Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines

Timothy Heil James E. Smith
University of Wisconsin – Madison
Madison, WI 53706
{heilt, jes}@ece.wisc.edu

Abstract

Virtual machine service threads can perform many tasks in parallel with program execution such as garbage collection, dynamic compilation, and profile collection and analysis. Hardware-assisted profiling is essential for providing service threads with needed information in a flexible and efficient way. A relational profiling architecture (RPA) is proposed for meeting this goal.

The RPA selects particular instructions for profiling, and communicates collected information to service threads through shared memory message queues. The RPA's capabilities lead to new profiling applications, such as concurrent garbage collection.

Simulations indicate that a low-cost implementation of the RPA should be able to profile four in-flight instructions simultaneously, and provide storage for eight profile records. Profiling overhead is less than 0.5% for concurrent garbage collection and edge profiling.

1 Introduction

Two important trends are shaping the processing paradigm of the future. First, technology and the demand for higher performance are leading to on-chip multithreading, both within a large single processor and across multiple on-chip processors. Second, the emergence of binary translation, dynamic optimization, and virtual machine (VM) technologies is leading to a re-definition of the traditional hardware/software interface.

Our research is targeted at this future environment and is centered on the development and application of co-designed virtual machines. Co-designed VMs combine hardware and software to implement a virtual instruction set architecture on hardware directly supporting an implementation-specific instruction set architecture [10, 11, 12, 18, 20]. Co-designed virtual machines and on-chip multithreading complement each other very well. In particular, the co-designed VM paradigm naturally leads to a form of thread-level parallelism (TLP) where *service threads* perform tasks such as dynamic profile collection and processing, dynamic re-compilation, and garbage collection in parallel with program execution (which may also be multithreaded).

Efficient hardware-assisted profiling is central to the dynamic optimization paradigm. We have taken a top-down approach to hardware-assisted profiling. The goal is to develop profiling mechanisms flexible enough to satisfy not only known applications, but also future applications that will develop as the VM paradigm evolves. At the top level, the *relational profiling architecture* (RPA) summarizes and passes profile information to service threads in a flexible and efficient way, enabling them to perform their tasks.

We propose a *relational profiling model*, which provides the framework for designing the RPA. The relational profiling model allows software to form general queries regarding program behavior. These queries may request information regarding instruction types, hardware events, specific instructions, or ranges of instructions, as well as various combinations. A hardware implementation processes the queries, collects the requested information, and passes it back to software in the form of standard format messages. Service threads read these messages to perform optimizations and other tasks to support the main computation thread(s).

The relational profiling model is discussed in Section 2. Section 3 describes the RPA using examples. Section 4 discusses an implementation of the RPA and analyzes hardware costs at the microarchitectural level. Sections 5 and 6 evaluate the utility and performance of the RPA using one traditional profiling application, edge profiling, and one new application, concurrent garbage collection. Related work is described in Section 7, and Section 8 concludes the paper.

2 The Relational Profiling Model

The relational profiling model can uniformly specify the collection of a wide range of information. Conceptually, the relational profiling model is similar to a table in a relational database. See Figure 1. Each column represents a dynamic instruction; each row represents a possible event. This model leads to two basic forms of queries.

1) Instruction-based queries. "For certain instructions, what events occurred?" These queries select columns from the table. To collect this information, the profile mechanism essentially follows the instruction as it flows

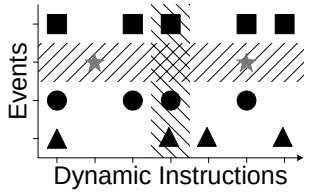

Figure 1. The relational profiling model organizes instructions and events in a table.

through the pipeline, collecting event information regarding its behavior. This is similar to ProfileMe [9].

2) Event-based queries. "For some events, what instructions were involved?" These queries select rows from the table. To collect this information, the profile mechanism essentially sits at some point(s) in the pipeline, recording information about instructions that flow past. This is similar to the counter profiling mechanisms common in processors today. In contrast to counter-based methods, however, the relational model can provide detailed information about specific dynamic instructions. Nevertheless, hardware counters may sometimes be useful as an efficient summarizing mechanism.

Hybrids queries are also possible. For instance, the definition of, "some instructions," in instruction-based queries may contain event-related conditions (i.e. "For all load instructions that miss in the cache...").

3 The Relational Profiling Architecture

Given the relational model, the goal was to produce a profile architecture that allows queries to be conveniently expressed, and that leads to an efficient implementation. This pursuit led to the development of the relational profiling architecture (RPA).

3.1 RPA Assembly Language

Profile queries are most easily expressed via an assembly language. An RPA assembly language program or *query* 1) describes records of information to be collected, 2) specifies a rate at which the information should be collected, 3) describes selection criteria for which a record should be checked, and 4) indicates actions to be taken for the selected records.

To facilitate RPA research, an assembler was developed using the ANTLR tool [16]. Unlike typical assembly languages, RPA queries invoke a number of machine level instructions that manage structures in the profile hardware. These structures are described in Section 3.4. Examples in Figure 2 describe the RPA and its usage.

An RPA program is broken into a series of queries. Each query begins with a *query header* that indicates which instructions should be profiled, what information should be collected and how often. Instructions are divided into the eight classes shown in Table 1. For example, the statement in Figure 2a specifies that conditional

```
a)  for opBRANCH * every 256 collect pc misc ;
        send 1 stop;

b)  for evL2DMISS * always collect pc res2 ;
        send 2 stop;

c)  for opSTORE 1 always collect op1 op2 op3;
        if op1 <> 0 then send 3 stop else stop;

d)  for opJMP  * opBRANCH * opLOAD *
            opSTORE * opALU * opMULT *
            opFLOAT  * opSYS *
            every 1024 collect pc rrate;
        send 4 stop;
```

Figure 2. Example RPA assembly language queries.

Table 1 Instruction profiling classes.

Mnemonic	Instructions
opJMP	Unconditional jumps
opBRANCH	Conditional branches
opLOAD	Load instructions
opSTORE	Store instructions
opALU	Simple arithmetic and logical instr.
opMULT	Multiply/divide instructions
opFLOAT	Floating point operations
opSYS	SYSCALL, BREAK, etc.

branch instructions should be profiled. Additional software-controllable classification is made available by a two-bit profile tag per instruction in the program binary being profiled. This yields a total of 32 instruction classes. Note that the VM paradigm allows instruction fields to be added to the implementation ISA. An alternative is to add additional hardware tables to hold software-controlled classification information. The statement in Figure 2c indicates that only stores with a profile tag of "1" should be profiled.

For the specified instruction classes the query header indicates the information to be collected and a random sampling rate. Simple random sampling reduces the rate at which profile information is collected. The header uses mnemonics from Table 2 to list the information collected. Types of collected information include both *architected* and *implementation* information. Each item in Table 2 represents one 32-bit word of information, which is collected and packed into a record by the RPA. The proposed RPA limits the information collected to seven words per record. This limits records to a manageable size, and still supports most profiling tasks. Example 2b collects the PC and Result 2 (the effective address) of L2 data misses.

Following the query header, *query clauses* select certain records and perform actions on them. Each query clause can perform up to two comparisons to check for properties within the record. Comparisons operate on 8,16 or 32 bit integer values. Both the size and location of the data within a record are encoded in the query comparison.

Table 2 Proposed information collectible by RPA.

Mnemonic	Information
pc	Instruction PC
thread	Thread ID
op1, op1up, op2, op2up, op3	Input operand values
res1, res1u, res2	Output results
misc	Exceptions and branch outcome
ftime	Cycle when instruction fetched
frate, drate, irate	Fetch, dispatch and issue rates
wlat	Execution window latency
elat	Execution latency
rtime	Cycle when instruction retired
rrate	Retire rate

Each comparison may use a 16-bit immediate value. Table 3 lists the comparison types available. All standard relational operations are available. Other comparisons check for set or cleared bits in the record or perform further random sampling. The example in Figure 2c checks if operand 1 has a nonzero value. If the comparison(s) match, an action may be performed, or there may be a branch to another query clause. Otherwise, execution falls through to the next sequential query clause. Query clauses can form if-then-else decision trees to compute arbitrary Boolean expressions. The *stop* keyword within the query clause indicates that query is completed.

Query clauses perform profile actions to communicate collected information to VM software. The most common action is the *message action*, indicated by the *send* keyword in the assembly code. The message action writes a copy of the record into a message queue where it can be examined by a service thread. Message queues are held in shared memory, and are accessed by service threads using normal loads and stores. Typically a single service thread is assigned to each queue to reduce synchronization between the service threads. The RPA can send different messages to different queues, so service threads can be specialized to a particular type of information. If processing power beyond a single service thread is needed, the RPA can disburse messages to multiple queues so multiple service threads can consume the messages.

A *counter action* increments a counter embedded in the query clause itself. This allows software to construct custom counters for arbitrary events using the query engine. When the counter is close to overflowing, the thread being profiled is interrupted.

An *interrupt action* interrupts a thread directly. The interrupt may be synchronous or asynchronous. For synchronous interrupts, the profiled instruction retires, but subsequent retirement stalls until the query is completed.

3.2 Examples

The four example queries shown in Figure 2 illustrate how the RPA can specify different profiling applications.

Table 3 Comparison types

Mnemonic	Comparison
< = >	All unsigned integer magnitude comparisons
BCLR	True if bits under mask are clear
BSET	True if bits under mask are set
FILTER 2/4/16/256	Random filtering. Succeed once in 2/4/16/256 executions.
TRUE/FALSE	Always/never succeed.

Example a) Edge Profiling -- Edge profiles, counts of how often conditional branches are taken and not-taken, are one of the most useful types of profiles, enabling or improving several important optimizations [2, 4, 26]. These profiles are also relatively easy to collect. Figure 2a shows the RPA assembly for the query. The *query header* indicates the PC and branch outcome (taken/not-taken) result as well as a branch misprediction flag (*misc*) for branch instructions should be collected. Random sampling is used to select one out of 256 branches. This query performs no comparisons. It simply sends the message to a service thread(s) that tabulates the information.

Example b) Prefetching -- It has been proposed that data prefetching can be done by *assistant threads* or *nanothreads* [19] which are essentially types of service threads. An RPA query for performing this profiling is in Figure 2b. The RPA is used to monitor L2 cache misses. For each L2 cache miss the instruction PC and effective address are collected. These records are sent to a service thread that executes prefetch instructions on behalf of the application. This is straightforward if the service thread and application thread share the L2 cache. In other situations, a dynamic optimizer running in a service thread could insert prefetch instructions into the binary, or the service thread could configure an existing hardware prefetch mechanism.

Example c) Garbage Collection (GC) -- Figure 2c shows the RPA query used in a low-overhead concurrent GC algorithm we proposed [13]. GC is the process of automatically reclaiming memory that is no longer needed by an application. Details of the algorithm are given in Section 5. Briefly, to perform GC without stopping the application, the GC thread must monitor certain application stores. This is typically done by adding *store-barrier* code around each store in the application, thereby slowing the application. RPA solves this problem by profiling all instructions that store references to the heap. By using the RPA, the time overhead of GC was reduced to 0.6% on average [13]. In Figure 2c, input operand 1, op1, contains the value being written. If this value is not null (zero), then the record is passed to GC service threads that execute the store barrier code.

Example d) Concurrency Metrics -- ProfileMe [9] uses paired sampling to estimate *concurrency metrics*, such as wasted issue slots in the vicinity of a given instruction. The RPA can compute similar concurrency metrics without using paired sampling. For instance, the query in Figure 2 can be used to compute cycles-per-instruction (CPI) for individual instructions and regions, such as loops and functions. The query collects the PC and retirement rate data for all instruction types. Retirement rate data, *rrate* in Figure 2d and Table 2, contains two basic pieces of information. The first is the number of instructions, *n*, that retired during the same cycle as the sampled instruction. The second is the number of preceding cycles in which no instructions retired, *c*. These cycles can be attributed to the sampled instruction only if it was the head instruction during these stalled cycles. Hence, *c* is zero if the sampled instruction was not the first instruction retired after the stalled cycles.

The execution cycles for one instruction sample is computed as $C = c+1/n$. The total cycles spent executing the instruction can be computed by summing all the samples, and dividing by the sample rate. The CPI for one instruction can be computed as the arithmetic average over all samples collected for that instruction.

The total time spent in a region of code can be computed by summing the total cycles for each instruction in the region. The CPI over a region can be computed using an arithmetic average weighted by relative instruction execution frequency.

To locate bottlenecks, RPA collects not only the number of retirement stall cycles (*c*), but categorizes stall cycles by the reason for the stall. Example categories are an empty window, the head instruction not yet issued, or the head instruction not yet completed. Because stalls often have multiple causes, information gained from such categorization is usually approximate. The RPA can collect stall information at other stages in the pipeline as well.

3.3 Simplifications

In the example applications given above, the queries are very simple -- one query clause for each, and message passing is used exclusively. We expect this to be the common case. Essentially, the query engine filters unnecessary messages, and passes only the necessary records on to service threads for more involved processing. This reduces the work for the service threads. A lower-cost design could replace the query engine with a table able to do a small fixed number of comparisons per record, followed by a message action.

In addition, messaging may make counter actions redundant. Counters can be maintained by service thread software. While this is less efficient than the counter increment operation, simple random sampling can reduce counter increments to a reasonable rate and still maintain good counter accuracy. Rare events use a high frequency sampling rate, while common events require a low sampling rate.

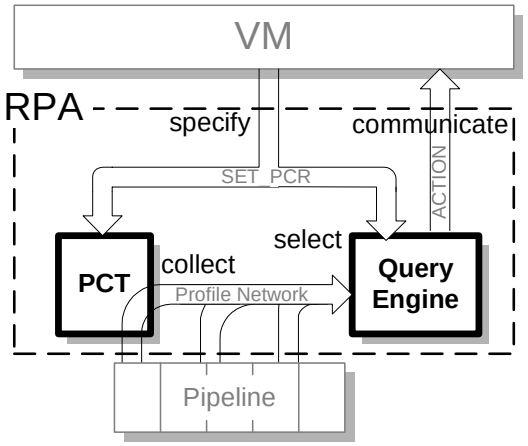

Figure 3. The RPA contains the profile control table (PCT) and the query engine.

3.4 Low-Level Architecture

Figure 3 shows an implementation of the RPA. Specific queries are formed by software -- virtual machine software, using the query language as described in the preceding section. The assembler divides query processing into two components shown in the figure. A configuration for the *profile control table* (PCT) is derived from the query headers. The PCT is a set of architected *profile control registers* (PCRs). Software loads the PCT using a special SET_PCR instruction, also used to configure other parts of the RPA.

Each query clause generates one *query instruction* to be executed by the *query engine*, a simple processor capable of performing the comparisons and actions dictated by the query. Query instructions may be stored in memory, or, to reduce implementation costs, in a special-purpose table constructed out of PCRs.

3.4.1 The Profile Control Table (PCT). The PCT implements two PCRs for each of the 32 instruction classes. The first PCR sets the sampling rate and selects information from Table 2 to be collected. The second PCR contains the starting *query PC* (QPC), the address of the initial query instruction for the query engine to execute.

The PCT also controls event-based profiling. Theoretically, event-based profiling hardware could be dispensed with entirely by collecting instruction records and using the query engine to select only those instructions with the desired event. For rare events this yields very long profile times, however. The proposed RPA specifies event-based profiling for ITLB, DTLB and L2 cache misses. Two PCRs are provided for each of these three events. Due to implementation constraints, only a subset of the informa-

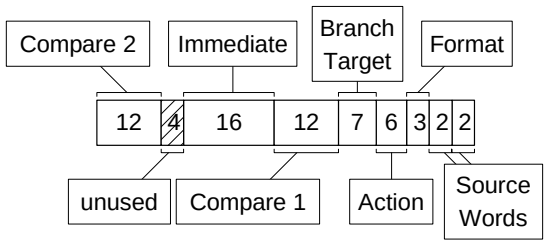

Figure 4. Query instructions contain up to two comparisons, a branch and an action.

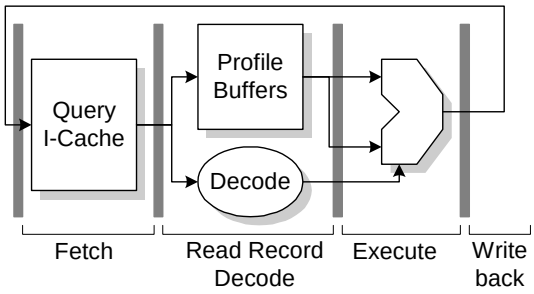

Figure 5. The query engine four stage pipeline.

tion in Table 2 may be collected for event-based profiling. The processor implementation may not detect an instruction needing to be profiled until the event occurs, which is likely to be too late to collect information from earlier pipeline stages. At least the instruction PC, a cycle count, and the effective address are expected to be available.

3.4.2 The Query Engine. Profile records collected at the behest of the PCT are passed to the query engine, which begins executing query instructions at the initial QPC location indicated in the PCT. Query instructions perform the comparisons, branches and actions specified by the equivalent query clauses.

Query instructions are encoded in a 64-bit two-comparison format shown in Figure 4. Instructions encode two comparisons, a 16 bit immediate value, branch target and an action. The query instruction reads two values from the record. To reduce implementation costs, one even word and one odd word from the profile record are selected.

The application program continues to execute in parallel with the query engine, and multiple comparisons may be executed in parallel. To simplify the implementation there is no guarantee of the order in which separate queries are executed or completed.

To implement message actions, the query engine manages message queues in memory. Each message uses eight words of memory, and the size of the queue is configurable up to 128 messages. To write a record into a queue, the query engine writes the seven words of the record. The eighth word is used to indicate that the record is available. To read a record, a service thread polls the ready word, reads the record from memory, and clears the ready word. The service thread also informs the query engine how many messages have been read by storing the total messages read into the *buffer read status* word in memory. The buffer read status word is examined periodically by the query engine. The query engine uses the number of messages read, along with the number written (which it knows), to determine if space is available for another message, reducing polling by the query engine.

4 RPA Implementation and Cost

The query engine pipeline that we simulated is shown in Figure 5. The query engine is designed to execute up to four queries simultaneously using a simple barrel-and-slot design [21]. The pipeline is four stages long, and one query instruction is executed for each active query once every four cycles in a round-robin fashion. The barrel-and-slot design eliminates all interlocks and dependences in the pipeline. As the profile records fill, the processing power of the query engine increases to one query instruction per cycle.

The first pipeline stage fetches the query instruction from the small query instruction array. At 512 bytes, this cache holds 64 instructions, which appears to be plenty. Decode is performed in stage 2. The record buffers are also accessed in this cycle. They are word-interleaved; the query engine accesses one word from the even bank, and one word from the odd bank each cycle. The query ALU performs the comparison(s) in stage 3 using a masked magnitude comparator. The fourth stage selects and drives the next query PC back to the fetch stage. Query actions are also initiated in the write-back stage.

An interconnection network carries profile information from the pipeline to the profile buffer. Though a complete design of this network is beyond the scope of this paper, the size of the network will scale linearly with the number of simultaneously profiled instructions [9]. The latency of this network is not a concern; the profile buffers can be relatively distant from the core pipeline.

A typical implementation contains a number of identical profile networks to carry information from the pipeline to the profile buffers. The number of networks determines how many in-flight instructions can be profiled simultaneously. Instructions are selected for profiling during instruction dispatch, and a profile network and profile buffer are allocated to the instruction. Instruction dispatch stalls if either is unavailable. Hence enough of each should be provided to make this a rarity. When the instruction retires the profile network is freed, but the profile buffer remains allocated until the query completes.

The fixed costs of the RPA are the 280-byte PCT and

the query engine with its 512B I-cache. Two remaining important cost variables are the number of profile networks and the number of 32-byte profile buffers. These are evaluated in the following sections.

5 Simulated Applications of the RPA

To determine how many profile networks and buffers are needed for low-overhead profiling, we simulated two example applications. These are the edge profiling and the concurrent GC applications shown in Figure 2.

The concurrent GC algorithm is described in detail in [13]. A short overview is provided here. Object-oriented languages such as Java organize memory into objects that contain *references* to other objects. This forms a graph with objects as nodes and references as directed arcs between the nodes. Tracing GC finds all objects reachable, directly or indirectly, from a set of *root references* -- object references in global or local variables. This reachability analysis involves *marking* objects directly reachable from the roots, and then iteratively marking all objects reachable (via references) from previously marked objects. This iterative processes terminates when no more unmarked objects can be reached from marked objects. At this point all unmarked objects are unreachable by the application and can be collected, freeing the storage space for future objects.

Concurrent GC algorithms must handle modifications to the object graph performed concurrently by the application to avoid erroneously collecting reachable objects. The RPA supports this function by profiling store instruction addresses and values. This information is sent to *store barrier* service threads that perform simple book keeping operations for the garbage collector, running on another service thread. Storing a reference to an object causes a store barrier thread to mark the object live so it will not be collected. This is called an *incremental update* mechanism in Wilson's survey of GC algorithms [24].

Only instructions that store object references need to be profiled. The VM distinguishes these stores from others using the profile tag described in Section 3.1. In addition, if a null reference store is performed, no profiling is needed. Hence, the query discards profile records in which the stored value is null (zero). Because the GC algorithm relies on observing every non-null reference store, random sampling is not appropriate.

The result is an extremely efficient garbage collector. Using the RPA, GC overhead is reduced to 0.6% of the total runtime [13].

5.1 Simulation Methodology

Simulation experiments are based on the Strata VM research infrastructure and the SimpleMP simulator [17]. SimpleMP is a version of the SimpleScalar [3] execution-driven timing simulator that was extended to simulate multiple processors. We extended it further to simulate multithreaded processors and the RPA. The Strata VM contains a static compiler from Java bytecodes to SimpleScalar PISA assembly code. The Strata compiler is itself written in Java and forms one of our benchmarks.

Table 4 Benchmark characteristics.

	Inputs	Duration Instr. (M)	Duration Cycles (M)
strata	Database	517	222
db	input/db2 input/src3	206	79
jack	10	648	234
javac	-verbose JavaLex.java	432	190
jess	zebra.clp wordgame.clp	596	220
raytrace	50 500 time-test.model	246	84

5.2 The Strata Compiler

The Strata compiler performs typical optimizations such as global register allocation, constant propagation, local common sub-expression elimination and global copy propagation. It also performs Java specific optimizations aimed at eliminating null-pointer checks, type checks and array bounds checks.

The runtime system, which contains the GC algorithm, is written in C. Running a Java application involves compiling the bytecodes using the Strata compiler, and then linking the resulting assembly with the runtime system.

5.3 Benchmarks

Six benchmarks shown in Table 4 are simulated. The first is the Strata compiler. The other five are taken from the SPECjvm98 suite. They are *jess*, an expert system, *raytrace*, a 3-D rendering tool, *db*, a simple relational database, *javac*, the Java compiler, and *jack*, a parser generator. Simulations began at about 10 million instructions before the first GC, until completion, except for *javac*, which was terminated at 200M cycles.

5.4 Processor models

Ways to exploit thread-level parallelism include chip multiprocessing (CMP) [15] and multithreading. The particular design explored in this paper, Figure 6, uses fine-grain multithreading (FGMT) [1] and CMP. On one chip, there is a large high-ILP processor supplemented by three service processors. The computation of greatest concern, the application, runs on the high-ILP processor. Lower priority VM tasks run on three service processors concurrently with application execution. One service processor consumes the edge profile information. A second processor consumes the store-barrier records for the garbage collector. A third service processor executes the garbage collector.

Parameters for the processor models are shown in Ta-

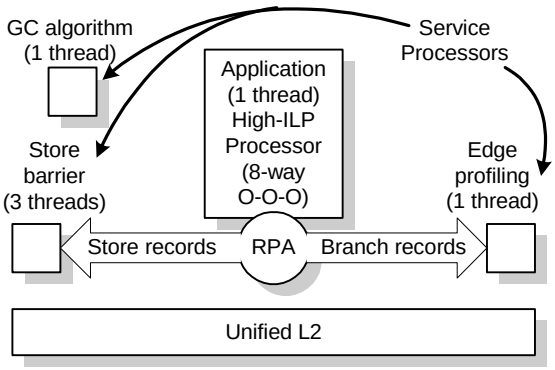

Figure 6. The system on a chip

Table 5 Processor model parameters

Parameter	High-ILP Proc.	Service Proc.	Units
Number	1	3	Proc.
Threads	1	3	Threads
Width	8	1	Instr.
Instr. Win.	128	(in-order)	Instr.
Br. Pred.	4KB gshare	Not-taken	
Min. penalty	8	4	Cycles
I-Cache	32KB 2-way	1KB 4-way	
D-Cache	64KB 4-way	2KB 4-way	
Unified L2	Perfect		

ble 5. The ILP processor is an 8-way super-scalar, out-of-order processor, running only one thread. The service processors are designed to maximize throughput per unit area, rather than single-thread performance. Each service processor is a six stage scalar pipeline capable of running three threads using FGMT. To keep these processors small and simple, they have small L1 caches and predict-not-taken branch prediction. All three processors connect to a perfect L2 with a 12 cycle round-trip access time.

6 Results

This section uses simulations to determine the number of profile buffers and networks needed by the profiling hardware. Further simulations examine the performance overhead of profiling and the sampling rates that the RPA implementation can support.

6.1 Resource Usage

Figure 7 shows results from simulating the edge profiling and GC queries from Figure 2. One out of 256 branches are sampled. Profile resources were essentially unrestricted with 32 profile buffers and 32 profile networks. The histograms on the left show the percentage of cycles (y-axis) in which at least *x* profile buffers were in use (x-axis).

The benchmarks *raytrace*, *db* and *jess* stress the RPA very little, rarely using more than a single profile buffer. Hence we omitted these plots. The three benchmarks *strata*, *javac* and *jack* load the RPA more heavily. We observe that eight profile buffers, 256 bytes of storage, appear to be sufficient. Only *jack* can utilize more than eight buffers, although eight are nearly enough.

The right histograms show the number of profiled instructions simultaneously in flight. The histograms show the percentage of cycles (y-axis) in which at least *x* profiled instructions were in flight (x-axis). This includes profiled instructions between the dispatch and retire stages. Under the interconnect model described in Section 4, one profile network is need for each in-flight instruction. The histograms are very similar to the left histograms, though they are slightly lower because profile buffers remain allocated longer than the flight time of an instruction. The benchmarks *strata* and *javac* should run well with four profile networks. The *jack* benchmark appears to need as many as eight. Direct measurement of profiling overhead in the next section shows far fewer networks are actually needed.

6.2 Profiling Performance Overhead

Based on the previous results, the number of profile buffers was reduced to eight, and the number of profile networks was varied from one to four. Figure 8a plots the percentage of cycles for which dispatch was stalled due to limited RPA resources. Only *jack* and *strata* are plotted in Figure 8; *javac* generally shows about half the stalls of *strata*, and the other benchmarks show very few or no stalls. Other simulation results not presented here indicate that limited profile networks cause these stalls. *Strata* stalls vary from 3.8% with one profile network, to 0.5% with four networks. *Jack* stalled dispatch 12.4% of the time with a single profile network. Despite the results in Figure 7, however, stalling was reduced to only 1.7% with four networks.

Figure 8b plots the percent slow-down resulting from these stalls, as compared to the unrestricted results in Section 6.1. Note that the y-axis scale changes between Figures 7 and 8; generally about half of the dispatch stalls are covered by the out-of-order execution window. Three profile networks reduce profiling overhead below 0.5% for all benchmarks. Four networks make it effectively zero. In fact, *jack* shows a slight speedup (negative slow-down), likely resulting from system-level interactions with the garbage collector.

To understand how profiling overhead varies with the sample rate, four sampling rates for edge profiling were simulated; rates simulated were one in 32, 64, 128 and 256 branches. Note that the GC query is still included. Figure 8c plots the slow-downs for these rates. *Strata* and *jack* show a mild increase in profiling overhead as the profiling rate is increased. This suggests that the RPA

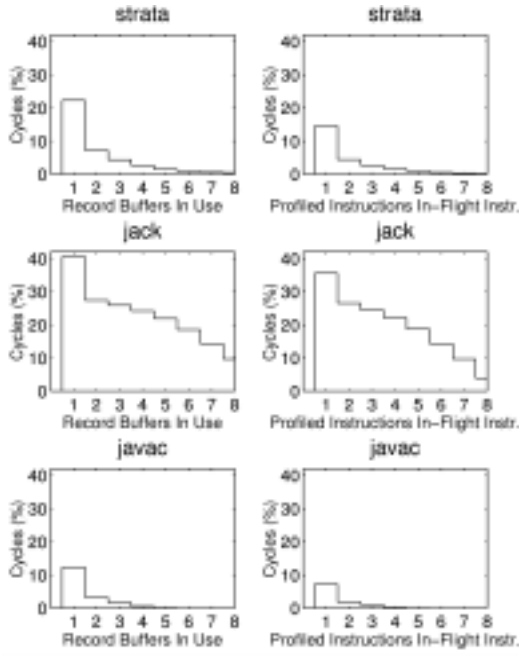

Figure 7. Resources used by the RPA for *strata*, *jack* and *javac*.

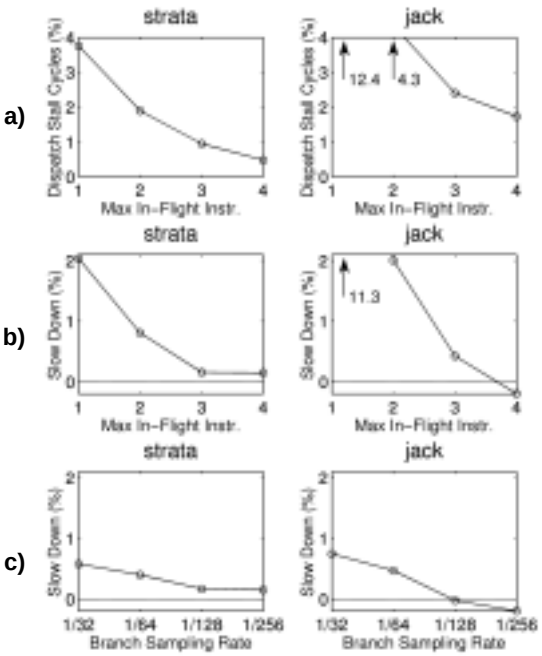

Figure 8. Profiling overhead for the *strata* and *jack* benchmarks. a) Percentage of cycles that the RPA stalled dispatch. b) Slow-down resulting from profiling. c) Variation in the slow-down with the sampling rate.

should monitor its own behavior. Sample rates can be throttled back if dispatch stalls increase.

Table 6 shows the load that these rates place on the profile mechanism. Table 6 contains the instructions executed per profiled instruction, and the cycles per profiled instruction. The instructions executed per cycle (IPCs) are generally a little above two for these benchmarks and this CPU model, so instructions-per-sample is about twice the cycles-per-sample.

Comparing Table 6 to Figure 8 shows *jack* can profile about one in 41 instructions, or one sample every 15 cycles. At higher rates, profiling begins to stall the processor. *Strata* also has measurable stalls for higher sampling rates, though Table 6 shows many fewer instructions are profiled than *jack* due to variation within the benchmark. Stalls in *strata* occur during bursts of profiling for the GC application.

6.3 Reducing Profile Stalls

The above results are based on a network model that collects information throughout the pipeline. However, information available at instruction retirement may often be sufficient. Such information could include the instruction PCs, branch targets and directions, and effective addresses. If a profile record only contains retire-time information, it is wasteful to allocate a profile network for the entire lifetime of the instruction.

Table 6 Profiling rates for different branch sampling rates.

	Instr. per sample				Cycles per sample			
	1/32	1/64	1/128	1/256	1/32	1/64	1/128	1/256
strata	110	149	182	205	47	64	78	88
db	113	173	237	290	44	67	91	112
jack	35	38	41	42	13	14	15	15
javac	128	212	317	420	57	94	139	184
jess	166	299	499	750	62	111	185	277
ray	289	468	676	869	99	160	230	296

This leads to a model that supports both cheap and expensive profiles. A cheap profile only contains information available during retirement. Instructions profiled this way must allocate profile buffers, but need not allocate a profile network. An expensive profile includes information not available during retirement and must allocate a profile network, as assumed in the above results. Mechanisms that collect all information for all instructions, like ProfileMe, cannot take advantage of this optimization.

7 Related work

Special-purpose mechanisms have been proposed for many of the profiling operations discussed above. These require special hardware, software, or both.

Conte et al. propose two hardware methods [6, 7] for edge profiling. The first samples the values of the branch-target-buffer and branch prediction array to derive estimates of the edge profile. The second method improves the accuracy using a small special-purpose array, the *Profile Buffer*, to count taken and not-taken branches indexed by PC. In both methods the tables are periodically read by software using interrupts.

Merten et al. [14] develop a scheme for identifying *hot spots* in programs. Their scheme works by collecting branch taken/not-taken counts in the Branch Behavior Buffer (BBB), a structure similar to the Profile Buffer, though much larger. The BBB also identifies frequently seen branches and uses this information to identify hot spots. A separate structure, the Monitor Table prevents hot spot re-detection.

As noted earlier, Song and Dubois [19] investigate data prefetching with *assisted execution*, a service thread-like paradigm. Both sequential and stride prefetching are implemented with assisted execution. Simultaneous Subordinate Microthreads (SSMT) [5] is another paradigm similar to service threads. SSMT executes multiple *microthreads* in parallel with the application using simultaneous multithreading [8, 21, 25].

Several elements of the ProfileMe mechanism [9] also appear in the RPA. ProfileMe provides a simplified form of instruction-based profiling. The hardware picks one instruction from the stream, using a software settable decrementing counter. Simple random sampling is performed by resetting the counter to randomized values. ProfileMe allows information to be collected for both retiring and squashed instructions.

To keep costs low, ProfileMe keeps sampling rates low, e.g. one sample every 10^3 to 10^5 instructions, much lower than the one in forty instructions sampled by the RPA. Although more research is needed to determine whether higher sampling rates will benefit dynamic optimization, some non-traditional applications, such as concurrent GC, will require a much higher sampling rate.

ProfileMe also supports *paired sampling*, a simple and powerful form of clustered sampling. Paired sampling collects two instruction samples close to each other using a *major* and *minor* sampling interval. Paired sampling is useful for measuring interactions between instructions.

RPA is not incompatible with paired sampling, and future research will consider adding paired sampling to the RPA. Paired sampling does have some drawbacks. First, it will likely increase the number of in-flight instructions that must be tracked. Second, paired sampling can further increase the profiling latency if representative samples must be gathered for all nearby pairs of instructions. Furthermore, the RPA can gather some of the same information as paired sampling by collecting the proper per-instruction data, as illustrated by the concurrency metric example in Section 3.1.

A method described in a patent by Westcott and White [23] is also very similar to both RPA and ProfileMe. The processor uses a counter to randomly select instructions for profiling. Like ProfileMe, a standard profile record is collected. Like RPA's query engine, *triggers* can be used to scan for records of interest. Also like RPA, the records are stored to a buffer in memory. However, the buffer is read following an interrupt when the buffer fills.

Contemporaneous with this work, Zilles and Sohi developed a profiling mechanism with many similarities to the RPA [27]. It selects instructions for profiling based on their opcode class and the low-order bits of the instruction PC using a *hardware filter* similar to the PCT. Collected information is held in a *sample buffer* (like RPA's profile buffers), until examined by a programmable *profile co-processor*. This co-processor summarizes information before it is relayed to software via an interrupt. The co-processor is a more complete processor than the query engine, having registers, a private data memory and an associative array for hash tables. The RPA off-loads much of this computation to service threads.

The RPA has some additional capabilities. Working in the context of service threads leads RPA to focus on communicating profiled information through shared memory. All previous profile mechanisms use interrupts to record or sample profiled information. More importantly, the RPA can guarantee some instructions are always profiled. This allows the profiler to be used for new applications that require correctness.

8 Conclusions

The relational profiling architecture (RPA) provides a powerful and flexible profiling mechanism. It can enable the same optimizations as several previous mechanisms, and has additional capabilities. Since the RPA can guarantee that certain instructions are always profiled it is useful when the information is required for correctness. Working in the context of service threads leads to messages communicating profile information, which increases the sampling rate and lowers profiling overhead, compared to previous interrupt-driven approaches. This leads to new applications for hardware-assisted profilers such as concurrent garbage collection.

Simulation results shed light on the implementation requirements and profiling overhead of the RPA. Profiling four simultaneous in-flight instructions and storing eight profile records is sufficient to make losses negligible. Eight profile records require only 256 bytes of storage. The profile interconnection network, which scales linearly with the number of simultaneously profiled instructions,

appears to be the greatest cost of profiling. The PCT reduces this cost by selectively profiling only the particular data that is needed. This reduces the size of the profile records, and could also be used to intelligently allocate profile resources.

The PCT allows the sampling rate to be tuned to match the frequency with which the profiled event occurs. For common events, low sampling rates reduce the bandwidth required. For rare events, high sampling rates reduce the time required to obtain a representative sample. In addition, different benchmarks stress the profiling mechanism to different degrees. This suggests that the RPA should monitor its own behavior to adapt the sampling rate to the application.

The query engine is a powerful and relatively low-cost mechanism for selecting and communicating collected profile information. Queries are generally simple and result in sending the profile record to a service thread. In fact, a simple table-driven engine which can do a small fixed number of comparisons, conditionally followed by sending the record to a service thread may be sufficient. The RPA effectively enables a virtual machine to exploit the abundant thread-level parallelism expected to be available on-chip in the future. This configuration will reduce the overheads of VM technology by running VM tasks such as dynamic compilation and garbage collection concurrently.

Acknowledgements

This work was supported by NSF Grant CCR-9900610, by Sun Microsystems, by an IBM Partnership Award, and by Intel Corporation.

References

[1] R. Alverson et al., "The Tera Computer System," 1990 Intl. Conf. on Supercomputing, pp. 1-6, 1990.

[2] R. Bodik, R. Gupta, M. L. Soffa, "Complete Removal of Redundant Expressions," 1998 Conf. on Programming Language Design and Implementation, pp.1-14, June 1998.

[3] D. C. Burger, T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," Univ. of Wisconsin - Madison Comp. Sci. Tech. Rep. #1342, June 1997.

[4] P. P. Chang, S. A. Mahlke, W. W. Hwu, "Using Profile Information to Assist Classic Code Optimizations," Software-Practice and Experience, vol. 21, pp. 1301-1321, Dec. 1991.

[5] R. S. Chappel et al., "Simultaneous Subordinate Microthreading (SSMT)," 26th Intl. Symp. on Computer Architecture, pp. 186-195, May 1999.

[6] T. M. Conte, B. A. Patel, J. S. Cox, "Using Branch Handling Hardware to Support Profile-Driven Optimization," 27th Intl. Symp. on Microarchitecture, pp. 12-21, Nov. 1994.

[7] T. M. Conte, K. N. Menezes, M. A. Hirsch, "Accurate and Practical Profile-Driven Compilation Using the Profile Buffer," 29th Intl. Symp. on Microarchitecture, pp. 36-45, Dec. 1996.

[8] G. E. Daddis, Jr., H. C. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors," Intl. Conf. on Parallel Processing, pp. I:76-83, Aug. 1991.

[9] J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," 30th Intl. Symp. on Microarchitecture, pp. 292-302, Dec. 1997.

[10] K. Ebcioglu, E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," IBM Research Rep. RC 20538, Aug. 1996.

[11] M. Gschwind et al., "Dynamic and Transparent Binary Translation," Computer, 33(3):54-59, Mar. 2000.

[12] A. Klaiber, "The Technology Behind Crusoe Processors," a Transmeta technical brief, 2000.

[13] T. Heil, J. E. Smith, "Concurrent Garbage Collection Using Hardware Assisted Profiling," to appear Intl. Symp. on Memory Management, Oct. 2000.

[14] M. C. Merten et al., "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," 26th Intl. Symp. on Computer Architecture, pp. 136-147, May 1999.

[15] K. Olukotun et al., "The Case for a Single-Chip Multiprocessor," 7th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems, pp. 2-11, Oct. 1996.

[16] T. Parr, ANother Tool for Language Recognition (ANTLR), available at http://www.ANTLR.org.

[17] R. Rajwar, A. Kagi, J. Goodman, private correspondence. The SimpleMP simulator was produced by the Galileo group at the University of Wisconsin - Madison.

[18] J. E. Smith, T. Heil, S. Sastry, T. M. Bezenek, "Achieving High Performance via Co-Designed Virtual Machines," Intl. Workshop on Innovative Architecture, pp. 77-84, Oct. 1999.

[19] Y. H. Song, M. Dubois, "Assisted Execution," Tech. Rep. CENG 98-25, EE-Systems, University of Southern California, Oct. 1998.

[20] "MAJC Architecture Tutorial", Sun Microsystems White Paper, May 1999.

[21] J. E. Thornton, "Design of a Computer – The Control Data 6600," Scott, Foresman and Co., 1970.

[22] D. M. Tullsen et al., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," 23rd Intl. Symp. on Computer Architecture, pp. 191-202, May 1996.

[23] D. W. Westcott, V. White, "Instruction Sampling Instrumentation," U.S. Patent #5151981, assigned to IBM, Sept. 1992.

[24] P. R. Wilson, "Uniprocessor Garbage Collection Techniques," 1992 Intl. Workshop on Memory Management, pp. 1-42, Sept. 1992.

[25] W. Yamamoto et al., "Performance Estimation of Multistreamed, Superscalar Processors," 27th Hawaii Intl. Conf. on System Sciences, pp. I:105-204, Jan. 1994.

[26] C. Young, D. S. Johnson, D. R. Karger, M. D. Smith, "Near-Optimal Intraprocedural Branch Alignment," 1997 Conf. on Programming Language Design and Implementation, pp. 183-193, June 1997.

[27] C. Zilles, G. Sohi, "A Programmable Co-processor for Profiling," to appear 7th Intl. Symp. on High Performance Computer Architecture, Jan. 2001.