Project Motivation and Goal

- Hardware/Software interaction is increasingly complex
- Understanding dynamic program behavior
  - Key to extracting performance
- Profiling systems
  - Summarize execution behavior
  - Capture and characterize profile-guided optimization opportunities
  - Online profile collection
    - Enables dynamic optimization, specialization, and reconfiguration
    - Introduces overhead
    - Sample-based techniques reduce overhead at a cost in accuracy

Goal of our work

- Online profiling support for the low-overhead collection of accurate sample-based profiles
Current Approaches to Online Profiling

Hardware
- EX: performance counters, special add-ons, co-processor
  - + Low overhead
  - - Consume die-area and additional resources (e.g. power)
  - - Inflexible
  - - No software context

Software
- EX: sampling framework, dynamic instrumentation
  - + Flexible
  - + Arbitrary profiles including fine-grained
  - + Usually more accurate (can capture high-level program info)
  - - Added overhead (cycles, memory, ...)
  - - Indirect effect on performance
Our Work: Hybrid Profiling Support

- **Combine** H/W and S/W
  - General-purpose, low-area, efficient hardware
  - Flexible sample-based profiling interface

- **To Achieve**
  - Low profiling overhead
  - Flexible online management facility
  - Arbitrary instruction-based profile type collection
  - Minimal indirect effect
  - No static analysis and instrumentation for most profile types
Outline

- Motivation and Overview
- Software Sampling
- Dynamic Instruction Stream Editing (DISE)
- Sample-based Profiling with DISE
- HPS Extensions to DISE
- Results
- Conclusion
### Duplicate code
- instrumented & checking
- Sample (switch to inst.) according to expired counter in checking code
  - Exec single pass of profiled code (until loop or call)

### Bursty extensions:
- Extend stays in the instrumented version
  - Requires additional counter(s) in profiled code

### Basic Overhead:
- 5% [1-9%] (VM), 16% [6-35%] (binary)
Dynamic Instruction Stream Editing

- **Processor extension** [Corliss’03]
  - Matches an instruction according to pattern (from PT)
  - Replace with alternate stream (from RT)

- User specifies patterns & replacements via specification language (DISE Productions).
Sampling Framework Using DI SE

- Emulate code transitions via **Sampling Flag**
  - Eliminate code duplication

- Dynamic instrumentation in hardware
  - Eliminate static analysis and code modification

- DI SE productions
  - @ Sampling boundaries => manage (toggle) flag
  - @ Profiled instruction => check flag and profile

- Drawback
  - **Overhead** from too many instrumentations
Breakdown of overheads of sample-based hot data stream profiling using DI$E$
HPS: DI SE extensions

- Basic overhead: simple counter manipulations and checks
- Instrumentation overhead: exclusively flag checking
- Our extension is H/W design + semantic definition for a conditional controller to
  - Manipulate counters and flags
  - Conditionally apply DI SE productions

Original: $\text{DISE} \_ \text{Production} ::= \text{Instruction} \_ \text{Pattern} \Rightarrow \text{Replacement} \_ \text{Sequence}$

HPS extended: $\text{DISE} \_ \text{Production} ::= \text{Instruction} \_ \text{Pattern} \&\& \text{Conditional} \Rightarrow (\text{Replacement} \_ \text{Sequence}|null) \&\& (\text{Conditional} \_ \text{Control}|null)$
HPS: Architecture

- Fetch
- Decode
- Execution engine in-order or out-of-order
- DI SE Engine
- Instantiator
- DI SE controller

CC counters and status flags
**HPS: Profiling Productions**

**Sampling Framework:**

**toggle profiling on/off**

```plaintext
# We are not profiling, check for procedure call and backward branch
# increment the sampling counter - CC1- and fail production
P1: T.OPCLASS == proc_call && !sample_flag ⇒ null, CC1
P2: T.OPCLASS == branch && T.PC < T.Target && !sample_flag ⇒ 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 && sample_flag && !burst_flag ⇒ null, CC2
P4: T.OPCLASS == branch && T.PC < T.Target && sample_flag && !burst_flag ⇒ 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 && sample_flag && burst_flag ⇒ null, CC3
P6: T.OPCLASS == branch && T.PC < T.Target && sample_flag && burst_flag ⇒ null, CC3

CC1: inc_sample_counter;
CC2: inc_burst_counter;
CC3: set_sample_counter(sampleFreq);
    set_burst_counter(burstLength);
```

**Profile Type Productions:**

**Collect the profiling data**

**Hot Data Stream Analysis**
P1: T.OPCLASS == mem_op && sample_flag ⇒ R1, null
R1: call(DataStream_handler)
    T.INSN

**Hot Call Pair Analysis**
P2: T.OPCLASS == proc_call && sample_flag ⇒ R2, null
R2: call(CallPair_handler)
    T.INSN

**Hot Method Analysis**
P3: T.OPCLASS == proc_call
    || (T.OPCLASS == branch && T.PC < T.Target)
    && sample_flag ⇒ R3, null
R3: call(HotMethod_handler)
    T.INSN
HPS: Profiling Productions

**Not Profiling**

- **P1**: \( T\).OPCLASS == proc\_call && !sample\_flag  
  \( \Rightarrow \) null, CC1
- **P2**: \( T\).OPCLASS == branch  
  && \( T\).PC < \( T\).Target && !sample\_flag  
  \( \Rightarrow \) null, CC1
- **CC1**: inc\_sample\_counter;

**Profiling**

- **P3**: \( T\).OPCLASS == proc\_call  
  && sample\_flag && !burst\_flag  
  \( \Rightarrow \) null, CC2
- **P4**: \( T\).OPCLASS == branch  
  && \( T\).PC < \( T\).Target  
  && sample\_flag &&!burst\_flag  
  \( \Rightarrow \) null, CC2
- **CC2**: inc\_burst\_counter;

**Hot Data Stream Analysis**

**P1**: \( T\).OPCLASS == mem\_op && sample\_flag  
\( \Rightarrow \) R1, null
- **R1**: call(DataStream\_handler)  
  T.INSN

**Profiling Not Profiling**

**Sample counter overflow**

**Reset**

**Burst counter overflow**

**Instrumented**

**Hot Data Stream Analysis**

**P1**: \( T\).OPCLASS == mem\_op && sample\_flag  
\( \Rightarrow \) R1, null
- **R1**: call(DataStream\_handler)  
  T.INSN
Experimental Methodology

- Cycle accurate simulation (SimpleScalar for Alpha)
  - 4-way MIPS R10000, 12 stage pipeline
  - DISE: 32 entry PT, 2K entry RT, Conditional Controls

- Benchmarks
  - SPECINT 2000 compiled for alpha EV6 with -O4

- Sampling rate
  - Frequencies from 1/100 to 1/10,000

- **Overhead** is increased execution time due to profiling
  - Baseline is execution time with no HPS instrumentation

- **Accuracy**: % overlap of sampled/exhaustive profile
HPS Overhead

Hot data
stream sampling

Hot call
pair sampling

Sampling frequency = 1/100
HPS Overhead/Accuracy Tradeoffs

Hot data stream sampling

Hot call pair sampling
Impact of Burst Length

Impact of longer sampling burst varies across:

- Profile types
- Effective sampling frequency
- Benchmark
Conclusions

- Hybrid Profiling Support
  - Manage sampling flag entirely in H/W
  - Insert profiling instruction via H/W instrumentation

- Primary contributions
  - “Pay as you go” profiling
    - Eliminate basic and instrumentation overhead
  - Efficient dynamic toggling
  - Flexible usage scenarios
    - Support arbitrary instruction-based profile types
  - 16-106% reduced overhead over direct DI SE