Architectural Support for Online Program Instrumentation

Kapil Vaswani Y N Srikant T Matthew Jacob

IISc-CSA-TR-2003-2
http://archive.csa.iisc.ernet.in/TR/2003/2/

Computer Science and Automation
Indian Institute of Science, India

February 2003

Architectural Support for Online Program Instrumentation

Kapil Vaswani, Y. N. Srikant and T. Matthew Jacob
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, India
{kapil,srikant,mjt}@csa.iisc.ernet.in

Abstract

For advanced profile-guided optimizations to be effective in online environments, fine-grained and accurate profile information must be available at low cost. In this paper, we propose a generic framework that makes it possible for instrumentation based profilers to collect profile data efficiently, a task that has traditionally been associated with high overheads. The essence of the scheme is to make the underlying hardware aware of instrumentation using a special set of profile instructions and tuned micro-architecture. This not only allows the hardware to provide the runtime with mechanisms to control the profiling activity, but also makes it possible for the hardware itself to optimize the process of profiling in a manner transparent to the runtime.

We propose *selective instruction dispatch* as one possible controlling mechanism that can be used by the runtime to manage the execution of profile instructions and keep profiling overheads under check. We propose *profile flag prediction*, a hardware optimization that complements the selective dispatch mechanism by not fetching profile instructions when the runtime has turned profiling off. The framework is light-weight and flexible. It eliminates the need for expensive book-keeping, additional recompilation or code duplication. Our simulations with benchmarks from the SPEC CPU2000 suite show that overheads for call-graph and basic block profiling can be reduced by 72.7% and 52.4% respectively with a negligible loss in accuracy.

Keywords

profiling, instrumentation, profile-guided optimization

I. Introduction

OVERCOMING the performance penalties associated with the dynamic compilation model in general and runtime systems in particular has been a target of much research recently. It is in this context that profile-guided optimizations have emerged as important tools in our bid to achieve performance comparable to what can be achieved using the traditional compilation model. The availability of online profile information in dynamic environments makes profile-guided optimizations more effective since optimization decisions are based on profile data representing the behavior of the program in its current execution. Furthermore, it is possible for optimizing compilers to generate high quality context-sensitive code by monitoring program characteristics over the entire length of the program's execution and adapting to changes in program behavior.

The applicability of profile-guided optimizations is governed by a complex cost-benefit trade off. Collecting fine-grained profile information such as call-graph and path profiles, for even a subset of methods involves substantial costs, both in terms of execution time and profiler design complexity. This factor has precluded the use of advanced profile-guided optimizations that rely on fine-grained profile information.

In this paper, we propose a generic framework that reduces instrumentation costs through a higher level of interaction between the compiler, the runtime and the underlying hardware. The framework incorporates instruction set support for online instrumentation in the form of *profile instructions*. These instructions allow the processor to explicitly identify instrumented code. We propose dedicated architectural support for online profiling in the form of *selective instruction dispatch* and *profile flag prediction* *mechanisms*. With support from the compiler, selective instruction dispatch allows the runtime to dynamically control the execution of profile instructions by switching between profiling and non-profiling modes of execution and hence regulate the profiling activity in the program. Profile flag prediction is a hardware optimization that fetches or skips a group of profile instructions based on the likelihood of their execution. Our framework (1) can be easily incorporated into existing architectures without substantial hardware costs, (2) supports all types of instrumentation based profiles and (3) involves no code duplication or code management overheads making advanced optimizations feasible in cost-sensitive dynamic recompilation environments.

The rest of the paper is organized as follows. Section II discusses the context and motivation for this work. In section III, we present the design aspects of the selective instruction dispatch mechanism. We take a step further in Section IV and propose a profile flag prediction mechanism that complements selective instruction dispatch to improve the efficiency of our technique. Section V presents a brief discusion on how our techniques can be scaled to support multiple types of profiles simulataneously. Section VI discusses the simulation methodology we used to evaluate our ideas. We present results of our detailed simulations with basic block and call-graph profiles as case studies in Section VII and conclude with section VIII.

II. Background and Motivation

Although most traditional profiling techniques are based on software mechanisms and instrumentation, there has been a flurry of activity over the recent years on hardware support for profiling. Conte *et. al* [1] propose an approach that uses the branch handling hardware present in most commercial processors for profiling purposes. The profiler samples the branch prediction hardware using kernel mode instructions at every kernel entry point to construct weights of a Weighted Control Flow Graph. Merten *et. al*[2] describe a mechanism that monitors instructions at the retirement stage of the processor pipeline to identify program hotspots. ProfileMe[3] is a hardware framework that profiles instructions by tracking their progress through the pipeline and collecting information about associated events. The Relational Profiling Architecture[4] provides mechanisms that allow the software to issue queries about information regarding program behavior These queries are serviced by profiling hardware that monitors the pipeline. Zilles and Sohi[5] suggest a solution that advocates dedicated hardware for profiling in the form of a programmable profiling co-processor that receives profile samples from the main processor, processes it and feeds it back to the main processor.

In general, these hardware profilers have advantages of low overheads and non-intrusiveness but they are restricted by the profiling capabilities built into the processor hardware and cannot cater to the range of profiling requirements raised by dynamic environments of the present and future. Moreover, these techniques are usually platform dependent, which makes it difficult to develop profilers for runtime systems that support multiple architectures.

We therefore believe that software techniques such as instrumentation will continue to play an important role in the design of profilers for dynamic environments. It has been recognized that instrumentation based profilers must be complemented with simple mechanisms that allow runtimes to control instrumentation overheads[6]. These overheads can be broken down into

- Overheads associated with determining where, what and how to instrument: Instrumentation procedures for most types of profiles do not require significant time and effort. However, in cases such as edge and path profiles, increasing the efficiency of the instrumented code might involve complex analysis[7][8].
- Recompilation space and time overheads: Some instrumentation strategies create duplicate instrumented copies of code and mechanisms for transferring control to the instrumented copies when profiling is to be performed.

- Overheads due to instruction and data cache pollution: The presence of the instrumentation causes code expansion and could degrade instruction cache performance. Also, the instrumented instructions data accesses could adversely effect the data cache hit ratio.
- Overhead of running the instrumented code: This is usually the main source of profiling overheads. We further break this overhead into (i) the overhead associated with *fetching* profile instructions and (ii) the overhead of *executing* them.
- If the environment provides a mechanism to dynamically control instrumentation, its overheads and complexity of add to those of the profiler.

Several existing techniques attempt to reduce the execution overheads of instrumentation by stopping instrumentation code from executing. Traub et. al[9] suggest a mechanism for dynamically hooking and unhooking instrumentation by code patching. In Convergent Profiling[10], the profiling code is conditioned on a boolean test before proceeding with data collection. During execution, the boolean is set based on the convergence of the profile. Arnold and Ryder[6] propose a software based framework for instrumentation sampling that uses code duplication and counter based sampling to switch between instrumented and non-instrumented code. This approach requires no hardware support but it involves significant recompilation space and time overheads. A drawback common to these techniques is the overhead of the instrumentation management mechanism itself. While the approach in [9] involves book keeping and code patching, [10] incurs overheads in maintaining individual booleans for every instruction being profiled. The instrumentation sampling framework[6] report a framework overhead of 4.9% due to the presence of checks on method entries and back edges, a 34% increase in recompilation time and requires 285KB of additional code space.

In our view, runtime environments should be able to concentrate on the eventual use of profile information rather than having to spend precious cycles dealing with the management of profile overheads. In this paper we propose a scheme where instrumentation overhead management is a service implemented in the hardware. The runtime environment utilizes this service depending on its requirements. As our results show, this facility does not come at the expense of profiling accuracy or flexibility.

III. Reducing profile execution overheads using selective instruction dispatch

It is a well-recognized fact that most programs show a well-defined phased or even periodic behavior[11]. So, instrumented code need not run over the entire duration of the profiling interval to gather data that is adequately representative of the actual program profile. Since most profile-guided optimizations use profile information to drive heuristics and decision making, they can function as effectively with such representative profiles. This observation has been the basis of sampling based profiling[12]. In a similar vein, we propose *selective instruction dispatch* infrastructure that provides the runtime environment with the means to sample the execution of instrumented code by periodically switching profiling on or off.

In selective instruction dispatch, the switching between profiling and not profiling happens in hardware under software control. Central to the implementation of selective instruction dispatch is the *profile flag*, which is supported by the processor as part of its dynamic state. The state of this flag determines the current profiling mode of the processor. When the software sets the profile flag, the processor runs in the *profiling mode* during which it fetches and executes instrumented code as normal instructions. When the profile flag is reset, profiling is switched off and the processor shifts to the *non-profiling mode.* During this mode, instrumented code is ignored and dropped from the dynamic instruction stream. Thus unlike the schemes surveyed in Section II, under selective instruction dispatch, the runtime environment can reduce profile instruction execution overheads by controlling the periodicity of switching instrumentation on and off without incurring additional recompilation, code duplication or instrumentation management overheads.

The rest of this section describes instruction set and architectural extensions required to support selective instruction dispatch. We also discuss modifications in the instrumentation procedures that allow the runtime environment to control and tune profiling activity.

A. Instruction Set Extensions

A.1 Profile Instructions (PI):

In order to selectively dispatch instrumented code, the processor must be able to distinguish between instrumentation code and ordinary code. This distinction can be made explicit by extending the ISA with a set of instructions referred to as *profile instructions.* These instructions are variants of the subset of the original instruction set that are generally used in profiling. The difference between the original instruction opcode and its profile version could be a single bit, an annotation or a prefix. The processor might even support a completely new set of opcodes depending on the instruction set and decode space available. Using annotations or a bit in the opcode would simplify the identification of these instructions. The instrumentation system on its part uses these special instructions instead of ordinary instructions to instrument code. In this study, we use annotations for lui, lw, sw, add , jal and mov instructions; these are sufficient for the kinds of profiling we report on here.

A.2 Profile Control Instructions (PCI):

These instructions operate on the profile flag and thus control the profiling activity during execution. The *setpf* instruction sets the profile flag while *resetpf* resets the flag. The *pushpf* operation pushes the value of the profile flag onto the runtime stack. *poppf* pops a value from the top of stack and sets the profile flag accordingly.

B. Micro-architecture Extensions

We propose the following extensions to the base architecture in order to support selective instruction dispatch.

B.1 Extensions to decode/dispatch logic:

The decode stage of the processor pipeline is where most of the selective dispatch logic is implemented. The decode logic is extended to support profile instructions and profile control instructions in the following manner. The *setpf* and *resetpf* instructions set and reset the profile flag during instruction decode. These instructions consume decode bandwidth but are not dispatched and hence have minimal overhead. Furthermore, the decode stage is extended with logic to identify profile instructions and process them according to the value of the profile flag. If the decode stage comes across a profile instruction and finds the profile flag set, the instruction is decoded and moved to the reorder buffer like any ordinary instruction. However, if the profile flag is off, the profile instruction is not dispatched. This action of filtering profile instructions from the dynamic instruction stream at the decode stage is referred to as *selective instruction dispatch.* Note that a marginal overhead is incurred even when a profile instruction is ignored in this way since the instruction has already consumed fetch and decode bandwidth.

Independent of whether the profile instruction has been dispatched or ignored, the decode stage checks for a possible *profile mis-fetch.* A *profile mis-fetch* is said to have occurred if the profile instruction satisfies the following three conditions:

1. The profile instruction is also a control transfer instruction
2. It was predicted taken by the branch predictor
3. It is being dropped by the decode stage due to the value of the profile flag.

In this situation, the fetch engine would already have speculatively fetched instructions from the taken address of the branch instruction whereas the instruction that should be processed next lies in the fall-through path. The mis-fetch is rectified by squashing the fetch queue and re-initializing the fetch engine to start fetching from the fall through address.

B.2 Branch prediction for profile instructions

To minimize the number of profile mis-fetches, the branch prediction mechanism must be modified to enable accurate predictions for branch instructions that are also profile instructions. The branch predictor must not only accurately predict the direction this instruction might take, but also whether the instruction is likely to be ignored by the selective instruction dispatch hardware. There are several ways of making the branch predictor aware of profile instructions. We evaluated the following three strategies.

- Branch prediction with ignored instruction updates: Profile instructions that are control instructions and have been ignored at the decode stage update the branch predictor indicating that they were not taken. This strategy works sufficiently well for profile instructions. However, it adversely effects the prediction accuracy for ordinary instructions since updates made by ignored instructions corrupt the branch predictor state.

- Profile flag based branch prediction: The branch predictor is modified to make a prediction based on the current value of the profile flag. A not taken prediction is made if the profile flag is currently off. If the profile flag is set, the existing branch prediction hardware is used to make a prediction. Also, instructions ignored by the selective dispatch hardware do not update the branch predictor at the decode stage. This modification is simple but not very accurate since the value of the profile flag seen by an instruction at the fetch stage could be different from the value it sees at decode due to intermediate *setpf* and *resetpf* instructions.

- Prediction bit based branch prediction: Here, the fetch stage maintains a prediction bit that is similar to the profile flag. This bit is set and reset by *setpf* and *resetpf* instructions when these instructions are fetched. The bit is different from the profile flag since the *setpf* and *resetpf* instructions that modify it might have been fetched speculatively and squashed later. If the prediction bit is off, profile instruction is predicted not taken. However if the bit is set, we fall back on the normal branch prediction hardware. Although the approach requires additional hardware at the fetch stage, the prediction accuracy is high for both profile instructions and ordinary control instructions. We use this strategy in simulation studies reported in Section VII.

B.3 Dependency Logic

The profile flag creates additional dependencies between PI's and PCI's. The dependency analysis logic must be extended to take these additional dependencies into account. A case that needs special consideration is the RAW dependency between a *poppf* and any following instruction that reads the profile flag. Since the profile flag acts at the decode stage and the result of the pop will be available at best in the next cycle, maintaining this dependency involves stalling the decode. Other profile flag dependencies are taken care of by the serial order of decode.

B.4 Branch mis-prediction recovery

Similar to other components of processor state, the profile flag is saved at points where the processor moves into speculative mode. The recovery mechanism restores the profile flag to the saved value when a mis-speculation is detected.

```
if (condition code check)
        setpf
else
        resetpf
```

Fig. 1. Sample implementation of a check-based trigger using PCIs

C. Compiler and runtime extensions

The instrumentation system and the runtime environment must be modified to exploit selective instruction dispatch. Recall that the instrumentation system must use profile instructions instead of ordinary instructions to instrument code. The instrumentation system must also insert PCI's in a manner dictated by the granularity of control required over profiling. A *setpf*/*resetpf* instruction is typically associated with a unit of code, which might be a function, a loop, a basic block or an individual instruction. The *setpf*/*resetpf* instruction precedes the unit of code and controls the profiling activity within the unit. The exact manner of placing the *setpf*/*resetpf* instruction depends on the mechanism used to trigger the switching of profiling modes, which we discuss later in this section. The *pushpf* and *poppf* instructions save and restore the state of the profile flag. Proper placement of these instructions is required to ensure that the scope of *setpf*/*resetpf* instructions is restricted to their intended areas. For example, a *pushpf*-*poppf* pair at the callee's method entry and exit ensure that an *setpf*/*resetpf* instruction in the callee does not interfere with profiling context in the caller. Sample methods detailing these modifications for basic-block and call-graph profiling are included later Section VI.

The mechanism that triggers the switching of profiling modes can either be built into the instrumented methods or implemented externally in the runtime. Arnold and Ryder[6] propose a scheme that involves placing checks at method entries and back-edges that transfer control to the instrumented code based on a condition in the check. A similar check implemented using *setpf* and *resetpf* instructions would resemble the code shown in Figure 1.

To implement the switching mechanism externally, the instrumentation system places a single *setpf*/*resetpf* instruction before units of code. The runtime maintains for each instrumented method, a set of addresses where *setpf*/*resetpf* instructions have been placed. To switch profiling off for a unit of code, the runtime dynamically swaps the associated *setpf* instruction with an *resetpf* and vice versa. The swapping routine could be invoked either by signals, hardware based interrupts or a dedicated thread. Independent of the trigger mechanism used, the *setpf*/*resetpf* instruction executed on the next entry into the unit of code dictates the manner in which profile instructions would be handled by the processor.

This amalgam of minor extensions provides the runtime with fine-grained control over the execution of profile instructions. To reduce overheads, the runtime could use a triggering mechanisms mentioned above to switch profiling on and off periodically. During this interval, profile instructions will not contribute to the execution overhead. As we shall see, the savings can be even more significant if the profile instructions involve calls to external routines.

As with any sampling technique, selective instruction dispatch involves a trade-off with the precision of the profile data. The runtime must ensure that profiling is switched on long enough for a representative and meaningful profile to be collected. Our results show that overheads for call-graph profiling can be reduced from 42% to 11% on average while retaining a profile accuracy of at least 96% if profiling is periodically toggled so as to enable profiling for 25% of the profiling interval.

Other salient features of the selective instruction dispatch are (1) the extensions required to support selective instruction dispatch do not effect the processor's critical path since their effect is limited to profile instructions, (2) selective instruction dispatch decreases resource contention since a number of profile instructions that might have prevented other instructions from using processor resources such as memory ports, registers and functional units are no longer dispatched and (3) the technique involves no recompilation or code duplication and runs in a manner transparent to the compiler.

IV. Reducing fetch overheads using profile flag prediction

Selective instruction dispatch of profile instructions helps in reducing the overheads of profile instruction execution. We next address the problem of trying to avoid fetching profile instructions that are going to be ignored due to selective instruction dispatch. These unnecessary instruction fetches can be a significant overhead to certain kinds of profiles. Basic block profiling is one such case. Here the profile instructions are largely independent of the rest of the code and hence have low execution overheads, particularly in out-of-order processors where they add to the available instruction level parallelism. It is the fetching of these instructions, usually large in number, that is expensive. Fetching and ignoring instructions has the effect of shrinking the instruction window seen by the decode stage. It also leads to an increase in pressure on the i-cache and a higher-than-necessary power consumption. We propose profile flag prediction as a solution to this problem. This mechanism predicts the value of the profile flag that a profile instruction will see during decode. This prediction is used to skip over groups of profile instructions that are unlikely to be dispatched by the selective instruction dispatch hardware, thereby eliminating unnecessary fetches.

We define a *profile block* as a sequence of profile instructions that appear statically together. The *leader* of a block is defined as the first instruction in that profile block. The leader "dominates" all other instructions in its profile block. The *block follow* as the instruction that immediately follows the profile block in static code. A profile block is uniquely defined by its leader and the block follow.

A. The Profile Flag Predictor

The predictor uses a hardware *block descriptor table*. The table has one entry per profile block consisting of the following fields:

1. The address of its leader instruction.
2. The address of its block follow.
3. A prediction bit that indicates the value of the profile flag the profile block is likely to see when fetched next.
4. A lock bit initialized to 1, which disallows reads from this table entry until reset to 0.

The block descriptor table is indexed by the address of the leader instruction. The table maintains descriptions of profile blocks seen during decode. It records past history of whether profile block was dispatched or ignored when previously encountered.

In order to detect profile blocks in the instruction stream without compiler support, the decode stage maintains a special internal register called the *leader address register* that keeps track of the leader address of the profile block being processed. Ordinary instructions reset the register. When the decode comes across a profile instruction and finds the leader address register reset, it identifies the instruction as a leader and updates the register with its address. Profile instructions that follow the leader do not alter the register.

The profile flag predictor interfaces with the pipeline stages in a fashion similar to the branch predictor. For every profile instruction encountered, the decode updates the profile flag predictor with the current instruction address, the profile flag value and the contents of the leader address register. The leader instruction address is used to index into the block descriptor table. If no entry for this address is found, a new table entry is created only if the profile flag is off. This ensures that control will flow through the current profile block even in the presence of profile branches and the predictor will eventually be updated with the last instruction of the block. The block follow address is initialized to the address of the instruction following the leader (subsequent profile instructions update this field). The prediction bit is initialized to predicted reset and the lock bit is set to disallow any reads for this entry until the predictor has made sure of having seen the entire profile block.

If an entry for the leader instruction address is found in the descriptor table, the predictor updates the block follow address if the current instruction address is greater than the existing block follow and the lock bit is set. The lock bit is reset to 0 if the current instruction address is the same as the leader instruction address. These operations ensure that the entry for a profile block is not read until we have seen the entire profile block and correctly determined block follow. The prediction bit is always updated to the current value of the profile flag.

During instruction fetch, the profile block predictor is consulted to check if the fetched instruction is the leader of a block of profile instructions. If so, the predictor uses the prediction bit corresponding to the block to predict whether the block is likely to be executed or not. If it predicts that the profile block is likely to be ignored and finds the lock bit 0, the predictor returns the block follow address corresponding to the leader address. The current fetch is dis-continued and the fetch engine is re-directed to the follow address, thus skipping over the block of profile instructions. The profile flag prediction overrides branch prediction if the leader happens to be a control flow instruction. The leader instruction is moved to the fetch-decode queue in all cases.

A.1 Profile flag mis-prediction and recovery

The decode stage is equipped with a profile flag mis-prediction detection mechanism which works as follows: A profile block mis-prediction is detected when the decode stage comes across a leader instruction whose profile block has been predicted ignored, but the profile flag is currently on. The mis-prediction is corrected by re-initializing the fetch engine to fetch remaining instructions in the profile block.

A.2 Profile flag prediction in dynamic environments

Profile flag prediction in its current form does not validate a correct prediction that involves skipping instructions. It assumes that profile blocks do not change over time. This assumption does not hold in dynamic recompilation environments (where it is not uncommon to have a memory management policy for compiled code) or for self modifying code. Here, a profile block could be replaced by another block of a different length but at exactly the same location. The predictor might cause the processor to skip genuine instructions or execute profile instructions it is not supposed to. This problem could be corrected if the compiler passes information about the length of every profile block to the processor, either as part of the profile instructions or through a special PCI instruction that precedes every profile block. The profile flag predictor could then detect a change in the profile block and remove stale entries in the block descriptor table.

V. Support for simultaneous profiling

The selective instruction dispatch logic can easily be scaled to allow the runtime to instrument a method for different types of profiles and still control each type of profiling independently of the other. To support n simultaneous profiles, the processor must maintain n profile flags, possibly in a dedicated profile flag register where each flag controls one type of profiling. The instruction set would be similarly extended to support n different annotations, one for each profile type. The *pushpf*, *poppf*, *setpf* and *resetpf* instructions take operands specifying a profile flag they need to operate on. Similarly, the fetch stage maintains n prediction bits and the branch predictor uses the prediction bit corresponding to the profile instruction for which a prediction is being made. For the purpose of this study, we restrict our attention to a single type of instrumentation at a time.

VI. Simulation Methodology

We use the SimpleScalar[13] simulator with SPEC CPU2000 benchmarks[14]. The base machine configuration (Table I) represents a typical 4-way out-of-order Superscalar processor[15]. The benchmarks were compiled using the gcc cross compiler targeted to the Simplescalar architecture with compiler flags as specified by SPEC for the Alpha processor (-O4 -ffast-math -funroll-loops for SPEC Integer benchmarks and -O4 for SPEC floating-point benchmarks). The compiler was extended with implementations for basic block and call-graph profiling. We used the SPEC test inputs for our simulations, except for *mcf* where we used the train input as the test input is too short. Detailed simulations were conducted for windows of 100 million instructions at various points spanning the entire length of program execution.

Fetch queue Size	32
Branch predictor	Bimodal and two-level combined, with speculative update in dispatch, 3 cycle penalty for branch mis-prediction
BTB config	256 entries, 2-way
Decode width	4
Issue width	4
Commit width	4
RUU size	64
LSQ size	64
L1 dcache	64KB, 2-way
L1 dcache latency	2 cycles
L1 icache	64KB, 2-way
L1 icache latency	1 cycle
L2 unified cache	2 MB, 4-way
L2 cache latency	10 cycles
Memory latency	100 cycles
Memory bandwidth	32 bytes
TLB	128 entry, 4-way associative, 8KB page size, 50 cycle penalty
Functional Units	4 integer ALUs, 2 integer multipliers, 2 FP adders, 2 FP dividers

TABLE I
Base processor configuration

A. Case Studies: Basic Block and Call-Graph profiling

Basic block and call-graph profiles[16] are representative of classes of instrumentation based profiles that are widely used as inputs to adaptive optimizations. These profiles find applications in optimizations such as profile-guided basic block placement and scheduling [17], dead code elimination and method inlining[18][19].

Figure 2 shows the layout of a function instrumented for basic block profiling under our framework. Each basic block is instrumented at entry with a set of 5 instructions (the two loads are expanded into two instructions by the assembler) to increment a counter corresponding to the block. Since all computation related to profiling happens inline without control being transferred outside the function, these instructions form an example of what is known as *inline* instrumentation[20]. The *pushpf* and *poppf* instructions ensure that the state of the profile flag is restored to its original value in the caller. The presence of the *resetpf* instruction after the prologue allows the runtime to control profiling at the method-level. The *resetpf* instruction is followed by the block profile initialization code. This piece of code is responsible for registering the function's basic block counters with a routine that prints out the block profile on program exit. We exclude the initialization code from overhead computations since a dynamic environment is unlikely to instrument methods with such code.

```
function_entry:
        <method prologue>
        pushpf                      // push the profile flag, save caller context
        resetpf                     // reset the profile flag
                                    // runtime keeps track of this address, patches
                                    // to setpf to turn profiling on and vice versa.
        <block profile init code>   // run only once per method
        ...
LPBX1:                              // block label
        lw/p     $26, ...           // basic block instrumentation instructions
        add/p    $26, $26, 1        // the /p annotation indicates profile instructions
        sw/p     $26, ...
        ...
        poppf                       // pop profile flag and restore caller context
        <method epilogue>
```

Fig. 2. Method instrumentation for basic block profiling

The instrumentation for call-graph profiling shown in Figure 3 involves a call to an external routine. The routine uses the two most recent return addresses to update the call-graph with an arc that corresponds to the current caller-callee pair. This is an example of *out-of-line* instrumentation.

As we shall shortly see, not only do inline and out-of-line instrumentation have a different component-wise distribution of overheads, but they respond differently to our overhead reducing mechanisms.

B. Simulating a runtime environment

We extend Simplescalar with a simulated runtime environment. Our objective is to simulate a runtime that identifies *hot methods*, i.e. methods that consume most of the execution time, and then instruments and collects profiles only for these methods. We achieve this through two separate runs of the simulator. Each run consists of a maximum of four phases, the initial fast-forward phase, method counting phase, detailed simulation phase and the fast-forward till completion phase. The first run uses a binary executable file which has all methods instrumented with just the PCI's in method prologue and epilogue. In both runs, the initial phase skips instructions in fast-forward mode until the simulation point. The method counting phase follows the initial phase and runs in fast-forward mode for 100M instructions.

```
function_entry:
        <method prologue>
        pushpf                           // push the profile flag, save caller context
        resetpf                          // reset the profile flag
        mov/p    $1, $31                 // save caller return address
        jal/p    _mcount                 // call external routine to update call-graph
        mov/p    $31, $1                 // restore caller return address
        ...
        poppf                            // pop profile flag and restore caller context
        <method epilogue>
```

Fig. 3. Method instrumentation for call graph profiling

During this phase, the processor informs the runtime of the occurrence of every *resetpf* instruction and the address at which it occurs. The runtime uses this to keep track of method invocation counts. These are uesd as a measure of method hotness. Information regarding the 10 hottest methods is hand-fed back to gcc, which generates a new binary that has only these 10 methods instrumented with instructions depending on the type of profile desired. The second simulation run uses this binary for detailed simulation. The last phase, fast forward until completion, is necessary as our instrumentation dumps accumulated profiles only at program exit.

The simulated runtime environment uses an external trigger mechanism based on dynamic patching to toggle profiling. To enable profiling during detailed simulation, the runtime patches the *resetpf* instructions in the 10 hot methods to *setpf.* Simulations are performed for different sampling window sizes by switching profiling on and off at specific instruction count intervals. Profiling is switched on after every 20 million instructions for a duration of 2, 5 and 10 million instructions, which results in profiling being enabled for 10, 25 and 50% of the instruction window respectively. Execution time overheads are calculated relative to the execution time for the binary with no instrumentation which is the *No Profiling* configuration. The overheads in the 100% profiling configuration are essentially the overheads that a program would incur in the absence of our framework.

The simulator's instruction counting strategy was modified to ensure that the detailed simulation phase ran over exactly the same instruction window, independent of the instrumentation code that might be present in the binary. Thus profile instructions and all other instructions executed as a result of instrumentation do not contribute to the simulation instruction window count. To ensure program correctness, the pipeline is drained after the detailed simulation phase before switching over to the run till completion.

C. Comparing profiles

Sherwood et. al in their work on characterizing program behavior[21], use Manhattan distance to compare basic block vectors. We use the same distance measure to quantify the similarity of profiles. Profile vectors are treated as points in n-dimensional space. Each vector is normalized by dividing each element of the vector by the sum of all elements. The Manhattan distance in n-dimensional space is the shortest distance between two points when the only paths taken are parallel to the axes. Given two normalized n-dimensional vectors a and b, the Manhattan distance MD between the vectors is given by

$$\mathrm{MD} = \sum_{i=1}^{n} (a_i - b_i)$$

Benchmark	*# Profile inst. executed*	*Execution time overhead(%)*
197.parser	3125956	0.87
164.gzip	3305409	-0.02
181.mcf	723526	0.09
255.bzip2	3241298	0.33
179.art	168	0.02
188.ammp	190535	0.01
183.equake	673270	0.06
Average	**1608595**	**0.19**

TABLE II
NUMBER OF PROFILE INSTRUCTIONS EXECUTED AND FRAMEWORK EXECUTION TIME OVERHEADS

Benchmark	*# branch mispredictions (no instrumentation)*	*# branch mispredictions (instrumented with PCIs)*
197.parser	1216905	1212741
164.gzip	939334	936898
181.mcf	3307418	3297356
255.bzip2	262674	263344
179.art	400046	399504
188.ammp	86790	91306
183.equake	169223	174746

TABLE III
NUMBER OF BRANCH MISPREDICTIONS BEFORE AND AFTER INSTRUMENTING WITH PCIS

The Manhattan distance is a number between 0 and 2, where 0 represents identical vectors. Given the Manhattan distance between two profiles, the relative accuracy is computed as

$$\% \text{ Accuracy} = (1 - MD/2) * 100$$

VII. Performance Results

In this section, we first present the overheads due to the extra instrumentation (PCIs) required to control profiling activity. We then evaluate the impact of the selective instruction dispatch mechanism on instrumentation overheads and profile accuracy. Finally, we present results on the effectiveness of the profile flag prediction mechanism.

A. Framework overheads

Table II shows the execution time overheads incurred by selected programs from the SPEC2000 benchmark suite due to the presence of PCIs in method prologues and epilogues and the average number of PCIs executed in a 100 million instruction window. This overhead is computed relative to the execution time of the binary without PCIs. The overheads for most benchmarks are extremely low inspite of a large number of PCIs encountered, partially indicative of the nature of these instructions and the way they are processed. The average execution time overhead is also a negligible 0.19%.

Overheads are also influenced by redistribution of branch instruction addresses caused by the instrumentation. Table III shows the impact of instrumenting code with PCIs on the number of branch

Benchmark	*# Basic blocks*	*# Calls to hot methods*
197.parser	5921520	1041985
164.gzip	8402845	1101803
181.mcf	26967411	241175
255.bzip2	6640489	1080432
179.art	16633082	56
188.ammp	34608741	63511
183.equake	1941593	224423

TABLE IV
NUMBER OF BASIC BLOCKS AND CALLS TO HOT METHODS

predictor accuracy. For benchmarks other than *ammp* and *equake*, the redistribution has a positive impact on the branch predictor performance and leads to lower mispredictions. We find that the number of mispredictions for *gzip* reduce from 939334 to 936988, which actually translates to a small improvement in performance. The impact of branch address redistribution on branch mispredictions increases in magnitude with the addition of instructions for basic block and call-graph profiling.

B. Selective instruction dispatch

Table V shows the execution time overheads and profile accuracy for call-graph profiling with various sampling window sizes. We infer that as the sampling window size is reduced, profiling overheads decrease proportionally. For instance, overheads drop by 72.7% from 41.81% to 11.41% when profiling at 25%.

We can draw other several insights from this table. The column for 100% profiling indicates that overheads for call-graph profiling are quite significant, especially for integer benchmarks where profiling can result in severely degraded performance. This is despite the fact that we only profile hot methods. The actual overhead depends on the number and locality of method calls. *parser*, *gzip* and *bzip2* incur substantial overheads because of a high number of calls(table IV) and the significant amount of work done to maintain the call-graph. On the other hand, *art* has very few calls to hot methods and does not incur any profiling overheads. Although *ammp* has a sufficiently high number of calls, its poor data cache behaviour overshadows the profiling overheads. The low average execution time overhead of 0.48% when profiling is completely disabled is partially due to the small number of profile instructions that are encountered during call-graph profiling. As shown in Figure 3, a method instrumented for call-graph profiling has only three additional profile instructions, executed once per method call. Also, fewer mispredictions due to the redistribution of branch instruction addresses contributes to the low overhead, even improving the overall performance in *art* and *ammp*. Table VI illustrates this effect. The branch predictor accuracy for some benchmarks increases due to the instrumentation for call-graph profiling. The increase in predictor accuracy as the sampling window size can be attributed to the effects of the *_mcount* routine.

Figure 4 illustrates the variation in instruction and data cache accesses as a result of sampling for call-graph profiling. It is evident that the reduction in overheads for *parser*, *gzip* and *bzip2* is due to a substantially lower number of instruction and data cache accesses. When the sampling window size is reduced, fewer instruction and data cache accesses result due to a higher number of profile blocks being ignored and a lower number of calls to the external *_mcount* routine.

The effects of sampling on the accuracy of the call-graph profile are varied. Sampling causes a negligible drop in profile accuracy for *mcf*, *bzip2*, *ammp* and *equake*. *parser* and *gzip* have a large number of call sites with distributed edge counts. Due to the phased behavior in these programs, sampling at 10%

Benchmark	*0% profiling*	*10% profiling*		*25% profiling*		*50% profiling*		*100% profiling*
	Overhead	Overhead	Accuracy	Overhead	Accuracy	Overhead	Accuracy	Overhead
197.parser	1.12	10.77	83.09	24.04	88.44	45.14	92.80	91.33
164.gzip	1.01	8.12	79.50	18.44	94.60	36.06	95.80	69.08
181.mcf	0.01	1.12	99.62	2.79	99.67	5.52	99.88	10.99
255.bzip2	1.25	13.07	99.96	30.44	99.95	59.23	99.99	104.66
179.art	-0.09	-0.08	71.82	-0.08	91.44	-0.08	97.11	-0.07
188.ammp	-0.02	0.03	99.45	0.11	99.62	0.23	99.90	0.49
183.equake	0.07	1.71	99.93	4.10	99.94	8.19	99.97	16.16
Average	**0.48**	**4.96**	**90.48**	**11.41**	**96.24**	**22.04**	**97.92**	**41.81**

TABLE V

Percentage overhead and accuracy for call-graph profiling with selective instruction dispatch

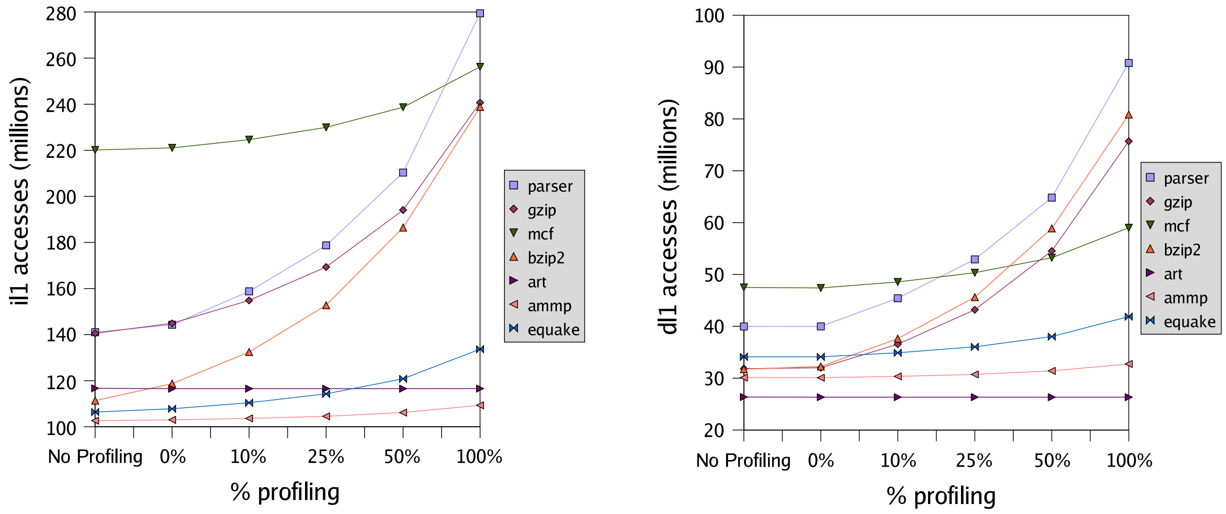


Fig. 4. Call-graph profiling with selective instruction dispatch: level 1 instruction and data cache accesses for different sampling window sizes

misses some call-sites alltogether, lowering the profile accuracy to 83% and 79% respectively. On the other hand, *art* has just four hot edges with extremely low edge counts. This leads to exaggerated error value even if sampling results in a few call-sites being missed. Profiling at 10% results in an average accuracy of 90.48%, which increases to a more acceptable level of 96.24% when profiling is on for 25% of the profile interval.

Table VII illustrates the impact on basic block profiling of sampling based on selective instruction dispatch. Profiling at 25% reduces average overheads from 26.42% to 16.31%, a drop of 38.3%. We also observe that the overheads are lower than those for call-graph profiling, mainly due to the characteristics of the instrumented code. In basic block profiling, the instrumented instructions are independent of the rest of the code, do not transfer control to external routines and show good data locality. On the other hand, the overheads when profiling is completely disabled are higher for basic block profiling than for call-graph profiling. These overheads also account for a higher fraction of the total overhead. This observation is substantiated by Figure 5, which plots the variations in instruction and data cache accesses for different sampling window sizes. We see that the number of data cache accesses decreases almost

Benchmark	*Base value*	*0% profiling*	*10% profiling*	*25% profiling*	*50% profiling*	*100% profiling*
197.parser	93.90	93.91	94.17	94.47	94.85	95.42
164.gzip	94.77	94.78	95.15	95.60	96.20	96.95
181.mcf	86.43	86.54	86.65	86.81	87.07	87.54
255.bzip2	98.36	98.37	98.42	98.49	98.57	98.67
179.art	97.19	97.21	97.21	97.21	97.21	97.21
188.ammp	99.73	99.72	99.72	99.72	99.71	99.71
183.equake	98.46	98.39	98.41	98.44	98.48	98.55

TABLE VI

CALL-GRAPH PROFILING WITH SELECTIVE INSTRUCTION DISPATCH: EFFECT OF SAMPLING ON BRANCH PREDICTION ACCURACY

proportionately with the decrease in the sampling window size. However, the number of instruction cache accesses does not follow the same trend. The number of i-cache accesses is close to its maximum even when profiling is completely disabled. This is due to the the very nature of selective instruction dispatch and the fact that unlike call-graph profiling, ignoring profile instructions here has no additional benefits apart from their own execution overheads. Since the number of profile instructions encountered in basic block profiling is fairly large (table IV), the overheads incured even when profiling is completely turned off are almost 50% of those when profiling is enabled. We conclude that in scenarios such as basic block profiling where overheads due to fetching profile instructions are not negligible, selective instruction dispatch must be complemented with the ability to skip profile instructions to have a significant effect on the overall profiling overheads.

Apart from a steady increase in overall overheads, we also notice a marginal rise in the number of i-cache accesses with the increase in sample window size. One might expect the number of i-cache accesses to remain constant as increasing the sampling window size does not cause any additional instructions to be fetched. Our results indicate that this might be due to an interesting side effect associated with increasing the sampling window size. We conjecture that a larger sample window causes a higher number of profile instructions to be executed and hence slows down the flow of instructions through the processor pipeline. Thus branch instructions, which would have executed earlier had the processor ignored more of the profile instructions, take longer to resolve. As a result, the processor speculatively fetches a larger number of instructions during this interval. This causes greater mis-speculation penalties and explains the marginal increase in i-cache accesses.

We also observe that the accuracy of block profiles reduces considerably for some benchmarks when sampled at 10%. The profiles indicate that the sampling procedure was not able to capture the phased behavior in these benchmarks. However, when the sampling window size is increased to cover 25% of the instruction window, we see a drop in profile error to less than 5% on average.

Table VIII shows the variations in branch predictor accuracy as the sampling window size is varied. We see that branch predictor has the highest accuracy when the program has been instrumented and profiling has been disabled. As the sampling window size increases, the predictor accuracy tends to decrease.

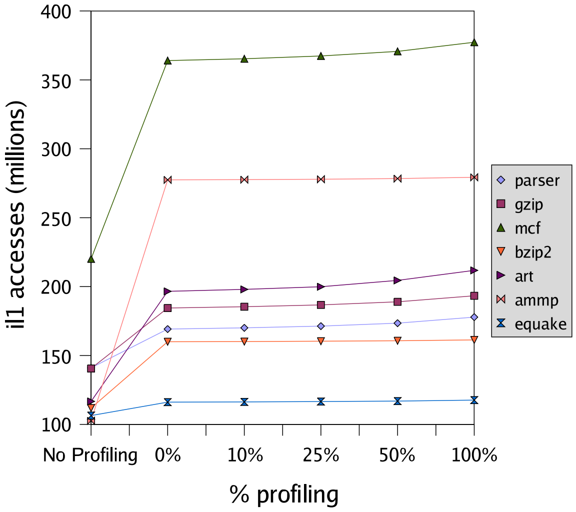
C. Selective instruction dispatch with profile flag prediction

In evaluating the performance of profile flag prediction, we choose an organization that does not incur substantial hardware costs while still being able to capture the behavior of most profile blocks encountered. We studied the impact of block descriptor table size on profile overhead and found that a 2-way associative table with 64 sets occupying 1KB to be adequate. Increasing the size or associativity

Benchmark	*0% profiling*	*10% profiling*		*25% profiling*		*50% profiling*		*100% profiling*
	Overhead	Overhead	Accuracy	Overhead	Accuracy	Overhead	Accuracy	Overhead
197.parser	7.75	8.51	86.49	9.55	92.62	11.31	94.91	14.96
164.gzip	16.39	17.46	82.68	19.00	94.38	21.63	96.19	26.57
181.mcf	18.98	22.98	98.13	28.82	98.82	38.46	99.35	57.78
255.bzip2	21.00	21.57	99.92	22.38	99.94	23.73	99.98	25.85
179.art	23.66	26.36	79.24	30.06	97.37	38.39	98.72	51.77
188.ammp	0.69	0.71	97.50	0.74	98.00	0.80	99.24	0.91
183.equake	2.38	2.86	79.54	3.63	92.27	4.77	94.11	7.08
Average	**12.98**	**14.35**	**89.07**	**16.31**	**96.20**	**19.87**	**97.50**	**26.42**

TABLE VII

PERCENTAGE OVERHEAD AND ACCURACY FOR BASIC BLOCK PROFILING WITH SELECTIVE INSTRUCTION DISPATCH



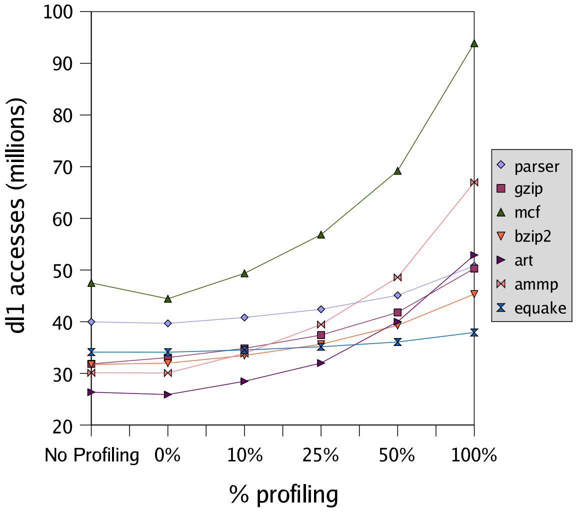


Fig. 5. Basic Block profiling with selective instruction dispatch: level 1 instruction and data cache accesses for different sampling window sizes

beyond these values help significantly.

Another predictor design issue was the minimum profile block size which is worth avoiding fetching. If a profile block is too small, it might be less expensive to not skip fetching the block and let the profile instructions be ignored by the selective dispatch hardware. We studied the performance impact of skipping fetches for profile blocks of various lengths. Our results indicate that skipping is profitable only for profile blocks of length 4 or more. We thus equip the predictor with logic to check the length of the profile block before initiating a skip and hence prevent smaller blocks from adversely effecting the execution time. Since the profile blocks in call-graph profiling have three instructions, profile flag prediction would then have no impact on overheads in call graph profiling. We thus restrict ourselves to its impact on basic block profiling in the rest of this section.

Table IX illustrates the overheads involved in basic block profiling with selective instruction dispatch supported by profile flag prediction. Profile flag prediction does not alter the accuracy of profiles since the only instructions skipped are the ones that would have been ignored by selective dispatch hardware.

We observe that when profiling is completely disabled, profile flag prediction is able to reduce average

Benchmark	*Base value*	*0% profiling*	*10% profiling*	*25% profiling*	*50% profiling*	*100% profiling*
197.parser	93.90	93.97	93.96	93.95	93.94	93.93
164.gzip	94.77	94.55	94.54	94.54	94.54	94.53
181.mcf	86.43	86.66	86.66	86.65	86.65	86.65
255.bzip2	98.36	98.36	98.36	98.36	98.36	98.36
179.art	97.19	98.55	98.36	98.10	97.49	96.52
188.ammp	99.73	99.73	99.73	99.73	99.72	99.71
183.equake	98.46	98.46	98.45	98.44	98.43	98.40

TABLE VIII

Basic block profiling with selective instruction dispatch: effect of sampling on branch prediction accuracy

Benchmark	*0% profiling*	*10% profiling*	*25% profiling*	*50% profiling*	*100% profiling*
197.parser	4.82	5.88	7.32	9.80	14.96
164.gzip	10.18	11.90	14.37	18.58	26.57
181.mcf	11.17	15.93	22.94	34.52	57.69
255.bzip2	14.75	15.98	17.76	20.72	25.59
179.art	13.11	16.82	21.86	33.22	51.39
188.ammp	0.52	0.55	0.61	0.71	0.90
183.equake	0.07	1.71	4.10	8.19	16.16
Average	**7.96**	**9.84**	**12.52**	**17.39**	**26.31**

TABLE IX

Percentage overhead for basic block profiling with selective instruction dispatch and profile flag prediction

overheads by 5% from 12.98% to 7.96%. As shown in Figure 6, the reduction is predominantly due to a decrease in the number of instruction cache accesses. As the sample window size increases and profiling is enabled for longer durations, the profile flag predictor has fewer opportunities to skip profile blocks. The predictor has negligible impact on performance when profiling is enabled throughout the execution window.

Observe that the average overhead is 7.96% on average even when profiling is completely disabled. Apart from the side-effects of instrumentation, this overhead can be attributed to the following reasons:

- Even when a profile block is predicted ignored, the leader instruction always goes through to the decode stage. This instruction might eventually be dropped by the decode but would have already used up fetch and decode bandwidth.
- A profile block is skipped by terminating the current fetch and re-initializing the fetch engine to start fetching instructions from the block follow address when the next cycle commences. On most architectures, the dis-continuous fetch has a small penalty associated with it.

These effects are more pronounced in basic block profiling because of a large number of profile blocks encountered, resulting in a non-negligible overhead even when profiling is completely disabled.

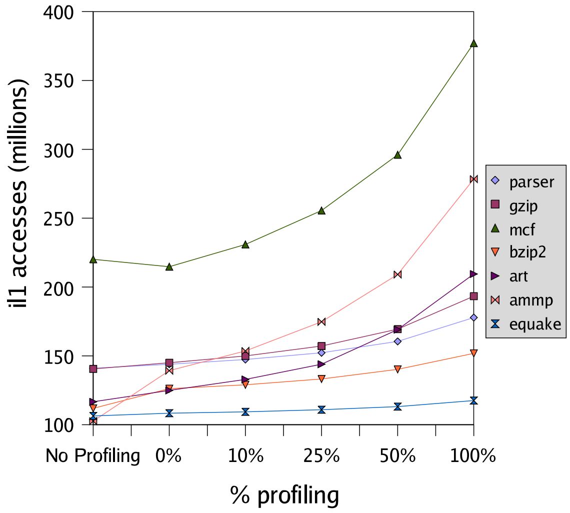


Fig. 6. Basic block profiling with selective dispatch and profile flag prediction: level 1 instruction cache accesses for different sampling windows

VIII. Conclusions

We believe that next generation processors could help dynamic environments achieve improved performance by catering to some of their special requirements. Given the importance of online profiling and instrumentation in such environments, this paper suggests a profile aware micro-architecture that provides dedicated support for online instrumentation. We propose minor but significant extensions to the processor micro-architecture in the form of instruction set support, a hardware profile flag, modified branch prediction, extended decode logic for selective instruction dispatch and profile flag prediction that combine to make a highly flexible, light-weight and tunable framework. The framework provides the runtime with simple mechanisms to reduce profiling overheads that involve no code duplication or instrumentation management. Results of our simulations for call-graph and basic block profiling show that accurate profiles can be collected at substantially lower overheads by sampling the execution of profile instructions using this framework. Average overheads for naieve implementations of call-graph and basic block profiling can be brought down from 41.81% and 26.42% to 11.41% and 12.52% respectively by profiling for 25% of the instruction window while retaining a profile accuracy of over 96%.

References

[1] Thomas M. Conte, Burzin Patel, Kishore N. Menezes, and J. Stan Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," in *International Journal of Parallel Programming*, 1996.

[2] Matthew C. Merten, Andrew R. Trick, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu, "A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.

[3] Jeffery Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos, "Profileme: Hardware support for instruction-level profiling on out-of- order processors," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, December 13 1997, pp. 292–302.

[4] Timothy Heil and James E Smith, "Relational profiling: Enabling thread-level parallelism in virtual machines," in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2000.

[5] Craig Zilles and Gurindar Sohi, "A programmable co-processor for profiling," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[6] Matthew Arnold and Barbara Ryder, "A framework for reducing the cost of instrumented code," in *Programming Languages Design and Implementation*, 20-22 June 2001, pp. 168–179.

[7] Thomas Ball and James R. Larus, "Optimally profiling and tracing programs," in *ACM Transactions on Programming Languages and Systems*, July 1994, pp. 16(4):1319–1360.
[8] Thomas Ball and James R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996, pp. 46–57.
[9] Omri Traub, Stuart Schechter, and Michael D. Smith, "Ephemeral instrumentation for lightweight program profiling," in *Technical report.* Harvard University, 2000.
[10] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Journal of Instruction Level Parallelism*, March 1999.
[11] Michael D. Smith, "Overcoming the challenges of feedback directed optimization," in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 2000)*, January 2000.
[12] S. Sastry, R. Bodik, and J.E. Smith, "Rapid profiling via stratified sampling," in *28th Annual International Symposium on Computer Architecture*, June 2001.
[13] Doug Burger, Todd M. Austin, and Steve Bennett, "Evaluating future microprocessors: The simplescalar toolset," in *Technical Report CS-TR-96-1308.* University of Wisconsin-Madison, July 1996.
[14] T. S. P. E. Corporation, "Spec cpu2000 benchmarks," http://www.spec.org/cpu2000/.
[15] James E. Smith and Gurindar Sohi, "The microarchitecture of superscalar processors," in *Proceedings of the IEEE*, December 1995, pp. 83(12):1609–24.
[16] Glenn Ammons, Thomas Ball, and James R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN '97 Conf. on Programming Languages Design and Implementation*, 1997, pp. 85–96.
[17] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, 20-22 June 1990, pp. 16–27.
[18] Mathew Arnold, "Online instrumentation and feedback-directed optimization of java," in *Technical Report DCS-TR-469,Department of Computer Science, Rutgers University.*
[19] Matthew Arnold, Stephan Fink, Vivek Sarkar, and Peter F. Sweeny, "A comparative study of static and profile-based heuristics for inlining," in *Proceedings of ACM SIGPLAN workshop on Dynamic and Adaptive Compilation and Optimization*, January 2000, pp. 52–64.
[20] Susan L. Graham, Steven Lucco, and Robert Wahbe, "Adaptable binary programs," in *Usenix*, 1995.
[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.