Compiler Techniques for Improving Global and Dynamic Cache Reuse

Memory Performance

- Problem
  - High memory latency
- Improvement 1
  - Fast cache
- Improvement 2
  - Data prefetching

Is there enough bandwidth?

Bandwidth Bottleneck

- Hardware trends
  - CPU speed improved 6400 times in 20 years
  - Memory bandwidth improved 139 times
- Software trends
  - Large data sets
  - Dynamic content and computation
  - Modularized programming

Performance Model

- Balance
- Machine balance
  - Max words per cycle divided by max flops per cycle
  - # load/store units divided by # floating-point units
- Program balance
  - # words accessed divided by # flops executed
  - Total loads/stores divided by total floating-point ops
- Consequences
  - MB = PB → full utilization
  - MB > PB → memory idle
  - MB < PB → CPU idle

Program and Machine Balance

<table>
<thead>
<tr>
<th>Program/machine balance</th>
<th>Program/machine balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-Reg</td>
<td>L2-L1</td>
</tr>
<tr>
<td>Convolution</td>
<td>6.4</td>
</tr>
<tr>
<td>Dmxy</td>
<td>8.3</td>
</tr>
<tr>
<td>Mmijki (-O2)</td>
<td>24.0</td>
</tr>
<tr>
<td>Mmijki (-O3)</td>
<td>8.1</td>
</tr>
<tr>
<td>FFT</td>
<td>8.3</td>
</tr>
<tr>
<td>SP</td>
<td>10.8</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>15.0</td>
</tr>
<tr>
<td>Origin2000</td>
<td>4.0</td>
</tr>
</tbody>
</table>

- Ratios of demand to supply

<table>
<thead>
<tr>
<th>Applications</th>
<th>Reg BW</th>
<th>Cache BW</th>
<th>Mem BW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>1.6</td>
<td>1.3</td>
<td>4.5</td>
</tr>
<tr>
<td>Dmxy</td>
<td>2.1</td>
<td>2.1</td>
<td>10.5</td>
</tr>
<tr>
<td>Mmijki (-O2)</td>
<td>6.0</td>
<td>2.1</td>
<td>7.4</td>
</tr>
<tr>
<td>FFT</td>
<td>2.1</td>
<td>0.8</td>
<td>3.4</td>
</tr>
<tr>
<td>SP</td>
<td>2.7</td>
<td>1.6</td>
<td>6.1</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>3.8</td>
<td>2.3</td>
<td>9.8</td>
</tr>
</tbody>
</table>

- Memory bandwidth is least sufficient
- Maximal CPU utilization: 10% to 33%
- The imbalance is getting worse
- Software solution: Better caching
Vector Architectures: Past, Present and Future
Roger Fujii, Matteo Fraternali, James L. Smith

Figure 3: Evolution of clock frequency, over the years for vector supercomputers and for several microprocessors (source: vendor information).

Figure 4: Evolution of peak main memory bandwidth for vector supercomputers and for Alpha-based workstations. The values for the Alpha main memory bandwidth are derived using the STREAM 2002 and 2005 chips [5].

Steve Wallach’s Fall 1999 Seminar

Instead of

How about

CPU
CROSSBAR
MEMORY

Earth Simulator 2002

Processing Node
640 nodes, 330 cabinets

IBM BlueGene/L 2005

Single-Chip Bottleneck

- Intel Core 2 Quad
  - four 2.40GHz cores
  - 8 64-bit flops per cycle per core
  - sustained memory bandwidth measured by STREAM benchmark
    - 5.3GB/s for four threads
    - 0.55 byte per cycle per core, 0.009 dword per flop
  - 8MB L2 cache

- Locality
  - How (in)frequent a program accesses main memory?
  - How much data does it actively use?
  - Must model long-range program behavior

- Program and machine balance
  - [Callahan, Cocke, Kennedy, JPDC 88] [Ding, Kennedy, JPDC 04]
Problem of Caching

- An example
  - 7 accesses
  - single-element cache
- Cache management
  - LRU: 1 reuse
  - Belady: 2 reuses
- Our approach
  - fuse computation on the same data: 4 reuses
  - group data used by the same computation

Computation Fusion

Function Initialize:
read input[1:N].data1
read input[1:N].data2
End Initialize

Function Process:
//Fused_step_1
for each i in [1:N]
A_tmp[i].data1 ← input[i].data1
end for
B_tmp[1:N].data1 ← input[1:N].data2
//Fused_step_2
for each i in [1:N]
B_tmp[i].data1 ← input[i].data2
end for
...

Data Grouping

//Fused_step_1
for each i in [1:N]
read input[i].data1
A_tmp[i].data1 ← input[i].data1
end for

//Fused_step_2
for each i in [1:N]
read input[i].data2
B_tmp[i].data2 ← input[i].data2
end for
...

But How?

- Programmers
  - loss of modularity
  - data layout depends on function
- Hardware/operating system
  - limited scope
  - run-time overhead
- Compilers
  - global scope
  - off-line analysis/transformation
  - imprecise information

Overall Fusion Process

for each statement in program
find its data sharing predecessor
try clustering them (fusion)
if succeed
  apply fusion recursively
end if
end for

Original

Function Initialize:
read input[1:N].data1
read input[1:N].data2
End Initialize

Function Process:
A_tmp[1:N].data1 ← input[1:N].data1
B_tmp[1:N].data1 ← input[1:N].data2
...
End Process

Transformed

for each i in [1:N]
read group1[i].data1
group1[i].data2
...
End Process

- Computation fusion recombines all functions
- Data grouping reshuffles all data

Memory

Original

y=x+2
z=x+y

Transformed

y=x+1
z=x+y
Difficulties
- incompatible shapes
- data dependence

Three cases of fusion
- between iteration & loop
  - embedding
  - interleaving + alignment
  - otherwise
  - iteration reordering,
  - e.g., loop splitting

Example Fusion
for i=2, N
  a[i] = f(a[i-1])
end for

for i=3, N
  b[i] = g(a[i-2])
end for

Example Fusion
for i=2, N
  a[i] = f(a[i-1])
if (i==3)
  a[2]=0.0
else if (i==N)
  a[1]= a[N]
end if
end for

Example Fusion
for i=2, N
  a[i]=f(a[i-1])
if (i==3)
  a[2]=0.0
else if (i==N)
  a[1]= a[N]
end if
end for

Example Fusion
for i=3, N
  b[i] = g(a[i-2])
end for

More on Fusion
- Multi-level fusion
  - gives priority to fusion at outer levels

- Optimal fusion
  - minimal data sharing
  - an NP-hard problem
  - hyper-graph formulation of data sharing
  - a polynomial-time case

Is Fusion Enough?
//Fused_step_1
for each i in [1:N]
  read input[i].data1
  A_tmp[i].data1
  // input[i].data1
end for

//Fused_step_2
for each i in [1:N]
  read input[i].data2
  B_tmp[i].data1
  // input[i].data2
end for
...
Data Regrouping

Program Analysis

- Computation phases
  - amount of data accessed
- Compatible arrays
  - array size
  - order of access
- Arrays are split into the smallest unit possible
  - A(3,N) -> A1(N), A2(N), A3(N)
- Regrouping is applied to individual set of compatible arrays

Computation phases | Arrays accessed
--- | ---
Constructing neighbor list | position
Smoothing attributes | position, velocity, heat, derivative, viscosity
density, momentum, volume, energy, cumulative totals
Hydrodynamics 1 | momentum, volume, energy, cumulative totals
Hydrodynamics 2 | Stress 1: volume, energy, strength, cumulative totals
density, strength
Stress 2 | Stress 2: volume, energy, strength, cumulative totals
density, strength

Regrouping Algorithm

- Requirements
  - 1. regroup as many arrays as possible
  - 2. do not introduce useless data
- Solution
  - group two arrays if and only if they are always accessed together
- Properties
  - time complexity is $O(N_{arrays} \times N_{phases})$
  - compile-time optimal
  - minimal page-table working set
  - eliminate all useless data in cache

More on Regrouping

- Multi-level regrouping
  - grouping array segments as well as array elements
  - consistent regrouping
- Extensions
  - allowing useless data
  - allowing dynamic remapping
  - NP-complete proofs
  - architecture-dependent data layout
**NAS/SP**

- Benchmark application from NASA
  - computational fluid dynamics (CFD)
  - class B input, 102x102x102
  - 218 loops in 67 loop nests, distributed into 482 loops
  - 15 global arrays, split into 42 arrays

- Optimizations
  - fused into 8 loop nests
  - grouped into 17 new arrays, e.g.
    - \( \text{ainv}[n,n,n], \text{us}[n,n,n], \text{qs}[n,n,n], \text{u}[n,n,n,1-5] \)
    - \( \text{lhd}[n,n,n,6-8], \text{lh}[n,n,n,11-13] \)

**Dynamic Optimizations**

- Unknown Access
  - “Every problem can be solved by adding one more level of indirection.”
  - Irregular and dynamic applications
    - Irregular data structures are unknown until run time
    - Data and their uses may change during the computation
  - For example
    - Molecular dynamics
    - Sparse matrix
  - Problems
    - How to optimize at run time?
    - How to automate?

**Example packing**

```
original array

data access
f[8], f[800], f[8], f[2], ...

transformed array
```

Software remapping:
- \( f[i] \rightarrow f[\text{remap}[i]] \rightarrow f'[i] \)
- \( f[i] \rightarrow f[\text{remap}[i]] \rightarrow f[i] \)
Dynamic Optimizations

- Locality grouping & Dynamic packing
  - run-time versions of computation fusion & data grouping
  - linear time and space cost
- Compiler support
  - analyze data indirections
  - find all optimization candidates
  - use run-time maps to guarantee correctness
  - remove unnecessary remappings
    - pointer update
    - array alignment
- The first set of compiler-generated run-time transformations

packing Directive: apply packing using interactions

for each pair (i,j) in interactions
  compute_force( force[i], force[j] )
end for

for each object i
  update_location( location[i], force[i] )
end for

apply_packing(interactions[*], force[*], inter_map[*])

for each pair (i,j) in interactions
  compute_force( force[inter_map[i]], force[inter_map[j]] )
end for

for each object i
  update_location( location[i], force[inter_map[i]] )
end for

apply_packing(interactions[*], force[*], inter_map[*], update_map[*])

update_indirection_array(interactions[*], update_map[*])

transform_data_array(location[*], update_map[*])

for each pair (i,j) in interactions
  compute_force( force[i], force[j] )
end for

for each object i
  update_location( location[i], force[i] )
end for

Indirection Analysis

pointer 4
data 1
pointer 2
data 2
data 5
data 4
DoD/Magi

- A real application from DoD Philips Lab
  - particle hydrodynamics
  - almost 10,000 lines of code
  - user supplied input of 28K particles
  - 22 arrays in major phases, split into 26

- Optimizations
  - grouped into 6 arrays
  - inserted 1114 indirections to guarantee correctness
  - optimization reorganized 19 more arrays
  - removed 379 indirections in loops
  - reorganized 45 arrays 4 times during execution

Overall Comparison

<table>
<thead>
<tr>
<th>programs</th>
<th>L2 misses</th>
<th>TLB misses</th>
<th>Speedup over SGI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoOpt SGI New</td>
<td>NoOpt SGI New</td>
<td></td>
</tr>
<tr>
<td>Swim</td>
<td>1.00  0.94</td>
<td>1.00  1.05</td>
<td>1.14</td>
</tr>
<tr>
<td>Tomcat</td>
<td>1.00  0.93</td>
<td>1.00  0.97</td>
<td>1.02</td>
</tr>
<tr>
<td>ASM</td>
<td>1.00  0.96</td>
<td>1.00  0.99</td>
<td>1.02</td>
</tr>
<tr>
<td>NAS/SP</td>
<td>1.00  0.97</td>
<td>1.00  0.98</td>
<td>1.02</td>
</tr>
<tr>
<td>Average</td>
<td>1.00  0.98</td>
<td>1.00  1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>Moldyn</td>
<td>1.00  0.91</td>
<td>1.00  0.97</td>
<td>1.02</td>
</tr>
<tr>
<td>Mesh</td>
<td>1.00  0.92</td>
<td>1.00  0.93</td>
<td>1.02</td>
</tr>
<tr>
<td>Magi</td>
<td>1.00  0.96</td>
<td>1.00  0.98</td>
<td>1.02</td>
</tr>
<tr>
<td>NAS/CsG</td>
<td>1.00  0.97</td>
<td>1.00  0.99</td>
<td>1.03</td>
</tr>
<tr>
<td>Average</td>
<td>1.00  0.97</td>
<td>1.00  0.99</td>
<td>1.03</td>
</tr>
</tbody>
</table>

Software Techniques Summary

<table>
<thead>
<tr>
<th>main steps</th>
<th>sub-steps</th>
<th>example techniques (<em>studied in my work</em>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>temporal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>reuse</td>
<td></td>
<td>*loop fusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local (single loop)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blocking, register allocation</td>
</tr>
<tr>
<td>spatial</td>
<td></td>
<td>*dynamic partitioning</td>
</tr>
<tr>
<td>reuse</td>
<td></td>
<td>global (inter-array)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*inter-array data regrouping</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local (intra-array)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>loop permutation, array reshaping,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>combined schemes</td>
</tr>
<tr>
<td>dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>*dynamic data padding</td>
</tr>
<tr>
<td>latency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tolerance</td>
<td></td>
<td>local (single loop)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>data prefetching, instruction scheduling</td>
</tr>
<tr>
<td>program</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tuning &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scheduling</td>
<td></td>
<td>*balance model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*bandwidth-based perf. tuning &amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>prediction</td>
</tr>
</tbody>
</table>

Summary

- Global transformations
  - Combining both computation and data reordering at a large scale
- Dynamic transformations
  - Combining compile-time and run-time analysis and transformation
- Compiling for locality
  - splits and regroups global computation and data
  - for the whole program and at all times