HPS: Hybrid Profiling Support

University of California, Santa Barbara Technical Report#2005-08

Hussam Mousa Chandra Krintz

Computer Science Department
University of California, Santa Barbara
{*husmousa,ckrintz*}*@cs.ucsb.edu*

Abstract

Key to understanding and optimizing complex applications, is our ability to dynamically monitor executing programs with low overhead and high accuracy. Toward this end, we present HPS, a Hybrid Profiling Support system. HPS employs a hardware/software approach to program sampling that transparently, efficiently, and dynamically samples an executing instruction stream. Our system is an extension and application of Dynamic Instruction Stream Editing (DISE), a hardware technique that macro-expands instructions in the pipeline decode stage at runtime.

HPS toggles profiling to sample the executing program as required by the profile consumer, e.g. a dynamic optimizer. Our system requires few hardware resources and introduces no "basic" overhead – the execution of instructions that triggers profiling. We use HPS to investigate the tradeoffs between overhead and accuracy of different profile types as well as different profiling schemes. In particular, we empirically evaluate hot data stream, hot call pair, and hot method identification using a number of parameterizations of bursty tracing, a popular sampling scheme used in dynamic optimization systems.

1 Introduction

The execution behavior and performance of high-end applications has become increasingly difficult to understand. The reason for this is the increased complexity in programs as well as that of the underlying hardware resources on which they execute. To facilitate better program understanding by application developers, and feedback-directed optimization by dynamic and adaptive optimization systems, we require novel techniques that enable efficient yet accurate collection of dynamic program behavior.

This technical report is an expanded version of the HPS paper appearing in the proceedings of The Fourteenth International Conference on Parallel Architectures and Compilation Techniques (PACT '05)

Toward this end, we present HPS, a *H*ybrid *P*rofiling *S*upport system. HPS is a hardware/software profiling system that toggles profile collection according to the dynamically changing behavior of the executing program. HPS combines existing hardware and software techniques into a single system and is able to extract the benefits, while avoiding the disadvantages of each.

HPS extends and applies Dynamic Instruction Stream Editing (DISE) [10], a hardware approach for macro-expansion, i.e., automatic replacement, of executing instructions. Our DISE extensions enable *conditional control* of instruction replacement. That is, we implement simple conditional checks inside the DISE engine so that they can be avoided in the the replacement process. We use DISE productions to replace instructions of interest with instrumentation streams that collect profile information.

HPS also implements bursty sampling, a technique proposed to enable efficient and accurate profile collection within a Java Virtual Machine [3] and later extended for use within a binary tracing and optimization system [6, 17]. Extant, software-based, approaches to bursty sampling are implemented using two copies of the executable code, one that contains profiling instructions and one that does not. At runtime, execution control alternates between the two copies according to execution behavior. Counters and conditional checks in the unprofiled copy determine when the number of method invocations or loop iterations warrant sampling. When this threshold is reached, the checks transfer control to the profiled copy. Execution continues in the profiled copy for a specified *burst length*. The burst length and return of control to unprofiled code is implemented using similar threshold-based, conditional checks.

We refer to the conditional checks and counter updates in the unprofiled code as *basic overhead*. Basic overhead is imposed in the software-only implementation of bursty sampling regardless of whether the code is being profiled or not. The *profiling overhead* is the cost of executing additional profile-collection instructions and checks in the profiled copy of the code.

As a result of employing a hybrid approach, HPS is able to eliminate the basic overhead of bursty sampling and to introduce very little profiling overhead. In addition, HPS does so without code duplication and with minimal indirect impact on program performance, e.g., memory hierarchy pollution, virtual memory paging, and pipeline interruption. HPS is also flexible, unlike extant hardware-based sampling systems, in that it can transparently and efficiently implement any profile type as well as multiple profile types concurrently. Moreover, HPS enables efficient and dynamic modification of profiling parameters, e.g., control-transfer thresholds, an on/off switch for profile collection, and the profile types collected.

To evaluate the generality of HPS, we use it to implement three different profiling types that are widely used in dynamic optimization systems [2, 6, 7, 18] and program memory behavior analysis [5, 17]: hot data stream, hot call pair, and hot method profiling. We measure and report the overhead and accuracy of each of these profile types using a range of bursty sampling thresholds. Our results identify an accuracy/overhead "sweet spot" for each of the profile types that we studied. They also show that the accuracy enabled by longer burst lengths is dependent upon the profile type.

In the following section, we detail background and related work on sample-based profiling and then describe the DISE system that we extend in section 3. In Sections 4 and 5, we describe the design and implementation of HPS. We then present our empirical evaluation of the overhead and accuracy of HPS for three different profile types and a number of bursty sampling parameterizations in Section 6. We present our conclusions and future work in Section 7.

2 Background and Related Work

The goal of our work is efficient and accurate, sample-based, program profiling using a combination of hardware and software. There is a significant amount of research in the area of program profiling. In this section, we describe sample-based profiling techniques that are similar to or that we employ in our system. Sampling enables low overhead but comes at a cost in profile accuracy. Profile accuracy is the degree to which a sample-based profile is similar to an exhaustive profile.

Sample-based techniques have been shown to be effective when implemented in either hardware or software. Examples of hardware-based profiling systems include custom hardware that collects specific profile types [12, 14, 16, 21], systems that utilize existing hardware to collect application-specific performance data [1, 19, 23], and programmable co-processors for profiling [27] and profile compression [24].

These approaches require no modification to application code. In addition, hardware profilers are highly efficient and in most cases avoid adverse, indirect effects such as memory hierarchy pollution. The disadvantage to such approaches in addition to increased hardware complexity and die area, is a lack of flexibility, i.e., the inability or difficulty to re-program the hardware. For example, most approaches implement a single profile type, e.g., instructions executed, branches, or memory accesses. Another important disadvantage of hardware profilers is their use of random and periodic sampling which cannot effectively capture the repeating patterns in program behavior, i.e., phases [22].

Software sampling systems offer improved flexibilty, portability, and accuracy over hardware-based systems. However, these systems also have significant limitations. In particular, software sampling requires program instrumentation via code duplication [3, 15, 17, 22] or dynamic binary modification [25, 26]. Such methodologies require complex instrumentation tools (compilers, loaders, binary instrumentors) and can consume significant computational resources.

A recent approach to software profiling and tracing samples an execution stream according to program behavior [3, 15, 17, 22]. Such systems collect accurate profiles with low overhead (as compared to other software-based approaches). These systems construct two copies of the code, one that is instrumented (i.e. profiled) and one that is not, and alternate execution control between them. Counters and conditional checks in the uninstrumented version determine when to commence sampling.

Execution continues in the instrumented copy for a specified *burst length*. Burst length is a threshold-based counter that dictates how much time the system spends in the instrumented version collecting profile information. Bursty tracing was originally proposed using a burst length of 1 for dynamic optimization in Java Virtual Machines [3], and then extended to employ longer burst lengths in a binary tracing system [15, 17].

There are three primary sources of overhead imposed by bursty sampling. The conditional checks and counter updates in the unprofiled code, i.e., the cost of deciding when to transfer control to instrumented code, is called *basic overhead*. Basic overhead is imposed in the software-only implementation of bursty sampling regardless of whether code is profiled or not. In addition, bursty sampling imposes indirect overhead in the form of cache, virtual memory, and pipeline pollution that results from alternating control between instrumented and uninstrumented versions of the code.

The third source of overhead imposed by bursty sampling is *profiling overhead*. Profiling overhead is the cost of executing additional instructions for profile collection and counter manipulation in the instrumented version. The amount of profile overhead imposed by the sampling system depends on the profile type and the frequency with which instrumented code versions are executed. The profile type dictates which instructions are inserted into the code and the points at which this code is inserted. The sampling frequency or rate depends both on application execution behavior and the accuracy requirements of the profile consumer.

Our Hybrid Profiling System (HPS) implements bursty sampling using both hardware and software to reduce these performance overheads. Moreover HPS is flexible in that it enables any type of profile to be collected. To enable this, we extend and employ the Dynamic Instruction Stream Editing (DISE) system from the University of Pennsylvania [10].

3 Dynamic Instruction Stream Editing (DISE)

DISE is a hardware mechanism that dynamically and transparently inserts instructions into the execution stream thereby enabling semantics similar to macro expansion in the C programming language [10]. To implement DISE, the DISE progenitors extend the *decode* stage of a hardware pipeline to identify interesting instructions and replace them with a specific stream of instructions efficiently.

DISE stores encoded instruction patterns and pre-decoded replacement sequences in individual DISE-private SRAMs called `the pattern table (PT)` and `the replacement table (RT)` respectively. The hardware compares (in parallel) each instruction that enters the decode stage against the entries in the PT. On a match, DISE replaces the original instruction with an alternate stream of instructions that commonly includes the original instruction. DISE retrieves this stream of instructions from the RT according to an address generated by the PT.

An `Instantiation Logic` (IL) unit is employed by DISE to parameterize the replacement sequence according to specific information extracted from the original matched instruction. DISE also supports a small number of DISE-private hardware registers to enable efficient and transparent execution of the replacement sequence instructions without impacting the application's registers.

The DISE mechanism operates within the single cycle bounds of the decode stage and imposes no overhead on instructions that are not replaced. For instructions that are replaced, DISE imposes a 1 cycle delay. DISE utilities operate concurrently with and transparently to the executing program and avoid polluting the memory hierarchy in use by the program for instructions and data.

DISE users build utilities called *Application Customization Functions*(ACFs). Users define ACFs by writing a set of DISE productions each of which consists of a *pattern specification* and a *parameterized replacement sequence*. The pattern specification includes a binary function computed from the various instruction fields. The parameterized replacement sequence is a list of instructions with fields that may contain either precise values or directives for dynamically computing the value based on the matched instruction. The IL unit processes the directives dynamically as it generates the replacement sequence.

The progenitors of DISE have built DISE utilities for software fault isolation [10], dynamic debugging [10, 11], and dynamic code decompression [9, 10]. The authors also suggest and evaluate a preliminary utility that uses DISE for exhaustive path profiling and code generation [8, 10].

4 Hybrid Profiling Support (HPS)

The profiling system that we propose uses a hardware/software (hybrid) approach for its implementation. Figure 1 shows the overview of our HPS design. HPS consists of an extension to DISE (dark grey region in the hardware (H/W) section) that enables highly efficient, conditional, dynamic profile collection. HPS also defines a flexible, software-based (S/W), sampling framework that includes a set of sampling productions that a profile consumer can use to specify instructions of interest and profiling actions via DISE-based, profile productions.

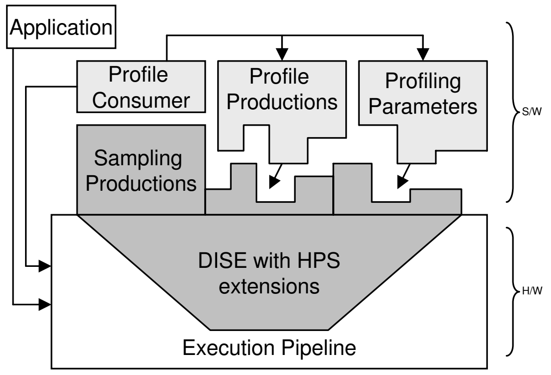


Figure 1. The Hybrid Profiling Support (HPS) system. HPS contains a hardware component which is an extension of the Dynamic Instruction Stream Editor(DISE). Software level productions control the sampling framework and provide facilities for the profile consumer to define their profiling productions (including instructions of interest and actions to be taken). HPS allows the profile consumer to control profiling parameters such as frequency, burst length, and profile toggling.

Our hybrid approach enables HPS to toggle profile collection (sampling) dynamically without code duplication, according to program behavior, and with low overhead. The HPS software interface also provides the profile consumer with control over profiling parameters such as frequency, burst length, and the toggle mechanism. We describe the implementation of these components in the following subsections.

4.1 Applying DISE for Sample-Based Profile Collection

HPS implements bursty sampling in which programs are executed without instrumentation and then periodically sampled in *bursts* [3, 17]. Unlike previous approaches to bursty tracing, however, HPS does so without code duplication. To enable this, we define a DISE ACF (Application Customization Function) to dynamically instrument the executing instruction stream according to a `sampling flag` that is controlled by a `sampling counter`, and a `burst counter`. The sampling flag indicates whether profiling instructions should be inserted by DISE for the program instructions of interest.

The sampling counter dictates when to set the sampling flag. This decrementing counter initially holds a sampling frequency value which is the number of loop iterations (basic block back-edges) and procedure invocations that must execute before HPS sets the sampling flag and commences profile collection. The burst counter is the same as the sampling counter except that it decrements the count when these instructions are executed *while the sampling flag is on* and unsets the flag when the counter expires. This implementation requires five dedicated DISE-private registers for the sampling flag, the sampling counter, the burst counter, and the initial sampling and burst frequencies.

To implement this sample-based, profiling scheme, we define two pattern/replacement production pairs. The first pattern is for procedure calls and back-edges. The replacement sequence for such instructions (that are successfully matched by DISE at runtime) is as follows. The inserted code checks the sampling flag and if it is set, an instruction unsets the flag. If the sampling flag is unset, an instruction decrements the sampling frequency counter and checks this resulting value against zero. If the counter is zero, instructions set the sampling flag and reset the counter value to the initial sampling frequency. The original instruction (call or back-edge) is the final instruction in the replacement sequence.

The second pattern/replacement production pair matches against the instruction of interest to the profile consumer, e.g., dynamic optimizer or program analyst. The replacement sequence for a matched instruction first checks whether the sampling flag is set. If it is, it executes a series of instructions (or alternately calls a special function) to collect the appropriate profiling data. The original instruction is the final instruction in the replacement sequence.

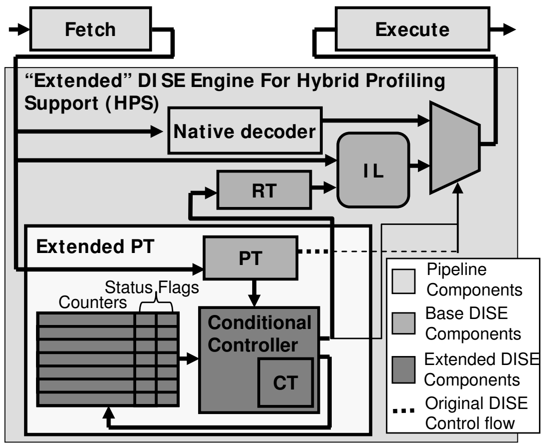


Figure 2. HPS extensions to DISE. HPS moves conditional control of instrumentation out of the critical path and into a dedicated controller.

The primary drawback of this implementation and the use of the original DISE design for sample-based profiling is the very large number of DISE replacements that must be performed. The system must replace every method call and back-edge. By doing so, sample-based profiling introduces significant overhead: a single cycle for each DISE replacement and 2-5 additional instructions (and potentially a pipeline flush) per procedure call, back-edge, and profiled instruction. Moreover, we must replace the profiled instruction regardless of whether the sampling flag is set – since DISE forces us to check the sampling flag within the replacement sequence. To reduce this overhead, we have optimized and extended the DISE design.

4.2 Optimizing DISE for Efficient Sampling

Given that the majority of the actions that we perform to enable profile sampling are simple decrements and zero checks, we modify DISE to perform these functions in the DISE engine as opposed to within the replacement sequence. We refer to this new DISE functionality as *conditional control*. Conditional control provides two primary benefits. First, the sampling interval boundaries (back-edges and procedure calls) require no replacement sequences (i.e. no overhead). Second, we only instrument profiled instructions when the sampling flag is set – we thus, are able to reduce the overhead of replacement to only instructions of interest when sampling is turned on. To enable conditional control, we modify the DISE production specification and the DISE engine.

Figure 2 shows the DISE extensions that enable our effi-

```
Extended DISE Production Specifications
DISE_Production ::=
   Pattern ⇒ Replacement , Conditional_Control
Pattern ::=
   Instruction_Pattern  && Conditional
Instruction_Pattern ::=
   [Original DISE Pattern Specifications]
Conditional ::=
   ( [ ! ] (overflow_N | zero_N) ) [ ( || | && ) Conditional ]
Replacement ::=
   null | [Original DISE Replacement Specifications]
Conditional_Control ::=
   (null | inc_N | dec_N | set_N(dise_reg))[ , Conditional_Control]
```

Figure 3. HPS pattern and replacement specification grammar (extended from the original DISE production specification grammar).

cient implementation of HPS. Instead of requiring that pattern specifications in the DISE ACF be implemented using unconditional matches, we allow *conditional pattern specification*. We add a masking layer which implements conditional matching based on the status flags of internal counters (we currently consider only overflow and zero status flags).

HPS manages internal counters using micro-instructions that are defined as part of the DISE productions. HPS stores these microinstructions in an associative table that we have defined called the Conditionals Table (CT). The CT is similar in structure (but smaller in size) to the pattern table (PT) defined as part of the original DISE infrastructure. The RT is also part of the original DISE implementation and holds the replacement instruction sequences.

We define a set of lightweight micro-instructions to manage the internal counters and extend the pattern specification language to allow for checking of the status bits. Figure 3 shows the extended DISE production specification that enables conditional control within a DISE ACF.

We also extended the root production, called DISE_Production, and the Pattern production to implement conditional control. We added Conditional_Control and Conditional productions that allow the ACF writer to specify simple conditional expressions. These expressions check the status flags for a particular counter for overflow and zero. The microinstructions that we defined are inc_N and dec_N which increment and decrement DISE counter N, respectively (where N varies between 0 and the number of DISE-private counters). The microinstruction set_N (dise_reg) sets counter N with a value retrieved from an internal dise register. HPS can also set the Conditional_Control to null if no conditional microinstruction are required.

Another DISE extension that we make for HPS is to enable specification of pattern productions without replacement sequences. This change is reflected in the Replacement production: null is an optional replacement sequence. When HPS encounters a pattern production rule with a null replacement specification, the pattern match fails upon completion. As a result, HPS simply forwards the current instruction through the decode stage unimpeded, while still permitting local conditional microinstructions. This implementation (match failure upon encountering a null replacement) is key to enabling low overhead in HPS since the 1-cycle penalty is imposed only when an instruction is replaced.

```
Sampling Framework Productions

# We are not profiling, check for procedure call and backward branch
# increment the sampling counter - CC1- and fail production

P1:T.OPCLASS == proc_call && !overflow_1 ⇒ null, CC1
P2:T.OPCLASS == branch && T.PC < T.Target && !overflow_1
                    ⇒ null, CC1

# We are profiling, check for procedure call and backward branch,
# increment the burst counter - CC2 - and fail production
P3:T.OPCLASS == proc_call && overflow_1 && ! overflow_2
                    ⇒ null, CC2
P4:T.OPCLASS == branch && T.PC < T.Target
                    && overflow_1 && ! overflow_2 ⇒ null, CC2

# We are done profiling because burst counter status (overflow_2)
# has overflowed. reset counters (this unsets overflow_1 and
# overflow_2) causing sampling to stop
P5:T.OPCLASS == proc_call && overflow_1 && ! overflow_2
                     ⇒ null, CC3
P6:T.OPCLASS == branch && T.PC < T.Target
                    && overflow_1 && overflow_2 ⇒ null, CC3

CC1: inc_1;
CC2: inc_2;
CC3: set_1(sampleFreq); set_2(burstLength);
```

Figure 4. Pattern and replacement productions for the HPS sampling framework.

Figure 4 shows the set of productions that we use to implement sample-based profiling in HPS. These productions use conditional control and microinstructions (demarked by CCx in the Figure) to decrement the sampling counter, to decide when to set and unset the sample flag, and to decide when a sampling interval has ended. Given this implementation, basic overhead is completely eliminated and no profiling overhead is imposed on the executing program, either direct or indirect, when the sampling flag is unset (i.e. sampling is turned off).

HPS employs two counters: counter_1 is the sampling counter and counter_2 is the burst counter. OF1 and OF2 are the overflow status flags that HPS uses as the sampling flag and burst flag. HPS originally sets the counters to the max-

Profile Type Productions

Hot Data Stream Analysis

```
P1: T.OPCLASS == mem_op && overflow_1 ⇒ R1, null
R1: call(DataStream_handler)
    T.INSN
```

Hot Call Pair Analysis

```
P2: T.OPCLASS == proc_call && overflow_1 ⇒ R2, null
R2: call(CallPair_handler)
    T.INSN
```

Hot Method Analysis

```
P3: T.OPCLASS == proc_call
    || (T.OPCLASS ==branch && T.PC < T.Target)
    && overflow_1 ⇒ R3, null
R3: call(HotMethod_handler)
    T.INSN
```

Figure 5. Pattern and replacement productions for the three different profile types that we investigated using the HPS sampling framework: hot data stream, hot call pair, and hot method profiling.

imum value minus the sampling frequency. This ensures that the counters overflow when the desired thresholds have been reached. This optimization requires only two DISE-private registers (as opposed to four in the unoptimized version), for the initial sampling value (sampleFreq) and burst length (burstLength). We can change these values dynamically in order to adapt to the changing needs of the profiling application.

When a procedure call (P1) or backward branch (P2) is encountered and sampling is turned off (OF1 is unset), the sampling counter is incremented. When a procedure call (P3) or backward branch (P4) is encountered and sampling is turned on (OF1 is set) while the burst length has not been reached (OF2 is unset), the burst counter is incremented. If either counter overflows, their OF flag will be implicitly set. If OF2 overflows, HPS has sampled for the appropriate burst length; P5 and P6 productions capture this. If OF1 overflows, HPS will commence sampling. The hardware implicitly unsets OF1 and OF2 when the counters are set to their initial values (which happens when a sampling interval terminates).

4.3 HPS Profile Type Specification

HPS is flexible in that it can implement any instruction-based profiling technique by specifying a DISE ACF for each profile type of interest. We use HPS to implement three different profile types: hot data stream, hot call pair, and hot method. These profiles are widely used in dynamic and adaptive optimization systems [2, 6, 7, 18]. Hot data stream profiling is also used in an offline setting to evaluate and analyze how the program accesses its data. This analysis is important for program and data placement optimizations [17].

We show the HPS productions for each of these profile types in Figure 5. We use the same format as our previous framework productions; however, these ACFs include a replacement sequence and execute no conditional microinstructions.

Currently we insert a call to the profile collection routine for each profile type. The typical size of a profile handler is a few hundred bytes. As such, several profile types can be implemented at once. We consider only a single profile type at a time in our evaluation of HPS. The only additional change required to enable collection of multiple profile types at once is to have multiple profile productions active simultaneously while merging the pattern specifications and replacement sequences of overlapping profiles.

5 Experimental Methodology

We employ a cycle-accurate simulation platform and simulator parameterization identical to that used in the original DISE studies [8, 9, 10, 11]. The platform is an extension to SimpleScalar [4] for the Alpha processor instruction set and system call definitions. Our simulation environment models a 4-way superscalar MIPS R10000-like processor. It simulates a 12 stage pipeline with 128 entry reorder buffers and 80 reservation stations. Aggressive branch and load speculation is performed and an on-chip memory with 32KB instruction and data caches and a unified 1MB L2 cache is modeled. The DISE mechanism is configured with 32 PT entries and 2K RT entries each occupying 8 bytes.

We model our DISE extension as straightforward counters for the sample frequency and burst length. We employ the DISE productions that we defined in Section 4 to direct the DISE engine. The actions that the system performs to compute the increment do not use the primary execution pipeline, and thus, do not impact simulated performance.

To generate the instructions for profile collection using each of our three profile types, we write the code using the C language and compile it for our target platform (Alpha EV6). We hand-optimize the generated assembly to ensure compactness. We also insert a no-op instruction to simulate single cycle stalls associated with each replacement.

We generate exhaustive profiles as well as sampled profiles for a variety of sampling frequencies and burst lengths. Sampling frequency is the number of events (back-edges and procedure calls) that must execute in between sampling intervals. We use the term `sampling sparsity` to mean the inverse of the sampling frequency. The relation-

	method Count	Call Sites	Call Pairs	Call Count (millions)	Dyn. Branch Cnt (millions)	Memory References (millions)	Dynamic Instructions (millions)	Unique Addresses (thousands)
bzip2	106	245	245	44.6	1,006	2,119	4,546	3,209
crafty	165	792	792	43.4	496	620	2,542	404
eon.cook	599	2,029	2,065	1.9	1	36	51	31
eon.kajiya	602	2,035	2,072	9.2	50	187	251	31
eon.rushmeier	602	2,046	2,083	2.7	2	54	72	31
gap	487	1,779	2,672	18.8	167	223	723	17,389
gcc	1,234	7,565	7,671	22.1	317	590	1,199	815
gzip	113	218	218	25.5	351	775	1,531	690
mcf	113	215	215	5.3	44	85	203	69
parser	338	1,149	1,151	82.0	690	922	2,798	1,387
perlbmk	719	3,138	3,338	5.1	48	79	178	2,728
twolf	234	1,047	1,047	2.7	29	61	165	25
vpr.place	180	627	627	13.9	1,647	437	897	35
vpr.route	264	1,054	1,054	6.0	77	196	469	435

Figure 6. Select Benchmark statistics relevant to the profiles collected.

ship between sampling frequency, sparsity, and burst length is expressed by the following formula:

$$frequency = \frac{burstlength}{(burstlength + sparsity)}$$

For our experimental evaluation, we investigate bursts of length 1, 10, 50 and 100. For each, we compute a sparsity value which would allow for a sampling frequency of 1/100, 1/500, 1/1000, 1/5000 and 1/10000.

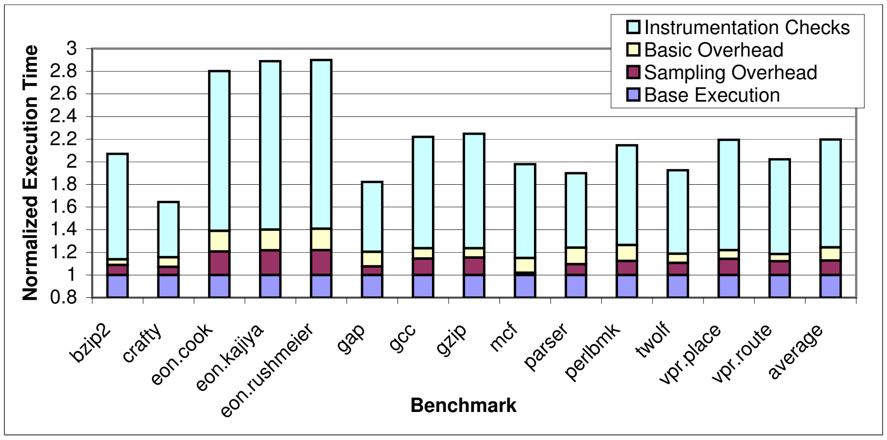
To evaluate accuracy, we compute the overlap of the sampled profiles and the exhaustive profiles by using the relative-contribution metric defined in [13]. We select the top members of the profile such that their combined total accounts for 90% of the total profile. We then compute the contribution percentage of each member in the profile total. Finally, we calculate the overlap of two profiles by iterating over all members of the profiles. For each member, we add the minimum of its contribution percentage to both profiles. The result is an accuracy value between 0% and 100%.

We evaluate the performance and profile quality of our system using the benchmarks of the SPECINT2000. We compile the benchmarks for the Alpha EV6 platform using GCC 3.2.2 with the -O4 optimization flag. We report results for complete runs on the test inputs. For hot call pair and hot method profiles we used all 15 benchmarks from SPECINT2000. For hot data stream analysis we measure the performance for all 15 benchmarks, but we compute the profile quality for only 6 of the benchmarks (eon.cook/kajiya/rushmeier, twolf, perlbmk), due to the large amount of memory required (and thus time) to analyze the other benchmark data.

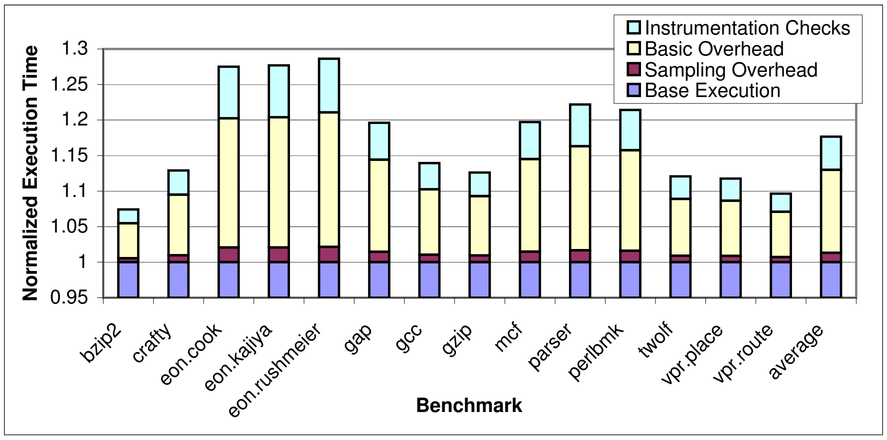
Figure 6 shows some of the dynamic behavior metrics of the studied benchmarks. *Method count* is the number of unique methods in the benchmark while *Call sites* is the number of static call operations. *Call Pairs* is the number of unique call site and target address pairing observed in the dynamic execution of the benchmark. The extra number of call pairs beyond the number of call sites indicate a number of indirect jumps. *Call count* is the number of dynamic calls made during the benchmark's execution. *Dynamic Branch Count* and *Dynamic Memory References* are the number of respective instruction executed while *Dynamic Instructions* is the total number of dynamic instructions executed. *Unique Addresses* is the number of distinct memory addresses accessed by a benchmark. The first two metrics are relevent to hot method profiling, while the first 5 are relevant to hot call pair profiling. The sixth and eighth are relevant to hot data stream profiling and the fourth and fifth metrics are relevant to the determination of when to sample (i.e. the sampling interval boundaries).

6 Evaluation

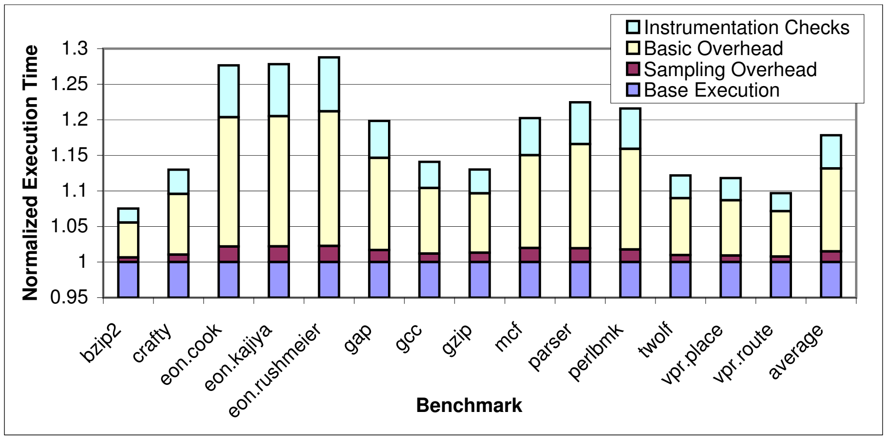
In this section, we present the performance results from our experimental evaluation of HPS and HPS-based sampling. As described previously, HPS imposes no basic overhead – the cost of deciding when to profile. In software sampling systems that transfer control between instrumented and uninstrumented code according to program behavior, e.g., [3] and [17], counters must be maintained in uninstrumented code to decide when to jump into instrumented code. Basic overhead accounts for 1-10% of execution time in a Java virtual machine sampling system [3] and 6-35% in a binary instrumentation system (3-18% using overhead reduction techniques that reduce the number of dynamic checks) [17]. HPS implements the same functionality as these prior systems using a hybrid hardware/software approach that eliminates the basic overhead and the need for code duplication.



(c) Hot Data Stream Profiling

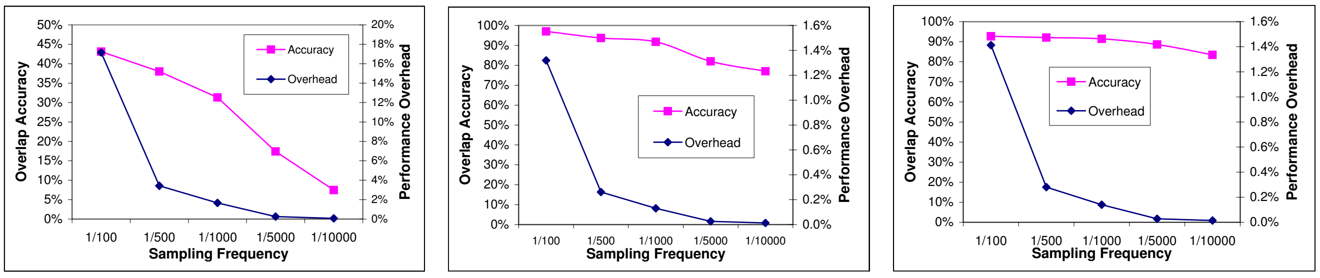


(a) Hot Call-Pair Profiling



(b) Hot Method Profiling

Figure 7. DISE vs HPS: The graphs breakdown the overheads exhibited by a direct DISE implementation of the sampling framework for the profiles studied. The overhead is measured against unprofiled executions of the benchmarks. The Instrumentation checks (top bar) are the compulsory instrumentation performed at every single instruction of interest (profile specific) to check weather its time to take a sample. Basic overhead (second bar from top) is the work done to determine the time to sample. Sampling overhead (second bar from bottom) is the aggregrate of profiling work performed to collect the data. Our extensions to DISE (HPS) eliminates all but the sampling overhead.



Hot data stream Hot call pair Hot method

Figure 8. Accuracy vs Overhead for hot data stream (left), hot call pair (middle), and hot method (right) profiling.

6.1 Profiling Overhead With and Without HPS Extensions

First we evaluate the performance impact of our DISE extensions which enables conditional control within the DISE engine. Figure 7 shows the overhead of sample-based hot data stream (top graph), hot call pair (middle graph), and hot method (bottom graph) profiling using DISE *without conditional controls*. Each bar is the execution time for each program normalized to execution without DISE and without profiling (bottom-most section of the bars). The second section from the bottom of the bar is the overhead of profiling the code using a sampling frequency of 1/100 and a burst length of 1 (burst length is 100 for the Data Stream results). The third section from the bottom of the bar is the basic overhead. The top section of the bar is the overhead for instrumentation checks.

Instrumentation checks are conditional instructions that check whether the sampling flag is set and, if so, invokes the profile handling routine. This instrumentation is applied to each instruction in which the profile being gathered is interested (e.g. procedure calls for hot call pair, and memory accesses for hot data stream profiling). The instrumentation checks are dominant in the hot data stream profiler since the frequency of memory operation is much higher (approximately 1 in 2 to 5 of all dynamic instructions), whereas procedure calls are much less frequent (approximately 1 in 25 to 100). HPS eliminates this overhead by performing these checks off the execution path as part of the DISE engine.

The only source of runtime overhead introduced by HPS is the sampling overhead (second section from the bottom of the bar). Software only sampling techniques will only exhibit basic and sampling overhead – however the overhead is larger due to the indirect impact of code duplication. HPS reduces the average overhead of DISE instrumentation for the programs and sampling parameterizations that we studied by 106% for hot data stream profiling and by 16% for hot call pair and hot method profiling.

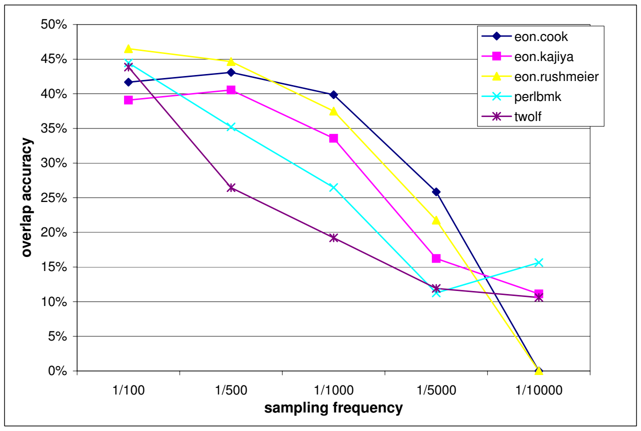
6.2 Overhead and Accuracy of HPS

We next evaluate the overhead and accuracy of HPS. Since the basic overhead of HPS is zero, the total performance overhead is proportional to the sampling frequency with a constant of proportionality equivalent to the cost of the inserted profiling code. The variance of performance between benchmarks corresponds to the dynamic density of profiled events within a sampling interval as well as the frequency of sampling intervals.

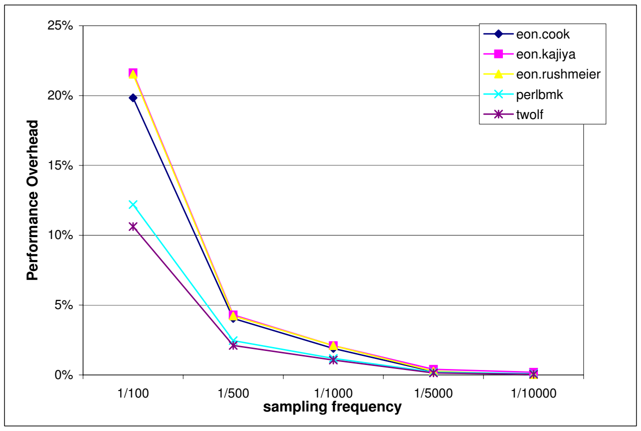
The graphs in Figure 8 show the relationship between overhead and accuracy for hot data stream (left graph), hot call pair (middle graph), and hot method (right graph) profiling. We employed a burst length of 100 for hot data stream profiling and a burst length of 1 for hot call pair and hot method profiling. We discuss the impact of burst length in the next section. This data is the average across all of our benchmarks. Figure 9 shows the equivalent results for individual benchmarks. We observe that most benchmarks have similar overhead graphs with varying offsets corresponding to the variance in the density of profiling events and number of sampling intervals.

The performance characteristics of sample-based hot data stream profiling are very different from hot method and call pair profiling. Such profiling is inaccurate and imposes higher overhead even when implemented in hardware. Our accuracy results are similar to those from recent studies that use software sampling for hot data stream profiling [6, 17].

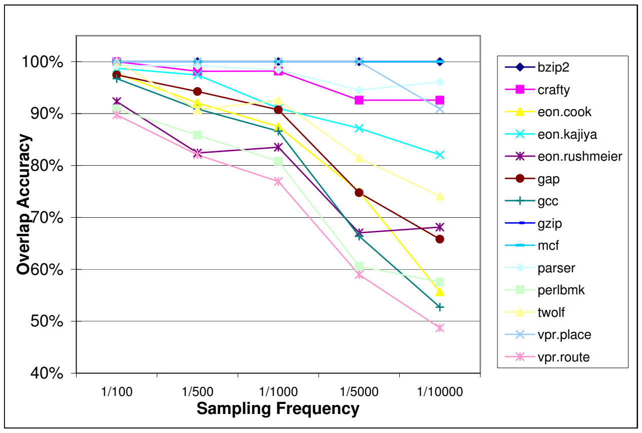
The most promising aspect of our results is the range of total overhead for hot method and hot call pair profiling. Our benchmarks exhibit an overhead range from 0% to 2.5% with an average of just over 1% for the most aggressive sampling rate (1/100) and about 0.2% for high quality (90% accurate) profiles. These overheads are very similar to hardware-based approaches to profile collection such as



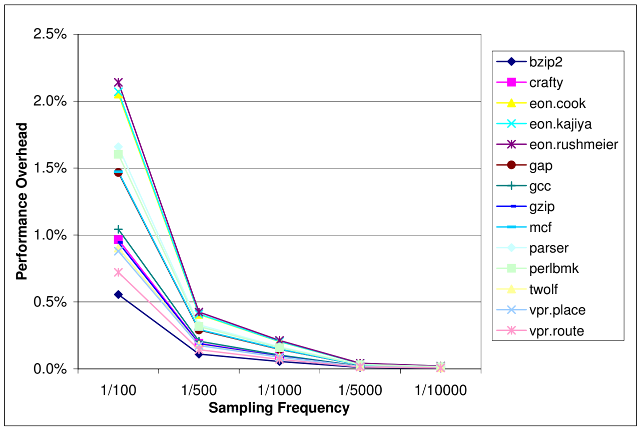
(e)Hot Data Stream Accuracy



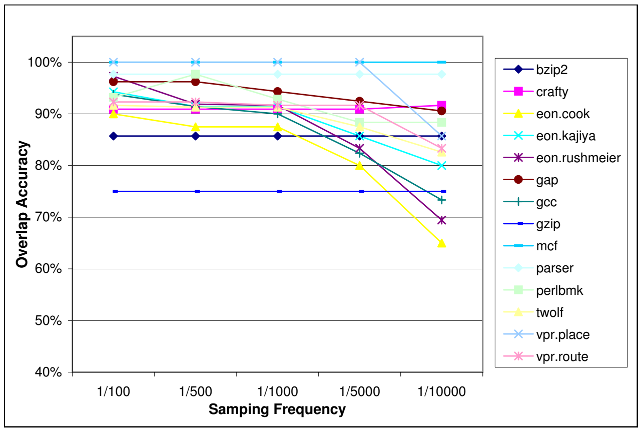
(f)Hot Data Stream Overhead



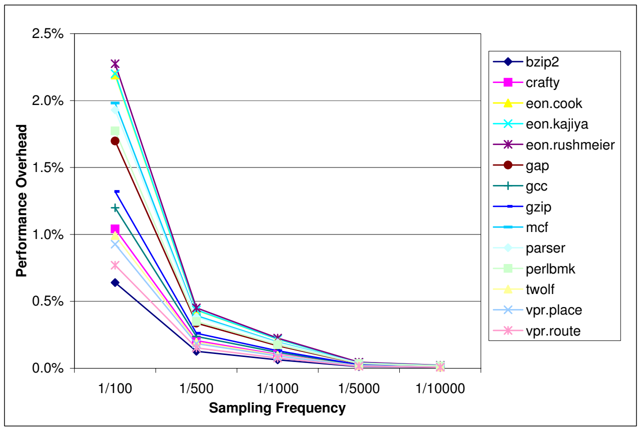
(c)Hot Call Pair Accuracy



(d)Hot Call Pair Overhead

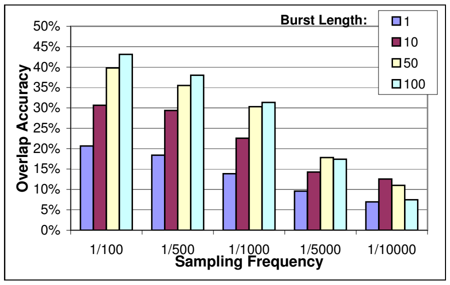


(a)Hot Method Accuracy

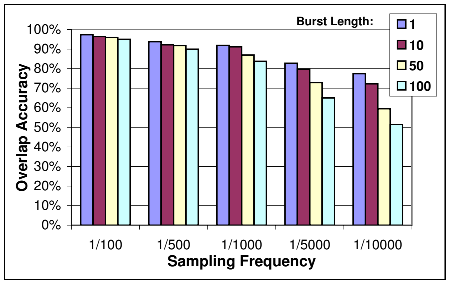


(b)Hot Method Overhead

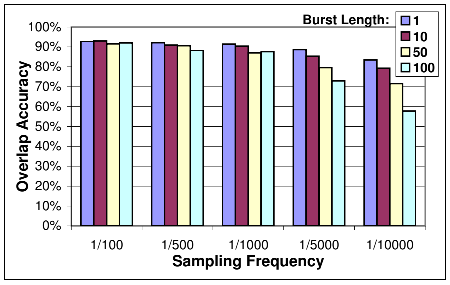
Figure 9. The Accuracy and overhead for the profiles of individual benchmarks collected using HPS



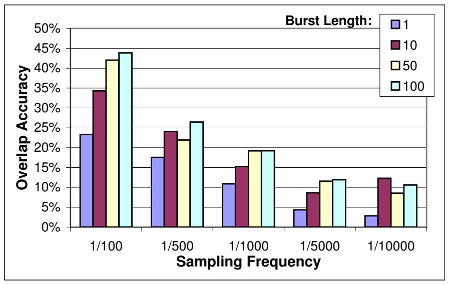
(a)DS:average



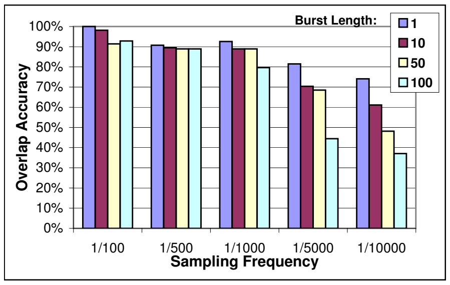
(b)CP:average



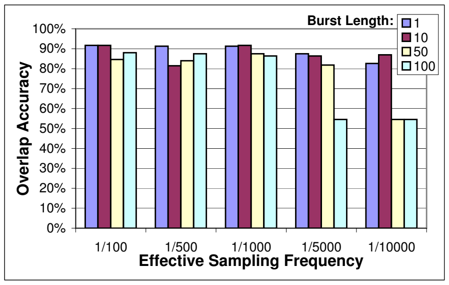
(c)HM:average



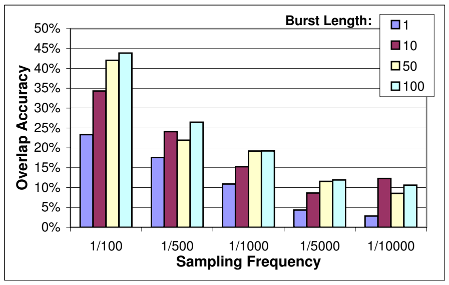
(d)DS:twolf



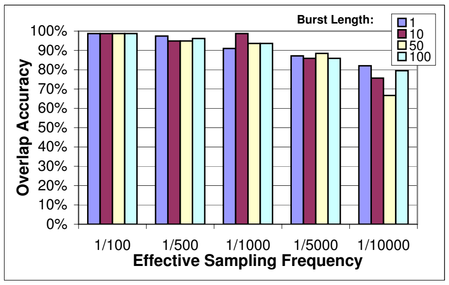
(e)CP:twolf



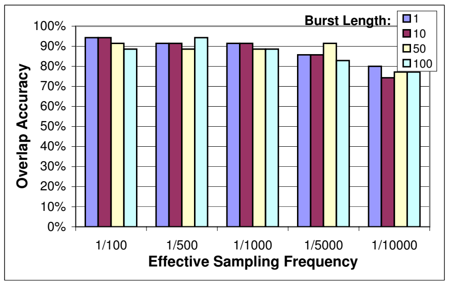
(f)HM:twolf



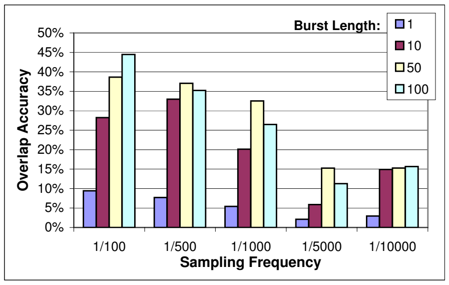
(g)DS:eon.kajiya



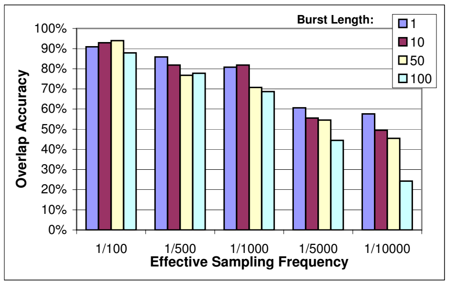
(h)CP:eon.kajiya



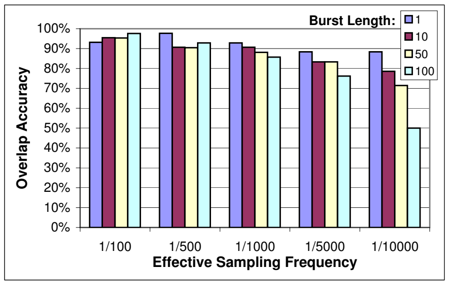
(i)HM:eon.kajiya



(j)DS:perlbmk



(k)CP:perlbmk



(l)HM:perlbmk

Figure 10. The graphs show the profile accuracy using variable burst lengths for hot data stream(DS) (left), hot call pair(CP)(middle), and hot method(HM)(right) profiling. The top row is the average accuracy across all studied benchmarks. While the bottom 3 rows show the accuracy for 3 selected representative benchmarks.

those that use hardware performance monitors. However, HPS enables flexible programming, collection, and parameterization of a wide range of application-specific profile types.

6.3 Impact of Burst Length

The authors in [17], argue that increasing the time spent in *instrumented* code during every sampling instance [17] improves profile accuracy at lower sampling rates. The authors investigate the use of longer burst lengths for hot data stream sampling.

The graphs in Figure 10 show the profile accuracy across benchmarks for different burst lengths (bars), sampling frequencies (x-axis), and profile types. The left clolumn graphs are for hot data stream, the middle graphs are for hot call pair, and the right graphs are for hot method sampling. The top row show the averaged data across all tested benchmarks, while the other 3 rows shows the same data for 3 representative benchmarks. The y-axis is the overlap accuracy; the maximum accuracy is 50% for the left graphs and 100% for the middle and right graphs. On average, hot data stream sampling performs significantly better using longer burst lengths as was found in the prior work. However, for hot call pair and hot method profiling, shorter bursts (i.e. a burst length of 1) performs significantly better.

The explanation for the improved accuracy of hot data stream sampling is straightforward. Since we are looking for consecutive streams of memory accesses, the longer consecutive sampling intervals we spend profiling, the longer the consecutive stream we are likely to observe and the higher our profile accuracy.

The explanation for the deteriorating performance by the other profiles is less apparent. We believe that this behavior is due to the sampling patterns exhibited by longer sampling bursts. With longer bursts, the profiler jumps to instrumented code less frequently but spends more time there every time it jumps. For benchmarks with high densities of sampling intervals, and many inner loops, this may result in a disproportionate amount of time spent profiling similar events, and thus skewing the data toward these events and missing other hot events because of the smaller sampling frequency.

Our conclusion is that for the majority of profiles, small sample bursts are likely to produce improved profile quality, unless there is an intrinsic property in the profile which will benefit from longer bursts. HPS implements per-profile-type burst length (as well as sampling frequency) as a parameter to the system that a user can set appropriately.

7 Conclusions and Future Work

The need to profile executing programs is becoming increasingly more important as software and hardware increase in complexity. *H*ybrid *P*rofiling *S*upport (HPS) is an efficient, flexible and accurate sample-based profiling framework. HPS adds conditional controls to the DISE system, a hardware approach for macro-expansion of dynamically executing instructions, and uses it to define a framework for the rapid implementation and deployment of profiling applications. Our empirical evaluations on 3 different types of profiles demonstrate the very low overhead and high accuracy of HPS.

Our future work includes the implementation and evaluation of different classes of profiling applications such as adaptive bug isolation [20], using HPS. We plan to investigate the use of HPS for concurrent collection of different profile types and to evaluate the effect of different parameters and configurations of HPS on dynamic optimization. We also plan to investigate the effect of incorporating phase aware profiling [22] on the performance and effectiveness of HPS.

Acknowledgments

We thank Marc Corliss, E Lewis, and Amir Roth for providing us with the baseline DISE simulator. We also thank Tim Sherwood for his valuable insights and advice regarding the architectural components of our design. Finally, we thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF grants No. EHS-0209195 and No. CNF-0423336, and an Intel/UCMicro research grant.

References

[1] J. Anderson, W. Weihl, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, and C. Waldspurger. Continuous profiling: Where Have all the Cycles Gone? In *Symposium on Operating Systems Principles (SOSP)*, pages 1–14, Oct 1997.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 47–65, Oct 2000.

[3] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 168–179, Jun 2001.

[4] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun 1997.

[5] T. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In

Conference on Programming Language Design and Implementation (PLDI), pages 191–202, Jun 2001.

[6] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 199–209, Jun 2002.

[7] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 13–26, Jun 2000.

[8] M. Corliss, E. Lewis, and A. Roth. DISE: Dynamic Instruction Stream Editing. Technical Report MS-CIS-02-24, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, Jul 2002.

[9] M. Corliss, E. Lewis, and A. Roth. A DISE Implementation of Dynamic Code Decompression. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 232–243, Jun 2003.

[10] M. Corliss, E. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *International Symposium on Computer Architecture (ISCA)*, pages 362–373, Jun 2003.

[11] M. Corliss, E. Lewis, and A. Roth. Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE. In *Symposium on High Performance Computer Architecture (HPCA)*, pages 303–314, Feb 2005.

[12] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Symposium on Microarchitecture (MICRO)*, pages 292–302, Dec 1997.

[13] P. Feller. Value Profiling for Instructions and Memory Locations. Technical Report CS98-581, Masters Thesis: University of California, San Diego, Apr 1998.

[14] A. Gordon-Ross and F. Vahid. Frequent Loop Detection Using Efficient Non-Intrusive On-Chip Hardware. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 117–124, Oct 2003.

[15] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 156–164, Oct 2004.

[16] T. Heil and J. Smith. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines. In *Symposium on Microarchitecture (MICRO)*, pages 281–290, Dec 2000.

[17] M. Hirzel and T. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec 2001.

[18] The Java HotSpot Virtual Machine, Technical White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.ps`.

[19] M. Itzkowitz, B. Wylie, C. Aoki, and N. Kosche. Memory Profiling using Hardware Counters. In *Supercomputing Conference (SC)*, pages 17–30, Nov 2003.

[20] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug Isolation via Remote Program Sampling. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154, Jun 2003.

[21] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *International Symposium on Computer Architecture (ISCA)*, pages 136–147, Jun 1999.

[22] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-Aware Remote Profiling. In *Conference on Code Generation and Optimization (CGO)*, Mar 2005.

[23] R. V. Peri, S. Jinturkar, and L. Fajardo. A novel technique for profiling programs in embedded systems. In *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-2)*, Nov 1999.

[24] S. Sastry, R. Bodik, and J. Smith. Rapid Profiling via Stratified Sampling. In *International Symposium on Computer Architecture (ISCA)*, pages 83–90, Jun 2001.

[25] M. Tikir and J. Hollingsworth. Efficient Instrumentation for Code Coverage Testing. In *International Symposium on Software Testing and Analysis*, 2002.

[26] O. Traub, S. Schecter, and M. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Department of Electrical Engineering and Computer Science, Harvard University,Cambridge,Massachusetts, Jun 2000.

[27] C. Zilles and G. Sohi. A Programmable Co-processor for Profiling. In *Symposium on High Performance Computer Architecture (HPCA)*, pages 241–253, Feb 2001.