS;
 cout << "Advancing forward through current Code Block." << endl;};
       break;
     case CNTRL_B:
       if (part_pos > PARTITIONS)
{part_pos--;
 inner_pos = BBS;
 cout << "Backing through current Code Block." << endl;};
       break;
     case CNTRL_S:
       if (part_pos >= 0)
{inner_pos = (inner_pos+1) % 3;
 cout << "Switching statistics mode." << endl;};
       break;
     case CNTRL_L:
       cout << "New run installed!" << endl;
       break;
     case ' ': case '\n':
       break;

     default:
       cout << "Unknown command.  ";
       cout << "Commands: C-n, C-p, C-f, C-b, C-s, C-L, RETURN.";
       cout << endl;};
     cout << "In Id Code Block " << cb_pos+1 << "/" << cb_count << "." << endl;

    };

  cout << "Done." << endl;

};

int plot_and_capture(int stat, DList<CB_data*>& cb_list) {

  String prog("/home/jj/sybok/Profiler/graphics/gnuplot ");
  String arg("/home/jj/sybok/Profiler/graphics/comfile ");
  String option1("-geometry 850x500+200+100 ");
  String option2("-fn 5x8 -geometry 850x500+200+100 ");

  FILE* filep;

  freopen("/dev/null","w",stderr);

  if (stat == BBS)
    filep = popen(prog+option1+arg,"w");
  else
    filep = popen(prog+option2+arg,"w");

  int in_char = grab_keyboard_char();

  if (in_char == CNTRL_L)
    {system("/home/jj/sybok/Profiler/main/qprof");
     cb_list = install();};

  putc('\n',filep);

  pclose(filep);

  return(in_char);
};


DList<CB_data*>& install() {

  ifstream f1("/home/jj/sybok/Profiler/Static_measures/002/qstat_short");
  if (f1.fail())
    sigerr("f1 failed.");

  ifstream f2("/home/jj/sybok/Profiler/Static_measures/002/qstat_long");
  if (f2.fail())
    sigerr("f2 failed.");

  ifstream f3("/home/jj/sybok/Profiler/Static_measures/002/qresults");
  if (f3.fail())
    sigerr("f3 failed.");

  int cb_count;
  f1 >> cb_count;

  DList<CB_data*>* cb_list = new DList<CB_data*>;

  for (int i = 0; i < cb_count; i++)
    cb_list->append(new CB_data(f1,f2,f3,cb_count));

  f1.close();
  f2.close();
  f3.close();

  return *cb_list;
};
```

```
// This ADT holds CB data during the postprof phase of execution

#ifndef CB_DATA_H
#define CB_DATA_H 1

#include <iostream.h>
#include <fstream.h>

#include "DList.h"
#include "QVHMap.h"
#include "i_mix.h"
#include "rtsstate.h"

enum plotstate {PARTITIONS = -3, CCALLS = -2, IDBCS = -1};

enum statstate {BBS, PPC, STCODE};

class CB_data
{
 public:

  CB_data(ifstream& f_qshort, ifstream& f_qstatic,
  ifstream& q_result, int called_cb_cnt);

  void display_CB_parts();

  void display_CB_fns();

  void display_CCBs();

  void display_BBs(String& p_name);

  void display_IM_ppc(String& p_name);
  void display_IM_sY(String& p_name);

  DList<String*>& get_p_names();

  int get_p_count() { return p_count;};

  void generate_plot(int pos, int stat);

  private:
  String cb_name;

  int ccb_count;
  int bb_count;
  int fn_count;
  int p_count;

  int invocations;
  double l_time;

  String bb_pname[MAX_BB_PER_CB];
  int    bb_ft_cnt[MAX_BB_PER_CB];
  int    bb_br_cnt[MAX_BB_PER_CB];
  double bb_l_time[MAX_BB_PER_CB];
  int    bb_ft_thy[MAX_BB_PER_CB];
  int    bb_br_thy[MAX_BB_PER_CB];
  i_mix  bb_i_mix[MAX_BB_PER_CB];

  String fn_name[MAX_CALLED_FN_PER_CB];
  double fn_l_time[MAX_CALLED_FN_PER_CB];
  int    fn_i_cnt[MAX_CALLED_FN_PER_CB];

  String ccb_name[MAX_CALLED_CB_PER_CB];
  double ccb_l_time[MAX_CALLED_CB_PER_CB];
  int    ccb_i_cnt[MAX_CALLED_CB_PER_CB];

  QVHMap<String,int>* registry;

  String p_name[MAX_BB_PER_CB];
  double p_l_time[MAX_BB_PER_CB];
  double p_ave_dur[MAX_BB_PER_CB];
  DList<int> bbs[MAX_BB_PER_CB];

  int sum_i_mix_elt(int i, DList<int>& dl);

};

inline
DList<String*>& CB_data::get_p_names()
{
  DList<String*>* p_list = new DList<String*>;

  for(int i = 0; i < p_count; i++)
    p_list->append(&p_name[i]);

  return(*p_list);
};

inline
int CB_data::sum_i_mix_elt(int i, DList<int>& dl)
{int sum = 0;

  for(DListIter<int> dlIter(dl); !dlIter.end_p(); dlIter++)
    sum += bb_i_mix[dlIter.value()].get_count_univ(i);

return sum;
};

#endif
```

```
// This ADT holds CB data during the postprof phase of execution

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include "err.h"

#include "CB_data.h"

CB_data::CB_data(ifstream& f_qshort, ifstream& f_qstatic,
 ifstream& f_qresult, int called_cb_cnt):
       ccb_count(called_cb_cnt)
{ int cb_num, proc_num, part_num, bb_num;
  int frame_slots;
  String called_name;

  int current_fn = 0;
  QVHMap<String,int> fn_registry(-1,10);
  int j;

  f_qshort >> bb_count >> fn_count;

  for(int i = 0; i < ccb_count; i++)
    f_qshort >> ccb_name[i];



  f_qresult >> cb_name >> frame_slots >> invocations >> l_time;

  registry = new QVHMap<String,int>(-1,10);

  for(j = 0; j < MAX_BB_PER_CB; j++)
    {p_l_time[j] = 0.0;
     p_ave_dur[j] = 0.0;};

  for(i = 0; i < bb_count; i++)
    {  f_qresult >> bb_ft_cnt[i] >> bb_br_cnt[i] >> bb_l_time[i];



      f_qstatic >> cb_num >> proc_num >> part_num >> bb_num;
      f_qstatic >> bb_pname[i];

      p_count = part_num;

      f_qstatic >> bb_ft_thy[i] >> bb_br_thy[i];

      for(j = 0; j < 5; j++)
f_qstatic >> bb_i_mix[i].ppc_count[j];

      for(j = 0; j < 8; j++)
f_qstatic >> bb_i_mix[i].sT_count[j];

      (*registry)[bb_pname[i]] = part_num-1;
      p_name[part_num-1] = bb_pname[i];
      bbs[part_num-1].append(i);
      p_l_time[part_num-1] += bb_l_time[i];

      if (p_ave_dur[part_num-1] < 0.1)
p_ave_dur[part_num-1] = bb_ft_cnt[i]+bb_br_cnt[i];


      int local_calls;
      f_qstatic >> local_calls;
      for(j = 0; j < local_calls; j++)
{f_qstatic >> called_name;
 if (!fn_registry.contains(called_name))
   {fn_registry[called_name] = current_fn;
    fn_name[current_fn++] = called_name;};};


    };

  for(i = 0; i < p_count; i++)
    if (p_ave_dur[i] > 0)
      p_ave_dur[i] = p_l_time[i]/p_ave_dur[i];

  for(i = 0; i < fn_count; i++)
    f_qresult >> fn_i_cnt[i] >> fn_l_time[i];

  for(i = 0; i < ccb_count; i++)
    f_qresult >> ccb_i_cnt[i] >> ccb_l_time[i];
};



void CB_data::display_CB_parts()
{
  int xleft = 0;
  int xright = p_count+1;


  double ytop = 20.0;
  double sc_factor = 0;

  for(int i = 0; i < p_count; i++)
    if (ytop < p_l_time[i])
      ytop = p_l_time[i];


  double ybottom = -ytop/20.0;

  for(i = 0; i < p_count; i++)
    if (sc_factor < p_ave_dur[i])
      sc_factor = p_ave_dur[i];



  if (sc_factor > 0)
    sc_factor = ytop/sc_factor;

  ytop = 1.25*ytop;

  ofstream f_out("/home/jj/sybok/Profiler/graphics/comfile");

  if (f_out.fail())
    sigerr("CB_data::display_CB_parts: Error in writing command file.");



  ofstream data_out1("/home/jj/sybok/Profiler/graphics/d1");
  ofstream data_out2("/home/jj/sybok/Profiler/graphics/d2");

  if (data_out1.fail() || data_out2.fail())
    sigerr("CB_data::display_CB_parts: Error in writing data.");

  f_out << "set boxwidth 0.4" << endl;
  f_out << "set xtics 0,1" << endl;
  f_out << "set xlabel 'Partition'" << endl;
  f_out << "set ylabel 'Live Time/Relative LT'" << endl;

  f_out << "cd '/home/jj/sybok/Profiler/graphics'" << endl;

  f_out << "set label '" << cb_name;
  f_out << " Partitions' at " << xright/2.0 << "," << ytop*.966;
  f_out << " center" << endl;

  f_out << "set label '" << invocations << " invocations' at " << xright/2.0;
  f_out << "," << ytop*.933 << " center" << setprecision(15) << endl;

  f_out << "set label 'Livetime " << l_time;
  f_out << " cpu cycles' at " << xright/2.0;
  f_out << "," << ytop*.9 << " center" << endl;


  double work_time = 0.0;


  for(i = 0; i < p_count; i++)
    {  f_out << "set label '" << p_name[i] << "' at " << i+i << ",";
       f_out << ybottom/2.0 << " center" << endl;

       double duration = (p_ave_dur[i] < 0.001) ? 0.001 : p_ave_dur[i];

       f_out << "set label '(" << p_l_time[i]/duration << "X)' at ";
       f_out << i+.80 << "," << p_l_time[i]+ytop/30 <<" center" << endl;
       f_out << "set label 'Av' at ";
       f_out <<i+1.20 << "," << p_ave_dur[i]*sc_factor+ytop/30;
       f_out <<" center" << endl;
       data_out1 << i+.80 << " " << p_l_time[i] << endl;
       data_out2 << i+1.20 << " " << p_ave_dur[i]*sc_factor << endl;

       work_time += p_l_time[i];

     };

  f_out << "set label 'Work time " << work_time;
  f_out << " cpu cycles' at " << xright/2.0;
  f_out << "," << ytop*.866 << " center" << endl;


  f_out << "plot [" << xleft << ":" << xright << "] [";
  f_out << ybottom << ":" << ytop;
  f_out << "] 'd1' title 'Tot. Live Time' with boxes";
  f_out << ", 'd2' title 'Ave LT/call (scaled)' with boxes" << endl;

  f_out << "pause -1 " << endl;

  f_out.close();
  data_out1.close();
  data_out2.close();

};


void CB_data::display_CB_fns()
{

  int xleft = 0;
  int xright = fn_count+1;


  double ytop = 20.0;
  double sc_factor = 0;

  for(int i = 0; i < fn_count; i++)
    if (ytop < fn_l_time[i])
      ytop = fn_l_time[i];
```

```
  double ybottom = -ytop/20.0;

  for(i = 0; i < fn_count; i++)
    if ((fn_l_time[i] > 0.001) && (sc_factor < fn_l_time[i]/fn_i_cnt[i]))
      sc_factor = fn_l_time[i]/fn_i_cnt[i];



  if (sc_factor > 0)
    sc_factor = ytop/sc_factor;

  ytop = 1.25*ytop;

  ofstream f_out("/home/jj/sybok/Profiler/graphics/comfile");

  if (f_out.fail())
    sigerr("CB_data::display_CB_FNs: Error in writing command file.");



  ofstream data_out1("/home/jj/sybok/Profiler/graphics/d1");
  ofstream data_out2("/home/jj/sybok/Profiler/graphics/d2");

  if (data_out1.fail() || data_out2.fail())
    sigerr("CB_data::display_CB_FNs: Error in writing data.");

  f_out << "set boxwidth 0.4" << endl;
  f_out << "set xtics 0,1" << endl;
  f_out << "set xlabel 'C Function'" << endl;
  f_out << "set ylabel 'Live Time/Relative LT'" << endl;
  f_out << "cd '/home/jj/sybok/Profiler/graphics'" << endl;

  f_out << "set label '" << cb_name << " C Calls' at " << xright/2.0;
  f_out << "," << ytop*.966;
  f_out << " center" << endl;

  f_out << "set label '" << invocations << " invocations' at " << xright/2.0;
  f_out << "," << ytop*.933 << " center" << endl;

  f_out << "set label 'Livetime " << setprecision(15);
  f_out << l_time << setprecision(15) << " cpu cycles' at " << xright/2.0;
  f_out << "," << ytop*.9 << " center" << endl;

  for(i = 0; i < fn_count; i++)
    {  f_out << "set label '" << fn_name[i] << "' at " << i+i << ",";
       f_out << ybottom/2.0 << " center" << endl;

       f_out << "set label '(" << fn_i_cnt[i] << "X)' at ";
       f_out << i+.80 << "," << fn_l_time[i]+ytop/30 <<" center" << endl;
       f_out << "set label 'Av' at ";

       double local_count = (fn_i_cnt[i] < 0.001) ? 0.001 : fn_i_cnt[i];

       f_out <<i+1.20 << "," << fn_l_time[i]*sc_factor/local_count+ytop/30;
       f_out  <<" center" << endl;
       data_out1 << i+.80 << " " << fn_l_time[i] << endl;
       data_out2 << i+1.20 << " " << fn_l_time[i]*sc_factor/local_count;
       f_out << endl;
     };

  f_out << "plot [" << xleft << ":" << xright << "] [";
  f_out << ybottom << ":" << ytop;
  f_out  << "] 'd1' title 'Tot. Live Time' with boxes";
  f_out << ", 'd2' title 'Ave LT/call (scaled)' with boxes" << endl;

  f_out << "pause -1 " << endl;

  f_out.close();
  data_out1.close();
  data_out2.close();

};


void CB_data::display_CCBs()
{

  int xleft = 0;
  int xright = ccb_count+1;

  double ytop = 20.0;
  double sc_factor = 0;

  for(int i = 0; i < ccb_count; i++)
    if (ytop < ccb_l_time[i])
      ytop = ccb_l_time[i];


  double ybottom = -ytop/20.0;

  for(i = 0; i < ccb_count; i++)
    if ((ccb_l_time[i] > 0.001) && (sc_factor < ccb_l_time[i]/ccb_i_cnt[i]))
      sc_factor = ccb_l_time[i]/ccb_i_cnt[i];



  if (sc_factor > 0)
    sc_factor = ytop/sc_factor;

  ytop = 1.25*ytop;

  ofstream f_out("/home/jj/sybok/Profiler/graphics/comfile");

  if (f_out.fail())
    sigerr("CB_data::display_CCBs: Error in writing command file.");

  ofstream data_out1("/home/jj/sybok/Profiler/graphics/d1");
  ofstream data_out2("/home/jj/sybok/Profiler/graphics/d2");

  if (data_out1.fail() || data_out2.fail())
    sigerr("CB_data::display_CCBs: Error in writing data.");

  f_out << "set boxwidth 0.4" << endl;
  f_out << "set xtics 0,1" << endl;
  f_out << "set xlabel 'Called CB Name'" << endl;
  f_out << "set ylabel 'Live Time/Relative LT'" << endl;
  f_out << "cd '/home/jj/sybok/Profiler/graphics'" << endl;

  f_out << "set label '" << cb_name << " Id CB Calls' at ";
  f_out << xright/2.0 << "," << ytop*.966;
  f_out << " center" << endl;

  f_out << "set label '" << invocations << " invocations' at " << xright/2.0;
  f_out << "," << ytop*.933 << " center" << endl;

  f_out << "set label 'Livetime " << setprecision(15);
  f_out << l_time << setprecision(15) << " cpu cycles' at " << xright/2.0;
  f_out << "," << ytop*.9 << " center" << endl;

  for(i = 0; i < ccb_count; i++)
    {  f_out << "set label '" << ccb_name[i] << "' at " << i+i << ",";
       f_out << ybottom/2.0 << " center" << endl;

       f_out << "set label '(" << ccb_i_cnt[i] << "X)' at ";
       f_out << i+.80 << "," << ccb_l_time[i]+ytop/30 <<" center" << endl;
       f_out << "set label 'Av' at ";

       double local_count = (ccb_i_cnt[i] < 0.001) ? 0.001 : ccb_i_cnt[i];

       f_out <<i+1.20 << "," << ccb_l_time[i]*sc_factor/local_count+ytop/30;
       f_out <<" center" << endl;
       data_out1 << i+.80 << " " << ccb_l_time[i] << endl;
       data_out2 << i+1.20 << " " << ccb_l_time[i]*sc_factor/local_count;
       f_out << endl;
     };

  f_out << "plot [" << xleft << ":" << xright << "] [";
  f_out << ybottom << ":" << ytop;
  f_out << "] 'd1' title 'Tot. Live Time' with boxes";
  f_out << ", 'd2' title 'Ave LT/call (scaled)' with boxes" << endl;

  f_out << "pause -1 " << endl;

  f_out.close();
  data_out1.close();
  data_out2.close();

};

void
CB_data::display_BBs(String& p_name)
{
  int p_num = (*registry)[p_name];

  DList<int> bb_list = bbs[p_num];

  int local_bb_count = bb_list.length();

  int xleft = 0;
  int xright = local_bb_count+1;

  double ytop = 20.0;

  double bb_tot_thy_time[MAX_BB_PER_CB];

  for(DListIter<int> iIter(bb_list); !iIter.end_p(); iIter++)
    {int idx = iIter.value();
     double local_live_time = bb_l_time[idx];
     bb_tot_thy_time[idx] = bb_ft_cnt[idx]*bb_ft_thy[idx]+
                            bb_br_cnt[idx]*bb_br_thy[idx];
     if (ytop < local_live_time)
       ytop = local_live_time;
     if (ytop < bb_tot_thy_time[idx])
       ytop = bb_tot_thy_time[idx];};

  double ybottom = -ytop/20.0;

  ytop = 1.25*ytop;

  ofstream f_out("/home/jj/sybok/Profiler/graphics/comfile");

  if (f_out.fail())
    sigerr("CB_data::display_BBs: Error in writing command file.");

  ofstream data_out1("/home/jj/sybok/Profiler/graphics/d1");
  ofstream data_out2("/home/jj/sybok/Profiler/graphics/d2");

  if (data_out1.fail() || data_out2.fail())
```

```
    sigerr("CB_data::display_BBs: Error in writing data.");

  f_out << "set boxwidth 0.4" << endl;
  f_out << "set xtics 0,1" << endl;
  f_out << "set xlabel 'BB Number'" << endl;
  f_out << "set ylabel 'Real LT/Ideal LT'" << endl;
  f_out << "cd '/home/jj/sybok/Profiler/graphics'" << endl;

  f_out << "set label '" << p_name << " Basic Blocks' at ";
  f_out << xright/2.0 << "," << ytop*.966;
  f_out << " center" << endl;

  f_out << "set label '" << bb_ft_cnt[bb_list[0]]+bb_br_cnt[bb_list[0]];
  f_out << " invocations' at " << xright/2.0;
  f_out << "," << ytop*.933 << " center" << endl;

  f_out << "set label 'Livetime " << setprecision(15);
  f_out << p_l_time[p_num] << setprecision(15);
  f_out << " cpu cycles' at " << xright/2.0;
  f_out << "," << ytop*.9 << " center" << endl;

 int i = 0;

  for(DListIter<int> iIter2(bb_list); !iIter2.end_p(); iIter2++,i++)
    {  int idx = iIter2.value();
       f_out << "set label '(" << bb_ft_cnt[idx]+bb_br_cnt[idx] << "X)' at ";
       f_out << i+1 << "," << ybottom/2.0 <<" center" << endl;

       data_out1 << i+.80 << " " << bb_l_time[idx] << endl;
       data_out2 << i+1.20 << " " << bb_tot_thy_time[idx] << endl;
     };

  f_out << "plot [" << xleft << ":" << xright << "] [";
  f_out << ybottom << ":" << ytop;
  f_out << "] 'd1' title 'Actual Live Time' with boxes";
  f_out << ", 'd2' title 'Ideal Time' with boxes" << endl;

  f_out << "pause -1 " << endl;

  f_out.close();
  data_out1.close();
  data_out2.close();


};

void
CB_data::display_IM_ppc(String& p_name)
{
  int p_num = (*registry)[p_name];

  DList<int> bb_list = bbs[p_num];

  int xleft = 0;
  int xright = 5+1;

  double ytop = 20.0;

  int summed_stats[13];

  for(int i = 0; i < 13; i++)
    {summed_stats[i] = sum_i_mix_elt(i,bb_list);
     if (ytop < summed_stats[i])
       ytop = summed_stats[i];};

  double ybottom = -ytop/20.0;

  ytop = 1.25*ytop;

  ofstream f_out("/home/jj/sybok/Profiler/graphics/comfile");

  if (f_out.fail())
    sigerr("CB_data::display_IM: Error in writing command file.");

  ofstream data_out1("/home/jj/sybok/Profiler/graphics/d1");

  if (data_out1.fail())
    sigerr("CB_data::display_IM: Error in writing data.");

  f_out << "set boxwidth 0.4" << endl;
  f_out << "set noxtics" << endl;
  f_out << "set xlabel 'IMix Statistic'" << endl;
  f_out << "set ylabel 'Frequency'" << endl;
  f_out << "cd '/home/jj/sybok/Profiler/graphics'" << endl;

  f_out << "set label '" << p_name << " Basic Blocks' at ";
  f_out << xright/2.0 << "," << ytop*.966;
  f_out << " center" << endl;

  f_out << "set label '" << bb_ft_cnt[bb_list[0]]+bb_br_cnt[bb_list[0]];
  f_out  << " invocations' at " << xright/2.0;
  f_out << "," << ytop*.933 << " center" << endl;

  f_out << "set label 'Livetime " << setprecision(15);
  f_out << p_l_time[p_num] << setprecision(15) << " cpu cycles' at ";
  f_out << xright/2.0;
  f_out << "," << ytop*.9 << " center" << endl;

  for(i = 0; i < 5; i++)
    {
       f_out << "set label '" << bb_i_mix[0].get_ppc_name(i) << "' at ";
       f_out << i+1 << "," << ybottom/2.0 <<" center" << endl;

       data_out1 << i+1 << " " << summed_stats[i] << endl;
     };

  f_out << "plot [" << xleft << ":" << xright << "] [";
  f_out << ybottom << ":" << ytop;
  f_out << "] 'd1' title 'PowerPC Imix' with boxes" << endl;

  f_out << "pause -1 " << endl;

  f_out.close();
  data_out1.close();

};


void
CB_data::display_IM_sT(String& p_name)
{
  int p_num = (*registry)[p_name];

  DList<int> bb_list = bbs[p_num];

  int xleft = 0;
  int xright = 8+1;

  double ytop = 20.0;

  int summed_stats[13];

  for(int i = 0; i < 13; i++)
    {summed_stats[i] = sum_i_mix_elt(i,bb_list);
     if (ytop < summed_stats[i])
       ytop = summed_stats[i];};

  double ybottom = -ytop/20.0;

  ytop = 1.25*ytop;

  ofstream f_out("/home/jj/sybok/Profiler/graphics/comfile");

  if (f_out.fail())
    sigerr("CB_data::display_IM: Error in writing command file.");

  ofstream data_out1("/home/jj/sybok/Profiler/graphics/d1");

  if (data_out1.fail())
    sigerr("CB_data::display_IM: Error in writing data.");

  f_out << "set boxwidth 0.4" << endl;
  f_out << "set noxtics" << endl;
  f_out << "set xlabel 'IMix Statistic'" << endl;
  f_out << "set ylabel 'Frequency'" << endl;
  f_out << "cd '/home/jj/sybok/Profiler/graphics'" << endl;

  f_out << "set label '" << p_name << " Basic Blocks' at ";
  f_out << xright/2.0 << "," << ytop*.966;
  f_out << " center" << endl;

  f_out << "set label '" << bb_ft_cnt[bb_list[0]]+bb_br_cnt[bb_list[0]];
  f_out << " invocations' at " << xright/2.0;
  f_out << "," << ytop*.933 << " center" << endl;

  f_out << "set label 'Livetime " << setprecision(15);
  f_out << p_l_time[p_num] << setprecision(15) << " cpu cycles' at ";
  f_out << xright/2.0;
  f_out << "," << ytop*.9 << " center" << endl;

  for(i = 0; i < 8; i++)
    {
       f_out << "set label '" << bb_i_mix[0].get_sT_name(i) << "' at ";
       f_out << i+1 << "," << ybottom/2.0 <<" center" << endl;

       data_out1 << i+1 << " " << summed_stats[i+5] << endl;
     };

  f_out << "plot [" << xleft << ":" << xright << "] [";
  f_out << ybottom << ":" << ytop << "] 'd1' title 'sT Imix' with boxes";
  f_out << endl;;


  f_out << "pause -1 " << endl;

  f_out.close();
  data_out1.close();

};


void CB_data::generate_plot(int pos, int stat)
{
  switch(pos) {
  case PARTITIONS:
    display_CB_parts();
    break;
  case CCALLS:
    display_CB_fns();
    break;
  case IDBCS:
```

```
    display_CCBs();
    break;
  default:
    switch(stat) {
    case BBS:
      display_BBs(p_name[(int) pos]);
      break;
    case PPC:
      display_IM_ppc(p_name[(int) pos]);
      break;
    case STCODE:
      display_IM_sT(p_name[(int) pos]);
      break;
    };
  };
};
```

```
// QPROF i_mix ADT -- used for storing info about instruction mix during
//                    the postprof phase of processing

#ifndef I_MIX_H
#define I_MIX_H 1

#include <String.h>
#include <err.h>

class i_mix {
 public:
  i_mix();

  int ppc_count[5];
  int sT_count[8];
  int get_count_univ(int idx);

  String get_ppc_name(int index) {return ppc_names[index];};
  String get_sT_name(int index) {return sT_names[index];};

 private:

  String ppc_names[5];
  String sT_names[8];

};

inline
int i_mix::get_count_univ(int idx)
{
  if ((idx < 0) || (idx > 12))
    sigerr("i_mix::get_count_univ: Imix statistics out of range.");

  if (idx < 5)
    return ppc_count[idx];
  else
    return sT_count[idx-5];
};

inline i_mix::i_mix()
{
  char* names[13] = {"fixed point arith",
      "logical ops",
      "control ops",
      "floating pt arith",
      "memory ops",
      "fixed ALU ops",
      "floating ALU ops",
      "heap ops",
      "i/m struct ops",
      "scheduling ops",
      "network ops",
      "local mem ops",
      "reg alloc ops"};


  for(int i = 0; i < 5; i++)
    {ppc_count[i] = 0;
     ppc_names[i] = *(new String(names[i]));};

  for(i = 0; i < 8; i++)
    {sT_count[i] = 0;
     sT_names[i] = *(new String(names[i+5]));};


};

#endif
```

Appendix E

Appendix: fact.st

```
.proc   MAIN
.cb     MAIN

.cthread        MAIN.init
        carg    FP, 1
        mov     tr1, 1
        mov     tr2, 3
        store   tr1, FP, 3
        store   tr2, FP, 4
        cret
.endcthread

.inlet  MAIN.inlet1
        recv    ir12, ir11
        recvdone
        store   ir11, IFP, 6
        store   ir12, IFP, 5
        bnjoin  IFP, 3, label0
        post    MAIN.part0, IFP
label0:
        bnjoin  IFP, 4, label1
        post    MAIN.part1, IFP
label1:
        halt
.endinlet

.inlet  MAIN.inlet2
        recv    ir44, ir43
        recvdone
        store   ir43, IFP, 9
        store   ir44, IFP, 8
        bnjoin  IFP, 4, label2
        post    MAIN.part1, IFP
label2:
        halt
.endinlet

.thread MAIN.part0
        load    tr31, FP, 5
        load    tr32, FP, 2
        padd    tr32, tr32, 7
        mkcont  tr29, tr30, 0, tr32, FP
        mov     tr27, 10
        mov     tr28, FACT..CBD
        call    tr35, alloc_frame_local, tr28, tr29, tr30
        padd    tr34, tr28, 6
        mkcont  tr34, tr35, 0, tr34, tr35
        mov     tr24, 0
        load    tr34, tr34, tr24
        send    tr34, tr35, tr29, tr27
        store   tr35, FP, 7
        bnjoin  FP, 4, label3
        post    MAIN.part1, FP
label3:
        halt
.endthread

.thread MAIN.part1
        load    tr55, FP, 6
        load    tr57, FP, 7
        load    tr59, FP, 8
        load    tr62, FP, 9
        mov     tr58, 0
        mov     tr56, tr57
        mov     tr1000000, 0
        call    tr54, dealloc_frame_at, tr1000000, tr56
        mov     tr51, 0
        load    tr61, FP, 0
        load    tr62, FP, 1
        mov     tr50, 0
        load    tr61, tr61, tr50
        send    tr61, tr62, tr51, tr52
        halt
.endthread


.proc   FACT
.cb     FACT

.cthread        FACT.init
        carg    FP, 1
        mov     tr1, 1
        store   tr1, FP, 3
        cret
.endcthread

.inlet  FACT.inlet1
        recv    ir10, ir9
        recvdone
        store   ir9, IFP, 5
        store   ir10, IFP, 4
        bnjoin  IFP, 3, label4
        post    FACT.part0, IFP
label4:
        halt
.endinlet

.thread FACT.part0
        load    tr73, FP, 4
        load    tr34, FP, 5
        mov     tr32, 1
        cmp     tc1000001, tr32, tr34
        ble     tc1000001, 10
        mov     tr23, 0
        jump    11
10:
        mov     tr23, 1
11:
        cmp     tc1000002, tr23, 0
        beq     tc1000002, 12
        mov     tr35, tr34
        mov     tr46, 1
        mov     tr52, 1
        mov     tr47, 0
        mov     tr37, tr52
        mov     tr36, tr46
        mov     tr56, tr35
        mov     tr59, tr36
        mov     tr64, tr37
14:
        mul     tr40, tr59, tr64
        mov     tr58, 1
        add     tr60, tr59, tr58
        cmp     tc1000003, tr60, tr56
        ble     tc1000003, 15
        mov     tr62, 0
        jump    16
15:
        mov     tr62, 1
16:
        cmp     tc1000004, tr62, 0
        beq     tc1000004, 17
        mov     tr59, tr60
        mov     tr64, tr40
        jump    14
17:
        mov     tr24, tr60
        mov     tr25, tr40
        mov     tr71, tr24
        mov     tr18, tr25
        jump    13
12:
        mov     tr29, tr34
        mov     tr27, 1
        mov     tr26, 1
        mov     tr71, tr26
        mov     tr18, tr27
13:
        mov     tr19, 0
        load    tr74, FP, 0
        load    tr75, FP, 1
        mov     tr16, 0
        mov     tr17, 0
        load    tr74, tr74, tr16
        send    tr74, tr75, tr17, tr18
        halt
.endthread
```

Bibliography

[1] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye. The IBM RISC System/6000 processor: Hardware overview. *IBM Journal of Research and Development*, 34(1):12–22, January 1990.

[2] Bernstein, et al. Performance Evaluation of Instruction Scheduling on the IBM RISC System/6000. *IEEE Transactions 0-8186-3175-9*, IBM Isreal Scientific Center, The Technion City, 1992.

[3] Culler, et al. TAM—A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[4] Digital Equipment Corporation. *pixie(1)*. Ultrix 4.0 General Information, Vol. 3A (Commands(1): M-Z).

[5] Aaron J. Goldberg. Reducing Overhead in Counter-Based Execution Profiling. Technical Report: CSL-TR-91-495, Stanford University, Stanford, October 1991.

[6] G. Grohoski, J. Kahle, and L. Thatcher. Branch and Fixed-Point Instruction Execution Units. In *IBM RISC System/6000 Technology*, Order No. SA23-2619, pages 24–33. IBM, 1990.

[7] G. F. Grohoski. Machine organization of the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, January 1990.

[8] Carl Kesselman. *Tools and Techniques for Performance Measurement and Performance Improvement in Parallel Programs.* Dissertation, University of California, Los Angeles, 1991.

[9] James R. Larus and Thomas Ball. Optimally Profiling and Tracing Programs. *ACM 089791-453-8*, University of Wisconsin, Madison, 1992.

[10] R. R. Oehler and R. D. Groves. The IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):23–36, January 1990.

[11] B. Olsson, R. Montoye, P. Markstein, and M. Nguyenphu. RISC System/6000 Floating-Point Unit. In *IBM RISC System/6000 Technology*, Order No. SA23-2619, pages 34–43. IBM, 1990.

[12] Unix User's Reference Manual. *gprof(1)*. Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, Berkeley, California, 1986.

[13] Unix User's Reference Manual. *prof(1)*. Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, Berkeley, California, 1986.

[14] H. S. Warren, Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34(1):85–92, January 1990.