PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications

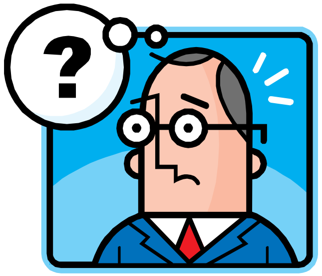
Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne

The University of Texas at Austin

>>>>>>>>>>>>>>>>>>>>>
>>> PerfExpert >>>>>
>>>>>>>>>>>>>>>>>>>>>>

Motivation

- Problem: HPC systems operate far below peak
  - Performance optimization complexity is growing
- Status: Most performance tools are hard to use
  - Require detailed performance and system expertise
  - HPC application developers are domain experts
- Result: HPC programmers do not use these tools
  - 75% of users haven't used performance tool on Ranger
  - Do not know how to apply information



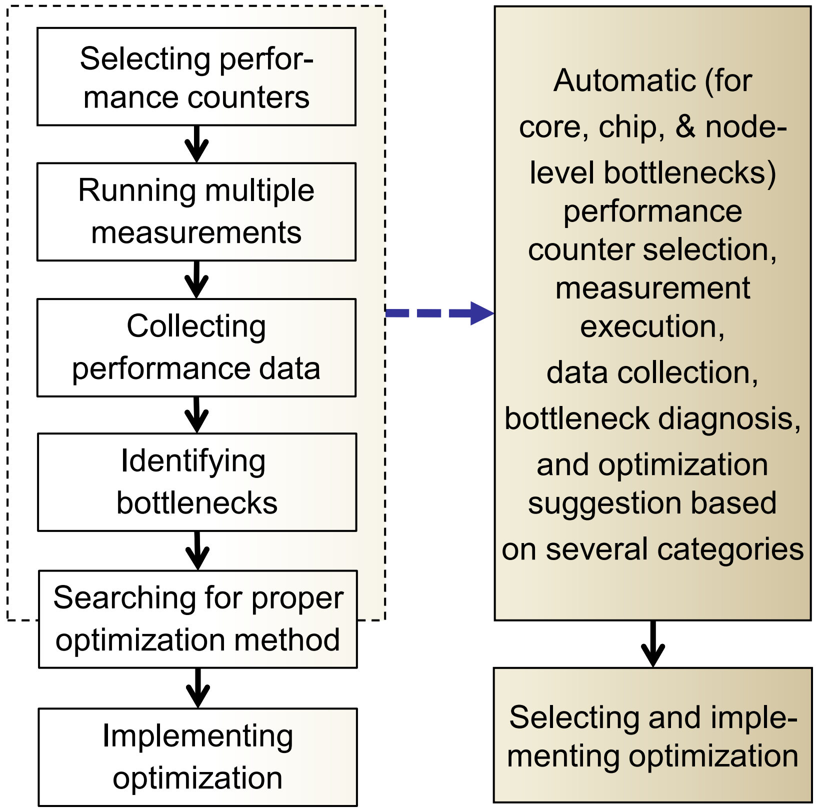
Performance Counter Tool Workflow

Basic tools [mostly manual]

Provide no aid with:

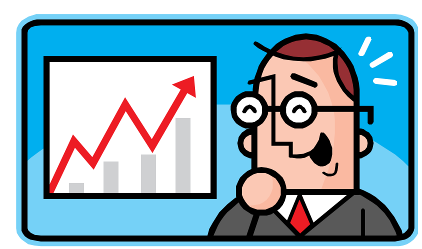
- Counter selection
  - 100s of possibilities
  - cryptic descriptions
  - unclear what counted
- Result interpretation
  - Is there a problem?
  - What is the problem?
- Solution finding
  - How do I fix it?

PerfExpert [mostly automated]



PerfExpert features:

- Automatic bottleneck detection & analysis
  - at core/chip/node level
- Recommends remedy
  - includes code examples & compiler switches
- Simple user interface
  - use provided job script
  - intuitive output



PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications

Overview

- PerfExpert case studies on four Ranger codes
  - Mangll: mantle advection production code (C)
  - Homme: atmospherics acceptance benchmark (F95)
  - Libmesh: Navier-Stokes example code (C++)
  - Asset: astrophysics production code (F90)
- Step-by-step usage example
- Internal operation and performance metric
- Future work
- Summary

Mantle Advection Case Study

- Found code to be memory bound
- 40% speedup due to node-level optimizations

```
total runtime in dgelastic.xml is 75.70 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
             tacc.utexas.edu/perfexpert/

dgae_RHS (59.8% of the total runtime)
----------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- instruction accesses   >>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions    >>
- data TLB               >
- instruction TLB        >
```

procedure identifier

URL to suggested optimizations

overall loop performance is bad

most of runtime is due to data accesses

Atmospheric Circulation Case Study

- Highlights scaling problem due to shared resources

```
total runtime in homme-4x64.xml is 356.73 seconds
total runtime in homme-16x16.xml is 555.43 seconds

...

prim_advance_mod_mp_preq_advance_exp_ (runtimes are 86.35s and 159.[illegible])
--------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>2222222222222222+
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- instruction accesses   >>>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>1
- branch instructions    >
- data TLB               >
- instruction TLB        >
```

comparing two experiments

second much worse than first

Navier-Stokes Case Study

- Illustrates optimization-benefit tracking ability

```
total runtime in ex18.xml is 144.78 seconds
total runtime in ex18-cse.xml is 137.91 seconds

...

NavierSystem::element_time_derivative (runtimes are 33.29s and 25.24s)
----------------------------------------------------------------------------
performance assessment   great.....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>222
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>2
- instruction accesses   >>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>1111111111
- branch instructions    >
- data TLB               >
- instruction TLB        >
```

optimization benefit

Astrophysical Case Study

- Code has already been aggressively optimized

```
total runtime in asset.xml is 52.25 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

calc_intens3s_vec_mexp (27.6% of the total runtime)
----------------------------------------------------------------------------
performance assessment   great....good......okay......bad.......problematic
- overall                >>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses          >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses   >>>>>>>>>
- floating-point instr   >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- branch instructions    >
- data TLB               >
- instruction TLB        >
```

performance is already good

Step-by-Step Usage Example

- Scenario
  - Developer’s HPC code performs poorly
  - May know code section but not how to accelerate it
- Example: matrix-matrix multiplication
  - Coded inefficiently for illustration purposes
- PerfExpert reports **where** the slow code is, **why** it performs poorly, and suggests **how** to improve it

PerfExpert Output for MMM Code

```
total runtime in mmm1.xml is 3.74 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

matrixproduct (100.0% of the total runtime)
----------------------------------------------------------------------------
...
loop at line 25 in matrixproduct (99.7% of the total runtime)
-----------------------------------------------------------------------------
performance assessment   LCPI good......okay......fair......poor......bad....
- overall                 9.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
upper bound by category
- data accesses          14.7 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- instruction accesses    0.6 >>>>>>
- data TLB                9.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
- instruction TLB         0.0 >
- branch instructions     0.1 >
- floating-point instr    3.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

loop identifier (if compiled with “-g”)

overall loop performance is bad

most of runtime is due to data TLB and data accesses

Optimize Critical Code Section

- Loop nest around line 25

```
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];
```

- Identified main bottleneck
  - Cause: memory accesses & data TLB
- Focus on data TLB problem first
  - No need to know what a data TLB is, just used as label to locate corresponding optimizations on web page

Data TLB Optimization Suggestions

1) Improve the data locality

a) use superpages (larger page sizes)

not yet enabled on all Ranger nodes

b) change the order of loops

loop i {...} loop j {...} → loop j {...} loop i {...}

c) employ loop blocking and interchange (change the order of the memory accesses)

loop i {loop k {loop j {c[i][j] = c[i][j] + a[i][k] * b[k][j];}}} →
loop k step s {loop j step s {loop i {for (kk = k; kk < k + s; kk++)
{for (jj = j; jj < j + s; jj++) {c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];}}}}}

2) Reduce the data size

a) use smaller types (e.g., float instead of double or short instead of int)

suggested remedy

double a[n]; → float a[n];

code example

use the "-fpack-struct" compiler flag

compiler flag example

b) allocate an array of elements instead of each element individually

loop {... c = malloc(1); ...} →
top = n; loop {if (top == n) {tmp = malloc(n); top = 0;} ... c = &tmp[top++]; ...}

Eliminate Inapplicable Suggestions

1) Improve the data locality

~~a) use superpages (larger page sizes)~~

~~not yet enabled on all Ranger nodes~~

b) change the order of loops

loop i {...} loop j {...} → loop j {...} loop i {...}

c) employ loop blocking and interchange (change the order of the memory accesses)

loop i {loop k {loop j {c[i][j] = c[i][j] + a[i][k] * b[k][j];}}} →
loop k step s {loop j step s {loop i {for (kk = k; kk < k + s; kk++)
{for (jj = j; jj < j + s; jj++) {c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];}}}}}

2) Reduce the data size

~~a) use smaller types (e.g., float instead of double or short instead of int)~~

~~double a[n]; → float a[n];~~

~~use the "-fpack-struct" compiler flag~~

~~b) allocate an array of elements instead of each element individually~~

~~loop {... c = malloc(1); ...} →~~
~~top = n; loop {if (top == n) {tmp = malloc(n); top = 0;} ... c = &tmp[top++]; ...}~~

Try Remaining Suggestions

- Start with suggestion 1b because it is simpler

1) Improve the data locality

b) change the order of loops

loop i {...} loop j {...} → loop j {...} loop i {...}

- Exchange the *j* and *k* loops of the loop nest

```
for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
      c[i][j] += a[i][k] * b[k][j];
```

- Assess transformed code with PerfExpert

Output after Loop Exchange

```
total runtime in mmm2.xml is 1.45 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

...

loop at line 25 in matrixproduct (99.5% of the total runtime)
-----------------------------------------------------------------------------
performance assessment   LCPI good......okay......fair......poor......bad....
- overall                 4.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
upper bound by category
- data accesses           3.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- instruction accesses    0.5 >>>>>
- data TLB                0.0 >
- instruction TLB         0.0 >
- branch instructions     0.1 >
- floating-point instr    3.3 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

runtime is lower

overall loop performance is better but still bad

data accesses should be optimized next

data TLB is no longer a problem

Data Access Optimization Suggestions

1) Reduce the number of memory accesses

a) move loop invariant memory accesses out of loop

loop i {a[i] = b[i] * c[j]} → temp = c[j]; loop i {a[i] = b[i] * temp;}

b) . . .

2) Improve the data locality

a) componentize important loops by factoring them into their own subroutines

... loop i {...} ... loop j {...} ... → void li() {...}; void lj() {...}; ... li(); ... lj(); ...

b) employ loop blocking and interchange (change the order of the memory accesses)

loop i {loop k {loop j {c[i][j] = c[i][j] + a[i][k] * b[k][j];}}} →
loop k step s {loop j step s {loop i {for (kk = k; kk < k + s; kk++)
{for (jj = j; jj < j + s; jj++) {c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];}}}}}

we will pick this one as it was already suggested before

c) . . .

3) Reduce the data size

. . .

Try Loop Blocking Suggestion

- Blocked loop code (blocking factor s = 70)

```
for (k = 0; k < n; k += s) {
  for (j = 0; j < n; j += s) {
    for (i = 0; i < n; i++) {
      for (kk = k; kk < k + s; kk++) {
        for (jj = j; jj < j + s; jj++) {
          c[i][jj] += a[i][kk] * b[kk][jj];
        }
      }
    }
  }
}
```

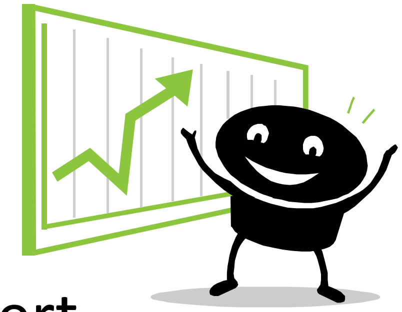
Output after Loop Blocking

```
total runtime in mmm3.xml is 0.28 seconds

Suggestions on how to alleviate performance bottlenecks are available at:
http://www.tacc.utexas.edu/perfexpert/

...

loop at line 28 in matrixproduct (98.8% [illegible]he total runtime)
-----------------------------------------------------------------------------
performance assessment   LCPI good.....okay.....fair.....poor.....bad....
- overall                 0.6 >>>>>>
upper bound by category
- data accesses           2.1 >>>>>>>>>>>>>>>>>>>>>
- instruction accesses    0.6 >>>>>>
- data TLB                0.0 >
- instruction TLB         0.0 >
- branch instructions     0.2 >>
- floating-point instr    2.5 >>>>>>>>>>>>>>>>>>>>>>>>>
```

runtime is much lower

overall loop performance is now good

Usage Example Summary



- Performance is greatly improved
  - Optimization process guided by PerfExpert
  - Runtime dropped by 13x
- Memory access and data TLB problems fixed
  - PerfExpert correctly identified these bottlenecks
  - Suggested useful code optimizations
  - Helped verify the resolution of the problem

Internal PerfExpert Operation

- Gather performance counter measurements
  - Multiple runs with HPCToolkit (PAPI & native counters)
  - Sampling-based results for procedures and loops
- Combine and check results
  - Check variability, runtime, consistency, and integrity
- Compute metrics and output assessment
  - Only for most important code sections
  - Correlate results from different runs

PerfExpert's Performance Metric

- Local Cycles Per Instruction (LCPI)
  - Compute upper bounds on CPI contribution for various categories (e.g., branches, memory) and code sections
    - (BR_INS * *BR_lat* + BR_MSP * *BR_miss_lat*) / TOT_INS
    - (L1_DCA * *L1_dlat* + L2_DCA * *L2_lat* + L2_DCM * *Mem_lat*) / TOT_INS
    - green = performance counter results, blue = system parameters
- Benefits
  - Highlights key aspects and hides misleading details
  - Relative metric (less susceptible to non-determinism)
  - Easily extensible (to refine or add more categories)

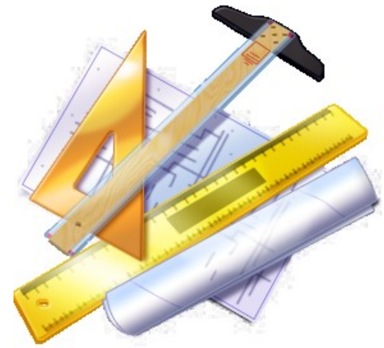
Related Work

- Automatic bottleneck analysis and remediation
  - PERCS project at IBM Research
    - Less automation for bottleneck identification and analysis
    - Not open source
  - PERI Autotuning project
  - Parallel Performance Wizard
    - Event trace analysis, program instrumentation
- Analysis tools with automated diagnosis
- Projects that target multicore optimizations

Future Work

- More case studies
  - Applications with various bottlenecks to harden tool
- Port to other systems: AMD, Intel, Power & GPU
  - Make PerfExpert available for general download
- Improve and expand capabilities
  - Finer-grained recommendations
  - Add data structure based analyses and optimizations
  - Automatic implementation of solutions to common core, chip and node-level performance bottlenecks

Summary



- PerfExpert performance diagnosis tool
  - Automates measurement and analysis
  - Uses new LCPI metric to compare counter results
  - Recommends optimizations for each bottleneck
  - Easy-to-use interface and understandable output
    - code sections sorted by importance
    - longer bars mean more important to optimize

Try it out on Ranger!

- Acknowledgments
  - Omar Ghattas' group, John Mellor-Crummey's group
  - National Science Foundation OCI award #0622780