Improving Performance by Using Memory Better

Jonathan Turpie

Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings

John Mellor-Crummey, David Whalley, Ken Kennedy
Problem

- The memory hierarchy makes things that seem slower faster
- Need to keep things in cache instead of evicting them
- Data Reordering
- Computation Reordering

First Touch Reordering

```
```

```
5 3 1 9 0 8 2 4 7 6
```
First Touch Reordering

1

5 3 1 9 0 8 2 4 7 6

D₀ D₁ D₂ D₃ D₄ D₅ D₆ D₇ D₈ D₉

First Touch Reordering

1 D₁

5 3 1 9 0 8 2 4 7 6

D₀ D₁ D₂ D₃ D₄ D₅ D₆ D₇ D₈ D₉
First Touch Reordering

- Traversing the entire data set requires two memory accesses.
- The first access is probably cached, but not the second.
- Reordering the data may result in better overall performance.
First Touch Reordering

- (3 memory accesses + 1 cache miss) once + 1 memory access for each use
- 2 memory accesses + 1 cache miss for each use
- After using all the data twice, this simple reordering is more efficient
- Requires that the data access pattern be known before computation

Data Reordering

- What if the data is organized 2 dimensionally?

![Diagram showing physical space and data space with particle information and interaction list.]
Space Filling Curve Reordering

- Use a space filling curve (fractal pattern)
- Data points that are near each other in physical space will probably be close in memory

Space Filling Curve Reordering

- Independent of the data access pattern
  - As long as it uses data points that are physically close together
- Additional computation is necessary to translate discrete physical coordinates into a memory address
- Sometimes data is not close in memory
Computational Reordering

- If possible, change the order that data is accessed to match memory
- Jump through loop iterations instead of jumping through memory
- Only works if the data can be processed out of order

Space Filling Curve Reordering

- Process physical data points in a space filling curve order
- If point \((x_i, y_j)\) was just processed, \((x_{i\pm1}, y_{j\pm1})\) are probably in cache
- If another point near \((x_i, y_j)\) is processed next, it will probably need some of \((x_{i\pm1}, y_{j\pm1})\)
Reordering by Blocking

- Try to work on chunks of data that fit in cache together

```
FOR i = 1 to number of blocks of particles DO
  FOR j = i to number of blocks of particles DO
    process interactions between all interacting particle pairs with the first particle in block i and the second in block j
```

- This only works for one level of cache
- Multiple nested blocking loops are needed to take advantage of each layer of memory cache
MemSpy: Analyzing Memory System Bottlenecks in Programs

Margaret Martonosi, Anoop Gupta, and Thomas Anderson

MemSpy

- MemSpy is a tool for looking at the performance of data access patterns
- MemSpy provides statistics for code blocks and for data blocks
- Provides information about what part of memory is performing badly
- Works on parallel programs
MemSpy

- Collection of statistics is done using hardware features
- Use fairly fine resolution
- Statistics are put into bins for the code that executed during that time and for the data that was used
- Each memory address is mapped to a bin

MemSpy

- Memory bins are associated with variables
- There can be a lot of variables to track
- Memory gets allocated dynamically
  - MemSpy only tracks heap allocated memory
- Many variables behave similarly
- MemSpy groups all memory that was allocated from the same part of the code
MemSpy

- Variables are named by their type, name, and if necessary, the call stack that produced it.
- In a more complex program it can be hard to tell what the different variables are.

MemSpy

- MemSpy tries to figure out why a cache miss occurred.
- MemSpy keeps two bits of data about each memory block:
  - Has the block been referenced before?
  - Has the block been evicted?
  - Has the block been invalidated?
- The data is stored in a small piece of memory which is indexed by the lower k bits of the memory address.
- There is a trade off between taking up more cache blocks with this array and having collisions.

Figure 1: Blocked matrix multiply code.
All the problems are cache misses
Almost all of the misses are being caused by Matrix Y
Probable cause is the blocks are too big to fit into cache or Y is not contiguous
MemSpy

- MemSpy has a lot of overhead
- Lots of context switches
- MemSpy also spends a lot of time finding the bin for each variable

Table 2: MemSpy execution time overhead.

<table>
<thead>
<tr>
<th>Application</th>
<th>Time (s) No Simulation</th>
<th>Time (s) Simulation</th>
<th>Time (s) Simulation and MemSpy</th>
<th>Simulation Overhead alone</th>
<th>MemSpy and Simulation Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID8</td>
<td>4.2</td>
<td>22.0</td>
<td>101.0</td>
<td>16.0</td>
<td>22.4</td>
</tr>
<tr>
<td>MatMult</td>
<td>34.3</td>
<td>544.0</td>
<td>1195.3</td>
<td>12.1</td>
<td>21.7</td>
</tr>
<tr>
<td>Pithor</td>
<td>9.0</td>
<td>312.0</td>
<td>521.4</td>
<td>34.8</td>
<td>57.9</td>
</tr>
</tbody>
</table>