Model-guided Automatic Performance Tuning

Apan Qasem
Texas State University
San Marcos, TX

*This work was supported by DOE, NSF, Intel and HP
A Recognized Problem

- Microprocessor architecture becoming increasingly complex
  - Many components impact performance
- Many program transformations
  - Transformations sensitive to architectural parameters and input program
  - Interaction between transformations

Lack of *performance portability*
Static heuristics used by traditional compilers unable to optimize programs effectively across architectures
Alternatives to Traditional Optimization

- Manual tuning of code
  - Costly: many person-months
  - Error prone
  - Maintainability can be an issue
- Empirically tuned libraries
  - More Promising
  - Close to hand-tuned performance
  - ATLAS, Spiral, FFTW has gained wide acceptance in their respective domains
Matrix-multiply Search Space

Best Point
Problem with Automatic Tuning

- Search space large and complex
  - Not always possible to exploit domain specific properties
  - Search space grows with program size
    - More loops imply more search dimensions
  - Transformations may have unconstrained numerical parameters
  - Search space representation is not always intuitive
    - Loop Fusion
    - Data Layout Strategies
- Evaluating all points on the search space is intractable
Outline

• Overview of framework
• Experimental study with heuristic search strategies
• Search space pruning
  – Loop fusion and tiling
• Conclusions and future work
LoopTool

- **LoopTool: A Source-to-source transformer**
  - Performs transformations such as loop fusion, tiling, unroll-and-jam
  - Applies transformations from source code annotation
  - Provides *fine-grain control* of transformations
  - Decouples search and code generation

```c
  cdir$ unroll 4
  do j = 1, N
      cdir$ block 16
      do i = 1, M
          cdir$ block 16
          do k = 1, L
              S1(k, i, j)
          enddo
      enddo
  enddo
```
Feedback

• HPCToolKit [Mellor-Crummey et. al., JOS02]
  – hpcrun - profiles executions using statistical sampling of
    hardware performance counters
  – bloop - retrieves loop structure from binaries
  – hpcview - correlates program structure information with
    sample-based performance profiles

• Key benefit
  – Performance measurements at loop-level granularity
  – Feedback beyond whole program execution time
    • Cache misses, loads, TLB misses
Search Module

• Implements a number of search strategies
  – Direct search
  – Simulated Annealing
  – Window Search
  – Random Search

• Flexible
  – Can operate on both transformation and architectural parameters
Outline

• Overview of framework
• Experimental study with heuristic search strategies
  – Loop fusion and tiling
• Search space pruning
• Conclusions and future work
Finding a Suitable Search Strategy

- Performed experimental study using a Direct Search strategy
- Explored search space of tiling and loop unrolling
  - Search space as large as $10^5$ points for some applications
- Goal was to evaluate the suitability of different search strategies
Why Direct Search?

• Search decision based solely on function evaluations
  – No modeling of the search space required
• Provides approximate solutions at each stage of the calculation
  – Can stop the search at any point when constrained by tuning time
• Flexible
  – Can tune step sizes in different dimensions
• Parallelizable
• Relatively easy to implement
Search Space: Bad Values
Search Space : Good Values
Findings

• Direct Search is able to find suitable tile sizes and unroll factors by exploring only a small fraction of the search space

• Search space pruning is essential for making searches more efficient
Outline

• Overview of framework
• Experimental study with heuristic search strategies
• Search space pruning
  – Loop fusion and tiling
• Conclusions and future work
Loop Fusion and Tiling

- Well-known and important transformations for improving memory hierarchy performance
- Making the right fusion and tiling choices is non-trivial
  - Tile size choice depends on fusion decision
  - Fusion decision influenced by tile size choice
  - Profitability depends on the underlying architecture
    - Conflict misses
    - Register Pressure
    - Different effects on different levels of the memory hierarchy
- If we do not get it right we might hurt performance!
LA: 
\begin{align*}
\text{do } & j = 1, N \\
\text{do } & i = 1, M \\
\quad & b(i,j) = a(i,j) + a(i,j-1) \\
\text{enddo} \\
\text{enddo}
\end{align*}

LB: 
\begin{align*}
\text{do } & j = 1, N \\
\text{do } & i = 1, M \\
\quad & c(i,j) = b(i,j) + d(j) \\
\text{enddo} \\
\text{enddo}
\end{align*}

(a) code before transformations
\[ \begin{align*}
L_{AB}: & \quad \text{do } j = 1, N \\
& \quad \text{do } i = 1, M \\
& \quad b(i, j) = a(i, j) + a(i, j-1) \\
& \quad c(i, j) = b(i, j) + d(j) \\
& \quad \text{enddo} \\
& \quad \text{enddo} \\
\end{align*} \]

- Lost reuse of \( a() \)
- Saved loads of \( b() \)
- Increased potential for conflict misses

(b) code after two-level fusion
\[ \text{do } i = 1, M, T \]
\[ \text{do } j = 1, N \]
\[ \text{do } ii = i, \text{MIN}(i + T - 1, M) \]
\[ b(ii, j) = a(ii, j) + a(ii, j-1) \]
\[ c(ii, j) = b(ii, j) + d(j) \]

\text{enddo}
\text{enddo}
\text{enddo}

How do we pick \( T \)?

\text{regained reuse of } a() \quad \text{reduced reuse of } d() \quad \text{Not too difficult if caches are fully associative}

\text{Can use models to estimate effective cache size for set-associative caches}

\text{Model unlikely to be totally accurate}
\quad - \text{Need a way to correct for inaccuracies}
Search Space Pruning

• Key Idea:
  
  Search for architecture-dependent model parameters rather than transformation parameters

• Fundamentally different way of looking at the optimization search space
• Implemented for loop fusion and tiling
Estimates of architectural parameters

New Search Space has only two dimensions!

Architectural Parameters

Register Set

F0 ………F 2L-1

L1 Cache

Effective Cache Capacity

Register Set

Fusion Config.

Effective Register Set

Tile Size

Cost Model

Search Space

Estimates of architectural parameters

Search Space has only two dimensions for New Search Space

(\(N-1\) dimensions)

(\(L + 1\))
Tuning Parameters

- Use a tolerance term to determine how much of a resource we can use at each tuning step

**Effective Register Set** = \[ t \times \text{Register Set Size} \]
\[ 0 < t \leq 1 \]

**Effective Cache Capacity** = \( E(a, s, t) \)
\[ 0.01 \leq t \leq 0.20 \]
Search Strategy

- Each tuning parameter constitutes a single search dimension
- Start off conservatively with a low tolerance value and increase tolerance at each step
- Search is *sequential* and *orthogonal*
  - stop when performance starts to worsen
  - use reference values for other dimension when searching a particular dimension
Fusion Parameter Search Space

Effective Register Set Search Space

Search Stops Here
Performance Across Architectures

Mean speedup over baseline

- MIPS
- Itanium
- Alpha
- PowerPC
- Opteron
- Pentium 4
- Pentium III
- Core 2 Duo
- Mean

model-based native
Performance Comparison with Direct Search

![Bar chart showing speedup over baseline for different benchmarks.]
Tuning Time Comparison with Direct Search

![Bar chart comparing tuning time for various benchmarks using model-based and direct methods.](chart.png)
Conclusions

– Approach of tuning for architectural parameters can significantly reduce the optimization search space
  • Single parameter captures the effects of multiple transformations
  • Search Space does not grow with program size
– Search space is more predictable
– Search compensates for inaccuracies in cost model
– Small penalty in terms of performance
Future Work

• Extend pruning strategy to include more transformations
  – Software Prefetching

• Explore other architectural parameters
  – Shared cache on multi-core platforms

• Investigate other objective functions
  – Power
Thank You
Problem Scope

- Target: Dense array computation
- Search Space: Numerical Parameters
- Method: Model-guided search
## Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Description</th>
<th>Fusible Loops</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sweep3d</td>
<td>LANL</td>
<td>3D neutron transport</td>
<td>0</td>
<td>2126</td>
</tr>
<tr>
<td>impact3d</td>
<td>H Sakagami</td>
<td>fluid dynamics simulation</td>
<td>27</td>
<td>1618</td>
</tr>
<tr>
<td>erlebacher</td>
<td>T Eidson</td>
<td>differential equation solver</td>
<td>30</td>
<td>686</td>
</tr>
<tr>
<td>advect3d</td>
<td>NCOMMAS</td>
<td>3D advection for weather modeling</td>
<td>21</td>
<td>545</td>
</tr>
<tr>
<td>riemann</td>
<td>P Woodward</td>
<td>hyperbolic equation solver (parabolic method)</td>
<td>22</td>
<td>494</td>
</tr>
<tr>
<td>vpenta</td>
<td>NAS kernel</td>
<td>3D pentadiagonal inversion</td>
<td>3</td>
<td>145</td>
</tr>
<tr>
<td>lud</td>
<td>netlib.org</td>
<td>LU decomposition (matrix vector multiply)</td>
<td>0</td>
<td>131</td>
</tr>
<tr>
<td>liv18</td>
<td>Livermore loops</td>
<td>2D explicit hydrodynamics fragment</td>
<td>6</td>
<td>130</td>
</tr>
<tr>
<td>mm</td>
<td>Textbook</td>
<td>N x N matrix multiply</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>swim</td>
<td>SPEC FP 2000</td>
<td>weather prediction</td>
<td>3</td>
<td>282</td>
</tr>
<tr>
<td>mgrid</td>
<td>SPEC FP 2000</td>
<td>multi-grid solver</td>
<td>1</td>
<td>344</td>
</tr>
<tr>
<td>sp</td>
<td>NAS 2.3 Serial</td>
<td></td>
<td>30</td>
<td>3028</td>
</tr>
<tr>
<td>bt</td>
<td>NAS 2.3 Serial</td>
<td></td>
<td>31</td>
<td>3711</td>
</tr>
</tbody>
</table>