Part III

- **HPCToolkit**: Low overhead, full code profiling using hardware counters sampling
- **MIAMI**: Performance diagnosis based on machine-independent application modeling
CHALLENGES FOR COMPUTATIONAL SCIENTISTS

• Execution environments and applications are rapidly evolving
  • Architecture
    • rapidly changing multicore microprocessor designs, increasing scale of parallel systems, growing use of accelerators
  • Applications
    • adding additional scientific capabilities to existing applications, MPI everywhere to threaded implementations

• Steep increase in application development effort to attain performance, evolvability, and portability

• Application developers need to
  • Assess weaknesses in algorithms and their implementations
  • overhaul algorithms & data structures as needed
  • Adapt to changes in emerging architectures
  • Improve scalability of executions within and across nodes
PERFORMANCE ANALYSIS CHALLENGES

• Complex architectures are hard to use efficiently
  • Multi-level parallelism: multi-core, ILP, SIMD instructions
  • Multi-level memory hierarchy
  • Result: gap between typical and peak performance is huge

• Complex applications present challenges
  • For measurement and analysis
  • For understanding and tuning

Performance tools can play an important role as a guide
HPCToolkit DESIGN PRINCIPLES

• Employ binary-level measurement and analysis
  • observe fully optimized, dynamically linked executions
  • support multi-lingual codes with external binary-only libraries

• Use sampling-based measurement (avoid instrumentation)
  • controllable overhead
  • minimize systematic error and avoid blind spots
  • enable data collection for large-scale parallelism

• Collect and correlate multiple derived performance metrics
  • diagnosis typically requires more than one species of metric

• Associate metrics with both static and dynamic context
  • loop nests, procedures, inlined code, calling context

• Support top-down performance analysis
  • natural approach that minimizes burden on developers
HPCToolkit WORKFLOW

1. compile & link
   - app. source
2. optimized binary
3. profile execution
   - profile
   - [hpcrun]
4. binary analysis
   - binary analysis
   - [hpcstruct]
5. call stack profile
   - call stack profile
6. program structure
   - program structure
7. presentation
   - presentation
   - [hpcviewer/hpctraceviewer]
8. interpret profile correlate w/ source
   - interpret profile correlate w/ source
   - [hpcprof/hpcprof-mpi]
For dynamically-linked executables on stock Linux
  - compile and link as you usually do: nothing special needed

For statically-linked executables (e.g. for Blue Gene, Cray)
  - add monitoring by using `hpclink` as prefix to your link line
  - uses “linker wrapping” to catch “control” operations
    - process and thread creation, finalization, signals, ...

HPCToolkit WORKFLOW

- compile & link
  - from source
  - to optimized binary

- profile execution
  - with `hpcrun`

- call stack profile

- binary analysis
  - with `hpcstruct`

- program structure

- interpret profile
  - correlate with source

- presentation
  - with `hpcviewer` and `hpctraceviewer`

- database

- [hpcrun]
- [hpcstruct]
- [hpcprof]
- [hpcprof-mpi]
**HPCToolkit WORKFLOW**

- Measure execution unobtrusively
  - launch optimized application binaries
    - dynamically-linked applications: launch with `hpcrun` to measure
    - statically-linked applications: measurement library added at link time
      - control with environment variable settings
  - collect statistical call path profiles of events of interest
HPCToolkit WORKFLOW

- Analyze binary with **hpcstruct**: recover program structure
  - analyze machine code, line map, debugging information
  - extract loop nesting & identify inlined procedures
  - map transformed loops and procedures to source
HPCToolkit WORKFLOW

- Combine multiple profiles
  - multiple threads; multiple processes; multiple executions
- Correlate metrics to static & dynamic program structure

- Interpret profile correlate w/ source
- Database
- Presentation
  - hpcviewer
  - hpctraceviewer
• Presentation
  • explore performance data from multiple perspectives
    • rank order by metrics to focus on what’s important
    • compute derived metrics to help gain insight
      • e.g. scalability losses, waste, CPI, bandwidth
  • graph thread-level metrics for contexts
  • explore evolution of behavior over time
ANALYZING RESULTS WITH hpcviewer

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

source pane
view control
metric display
navigation pane
metric pane
PRINCIPAL VIEWS

• Calling context tree view - “top-down” (down the call chain)
  • associate metrics with each dynamic calling context
  • high-level, hierarchical view of distribution of costs
  • example: quantify initialization, solve, post-processing

• Caller’s view - “bottom-up” (up the call chain)
  • apportion a procedure’s metrics to its dynamic calling contexts
  • understand costs of a procedure called in many places
  • example: see where PGAS put traffic is originating

• Flat view - ignores the calling context of each sample point
  • aggregate all metrics for a procedure, from any context
  • attribute costs to loop nests and lines within a procedure
  • example: assess the overall memory hierarchy performance within a critical procedure
• Comprehensive user manual:
• Quick start guide
  • essential overview that almost fits on one page
• Using HPCToolkit with statically linked programs
  • a guide for using hpctoolkit on BG/P and Cray XT
• The hpcviewer user interface
• Effective strategies for analyzing program performance with HPCToolkit
  • analyzing scalability, waste, multicore performance ...
• HPCToolkit and MPI
• HPCToolkit Troubleshooting
  • why don’t I have any source code in the viewer?

• Installation guide
USING HPCToolkit

- Add hpctoolkit’s bin directory to your path
  - Download, build and usage instructions at http://hpctoolkit.org
  - Installed on ICL machines in “/iclscratch1/homes/hpctoolkit”

- Perhaps adjust your compiler flags for your application
  - sadly, most compilers throw away the line map unless -g is on the command line. add -g flag after any optimization flags if using anything but the Cray compilers/ Cray compilers provide attribution to source without -g.

- Decide what hardware counters to monitor
  - dynamically-linked executables (e.g., Linux)
    - use hpcrun -L to learn about counters available for profiling
    - use papi_avail
      - you can sample any event listed as “profilable”
USING HPCToolkit

- Profile execution:
  - `hpcrun -e <event1@period1> [-e <event2@period2> ...] <command> [command-arguments]`
  - Produces one `.hpcrun` results file per thread

- Recover program structure
  - `hpcstruct <command>`
  - Produces one `.hpcstruct` file containing the loop structure of the binary

- Interpret profile / correlate measurements with source code
  - `hpcprof [-S <hpcstruct_file>] [-M thread] [-o <output_db_name>] <hpcrun_files>`
  - Creates performance database

- **Use** `hpcviewer` to visualize the performance database
  - Download `hpcviewer` for your platform from [https://outreach.scidac.gov/frs/?group_id=22](https://outreach.scidac.gov/frs/?group_id=22)
HANDS-ON DEMO

• Recall the matrix-multiply example compiled with two different compilers from Part I of the class

```c
void compute(int reps) {
    register int i, j, k, r;
    for (r=0 ; r<reps ; ++r) {
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                for (k = 0; k < N; k++) {
                    C(i,j) += A(i,k) * B(k,j);
                }
            }
        }
    }
}
```

• Performance questions
  • What is causing performance to vary with matrix size?
  • What factors are limiting performance for each binary?
  • The more efficient version runs at < 50% of peak FLOPS
HANDS-ON DEMO: MAT-MUL PERFORMANCE

Why the gap?

Why the change?

Why the difference?
HANDS-ON DEMO: USING HPCToolkit

• Recall performance inefficiencies from Part I
• Some native performance events for AMD K10

CPU_CLK_UNHALTED – CPU clock cycles / CPU time
RETIRED_INSTRUCTIONS – # instructions retired
RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS – # mispredicted branches
DATA_CACHE_ACCESSSES – # accesses to L1
DATA_CACHE_MISSES – L1 D-cache misses
DATA_CACHE_REFILLS:ALL – L1 cache refills (L1 misses)
DATA_CACHE_REFILLS_FROM_SYSTEM:ALL – L1 refills from system (L3+memory)

L1_DTLB_MISS_AND_L2_DTLB_HIT:ALL – L1 DTLB misses that hit in L2 DTLB
L1_DTLB_AND_L2_DTLB_MISS:ALL – L2 DTLB misses

DATA_PREFETCHES:ATTEMPTED – prefetches initiated by the DC prefetcher
REQUESTS_TO_L2:DATA – requests to L2 from the L1 data cache (includes
L1 misses and DC prefetches)
REQUESTS_TO_L2:HW_PREFETCH_FROM_DC – requests to L2 from the DC
prefetcher
L2_CACHE_MISS:DATA – L2 data cache misses
HANDS-ON DEMO: USING HPCToolkit

INSTRUCTION_CACHE_FETCHES — accesses to L1 I-cache
INSTRUCTION_CACHE_MISSES — L1 I-cache misses
INSTRUCTION_CACHE_REFILLS_FROM_L2 — L1 I-cache refills from L2
INSTRUCTION_CACHE_REFILLS_FROM_SYSTEM — L1 I-cache refills from system

L1_ITLB_MISS_AND_L2_ITLB_HIT — L1 ITLB misses that hit in L2 ITLB
L1_ITLB_MISS_AND_L2_ITLB_MISS:ALL — L2 ITLB misses

INSTRUCTION_FETCH_STALL — CPU cycles when instruction fetch stalled
DECODER_EMPTY — CPU cycles when decoder is idle
DISPATCHSTALLS — CPU cycles when dispatched was stalled
DISPATCH_STALL_FOR_REORDER_BUFFER_FULL — dispatch stalled due to full ROB
DISPATCH_STALL_FOR_RESERVEDATION_STATION_FULL — dispatch stalled due to full reservation station

DISPATCH_STALL_FOR_FPU_FULL
DISPATCH_STALL_FOR_LS_FULL — dispatch store due to LS buffer full

MEMORY_CONTROLLER_REQUESTS:READ_REQUESTS — read memory requests
MEMORY_CONTROLLER_REQUESTS:WRITE_REQUESTS — write memory requests
MEMORY_CONTROLLER_REQUESTS:PREFETCH_REQUESTS — memory prefetch requests
L3_CACHE_MISSES:ANY_READ — data reads that miss in L3
PERFORMANCE ANALYSIS CHALLENGES

• Current tools measure performance effects
  • How much time is spent and how many cache misses are in a loop / routine
  • Pinpoint hotspots
• Do not tell you if what you see is good or bad
• User must determine what factors are limiting performance
• Performance modeling tool
  • MIAMI: Machine Independent Application Models for performance Insight

• Automatically extract application features
  • Works on fully-optimized binaries
  • No performance effects are measured directly

• Separately model target architecture
  • Done manually once per machine

• Compute application performance from first order principles
WHAT IT SOLVES

• Identifies performance limiting factors
• Enables “what if” analysis
• Reveals performance improvement potential
  • Useful for prioritizing work and for understanding if “fixing” is worth the effort
MIAMI DIAGRAM

CSV files / XML performance database

Performance predictions, performance limiters, potential for performance improvement
*map metrics to source code and data structures*

Dependence graph customized for machine

*instruction latencies, idiom replacement*

Loop nesting structure
Dependence graph at loop level

CFGs, edge counts

MIAMI code IR in miasm, ureg / registers

Machine model (MDL)

x86 object code

Prefetching effectiveness

Cache miss predictions

*data reuse insight*

Memory reuse distance analysis

PIN

Streaming concurrency sim.

PIN

Binutils

SymtabAPI

hpcviewer

Dependency graph customized for machine instruction latencies, idiom replacement

Performance predictions, performance limiters, potential for performance improvement
*map metrics to source code and data structures*

Loop nesting structure
Dependence graph at loop level

CFGs, edge counts

MIAMI code IR in miasm, ureg / registers

Machine model (MDL)

x86 object code

Prefetching effectiveness

Cache miss predictions

*data reuse insight*

Memory reuse distance analysis

PIN

Streaming concurrency sim.

PIN

Binutils

SymtabAPI

hpcviewer

Dependency graph customized for machine instruction latencies, idiom replacement

Performance predictions, performance limiters, potential for performance improvement
*map metrics to source code and data structures*

Loop nesting structure
Dependence graph at loop level

CFGs, edge counts

MIAMI code IR in miasm, ureg / registers

Machine model (MDL)

x86 object code

Prefetching effectiveness

Cache miss predictions

*data reuse insight*

Memory reuse distance analysis

PIN

Streaming concurrency sim.

PIN

Binutils

SymtabAPI

hpcviewer
MIAMI DIAGRAM

Diagnose utilization of CPU cores
- Model CPU back-end
- Identify instruction schedule inefficiencies
- Understand potential for improvement

Diagnose cache reuse
- Understand data reuse at each memory level
- Identify memory access patterns with poor locality
- Understand what code and data layout transformations are needed

Diagnose stream prefetching perf.
- Understand data streaming behavior and number of concurrent streams
- Identify memory access patterns unfriendly to the hardware prefetchers

Loop nesting structure
- Dependence graph at loop level

CFGs, edge counts
- PIN

MIAMI code IR intr, µop / registers
- XED

Machine model (MDL)

Memory reuse distance analysis

PIN

CSV file / performance database

PIN

Streaming concurrency sim.

PIN

x86 object code

Performance predictions, performance limiters

Binutils SymtabAPI

hpcviewer

Instruction latencies, idiom replacement

Dependence graph customized for machine

Cache miss predictions

Data reuse insight

Diagnose utilization of CPU cores

Diagnose cache reuse

Diagnose stream prefetching perf.
MACHINE DESCRIPTION LANGUAGE (MDL)

Construct a model of the target architecture

- Enumerate back-end CPU resources
  - Baseline performance limited by the back-end
- Describe instruction execution templates & resource usage
- Scheduling constraints between resources
- Idiom replacement
  - Account for differences in ISAs, micro-architecture features / optimizations
- Memory hierarchy characteristics
- Other machine features
UNDERSTAND CPU CORES UTILIZATION

- Recover application CFG and understand execution frequency of paths in CFG
- Decode native x86 instructions to MIAMI IR
- Map application micro-ops to target machine resources
  - Identify the factors limiting schedule length
    - Application: insufficient ILP, instruction mix, SIMD
    - Architecture: resource contention, retirement rate
  - Idealize the limiting constraints to understand the maximum potential for improvement
MATRIX MULTIPLY
HANDS-ON DEMO
INSIGHT FROM MIAMI

• Understand losses due to insufficient ILP
• Utilization of various machine resources
• Instruction mix
  • Understand if vector instructions are used
• Contention on machine resources
  • Few options from an application perspective, must change instruction mix
  • Contention on load/store unit -> improve register reuse
SUMMARY

• Performance tools help us understand application performance
• HPCToolkit: low overhead, full-code profiler
  • Uses hardware counter sampling through PAPI
  • Maps performance data to functions, loops, calling contexts
  • Intuitive viewer
    • Enables top-down analysis
    • Custom derived metrics enable quick performance analysis at loop level
• MIAMI: performance diagnosis based on performance modeling
  • Uses profiling and static analysis of full application binaries
  • Models CPU back-end to understand the main performance inefficiencies
  • Data reuse and data streaming analysis reveal opportunities for optimization
  • It is a research tool, not publicly available yet