A Scalable Auto-Tuning Framework for Compiler Optimization

Ananta Tiwari\textsuperscript{1}, Chun Chen\textsuperscript{2}, Jacqueline Chame\textsuperscript{3}, Mary Hall\textsuperscript{2} and Jeff Hollingsworth\textsuperscript{1}

tiwari@cs.umd.edu

\textsuperscript{1}Department of Computer Science
University of Maryland, College Park, MD

\textsuperscript{2}School Of Computing
University of Utah, Salt Lake City, UT

\textsuperscript{3}Information Sciences Institute
University of Southern California, Marina del Ray, CA

IPDPS 2009 (Rome, Italy)
The Rationale

- Complex and diverse architectures
  - Processor utilization
  - Cost

- Manual tuning
  - Focus on key computational units
  - Write/Select among alternate implementations

- Pitfalls
  - Maintenance/Programmer productivity
  - Porting
Existing Tools

● Three Categories
  - Self tuning libraries (ATLAS, OSKI)
  - Compiler-based auto-tuners (POET, Orio)
  - Application-level auto-tuners (Active Harmony)

● Common Theme
  - Search among a set of alternative mappings

● Ultimate goal
  - A unified end-to-end framework for auto-tuning
Our Contribution

● *Active Harmony combined with CHiLL*
  - A general-purpose auto-tuning framework for compiler generated code that:
    • allows the development of self-tuning apps
    • explores a much richer code transformation space

● * Powerful parallel search backend that:*
  - can simultaneously evaluate different compiler optimizations
  - converges in only few tens of steps
Why is this hard?

• Tuning for complex architectures
  - Deep memory hierarchy
  - Performance for even MM is well below hand-tuned libraries

• Parameter interactions
  - Optimization strategies interact
  - Need to find a balance between multiple strategies
Empirical Optimization

- Systematic search over a collection of automatically generated code-variants
  - Can compensate for the lack of precise analytical models

- Enabling Concepts
  - Each transformation parameter $\rightarrow$ variable in an independent dimension
  - A set of transformation parameters $\rightarrow$ “point” in search space
  - Performance of the code-variant $\rightarrow$ Obj. function value
The Search Algorithm

- **Goals**
  - Minimize obj. function value
  - Explore only a small fraction of the search space
  - Avoid "bad" regions

- **Parallel Rank Ordering**
  - Online tuning of SPMD programs
  - $kN$ simplex: Parallel evaluation of candidate points
  - Can try multiple configurations simultaneously
Key Improvements

● Parallelization of expansion check step
  - Offline vs. online
  - Reduces the number of search-steps

● Projection operation
  - Constrained search space
  - Project illegal points to nearest legal point
  - Nearest neighbor defined w/ $L_1$ distance
  - ANN library for nearest neighbor calculation
Key Improvements - II

- **Initial simplex construction**
  - Can generate arbitrary # of independent search directions
  - Can be model-guided

- **Collaborative approach**
  - Bring compiler experts and programmers on-board
  - Guidance via models & architectural parameters
Constraint Specification - Example

parameter space tiling {
    code_region loopI;
    code_region loopJ;
    code_region loopK;
}

region_set loop
    [loopI, loopJ, loopK];

# declare tile_size parameter
parameter tile_size int {
    range [2:2:256];
    default 32;
    region loop;
}

constraint c1 {
    loopK.tile * loopJ.tile
    <= (l1_cache*1024)/16;
}

# rectangular tiles better.
    constraint c2 {
        loopI.tile_size >
        loopJ.tile_size;
    }

    constraint c3 {
        loopJ.tile_size >
        loopI.tile_size;
    }

    specification {
        (c1 AND c2) OR
        (c1 AND c3);
    }
**CHiLL** *(Composable High-Level Loop Transformation Framework)*

- **Polyhedral representation of loop-nests**
  - Each statement has its own iteration space, derived from its enclosing loops respectively
  - A loop-nest is represented by a collection of iteration spaces of the statements within the loop
  - Built Upon Omega Library Plus, an improved Omega Library from UMD

- **CHiLL features**
  - Provides a rich set of loop transformations
  - High-Level script interface
**CHiLL Script Interface - Example**

---

**<code-snippet>**

DO I=1,14,3  
   X(I)=0

---

**<CHiLL-script-1>**

**original()**

#unroll(statement #, loop-level, factor)

unroll(0,1,2)

---

**<transformed code>**

DO T2=1,7,6  
   X(T2)=0  
   X(T2+3)=0  
   X(13)=0

---

**<CHiLL-script-2>**

**original()**

unroll(0,1,5)

---

**<transformed code>**

X(1)=0  
X(1+3)=0  
X(1+6)=0  
X(1+9)=0  
X(13)=0
The Design
Experiments

● Three Kernels - MM, TRSM & Jacobi
  - For the two linear algebra kernels (MM & TRSM), comparisons to well-tuned ATLAS code available
  - All three exhibit the complex parameter trade-offs

● Full Application - SMG2000
  - Part of ASC purple benchmarks
  - Representative of a wide variety of SciDAC apps

● Platform
  - 64-node Linux cluster with dual Intel® Xeon
  - 128 KB L1 cache, 4096 KB L2 cache
  - Native compiler: ifort 10.0.026 w/ -O3 -xN
**MM Optimization Strategy**

**<naïve MM>**
DO K = 1, N
  DO J = 1, N
    DO I = 1, N

**<CHiLL Script>**

tile(0,2,TJ)
tile(0,2, TI)
tile(0,5, TK)
datacopy(0,3,2,1)
datacopy(0,4,3)
unroll(0,4, UI)
unroll(0,5, UJ)

**<Transformed MM>**
! Exploit the reuse of B(K,J) & ! and A(I,K) in L1 and L2 caches ! respectively
DO KK = 1, N, TK
  DO II = 1, N, TI
    DO K = KK, MIN(KK+TK-1,N)
      DO I = II, MIN(II+TI-1,N)
        DO J = JJ, MIN(JJ+TJ-1,N)
          DO K = KK, MIN(KK+TK-1,N)
            P1(K-KK+1, I-II+1) = A(I,K)
          
          DO J = JJ, MIN(JJ+TJ-1,N)
            DO K = KK, MIN(KK+TK-1,N)
              P2(K-KK+1,J-JJ+1) = B(K,J)

! Exploit C(I,J) reuse in registers
  DO I = II, MIN(II+TI-1,N), UI
    DO J = JJ, MIN(JJ+TJ-1,N), UJ
      DO K = KK, MIN(KK+TK-1,N)
        multiply P1's and P2's to registers

**<Constraints>**
TK x TI ≤ size(L2 cache)
TK x TJ ≤ size(L1 cache)
UI x UJ ≤ register set size

5-dimensional search space - TI, TJ, TK, UI, UJ
Experimental Questions

• Does parallel search help?

• How do the perf of our code-variants compare to well-tuned libraries?
  - Comparison with ATLAS (3.8)
    • Search-only
      - use of architectural defaults disabled
      - use of hand-coded BLAS turned off
    • Full-version
Does Parallelism Help?

Effects of Simplex Size on the Convergence of the Search Algorithm

- More parallelism leads to fewer search steps

<table>
<thead>
<tr>
<th></th>
<th>2N</th>
<th>4N</th>
<th>8N</th>
<th>12N</th>
</tr>
</thead>
<tbody>
<tr>
<td># of function Evaluations</td>
<td>276</td>
<td>571</td>
<td>750</td>
<td>961</td>
</tr>
<tr>
<td># of search steps</td>
<td>49</td>
<td>32</td>
<td>22</td>
<td>18</td>
</tr>
<tr>
<td>Speedup over native</td>
<td>2.30</td>
<td>2.33</td>
<td>2.32</td>
<td>2.33</td>
</tr>
</tbody>
</table>

[TI, TJ, TK, UI, UJ]
Result Quality

- **Experiment**
  - Selected & evaluated 100K evenly distributed configurations

- **Performance**
  - Our code-variant: 3.17 GFLOPS
  - The best among random samples: 3.22 GFLOPS
Comparison to ATLAS and ifort

- Performance of our code variant is:
  - within 20% of full-ATLAS
  - 2.36 times faster than ifort
  - 1.66 times faster than ATLAS search-only
Triangular Solver (TRSM)

- Strategy includes:
  - Loop Interchange
  - Unrolling
  - Tiling
  - Loop Split
  - Data Copy
- 7-dimensional search space
- 55 search-steps
- Performance
  - 3.62 times faster than ifort
  - 1.07 times faster than ATLAS search only

<TRSM Kernel>

\[
\begin{align*}
\text{DO } & J = 1, N \\
\text{DO } & K = 1, N \\
\text{DO } & I = K + 1, N \\
B(I,J) &= B(I,J) - B(K,J) * A(I,K)
\end{align*}
\]
Jacobi Kernel

- Strategy involves:
  - Tiling
  - Unrolling
- 4-dimensional search space
- 23 search steps
- Performance:
  - 1.35 times faster than ifort

\[
<\text{Jacobi Kernel}>
\]
\[
\text{DO } K = 2, \text{ N-1} \\
\text{DO } J = 2, \text{ N-1} \\
\text{DO } I = 2, \text{ N-1} \\
\]
Real Application Case Study: SMG2000

- Semi-coarsening multigrid on structured grids
  - Residual computation contains sparse matrix-vector multiply bottleneck, expressed in 4-deep loop nest
  - Key computation identified by HPCToolkit and outlined by ROSE Compiler

```c
for si = 0 to NS-1
  for k = 0 to NZ-1
    for j = 0 to NY-1
      for i = 0 to NX-1
        r[i + j*JR + k*KR] -=
        A[i + j*JA + k*KA + SA[si]] * x[i + j*JX + k*KX + Sx[si]]
```
Smg2000 Results

- **Strategy involves:**
  - Choice of compiler (gcc or icc)
  - Tiling
  - Unrolling

- **5-dimensional search space**

- **20 search steps**

- **Perf gain is:**
  - 2.37 times on residual computation
  - 27.23% on full application
Future Work

• More Large-scale applications
  - Aggressive pruning strategies to reduce the number of implementations
  - Experimenting on new architectures

• Run-time tuning during production execution - merger of just-in-time and traditional compilation
Thank You!

Project Website: www.dyninst.org/harmony
Backup Slides
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 ≤ TI , TJ, TK ≤ 122
0 ≤ UI ≤ 16
0 ≤ US ≤ 10
compilers ∈ {gcc, icc}

Search space:
122^3x16x10x2 = 581M points

University of Maryland
Run-time Tuning For Symbolic Params

```
[Notspot]
<start timer>
  do JJ=2,N-1,T0
  do II=2,N-1,T1
  do K=2,N-1
    do J=JJ,min(JJ+TJ-1,N-1)
      do I=II,min(II+TI-1,N-1)
        R(I,J,K) = stencil-operation
      <end timer>
<end timer>
```
How Runtime Tuning Works?

![Diagram of runtime tuning process]

- **PM₁, PM₂, ..., PMₙ**: Performance Measurements collected from the execution environment.
- **Active Harmony's Search Module**: Evaluates different versions of the code.
- **Code Server**: Stores multiple versions of the code.
- **Code Generator**: Generates code versions for compilation.
- **READY Signal**: Indicates when the code is ready for execution.
- **A Running SPMD-Based Parallel Application**: Executes the compiled versions of the code.
- **Execution**: The process continues until the best version is selected.
Exploiting PRO's Spatial Locality

Parameter Configurations

Parameter 2

Parameter 1

Black: Reflection Points
Grey: Expansion/Shrink Points
White: Original Simplex Points

(Expansion Not Shown)
CHiLL Features

- **Iteration Space Alignment**

- **Auxiliary Loops**

  <code-snippet>
  ```
  DO I=2,N
  S1:   SUM(I)=0
        DO J=1,I-1
  S2:   SUM(I)=SUM(I)+A(J,I)*B(J)
  S3:   B(I)=B(I)-SUM(I)
  ```
  </code-snippet>

  <Aligned Iteration Spaces>
  IS1: \{[i,j] | 2 \leq i \leq N \land j=1\}
  IS2: \{[i,j] | 1 \leq j < i \leq j=1\}
  IS3: \{[i,j] | 2 \leq i \leq N \land j=i-1\}

  <Omega Transformation Relations with auxiliary loops>
  TS1: \{[*,i,*,j,*] \rightarrow [0,i,0,j,0]\}
  TS2: \{[*,i,*,j,*] \rightarrow [0,i,1,j,0]\}
  TS3: \{[*,i,*,j,*] \rightarrow [0,i,2,j,0]\}
  ```
Empirical Optimization - II

- Enabling Components
  - Active Harmony (Search component)
    - Permits programmers to expose tunable parameters and automates the search among a set of alternatives
    - Has shown its capability in offline application level parameter tuning
  - CHiLL (automated code-generation component)
    - Provides a high-level script interface that allows compilers and/or programmers to describe code transformations
Real Application Case Study: SMG2000

- Semi-coarsening multigrid on structured grids
  - Residual computation contains sparse matrix-vector multiply bottleneck, expressed in 4-deep loop nest
  - Key computation identified by HPCToolkit

```plaintext
for si = 0 to NS-1
  for k = 0 to NZ-1
    for j = 0 to NY-1
      for i = 0 to NX-1
        r[i + j*JR + k*KR] =
        A[i + j*JA + k*KA + SA[si]] * x[i + j*JX + k*KX + Sx[si]]
```

2D 6-point Stencil

- \((-1, 1)\) to \((0, 1)\)
- \((-1, 0)\) to \((0, 0)\)
- \((-1, -1)\) to \((0, -1)\)

\(S = \{(-1,1),(-1,0),(-1,-1),(0,1),(0,0),(0,-1)\}\)