PerfExpert: (Partial) Automation of Performance Optimization for Multi/Many-Core Chips and Nodes

Ashay Rane – ashay.rane@tacc.utexas.edu
Martin Burtscher – burtscher@txstate.edu
Jim Browne – browne@cs.utexas.edu
Knowledge Bases and Contributors

• Architecture, compilers, source code structures, experimental analyses, validations
• J. Diamond, B. D. Kim, S. Keckler and K. Pingali, J. McCalpin and L. Koesterke and O. A. Sopeju
Goals for Tutorial

- Introduction to PerfExpert
  - See how easy it is to use

- Understand how PerfExpert works
  - Appreciate the sophistication of its engine

- Feedback to us on all aspects
  - Improving PerfExpert’s features and capabilities

- Go forth and use PerfExpert on your codes!
Talk Overview

- Why PerfExpert?
- What is PerfExpert?
- Simple Example
- Extension to Data Structure Analysis
- Extension to Heterogeneous Architectures
- Future Development
Why PerfExpert?

• Problem: HPC systems operate far below peak
  – Chip/node performance optimization complexity is growing

• Status: Use of current tools is limited
  – Powerful in the hands of experts
  – Require architecture and compiler expertise
  – Tell where and what but not how to optimize

• 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
What is PerfExpert - Design Goal

• Simplicity is paramount
  – Focus on chip/node level optimization
  – Follow through to optimization
  – Trivial user interface, easily understandable output
  – No annotations of source code, etc.

• The four stages of automatic perf. optimization
  i. Measurement and attribution (HPCToolkit)
  ii. Diagnosis of bottlenecks (PerfExpert)
  iii. Selection of effective optimizations (PerfExpert)
  iv. Implementation of optimizations (future work)
PerfExpert Tool

• Not only measures but also analyses performance
  – Tells user **where** the slow code sections are as well as **why** they perform poorly

• Suggests source-code changes
  – Tells user **how** to fix the problem

• Implemented using **HPCToolkit** for measurement and source code mapping
Simple Example

• Application developer writes program
  – Implements sophisticated algorithm
  – Processes large amount of data

• Simple example: 3D integration \((n = 1 \text{ billion})\)

```c
#pragma omp parallel for num_threads(threads) default(none) \
private(i) shared(n, dt, posx, posy, posz, velx, vely, velz)
  for (i = 0; i < n; i++) {
    posx[i] += velx[i] * dt;
    posy[i] += vely[i] * dt;
    posz[i] += velz[i] * dt;
  }
```
Program Execution

• Developer tests program on target machine
  – For example, Ranger node with 16 cores

• Is performance good?
  – Takes 3.2 seconds per time step using 16 threads

• Developer applies PerfExpert

PerfExpert (Ranger)

Loop in function main() at Integrator.c:81 (98.6% of the total runtime)

(Loop at line 81 is the 3D integration loop)
PerfExpert Process - Diagnosis

• Metrics
  – Computational work – percent of “optimal”
    • How good is performance of code segment
  – Contribution of each step of instruction execution to CPI for each code segment – LCPI
    • How “optimal” is each step of instruction execution
  – How do we know “optimal?” Answer in Mr. Rane’s lecture.
PerfExpert Analysis Output

<table>
<thead>
<tr>
<th>Loop in function main() at Integrator.c:81 (98.6% of the total runtime)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio to total instrns</td>
</tr>
<tr>
<td>- floating point</td>
</tr>
<tr>
<td>- data accesses</td>
</tr>
<tr>
<td>* GFLOPS (% max)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>performance assessment</th>
<th>LCPI good......okay......fair......poor......bad....</th>
</tr>
</thead>
<tbody>
<tr>
<td>* overall</td>
<td>4.0 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>upper bound estimates</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>* data accesses</td>
<td>36.3 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;+</td>
</tr>
<tr>
<td>- L1d hits</td>
<td>2.0 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
<tr>
<td>- L2d hits</td>
<td>2.6 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;+</td>
</tr>
<tr>
<td>- L2d misses</td>
<td>31.7 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;+</td>
</tr>
<tr>
<td>* instruction accesses</td>
<td>0.4 &gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
<tr>
<td>- L1i hits</td>
<td>0.4 &gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
<tr>
<td>- L2i hits</td>
<td>0.0 &gt;</td>
</tr>
<tr>
<td>- L2i misses</td>
<td>0.0 &gt;</td>
</tr>
<tr>
<td>* data TLB</td>
<td>0.0 &gt;</td>
</tr>
<tr>
<td>* instruction TLB</td>
<td>0.0 &gt;</td>
</tr>
<tr>
<td>* branch instructions</td>
<td>0.1 &gt;&gt;</td>
</tr>
<tr>
<td>- correctly predicted</td>
<td>0.1 &gt;&gt;</td>
</tr>
<tr>
<td>- mispredicted</td>
<td>0.0 &gt;</td>
</tr>
<tr>
<td>* floating-point instr</td>
<td>1.2 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
<tr>
<td>- fast FP instr</td>
<td>1.2 &gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;</td>
</tr>
<tr>
<td>- slow FP instr</td>
<td>0.0 &gt;</td>
</tr>
</tbody>
</table>

overall loop performance is bad
biggest problem is data accesses that miss in the L2 cache
remaining performance categories are pretty good
Recommendations for Optimization

• Status
  – Know that there is a performance problem
  – Know where the slow code is (3D integration loop)
  – Know why it performs poorly (L2 misses)
  – Still do not know how to improve performance

• Optimization Phase of PerfExpert
  – Suggests remedies based on analysis results
    • Including code examples and compiler flags
  – Recommends optimizations for integration loop
PerfExpert Process – Select Optimization

• Library of code templates and compiler switches

• Set of rules for mapping each cause of “bad” performance to a near optimal code pattern
  – Based on knowledge of architecture, compilers and code pattern behaviors
  – Steadily growing database
PerfExpert with Optimization
Recommendations

• Running with PerfExpert Optimization
  – The maximum of three limits the number of recommendations per hot code sequence to top 3

```
perfexpert_run_exp ./integrator 1073741824 10 16
perfexpert -r=3 0.1 experiment-integrator.xml
```
Optimization Recommendations

Loop in function main() at Integrator.c:81  (98.9% of the total runtime)

apply loop fission so every loop accesses just a couple of different arrays

```
loop i {a[i] = a[i] * b[i] - c[i];} →
loop i {a[i] = a[i] * b[i];}  loop i {a[i] = a[i] - c[i];}
```

pad memory areas so that temporal elements do not map to same set in cache

```
double a[const * cache_size/8], b[const * cache_size/8];
loop i {... a[i] + b[i] ...} →
double a[const * cache_size/8 + 16], b[const * cache_size/8 + 16];
loop i {... a[i] + b[i] ...}
```

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++]; ...
}
```
Choose Applicable Suggestions

Loop in function main() at Integrator.c:81 (98.9% of the total runtime)

apply loop fission so every loop accesses just a couple of different arrays

```c
loop i {a[i] = a[i] * b[i] - c[i];}
loop i {a[i] = a[i] * b[i];}  loop i {a[i] = a[i] - c[i];}
```

pad memory areas so that temporal elements do not map to same set in cache

```c
double a[const * cache_size/8], b[const * cache_size/8];
loop i {... a[i] + b[i] ...}
double a[const * cache_size/8 + 16], b[const * cache_size/8 + 16];
loop i {... a[i] + b[i] ...}
```

allocate an array of elements instead of each element individually

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

let’s try this optimization

doesn’t apply, we have no mallocs
Apply Code Transformation

• Every loop should access just two different arrays

```c
#pragma omp parallel num_threads(threads) default(none) \ 
private(i) shared(n, dt, posx, posy, posz, velx, vely, velz) {
    #pragma omp for
    for (i = 0; i < n; i++) {
        posx[i] += velx[i] * dt;
    }
    #pragma omp for
    for (i = 0; i < n; i++) {
        posy[i] += vely[i] * dt;
    }
    #pragma omp for
    for (i = 0; i < n; i++) {
        posz[i] += velz[i] * dt;
    }
}
```
Performance Comparison

<table>
<thead>
<tr>
<th>threads</th>
<th>runtime/step [s] before</th>
<th>runtime/step [s] after</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>10.5</td>
<td>10.5</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>39.3</td>
<td>16.6</td>
<td>2.36</td>
</tr>
<tr>
<td>2</td>
<td>20.2</td>
<td>8.9</td>
<td>2.28</td>
</tr>
<tr>
<td>4</td>
<td>10.4</td>
<td>4.9</td>
<td>2.12</td>
</tr>
<tr>
<td>8</td>
<td>5.6</td>
<td>2.9</td>
<td>1.91</td>
</tr>
<tr>
<td>16</td>
<td>3.2</td>
<td>2.3</td>
<td>1.40</td>
</tr>
</tbody>
</table>

- Serial performance
  - Compiler converts both versions to same code
  - Optimization not useful

- Parallel performance
  - At least 40% faster
  - DRAM banks can stay open longer, making memory accesses faster
PerfExpert - Status

- Has been used to successfully optimize dozens of production codes for multicore execution
- Has been installed at more than 25 large sites in at least 6 countries
- Has tutorial support and documentation
- Can be downloaded from http://www.tacc.utexas.edu/perfexpert
- But is still very much a work in progress
Data Structure Execution Behavior

• PerfExpert:Current – Code segment, counter-based measurement and analysis - optimizations limited to code refactoring
• What about optimizations based on data structure modifications
• Memory Access Characterization for Performance Optimization – MACPO
MACPO Tool

• Generates traces of memory accesses and times for each **data structure** in an “important” code segment
• Derives strides, reuse distance, access latencies for each level of cache and NUMA accesses.
• Uses:
  – Enhanced bottleneck diagnoses – can diagnosis some problems not revealed by code behavior
  – Enhanced optimization specifications - blocking factors
  – Basic to process for identifying code segments for accelerator execution
New Optimization - Mapping to Accelerators

1. Heterogeneous nodes – multicore + manycore chips

2. Use of accelerators – best optimization for some code segments

3. Which code segments should be mapped to accelerators?
Which Code Segments?

1. Optimize for multicore chip execution - PerfExpert Why?
2. Eliminate kernels not suitable for accelerator execution. How?
3. Identify the kernels suitable for accelerator execution – What properties?
Unsuitable Kernels

- Frequent TLB misses
- High fraction of mis-predicted branches
- Cache conflicts across cores
- Irregular access strides for kernel data structures
“Good” Kernels

- Computational intensity
- Scalable “node-local” SPMD parallelism
- Streaming parallelism or vectorization
- Regular access strides for data structures
- High data reuse factor and low data transfer volume
Moil Example - Validation

Moil supports the usual set of tools for molecular modeling by classical mechanics, including energy calculations, energy minimization, molecular dynamics, and more.
Moil Example - Validation

• Loops with both computational intensity and significant FLOP count
  – Loop in passf4 dp() 46%
  – Loop in passb4 dp() 45%
  – Loop in CalcCpuEForceNBParallelListWater jGTi() 23%
  – Loop in CalcCpuEForceNBParallelListNW jGTi() 19%
  – Loop in CalcCpuEForceNBParallelListNW jGTi() 17%
Moil Example - Validation

• Scaling – Speedup with 1,2,3,4,5 and 8 cores
  – Loop in passf4 dp() 1.0 2.1 3.0 4.3 5.0 8.3
  – Loop in passb4 dp() 1.0 1.9 3.0 4.0 4.9 8.0
  – Function CalcCpuEForceNBParallelListWater jGTi()
    1.0 2.0 3.0 4.0 5.0 8.0
  – Function CalcCpuEForceNBParallelListNW jGTi()
    1.0 2.0 3.0 3.9 4.9 -
Moil Example - Validation

• Reuse and Strides
  – Loop in passf4 dp() 6 cache lines 0.7
  – Loop in passb4 dp() 6 cache lines 0.2 to 0.6
  – Function CalcCpuEForceNBParallelListListWater jGTi() 10 cache lines 0.9
  – Function CalcCpuEForceNBParallelListListNW jGTi() 12 cache lines 1.0
  – Loop in convolution z dp() 5 cache lines 1.0
Scaling of Recommended Functions

![Graph showing speedup vs. threads for different calculations (CPU Water & non-Water calc, MIC Water & non-Water calc).]
Moil – Entire Application

![Graph showing speedup vs. threads for CPU and MIC with 31 threads. The graph illustrates the comparison of CPU and MIC speedup as the number of threads increases, highlighting the performance advantage of MIC over CPU.]
Summary and Future Work

• Summary - PerfExpert is an easy to use and proven tool that automates much of performance optimization for multicore chips and multichip nodes.

• Future Work
  – Add data structure based optimizations
  – Optimization by mapping to accelerators will be added to PerfExpert
  – Automate some optimizations
Future Work

• Generate portable/releasable version of MACPO
• Complete integration of MACPO measurements into PerfExpert
• Extend optimizations to include heterogeneous execution environments
• Automate implementations for subset of optimization recommendations including translation to CUDA/OpenCL and OpenMP
References

• Papers, tutorials and a Quick Start Guide for PerfExpert can be found on the PerfExpert web page.
  http://www.tacc.utexas.edu/perfexpert

• An extended abstract on MACPO can be found at:

• Papers on selection of code segments for accelerator chip execution can be found at:
  http://dl.acm.org/citation.cfm?id=2148629&dl=ACM&coll=DL&CFID=83017535&CFTOKEN=47476358
  and
  http://www.tacc.utexas.edu/documents/13601/b9cc6eaa-d79e-4d78-80ef-cf1bec05085a