Friends Don’t Let Friends Tune Code

Jeffrey K. Hollingsworth
University of Maryland
hollings@cs.umd.edu

Ananta Tiwari (UMD)
About the Title
Why Automate Performance Tuning?

- Too many parameters that impact performance.
- Optimal performance for a given system depends on:
  - Details of the processor
  - Details of the inputs (workload)
  - Which nodes are assigned to the program
  - Other things running on the system
- Parameters come from:
  - User code
  - Libraries
  - Compiler choices

*Automated Parameter tuning can be used for adaptive tuning in complex software.*
Automated Performance Tuning

- **Goal:** Maximize achieved performance

- **Problems:**
  - Large number of parameters to tune
  - Shape of objective function unknown
  - Multiple libraries and coupled applications
  - Analytical model may not be available

- **Requirements:**
  - Runtime tuning for long running programs
  - Don’t try too many configurations
  - Avoid gradients
Active Harmony

- Runtime performance optimization
  - Can also support training runs

- Automatic library selection (code)
  - Monitor library performance
  - Switch library if necessary

- Automatic performance tuning (parameter)
  - Monitor system performance
  - Adjust runtime parameters

- Hooks for Compiler Frameworks
  - Working to integrate USC/ISI Chill
  - Looking at others too
Parallel Rank Ordering Algorithm

- All, but the best point of simplex moves.
- Computations can be done in parallel.
Application Parameter Tuning: GS2

- Physics application (DOE SciDAC project)
- Developed to study low-frequency turbulence in magnetized plasma
- Performance (execution time) improvement by changing layout and three parameters (ngrid, ntheta, nodes)
- Data layout analysis (benchmarking runs)
  - 55.06s → 16.25s (3.4x faster, W/O collision)
  - 71.08s → 31.55s (2.3x faster, W collision)
Generate and evaluate different optimizations that would have been prohibitively time consuming for a programmer to explore manually.

SMG2000 Optimization

Outlined Code
for (si = 0; si < stencil_size; si++)
    for (kk = 0; kk < hypre__mz; kk++)
        for (jj = 0; jj < hypre__my; jj++)
            for (ii = 0; ii < hypre__mx; ii++)
                rp[((ri+ii)+(jj*hypre__sy3))+(kk*hypre__sz3)] -=
                    ((Ap_0[((ii+(jj*hypre__sy1))+(kk*hypre__sz1))]+ 
                        (((A->data_indices)[i])[si]))*
                    (xp_0[((ii+(jj*hypre__sy2))+(kk*hypre__sz2)) + 
                        (( *dxp_s)[si])));  

CHiLL Transformation Recipe
permute([2,3,1,4])
tile(0,4,TI)
tile(0,3,TJ)
tile(0,3,TK)
unroll(0,6,US)
unroll(0,7,UI)

Constraints on Search
0 \leq TI , TJ, TK \leq 122
0 \leq UI \leq 16
0 \leq US \leq 10
compilers \in \{gcc, icc\}

Search space:
122^3 \times 16 \times 10 \times 2 = 581M points
SMG2000 Search and Results

Parallel search evaluates 490 points and converges in 20 steps

Selected parameters:
TI=122, TJ=106, TK=56, UI=8, US=3, Comp=gcc

Performance gain on residual computation:
2.37X

Performance gain on full app:
27.23% improvement
Auto Tuning For Different Platforms

- **Fixed parameters:**
  - Code: PMLB
  - Processors: 64

- **Study how parameters differ for the two systems**

- **Use harmony determined parameters from one system**
  - Run a post-line (fix parameters for entire run) run on another

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Speedup (post-line) run on UMD Cluster</th>
<th>Speedup (post-line) run on Carver Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UMD Best Config</td>
<td>Carver Best Config</td>
</tr>
<tr>
<td>384³</td>
<td>1.44</td>
<td>1.19</td>
</tr>
<tr>
<td>448³</td>
<td>1.42</td>
<td>1.13</td>
</tr>
<tr>
<td>512³</td>
<td>1.30</td>
<td>1.26</td>
</tr>
<tr>
<td>576³</td>
<td>1.38</td>
<td>1.16</td>
</tr>
</tbody>
</table>
Autotuning PFloTran (Trisolve)

Outlined Code
#define SIZE 15
void forward_solve_kernel( ... ) {
    ....
    for (cntr = SIZE - 1; cntr >= 0; cntr--) {
        x[cntr] = t + bs * (*vi ++);
        for (j=0; j<bs; j++)
            for (k=0; k<bs; k++)
                s[k] -= v[cntr][bs* j+k] * x[cntr][j];
    }
}

Constraints on Search
0 <= u1 <= 16
0 <= u2 <= 16
compilers ∈ {gnu, pathscale, cray, pgi}

CHiLL Transformation Recipe
original()
known(bs > 14)
known(bs < 16)
unroll(1,2,u1)
unroll(1,3,u2)

Search space:
17x17x4 = 1156 points
# PFloTran: Trisolve Results

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Original Time</th>
<th>Original Time</th>
<th>Active Harmony (u1,u2)</th>
<th>Speedup</th>
<th>Exhaustive Time</th>
<th>Exhaustive (u1,u2)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathscale</td>
<td>0.58</td>
<td>0.32</td>
<td>(3,11)</td>
<td>1.81</td>
<td>0.30</td>
<td>(3,15)</td>
<td>1.93</td>
</tr>
<tr>
<td>gnu</td>
<td>0.71</td>
<td>0.47</td>
<td>(5,13)</td>
<td>1.51</td>
<td>0.46</td>
<td>(5,7)</td>
<td>1.54</td>
</tr>
<tr>
<td>pgi</td>
<td>0.90</td>
<td>0.53</td>
<td>(5,3)</td>
<td>1.70</td>
<td>0.53</td>
<td>(5,3)</td>
<td>1.70</td>
</tr>
<tr>
<td>cray</td>
<td>1.13</td>
<td>0.70</td>
<td>(15,5)</td>
<td>1.61</td>
<td>0.69</td>
<td>(15,15)</td>
<td>1.63</td>
</tr>
</tbody>
</table>

Trisolve Optimization (with cray)

Trisolve Optimization (with gnu)
Compiling New Code Variants at Runtime

Outlined code-section

PM₁, PM₂, ..., PMₙ

Search Steps (SS)

Harmony Timeline

Active Harmony

Code Server

Code Transformation Parameters

Code Generation

Tools

v₁so

v₂so

vₙso

Compiler

Application

Execution timeline

Performance Measurements (PM)

PM₁

PM₂

PMₙ

Application Timeline

Harmony

SS₁ᵢ

SS₂ᵢ

SSₙᵢ

Application Execution timeline

SS₁ᵢ

SS₂ᵢ

SSₙᵢ

READY Signal

Code Transformation Parameters

PM₁, PM₂, ..., PMₙ

Search Steps (SS)
Online Code Generation Results

- **Two platforms**
  - umd-cluster (64 nodes, Intel Xeon dual-core nodes) – myrinet interconnect
  - Carver (1120 compute nodes, Intel Nehalem. two quad core processors) – infiniband interconnect

- **Code servers**
  - UMD-cluster – local idle machines
  - Carver – outsourced to a machine at umd

- **Codes**
  - Poisson Solver
  - PMLB Parallel Multi-block Lattice Boltzman
  - SMG2000
## How Many Nodes to Generate Code?

- **Fixed parameters:**
  - Code: poission solver
  - problem-size \( (1024^3) \)
  - number of processors (128)

- **Up to 128 new variants are generated at each search step**

<table>
<thead>
<tr>
<th>Code Servers</th>
<th>Search Step s+</th>
<th>Stalled steps+</th>
<th>Variations evaluated+</th>
<th>Speedup+</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6*</td>
<td>46</td>
<td>502</td>
<td>0.75</td>
</tr>
<tr>
<td>2</td>
<td>17*</td>
<td>13</td>
<td>710</td>
<td>0.97</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>7.2</td>
<td>928</td>
<td>1.04</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>4.5</td>
<td>818</td>
<td>1.23</td>
</tr>
<tr>
<td>12</td>
<td>22</td>
<td>4.1</td>
<td>833</td>
<td>1.21</td>
</tr>
<tr>
<td>16</td>
<td>26</td>
<td>3.6</td>
<td>931</td>
<td>1.24</td>
</tr>
</tbody>
</table>

* Search did not complete before application terminated
+ Mean of 5 runs
Conclusions and Future Work

- **Ongoing Work**
  - More end-to-end Application Studies
  - Continued Evaluation of Online Code Generation

- **Conclusions**
  - Auto tuning can be done at many levels
    - Offline – using training runs (choices fixed for an entire run)
      - Compiler options
      - Programmer supplied per-run tunable parameters
      - Compiler Transformations
    - Online – training or production (choices change during execution)
      - Programmer supplied per-timestep parameters
      - Compiler Transformations
  - It Works!
    - Real programs run faster