Autotuning

John Cavazos
University of Delaware
What is Autotuning?

- Searching for the “best” code parameters, code transformations, system configuration settings, etc.

- Search can be
  - Quasi-intelligent: genetic algorithms, hill-climbing
  - Random (often quite good!)
Parameters to tune in all of these

- application
- compiler
- runtime system
- operating system
- virtualization
- hardware
Traditional Compilers

- “One size fits all” approach
- Tuned for average performance
- Aggressive opts often turned off
- Target hard to model analytically
Solution: Random Search!

- Identify large set of optimizations to search over
  - Some optimizations require parameter values, search over those values also!
  - Out-performs state-of-the-art compiler
Optimization Sequence Representation

- Use random number generator to construct sequence

Example:

-LNO:interchange=1:prefetch=2:blocking_size=32:fusion=1:...

Generate each of these parameter values using a random number generator

Note: need to define a range of interesting values a-priori
Case Study: Random vs State-of-the-Art

- PathScale compiler
  - Compare to highest optimization level
  - 121 compiler flags
- AMD Athlon processor
  - *Real* machine; Not simulation
- 57 benchmarks
  - SPEC (INT 95, INT/FP 2000), MiBench, Polyhedral
Evaluated Search Strategies

- **RAND**
  - Randomly select 500 optimization sequences

- **Combined Elimination [CGO 2006]**
  - Pure search technique
    - Evaluate optimizations one at a time
    - Eliminate negative optimizations in one go
  - Out-performed other pure search techniques

- **PC Model [CGO 2007]**
  - Machine learning model using performance counters
  - Mapping of performance counters to good optimizations
Performance vs Evaluations

Performance versus Number of Evaluations (PC Model, CE, RAND)

- PC Model (17%)
- Random (17%)
- Combined Elimination (12%)
Some recommendations

- Use small input sizes that are representative
  - Be careful as tuning on small inputs may not give you the best performance on regular (larger) inputs
- Reduce application to most important kernel(s) and tune those
- If kernels can be mapped to highly-tuned library implementations, use those!
Some recommendations

- No optimization search will help a bad algorithm!
- Chose the correct algorithm first!
Using quasi-intelligent search

- Can easily setup a genetic algorithm or hill-climbing to perform search over optimization space
- We can help you set this up.
- Random often does as well!
Performance counters

- Can be used to narrow search to particular set of optimizations
- Lots of cache misses may require loop restructuring, e.g., blocking
- Lots of resource stalls may require instruction scheduling
### Most Informative Perf Counters

<table>
<thead>
<tr>
<th>1. L1 Cache Accesses</th>
<th>9. L2 Dcache Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. L1 Dcache Hits</td>
<td>10. L2 Cache Accesses</td>
</tr>
<tr>
<td>3. TLB Data Misses</td>
<td>11. L1 Dcache Accesses</td>
</tr>
<tr>
<td>4. Branch Instructions</td>
<td>12. Hardware Interrupts</td>
</tr>
<tr>
<td>5. Resource Stalls</td>
<td>13. L2 Cache Hits</td>
</tr>
<tr>
<td>6. Total Cycles</td>
<td>14. L1 Cache Hits</td>
</tr>
<tr>
<td>7. L2 Icache Hits</td>
<td>15. Branch Misses</td>
</tr>
<tr>
<td>8. Vector Instructions</td>
<td></td>
</tr>
</tbody>
</table>
Dependence Analysis and Loop Transformations

John Cavazos
University of Delaware
Lecture Overview

- Very Brief Introduction to Dependences
- Loop Transformations
The Big Picture

What are our goals?

- **Simple Goal:** Make execution time as small as possible

Which leads to:

- Achieve execution of many (all, in the best case) instructions in parallel
- Find *independent* instructions
Dependences

- We will concentrate on data dependences
- Simple example of data dependence:

  \[ S_1 \quad \text{PI} = 3.14 \]
  \[ S_2 \quad \text{R} = 5.0 \]
  \[ S_3 \quad \text{AREA} = \text{PI} \times \text{R} ** 2 \]

  Statement \( S_3 \) cannot be moved before either \( S_1 \) or \( S_2 \) without compromising correct results
Dependences

Formally:

There is a data dependence from statement $S_1$ to statement $S_2$ ($S_2$ depends on $S_1$) if:

1. Both statements access the same memory location and at least one of them stores onto it, and
2. There is a feasible run-time execution path from $S_1$ to $S_2$
Load Store Classification

- Quick review of dependences classified in terms of load-store order:
  1. True dependence (RAW hazard)
  2. Antidependence (WAR hazard)
  3. Output dependence (WAW hazard)
Dependence in Loops

Let us look at two different loops:

```
DO I = 1, N
    S_1  A(I+1) = A(I) + B(I)
ENDDO
```

```
DO I = 1, N
    S_1  A(I+2) = A(I) + B(I)
ENDDO
```

• In both cases, statement $S_1$ depends on itself
Transformations

- We call a transformation safe if the transformed program has the same "meaning" as the original program.

- But, what is the "meaning" of a program?

For our purposes:

- Two computations are equivalent if, on the same inputs:
  - They produce the same outputs in the same order.
Reordering Transformations

- Is any program transformation that changes the order of execution of the code, without adding or deleting any executions of any statements.
Properties of Reordering Transformations

- A reordering transformation does not eliminate dependences
- However, it can change the ordering of the dependence which will lead to incorrect behavior
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.
Loop Transformations

- Compilers have always focused on loops
  - Higher execution counts
  - Repeated, related operations
- Much of real work takes place in loops
Several effects to attack

- Overhead
  - Decrease control-structure cost per iteration

- Locality
  - Spatial locality ⇒ use of co-resident data
  - Temporal locality ⇒ reuse of same data

- Parallelism
  - Execute independent iterations of loop in parallel
Eliminating Overhead

Loop unrolling (the oldest trick in the book)

- To reduce overhead, replicate the loop body

\[
\text{do } i = 1 \text{ to } 100 \text{ by } 1 \\
a(i) = a(i) + b(i) \\
\text{end}
\]

\[
\text{becomes}
\]

\[
\text{do } i = 1 \text{ to } 100 \text{ by } 4 \\
a(i) = a(i) + b(i) \\
a(i+1) = a(i+1) + b(i+1) \\
a(i+2) = a(i+2) + b(i+2) \\
a(i+3) = a(i+3) + b(i+3) \\
\text{end}
\]

(unroll by 4)

Sources of Improvement

- Less overhead per useful operation
- Longer basic blocks for local optimization
Eliminating Overhead

Loop unrolling with unknown bounds

Generate guard loops

\[
\text{do } i = 1 \text{ to } n \text{ by } 1 \\
\quad a(i) = a(i) + b(i) \\
\text{end}
\]

becomes

\[
\text{(unroll by 4)}
\]

\[
i = 1 \\
\text{do while } (i+3 < n) \\
\quad a(i) = a(i) + b(i) \\
\quad a(i+1) = a(i+1) + b(i+1) \\
\quad a(i+2) = a(i+2) + b(i+2) \\
\quad a(i+3) = a(i+3) + b(i+3) \\
\quad i = i + 4 \\
\text{end}
\]

\[
\text{do while } (i < n) \\
\quad a(i) = a(i) + b(i) \\
\quad i = i + 1 \\
\text{end}
\]
Eliminating Overhead

One other use for loop unrolling

- Eliminate copies at the end of a loop

\[
t_1 = b(0) \\
do \ i = 1 \ \text{to} \ 100 \\
\quad t_2 = b(i) \\
\quad a(i) = a(i) + t_1 + t_2 \\
\quad t_1 = t_2 \\
\text{end}
\]

becomes

\[
t_1 = b(0) \\
do \ i = 1 \ \text{to} \ 100 \ \text{by} \ 2 \\
\quad t_2 = b(i) \\
\quad a(i) = a(i) + t_1 + t_2 \\
\quad t_1 = b(i+1) \\
\quad a(i+1) = a(i+1) + t_2 + t_1 \\
\text{end}
\]
Loop Unswitching

- Hoist invariant control-flow out of loop nest
- Replicate the loop & specialize it
- No tests, branches in loop body
- Longer segments of straight-line code
Loop Unswitching

If test then
  loop
  statements
  then part
else
  else part
endif
more statements
endloop

becomes
(unswitch)
Loop Unswitching

\[
\begin{align*}
\text{do } i & = 1 \text{ to } 100 \\
& \quad \text{a}(i) = \text{a}(i) + \text{b}(i) \\
& \quad \text{if (expression) then} \\
& \quad \quad \text{d}(i) = 0 \\
\text{end}
\end{align*}
\]

(becomes)

\[
\begin{align*}
\text{if (expression) then} \\
\text{do } i & = 1 \text{ to } 100 \\
& \quad \text{a}(i) = \text{a}(i) + \text{b}(i) \\
& \quad \text{d}(i) = 0 \\
\text{end} \\
\text{else} \\
\text{do } i & = 1 \text{ to } 100 \\
& \quad \text{a}(i) = \text{a}(i) + \text{b}(i) \\
\text{end}
\end{align*}
\]
Loop Fusion

- Two loops over same iteration space ⇒ one loop
- Safe if does not change the values used or defined by any statement in either loop (i.e., does not violate deps)

\[
\begin{align*}
    \text{do } i &= 1 \text{ to } n \\
    c(i) &= a(i) + b(i) \\
    \text{end} \\
    \text{do } j &= 1 \text{ to } n \\
    d(j) &= a(j) \times e(j) \\
    \text{end}
\end{align*}
\]

becomes

(fuse)

\[
\begin{align*}
    \text{do } i &= 1 \text{ to } n \\
    c(i) &= a(i) + b(i) \\
    d(i) &= a(i) \times e(i) \\
    \text{end}
\end{align*}
\]

For big arrays, \(a(i)\) may not be in the cache

\(a(i)\) will be found in the cache
Loop Fusion Advantages

- Enhance temporal locality
- Reduce control overhead
- Longer blocks for local optimization & scheduling
- Can convert inter-loop reuse to intra-loop reuse
Loop Fusion of Parallel Loops

- Parallel loop fusion legal if dependences loop independent
  - Source and target of flow dependence map to same loop iteration
Loop distribution (fission)

- Single loop with independent statements $\Rightarrow$ multiple loops
- Starts by constructing statement level dependence graph
- Safe to perform distribution if:
  - No cycles in the dependence graph
  - Statements forming cycle in dependence graph put in same loop
Loop distribution (fission)

\[
\begin{align*}
d & = 1 \text{ to } n \\
a(i) & = b(i) + c(i) \\
d(i) & = e(i) \times f(i) \\
g(i) & = h(i) - k(i)
\end{align*}
\]

Reads \( b, c, e, f, h, \) & \( k \)

\{ Writes \( a, d, \) & \( g \) \}

\[
\begin{align*}
d & = 1 \text{ to } n \\
a(i) & = \text{reads } b(i) + c(i) \\
d(i) & = \text{reads } e(i) \times f(i) \\
g(i) & = \text{reads } h(i) - k(i)
\end{align*}
\]

\{ Reads \( b, c \)

\{ Writes \( a \) \}

\{ Reads \( e, f \)

\{ Writes \( d \) \}

\{ Reads \( h, k \)

\{ Writes \( g \) \}
Loop distribution (fission)

(1) for \( i = 1 \) to \( N \) do


(3) \( B[i] = C[i-1]*X+C \)

(4) \( C[i] = 1/B[i] \)

(5) \( D[i] = \sqrt{C[i]} \)

(6) endfor
Loop distribution (fission)

(1) for \( i = 1 \) to \( N \) do


(3) \( B[i] = C[i-1] \times X + C \)

(4) \( C[i] = 1/B[i] \)

(5) \( D[i] = \sqrt{C[i]} \)

(6) endfor

(1) for \( i = 1 \) to \( N \) do


(3) endfor

(4) for

(5) \( B[i] = C[i-1] \times X + C \)

(6) \( C[i] = 1/B[i] \)

(7) endwhile

(8) for

(9) \( D[i] = \sqrt{C[i]} \)

(10) endwhile
Loop Fission Advantages

- Enables other transformations
  - E.g., Vectorization
- Resulting loops have smaller cache footprints
  - More reuse hits in the cache
Loop Interchange

\[
do \ i = 1 \ to \ 50 \\
\quad do \ j = 1 \ to \ 100 \\
\quad \quad a(i,j) = b(i,j) \times c(i,j) \\
\quad end
\]
end

\[
do \ j = 1 \ to \ 100 \\
\quad do \ i = 1 \ to \ 50 \\
\quad \quad a(i,j) = b(i,j) \times c(i,j) \\
\quad end
\]
end

- Swap inner & outer loops to rearrange iteration space

Effect
- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible
Loop Interchange Effect

- If one loop carries all dependence relations
  - Swap to outermost loop and all inner loops executed in parallel
- If outer loops iterates many times and inner only a few
  - Swap outer and inner loops to reduce startup overhead
- Improves reuse by using more elements per cache line
- Goal is to get as much reuse into inner loop as possible
Reordering Loops for Locality

In row-major order, the opposite loop ordering causes the same effects.

In Fortran’s column-major order, \(a(4,4)\) would lay out as:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
<td>4,3</td>
</tr>
<tr>
<td>1,4</td>
<td>2,4</td>
<td>3,4</td>
<td>4,4</td>
</tr>
</tbody>
</table>

As little as 1 used element per line.

After interchange, the direction of iteration is changed:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>2,1</td>
<td>3,1</td>
<td>4,1</td>
</tr>
<tr>
<td>1,2</td>
<td>2,2</td>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
<td>4,3</td>
</tr>
<tr>
<td>1,4</td>
<td>2,4</td>
<td>3,4</td>
<td>4,4</td>
</tr>
</tbody>
</table>

Runs down cache line.
Loop permutation

- Interchange is degenerate case
  - Two perfectly nested loops
- More general problem is called permutation

Safety

- Permutation is safe iff no data dependences are reversed
  - The flow of data from definitions to uses is preserved
Loop Permutation Effects

- Change order of access & order of computation
- Move accesses closer in time ⇒ increase temporal locality
- Move computations farther apart ⇒ cover pipeline latencies
Strip Mining

- Splits a loop into two loops

```plaintext
do j = 1 to 100
  do i = 1 to 50
    a(i,j) = b(i,j) * c(i,j)
  end
end

becomes

do j = 1 to 100
  do ii = 1 to 50 by 8
    do i = ii to min(ii+7,50)
      a(i,j) = b(i,j) * c(i,j)
    end
  end
end
```

Note: This is always safe, but used by itself not profitable!
Strip Mining Effects

- May slow down the code (extra loop)
- Enables vectorization
Loop Tiling (blocking)

\[
\begin{align*}
    & \text{do } t = 1, T \\
    & \quad \text{do } i = 1, n \\
    & \quad \quad \text{do } j = 1, n \\
    & \quad \quad \quad \ldots \ a(i,j) \ldots \\
    & \quad \text{end do} \\
    & \text{end do} \\
    & \text{end do}
\end{align*}
\]

Want to exploit temporal locality in loop nest.
Loop Tiling (blocking)

```plaintext
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1,T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n,jc+B-1), 1
          ... a(i,j) ...
        end do
      end do
      end do
    end do
  end do
end do
```

B: Block Size
Loop Tiling (blocking)

```
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n,jc+B-1), 1
          ... a(i,j) ...
        end do
      end do
    end do
  end do
end do
```

B: Block Size
Loop Tiling (blocking)

```plaintext
do ic = 1, n, B
  do jc = 1, n, B
    do t = 1, T
      do i = ic, min(n,ic+B-1), 1
        do j = jc, min(n, jc+B-1), 1
          ... a(i,j) ...
        end do
      end do
    end do
  end do
end do
```

B: Block Size
Loop Tiling (blocking)

\[
\begin{align*}
\text{do } ic &= 1, n, B \\
\text{do } jc &= 1, n, B \\
\text{do } t &= 1, T \\
\text{do } i &= ic, \min(n,ic+B-1), 1 \\
\text{do } j &= jc, \min(n, jc+B-1), 1 \\
\ldots & a(i,j) \\
\text{end do} \\
\text{end do} \\
\text{end do} \\
\text{end do} \\
\text{end do}
\end{align*}
\]

B: Block Size
When is this legal?
Loop Tiling Effects

- Reduces volume of data between reuses
  - Works on one “tile” at a time \((tile \; size \; is \; B \; by \; B)\)
- Choice of tile size is crucial
Scalar Replacement

- Allocators never keep c(i) in a register
- We can trick the allocator by rewriting the references

The plan
- Locate patterns of consistent reuse
- Make loads and stores use temporary scalar variable
- Replace references with temporary’s name
Scalar Replacement

\[
\begin{align*}
\text{do } i &= 1 \text{ to } n \\
\text{ do } j &= 1 \text{ to } n \\
\quad & \quad a(i) = a(i) + b(j) \\
\text{ end } \\
\text{ end }
\end{align*}
\]

becomes

\[
\begin{align*}
\text{do } i &= 1 \text{ to } n \\
\quad & \quad t = a(i) \\
\text{ do } j &= 1 \text{ to } n \\
\quad & \quad t = t + b(j) \\
\text{ end } \\
\quad & \quad a(i) = t \\
\text{ end }
\end{align*}
\]

Almost any register allocator can get \( t \) into a register
Scalar Replacement Effects

- Decreases number of loads and stores
- Keeps reused values in names that can be allocated to registers
- In essence, this exposes the reuse of a(i) to subsequent passes