Sampling-based Strategies for Measurement and Analysis

John Mellor-Crummey
Department of Computer Science
Rice University
Collaborators

• Rice
  – Nathan Tallent
  – Mark Krentel
  – Mike Fagan
  – Gabriel Marin

• Former students
  – Nathan Froyd
  – Cristian Coarfa

• RENCI
  – Rob Fowler
Rice’s HPCToolkit Philosophy

• Work at binary level for language independence
  – support multi-lingual codes with external binary-only libraries
• Profile rather than adding code instrumentation
  – minimize measurement overhead and distortion
  – enable data collection for large-scale parallelism
• Collect and correlate multiple performance measures
  – can’t diagnose a problem with only one species of event
• Compute derived metrics to aid analysis
• Support top down performance analysis
  – intuitive enough for scientists and engineers to use
  – detailed enough to meet the needs of compiler writers
• Aggregate events for loops and procedures
  – accurate despite approximate event attribution from counters
  – loop-level info is more important than line-level info
HPCToolkit Workflow

- Application source
  - Source correlation
    - Hyperlinked database
      - Hpcviewer
    - Binary analysis
      - Program structure
      - Interpret profile
      - Profile execution
      - Performance profile
HPCToolkit Workflow

- launch optimized application binaries
- collect statistical profiles of events of interest
HPCToolkit Workflow

- decode instructions and combine with profile data
HPCToolkit Workflow

- extract loop nesting & inlining from executables
HPCToolkit Workflow

- synthesize new metrics as functions of existing metrics
- relate metrics and structure to program source
HPCToolkit Workflow

- support top-down analysis with interactive viewer
- analyze results anytime, anywhere
Outline

• Sampling based measurement
• Binary analysis
• User interface
• Scalability analysis
• Components
  – ours
  – our desires
• Related modeling activities
HPCToolkit Workflow

1. **Application Source**: The starting point is the source code of the application.
2. **Compilation and Linking**: The source code is compiled into object code.
3. **Binary Code**: The object code is linked to create a binary executable.
4. **Binary Analysis**: The binary code is analyzed to understand its structure and performance characteristics.
5. **Program Structure**: This step likely involves understanding the architecture and design of the program.
6. **Profile Execution**: The program is executed with profiling enabled.
7. **Profile Interpretation**: The profiles are interpreted to extract meaningful performance data.
8. **Performance Profile**: The final step is to generate a detailed performance profile of the application.
9. **Hyperlinked Database**: Throughout the process, data is stored in a hyperlinked database for easy access and correlation.
10. **HPCViewer**: The final output is visualized using an HPCToolkit viewer, allowing for detailed analysis and comparison of performance metrics.
Measurement Challenges

Performance often depends upon context

- Layered design
  - math libraries
  - communication libraries in parallel programs

- Generic programming, e.g. C++ templates
  - both data structures and algorithms

- Goals
  - identify and quantify context-sensitive behavior
  - differentiate between types of performance problems
    - cheap procedure called many times
    - expensive procedure called few times
Understanding Costs In Context

Call Path Profiling

- Measure time spent in each procedure
- Attribute time upward along call chain
- Report average time per call per calling context

Call Graph

```
  main
   /\   \\
  /  \  /
 a   b c
   \   /
    \ d
```

Calling Context Tree

```
  main
    /\   \\
   /  \  /
 c    b
   \   /
    \ d
```

```
  main
   /\   \\
  /  \  /
 a   c
   \   /
    \ d
```

```
  main
    /\   \\
   /  \  /
 c    c
   \   /
    \ d
```
A Torture Test

```c
#define HUGE (1<<28)

void d() {}
void c(long n) {
    for(int j=0; j<HUGE/n; j++) d();
}
void a(void (*)(long)) { f(1); f(1); }
void b(void (*)(long)) { f(2); f(2); f(2); f(2); }
void main() { a(c); b(c); }
```
Results with Existing Tools

(for the torture test)

- Instrumentation-based profilers
  - gprof: dilates execution by a factor of 3-14
    - cannot distinguish different costs for calling contexts
  - Vtune: dilates execution by a factor of 31 (Linux+P4)!

- Call stack sampling profilers
  - e.g., Apple’s Shark, HP’s scgprof
    - can’t distinguish different costs for calling contexts

\textbf{csprof: 1.5\% overhead; accurate context-based attribution}
Call Path Profiling Overview

• At each sample event
  – use call stack unwinding to identify full context
    • [vector of return addresses; PC]
  – record sample in a calling context tree (CCT)
    • captures common context between samples
  – “mark the current procedure frame”
    • replace frame’s return address with address of a “trampoline”
  – remember CCT path to marked frame

• When returning from a marked procedure frame
  – increment edge count of the last call edge in the memoized path
  – pop the last edge in the memoized path
  – mark the caller’s frame with the trampoline
  – return control to caller

• Low-overhead unwinding: need not unwind beyond marked frame
SPECint 2000 Benchmarks

Average overhead: gprof 82%, csprof 2.7%

(Opteron, gcc 4.1)
SPECfp 2000 Benchmarks

Average overhead: gprof 31%, csprof 3.2%

(Opteron, gcc 4.1)
Ongoing Call Path Profiler Refactoring

- Platform: OS, architecture
- Profiling flavor
  - flat vs. calling context (CC)
    - CC: precise vs. summary
    - CC: naive vs. smart unwinding (SU)
      - SU: compiler information vs. binary analysis (BA) vs. emulation
        - BA: eager vs. lazy
      - SU: edge counting vs. pure call stack sampling
    - threaded vs. non-threaded
- Initiation: preloading vs. static vs. attaching
- Synchronous vs. asynchronous events
- Asynchronous sample sources
  - timers, counters
  - instruction-based sampling
- Online control API
HPCToolkit Workflow

1. **application source**
2. **source correlation**
3. **binary analysis**
4. **program structure**
5. **interpret profile**
6. **profile execution**
7. **binary object code**
8. **compilation linking**
9. **hyperlinked database**
10. **hpcviewer**

Flow:
- Application source → Source correlation → Binary analysis → Program structure → Interpret profile → Profile execution
- Source correlation → Hyperlinked database → Hpcviewer
- Binary analysis → Program structure
- Interpret profile → Performance profile
Why Binary Analysis?

- Understanding a program’s performance requires understanding its structure.
- Program structure after optimization may only vaguely resemble the program source:
  - Complex patterns of code composition
    - E.g. C++ expression templates
  - Understanding loops is important for understanding performance
    - Account for significant time in data-intensive scientific codes
    - Undergo significant compiler transformations

Goal: understand transformed loops in the context of transformed routines.
Program Structure Recovery with \texttt{bloop}

Analyze an application binary

- Construct control flow graph from branches
- Identify natural loop nests using interval analysis
- Map instructions to source lines, procedures
  - leverage line map + DWARF debugging information
- Recover procedure boundaries
- Identify inlined code & its nesting in procedures and loops
- Normalize loop structure information to recover source-level view
Sample Flowgraph from an Executable

Loop nesting structure
- blue: outermost level
- red: loop level 1
- green loop level 2

Observation
optimization complicates program structure!
Data Correlation

• Problem
  – any one performance measure provides a myopic view
    • some measure potential causes (e.g. cache misses)
    • some measure effects (e.g. cycles)
    • cache misses not always a problem
  – event counter attribution is often inaccurate

• Approaches
  – multiple metrics for each program line
  – computed metrics, e.g. peak FLOPs - actual FLOPS
    • eliminates mental arithmetic
    • serves as a key for sorting
  – hierarchical structure
    • errors with line level attribution still yield good loop-level information
HPCToolkit System Overview

- **application source**
  - source correlation
  - hyperlink database
  - hpcviewer

- **binary object code**
  - binary analysis
  - program structure
  - interpret profile
  - performance profile

- **profile execution**

- **compilation**
  - linking
hpcviewer User Interface

- source pane
- view control
- navigation pane
- metric pane

flatten/zoom control
**hpcviewer Views**

- **Calling context view**
  - top-down view shows dynamic calling contexts in which costs were incurred
- **Caller’s view**
  - bottom-up view apportions costs incurred in a routine to the routine’s dynamic calling contexts
- **Flat view**
  - aggregates all costs incurred by a routine in any context and shows the details of where they were incurred within the routine
Calling Context View: Chroma Lattice QCD

- static + dynamic structure
- costs for loops in CCT
- costs for inlined procedures
- inclusive and exclusive costs
- Dynamically nested loops
- Routines marked inline, not inlined
- Inlined routines

6 loops around operator evaluations
Caller’s View: Chroma Lattice QCD

show attribution of procedure costs to calling contexts
Flattening Static Hierarchies

- **Problem**
  - hierarchical view of a program is too rigid
  - sometimes want to compare children of different parents
    - e.g. compare all loops, regardless of the routine they are inside

- **Solution**
  - flattening elides a scope and shows its children instead
Flat View: S3D Combustion Code

attribute costs to loops implicit with F90 vector syntax
fine-grain attribution to loops within a loop nest
Another Flat View of S3D

highlights costs for an implicit loop that copies non-contiguous 4D slice of 5D data to contiguous storage
Computed Metrics for S3D

Overall performance (15% of peak)
2.05 x 10^{11} FLOPs / 6.73 x 10^{11} cycles = 0.305 FLOPs/cycle

Wasted Opportunity
(Maximum FLOP rate * cycles - (actual FLOPs))

highlighted loop accounts for 11.4% of total program waste
Outline

• Sampling based measurement
• Binary analysis
• User interface

œ Scalability analysis
• Components
  – ours
  – our desires
• Related modeling activities
The Lump Under the Rug: Scaling Bottlenecks

![Graph showing efficiency vs. number of CPUs]

- **CPU Usage:**
  - **Ideal efficiency** (blue line)
  - **Actual efficiency** (red line)

**Note:** higher is better

**Synthetic Example**
Impediments to Scalability

• Communication overhead
  – synchronization
  – data movement

• Computation overhead
  – replicated initialization
  – partially replicated computation

• Parallelization deficiencies
  – load imbalance
  – serialization

• Algorithmic scaling
  – e.g. reductions: time increases as $O(\log P)$
Goal: Automatic Scaling Analysis

- Pinpoint scalability bottlenecks
- Guide user to problems
- Quantify the magnitude of each problem
- Diagnose the nature of the problem
Challenges for Pinpointing Scalability Bottlenecks

- **Parallel applications**
  - modern software uses layers of libraries
  - performance is often context dependent

- **Monitoring**
  - bottleneck nature: computation, data movement, synchronization?
  - size of petascale platforms demands acceptable data volume
  - low perturbation for use in production runs

---

Example climate code skeleton

```
main
  |   
  v   v
land sea ice ocean atmosphere
  |   |   |   |
  v   v   v   v
wait wait wait wait
```
Performance Analysis with Expectations

- Users have performance expectations for parallel codes
  - strong scaling: linear speedup
  - weak scaling: constant execution time

- Putting expectations to work
  - define our expectations
  - measure performance under different conditions
    - e.g. different levels of parallelism or different inputs
  - compute the deviation from expectations for each calling context
    - for both inclusive and exclusive costs
  - correlate the metrics with the source code
  - explore the annotated call tree interactively
Weak Scaling Analysis for SPMD Codes

Performance expectation for weak scaling
- work increases linearly with # processors
- execution time is same as that on a single processor

• Execute code on $p$ and $q$ processors; without loss of generality, $p < q$
• Let $T_i = \text{total execution time on } i \text{ processors}$
• For corresponding nodes $n_q$ and $n_p$
  - let $C(n_q)$ and $C(n_p)$ be the costs of nodes $n_q$ and $n_p$

• Expectation: $C(n_q) = C(n_p)$

• Fraction of excess work: $X_w(n_q) = \frac{C(n_q) - C(n_p)}{T_q}$ parallel overhead
  total time
Strong Scaling Analysis for SPMD Codes

Performance expectation for strong scaling
- work is constant
- execution time decreases linearly with # processors

- Execute code on p and q processors; without loss of generality, p < q
- Let $T_i = \text{total execution time on } i \text{ processors}$
- For corresponding nodes $n_q$ and $n_p$
  - let $C(n_q)$ and $C(n_p)$ be the costs of nodes $n_q$ and $n_p$

- Expectation: $qC_q(n_q) = pC_p(n_p)$

- Fraction of excess work: $X_s(C,n_q) = \frac{qC_q(n_q) - pC_p(n_p)}{qT_q}$ parallel overhead
  total time
Scaling Analysis with Expectations

- Excess work metrics are intuitive
  - $= 0$ ideal scaling
  - $> 0$ suboptimal scaling

- Using excess work metrics
  - $X(I,n) \approx X(E,n)$: scaling loss due to computation in $n$
  - $X(I,n) >> X(E,n)$: scaling loss due $n$’s callees
  - using multiple views
    - losses associated with few calling contexts $\Rightarrow$ CCT view suffices
    - losses spread across many contexts $\Rightarrow$ use callers view
LBMHD size $1024^2$
Strong Scaling Analysis of LBMHD

- 53% excess work = 47% efficiency
- 14% scalability loss due to computation
- 17% scalability loss due to barrier-based reductions
LANL’s Parallel Ocean Program (POP)

successive global reductions on scalars degrade parallel efficiency (7 total)

12% loss in scaling due to scalar reductions

7% in this routine alone
UPC NAS CG class B (size 75000)

63% excess work = 36% efficiency

Remote data prefetch
UPC NAS CG class B (size 75000)

loss of efficiency due to barrier-based implementation of sum reduction
Weak Scaling Analysis of MILC’s su3_rmd
Scalability Analysis Using Expectations

- Broadly applicable
  - independent of programming model
  - independent of bottleneck cause
  - applicable to a wide range of applications and architectures

- Easy to understand and use
  - fraction of excess work is intuitive and relevant metric
  - attribution to calling context enables precise diagnosis of bottlenecks
  - provides quantitative feedback

- Perfectly suited to petascale systems
  - call stack sampling is efficient enough for production use
  - uses only local performance information
  - data volume is modest and scales linearly

- Drawback
  - pinpoints bottleneck, but provides no intuition into cause
Outline

- Sampling based measurement
- Binary analysis
- User interface
- Scalability analysis

Components
  - ours
  - our desires

- Related modeling activities
Components to Share

• **libmonitor** - infrastructure for augmenting program with monitoring
  – what
    • monitors program launch thread creation/termination, fork/exec, exit
  – how
    • preloaded library for dynamically linked executables
    • static library for statically-linked executables

• **hpcviewer** user interface
  – three views: calling context, caller’s view, flat view
  – scalability analysis

• **bloop** binary analyzer
  – identify loops, inlined code

• **OpenAnalysis** - representation-independent program analysis tools
  – call graph and control-flow graph construction
  – dataflow analysis
Component Needs

- Metadata collection
- Standard OS interface for sampling-based measurement
- Ubiquitous stack unwinder for fully-optimized code
  - instruction cracker
  - engine for recovering frame state info at any point in an execution
Outline

- Sampling based measurement
- Binary analysis
- User interface
- Scalability analysis
- Components
  - ours
  - our desires

Related modeling activities
Analysis and Modeling of Node Performance

Object Code

Binary Code

Binary Analyzer

• Control flow graph
• Loop nesting
• Instruction dependences
• BB instruction mix

Static Analysis

Binary Instrumenter

Instrumented Code

Dynamic Analysis

Execute

• BB & Edge Counts
• Memory Reuse Distance
• Communication Volume & Frequency

Dynamic Analysis

Scalable Models

Modeling Program

Architecture neutral model

Evaluate

Cross Architecture Models

IR code

Modulo Scheduler

Performance Prediction for Target Architecture

Architecture Description
Capabilities of Modeling Toolkit

Loop level attribution of metrics

- Attribute execution costs to underlying causes
  - data dependencies that serialize operations
  - insufficient CPU resources
  - memory delays (latency and bandwidth)

- Explain patterns of data reuse
  - pinpoint opportunities for enhancing temporal reuse
  - pinpoint low spatial reuse

- Automatic “what if” scenarios
  - infinite number of CPU resources
  - no register or memory dependencies
  - no memory delays