• Compile and link for production
  – with full optimization
• For statically-linked executables (e.g. for Cray XT or BG/P)
  – use \texttt{hpclink} script to incorporate our monitoring library
Measure execution unobtrusively
- launch optimized application binaries
- collect call path profiles of events of interest
HPCToolkit Performance Tools

Analyze binary to recover program structure
- analyze machine code, line map, and debugging information
- extract loop nesting information and identify inlined procedures
- map transformed loops and procedures back to source
HPCToolkit Performance Tools

- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate measurements to static & dynamic pgm structure
• Explore performance data from multiple perspectives
• Rank order by metrics to focus on what’s important
• Compute derived metrics to gain insight
• Explore call stack traces to understand transient behavior
Attribution to Static + Dynamic Context

calling context view

costs for
- inlined procedures
- loops
- function calls in full context
Outline

- GPU profiling
- Detecting memory leaks
- Call path tracing
- More work on scaling
- Applying HPCToolkit to FY2011 “Joule Metric” applications
- Data-centric measurement and analysis
- Static analysis of memory access patterns
- Some challenges ahead
Call Path Profiling for GPU-based Systems

- Why call path profiling? Flat context often isn’t enough
  - same operations used differently in multiple places
- Many apps experimenting w/ GPU acceleration
  - call path of GPU kernel is separated in space
    - host stack + GPU kernel
  - call path of GPU kernel is separated in time
    - kernels may be executed asynchronously
  - GPUs contain interesting hardware performance counters

Adapt HPToolkit profiling to CUDA-accelerated executions
Prototype of GPU-Enabled Profiler

- Use PAPI + NVIDIA’s CUPTI profiling interface

On entering a CUDA “kernel launch”
  - `cudaThreadSynchronize()` // wait for GPU to finish
  - start GPU performance counters

On exiting a CUDA “kernel launch”
  - `cudaThreadSynchronize()` // wait for GPU to finish
  - stop GPU performance counters
  - gather calling context of kernel (synchronously)
  - associate GPU performance with kernel (in context)

Limitations
  - counters are not kernel-specific (hardware limitation)
    - must either serialize kernels or work with throughput metrics
  - `cudaThreadSynchronize()` on entry/exit
    - destroys CPU/GPU overlap
    - shouldn’t affect GPU measurements of individual kernels
  - kernel is finest granularity of GPU counter metrics
    - no line-level attribution within GPU code
GPU-Aware Call Path Profiles

attribute GPU metrics to CUDA kernel source in its full calling context using CPU metrics as well (e.g. WALLCLOCK)
GPU Profiling Support: What Next?

- Look at overall CPU and GPU utilization
- Quantify overlap of
  - CPU execution
  - data movement to accelerator
  - GPU execution
- Look at gap between potential vs. realized performance
  - compute derived metrics to understand GPU performance
    - degree of multithreaded parallelism utilized
    - fraction of compute capability utilized (instructions per cycle)
    - fraction of available memory bandwidth consumed
    - fraction of memory accesses that hit in cache
    - balance of reads and writes across cache and memory slices
    - fraction of divergent branches
    - ...

Outline

- GPU profiling
- Detecting memory leaks
- Call path tracing
- More work on scaling
- Applying HPCToolkit to FY2011 “Joule Metric” applications
- Data-centric measurement and analysis
- Static analysis of memory access patterns
- Some challenges ahead
Correctness Tool: Memory Leak Detector

- Intercept malloc() and free() (and variants)
  - malloc: gather calling context (synchronously)
  - free: note that the corresponding allocation point is freed
- Storing metadata: in-band vs. out-of-band
  - associate malloc calling context with allocated block
  - out-of-band: process-wide splay tree (with locks)
    - advantage: easy to implement
    - disadvantage: overhead
  - in-band: add header or footer to memory block [our approach]
    - prefer headers: constant time lookup, no synchronization
    - use footers as needed
      - advantage: avoids disturbing specified memory block alignment
      - disadvantage: synchronized lookup
- Can trade monitoring overhead for incompleteness
  - monitor every $n$th malloc; monitor all frees
- Detail: getcontext() is surprisingly expensive; write our own
Confirming OMEN Has No Leaks
• GPU profiling
• Detecting memory leaks
• Call path tracing
• More work on scaling
• Applying HPCToolkit to FY2011 “Joule Metric” applications
• Data-centric measurement and analysis
• Static analysis of memory access patterns
• Some challenges ahead
Understanding Temporal Behavior

• **Time-dependent behavior is often invisible in profiles**
  — but tracing is difficult to scale to long or large executions

• **What can we do? Trace call path samples:**
  — on each sample, record call path of each thread
  — organize the samples for each thread along a time line
  — view how the execution hierarchically evolves
    • assign each procedure a color; view a depth slice of an execution
  — use **sampling** to scalably render large-scale traces

ICS 2011
Exposing Temporal Call Path Patterns

PFLOTRAN, 8184 processes, Cray XT5

Process-time view at selected depth

Depth-time view for selected rank
Presenting Large Traces on Small Displays

• How to render an arbitrary portion of an arbitrarily large trace?
  — we have a display window of dimensions $h \times w$
  — typically many more processes (or threads) than $h$
  — typically many more samples (trace records) than $w$

• Solution: sample the samples!

![Diagram showing how to render a large trace on a small display](image)

Each sample defines a pixel.
Will Sampling Miss Something Important?

- Sampling may miss the precise cause of an anomaly...
  - but, important anomalies will have (local/non-local) effects
- Sampling exposes effects of the important anomalies

In an unusual execution, 8184 processes took 190 s to complete MPI_Init!
(FLASH, JaguarPF, Cray XT5)

Using sampling for both measurement and presentation clearly exposed the problem.
Outline

- GPU profiling
- Detecting memory leaks
- Call path tracing
- More work on scaling
- Applying HPCToolkit to FY2011 “Joule Metric” applications
- Data-centric measurement and analysis
- Static analysis of memory access patterns
- Some challenges ahead
Sample processes within SPMD applications
- record data on a process with probability $p$
- simplification of Gamblin et al., IPDPS ’08
- effective

Pinpoint strong scaling bottlenecks in PFLOTRAN, 4K — 32K cores, running on a Cray XT5 (JaguarPF).

This Allreduce accounts for 21.6% of scaling loss — 85.7% of the cost of 4K run

Loop accounts for 86.4% of scaling loss — 343% of the cost of 4K run
Real Tools Must Address the ‘D’ in R&D

- First version of tracer used one trace file per process
- Problem: File systems don’t handle 1000s of files per directory
  - FSs optimize for data integrity rather than for fast file lookup
    - typical: store files in order of creation and use linear search instead of data structure optimized for lookup
- Bleeding-edge version of tracer
  - fast and scalable trace record lookup
    - merge all trace files into one file
      - index + trace files
    - resolve several inefficiencies
      - e.g.: eliminate unnecessary duplication of call path data
      - one can only expect so much from high school seniors
- TODO: use SionLib or PLFS to write profile and trace data
Refining Analysis and Presentation

- Current scalable database requires \( O(1 \text{ CCT}) \) space
  - non-distributed data structure \( \rightarrow \) per-process requirement

- Many opportunities for refining database
  - never, ever use XML
    - replacing with Google Protocol Buffers
    - expect 1–2 orders of magnitude in space savings
  - use appropriate (sub) data structures
    - use dense vectors for dense data (e.g., inclusive metric values)
    - use sparse vectors for sparse data (e.g., exclusive metric values)
  - post-process data to accelerate performance of user interface
    - scatter plots: better to have per-thread metric values for CCT node instead of all CCT-node metric values for a thread
  - incrementally prune irrelevant profile data
    - reduce the high-water space requirement for building a CCT
  - possibly another order of magnitude (on top of XML change)
• GPU profiling
• Detecting memory leaks
• Call path tracing
• More work on scaling

• Applying HPCToolkit to FY2011 “Joule Metric” applications

• Data-centric measurement and analysis
• Static analysis of memory access patterns
• Some challenges ahead
Work on FY2011 ‘Joule Metric’ Codes

- Identified load imbalance:
  - using tracing of call path samples
    - found early/late arrivers at an MPI_Allreduce
  - using differential profiling & load imbalance analysis
    - compare early/late arrivers
    - confirm that exposed idleness is fully offset by FP computation
  - simple case of load imbalance
    - mismatch between input data and # of processors

K. Roche: Given OMEN, a highly tuned app, can you find anything?
Work on FY2011 ‘Joule Metric’ Codes

• Improved performance of array copies:
  — most inefficiency was in Goto BLAS xcopy wrappers
    • 5% of execution time; 18% of resource stalls
    • xcopy: assembly — no unwind information!
  — specialized calls to xcopy to use memcopy when possible
    • Goto BLAS copy didn’t exploit memory parallelism, prefetching
  — improved cost of copies by 25% (1.3% overall)
Outline

- GPU profiling
- Detecting memory leaks
- Call path tracing
- More work on scaling
- Applying HPCToolkit to FY2011 “Joule Metric” applications
  - Data-centric measurement and analysis
- Static analysis of memory access patterns
- Some challenges ahead
Data-centric View

- **Root of all uses**
- **Allocation point**
- **Data structures**

- **Source code pane**
- **Navigation pane**
- **Metrics pane**
Linux Kernel Support for AMD’s IBS

• Why perfmon2 & libpfm3 vs. perf_events?
  — perfmon2 supports per-thread mode for IBS
  — HPCToolkit monitors threads separately

• Problem in perfmon2 driver for Linux 2.6.30
  — runaway kernel process (kondemand/12)
    • causes system crash
  — occurs very few times when run sequentially
  — always occurs when monitoring parallel programs

• Patches (from Oprofile kernel and already known workarounds)
  — erratum 420: set IbsOpMaxCnt & IbsOpEn bits in two steps
  — UBTS 227027: enable/disable LBR
  — UBTS 299030: Read IP immediately after setting the IBS OP

No errors, no crashes
Outline

• GPU profiling
• Detecting memory leaks
• Call path tracing
• More work on scaling
• Applying HPCToolkit to FY2011 “Joule Metric” applications
• Data-centric measurement and analysis
  • Static analysis of memory access patterns
• Some challenges ahead
Memory Access Patterns Matter

• Understand cache usage
  — non-unit stride → poor spatial locality
  — data access = non-unit stride% + unit stride%
    • combine data-centric analysis: whether to transpose an array’s layout

• Understand memory access patterns at the loop level
  — e.g. memory footprint of an access
  — recording all memory accesses has large overhead
    • instrument all loads/stores and collect them in a buffer
    • compute the reuse distance for each element in the buffer
    • more than 100x slow down
  — use combination of static analysis + dynamic information instead
    • use stride analysis to reduce instrumentation necessary
Goal and Approach

• **Goal:** feed pattern information to HPCToolkit (future work)
  — data-centric measurement and analysis
  — memory footprint analysis

• **Approach:** analyze memory access stride in loops
  — perform static analysis of an application binary
    • only analyze indexed accesses
  — use Dyninst
    • parseAPI & instructionAPI
      – extract loop information and memory access instructions
    • dataflowAPI
      – perform data flow analysis using program slicing
Precursor to Stride Analysis: Loop Analysis

- parseAPI: analyze the control flow, build CFG
- Return all basic blocks in the loop exclusively or inclusively
- Find loop headers
Stride Analysis Algorithm

• Analyze each memory access instruction in the loop
  — filter out the scalar instructions
    • access the memory using the unchanged register as the index (bp)
  — get the multiplier for indexed operations: (%rbp,%rax,4) is 4

• Find the instruction, compute the index
  — backward slice from the memory access instruction in the loop
    • backward slice on the index register (rax is definitely the index)
      – movss 0x602080(,%rax,4),%xmm0
      – movss (%rbp,%rax,4),%xmm0
      – movss (%rax),%xmm0
    • backward slice on both index registers (rax, rbx are possible indexes)
      – movss (%rbx,%rax,1),%xmm0
  — symbolic evaluation
    • compute the symbolic expression using slicing
    • find how the index register changes in the loop body
    • return an AST
Simplify Results of Stride Analysis

- Raw AST data using ROSE symbols
  - \(<\text{extract:32}>(<\text{add}>(<\text{extMSB}>(<\text{V}[S[_Z19initialize_matricesv,-24,0]]:80487c8>),<33:32>,),<\text{add}>(<\text{extMSB}>(<4:32>,<33:32>,),<0:1>,),,<0:33>,<32:33>,)\)

- Simply the AST
  - remove unnecessary operators
  - handle the value from the memory
    - index
      - if the value comes from an LHS of an instruction in the loop
    - constant
      - if the value comes from an LHS of an instruction outside the loop
      - if the value is not an LHS \(\rightarrow\) it does not change in the loop
    - indirect
      - if the value is from an unknown location which is an indirect reference
  - simplified version of above expression: \((\text{index}+(0x4+0x0))\)
Eliminate Extraneous Details

- Inner loop details are irrelevant when analyzing outer loop
  - slicing in the inner loop generates extraneous detail
  - for example: \((\text{constant} \times 0x3e8) + ((\text{index} + (0x1 + 0x0)) + 0x0)\)

- Prune the AST to eliminate the extraneous detail
  - keep a sub-AST only if it is related to the index
Preliminary Experimental Results

- **Kernel of matrix multiplication**

  ```c
  for (i = 0 ; i < ROWS ; i++)
  {
    for (j = 0 ; j < COLUMNS ; j++)
    {
      float sum = 0.0;
      for (k = 0 ; k < COLUMNS ; k++)
      {
        sum = sum + matrix_a[i][k] * matrix_b[k][j];
        matrix_r[i][j] = sum;
      }
  }
  ```

  (0x8048660, 0x8048678)

  8048660(0x4): ((index+(0x4+0x0))+(0x4+0x0)) multiplier: 1
  8048668(0xfa0): ((index+(0xfa0+0x0))+(0xfa0+0x0)) multiplier: 1

  (0x8048648, 0x8048688)

  8048660(0x0): constant multiplier: 1
  8048668(0x4): (((0x8419a80+((index+(0x1+0x0))*0x4))+(0xfa0+0x0))+(0xfa0+0x0)) multiplier: 1
  804867d(0x4): ((index+(0x4+0x0))+(0x4+0x0)) multiplier: 1

  (0x804863a, 0x8048693)

  8048660(0xfa0): (((index+(0x1+0x0))*0xfa0)+(0x8049180+0x0)) multiplier: 1
  8048668(0x0): constant multiplier: 1
  8048668(0x0): constant multiplier: 1
  804867d(0xfa0): (((index+(0x1+0x0))*0xfa0)+0x87ea380)+(0x4+0x0)) multiplier: 1
Outline

- GPU profiling
- Detecting memory leaks
- Call path tracing
- More work on scaling
- Applying HPCToolkit to FY2011 “Joule Metric” applications
- Data-centric measurement and analysis
- Static analysis of memory access patterns
- Some challenges ahead
Some Challenges Ahead

• Scale measurement and analysis to > 1M cores
• Handle requirements for asymmetric measurement
• Understand usage of shared resources
  — examples
    • shared cores (SMT)
    • shared cache
    • memory bandwidth
    • network
  — quantify utilization
  — quantify impact of contention
    • aggregate
    • over time
  — attribute metrics to code
• Complete analysis of hybrid programs
• From metrics to bottleneck diagnosis
  — work with PerfExpert team at UT and Texas State