Compile-Time Detection of False Sharing via Loop Cost Modeling

Munara Tolubaeva, Yonghong Yan and Barbara Chapman

High Performance Computing and Tools Group (HPCTools)

Computer Science Department
University of Houston
OUTLINE

- Introduction and Motivation
- Methodology
- Experiment
- Conclusion
Introduction & Motivation

- Compiler Transformation Example

```c
for (i=0; i<N; i++) {
    A[i] = B[i] + C[i];
}
```

Loop Unrolling

```
for (i=0; i<N; i+=3) {
    A[i+0] = B[i+0] + C[i+0];
    A[i+1] = B[i+1] + C[i+1];
    A[i+2] = B[i+2] + C[i+2];
}
```
Introduction & Motivation

- **Compiler Cost Model**

  - Estimates the time needed to execute a specific section of code on a given system
  - Considers performance impacting architectural features (Processor, Cache, Memory bandwidth, etc)
  - Open64 cost models – the most sophisticated models among open source compilers
Introduction & Motivation

Open64 cost models

- Processor model
  - Computational resource cost
    - Operation cost
    - Issue cost
    - Mem_ref cost
  - Dependency latency cost
  - Register spilling cost

- Cache model
  - Cache cost
  - TLB cost

- Parallel model
  - Machine cost
  - Cache cost
  - Loop overhead
  - Parallel overhead
  - Reduction cost

For loops only...
Introduction & Motivation

- Processor model - predicts the scheduling of instructions given the available amount of resources
- Guides loop unrolling
- Finds the optimal loop unrolling level and factor

\[
\text{Machine}_{\text{per iter}} = \text{Resource}_c + \text{Dependency latency}_c + \text{Register spilling}_c
\]
\[
\text{Resource}_c = \max(\text{Op}_c, \text{MEM ref}_c, \text{Issue}_c)
\]
\[
\text{MEM ref}_c = (\text{Num fp refs} + \text{Num int refs})/\text{Num mem units}
\]
\[
\text{Issue}_c = \text{Num inst}/\text{Issue rate}
\]
\[
\text{Dependency latency}_c = \max(\text{Sum of latencies}/\text{Sum of distances})
\]
\[
\text{Register spilling}_c = (\text{Reg used} - \text{Target regs}) \ast \frac{[\text{Num reg refs}/(\text{Scalar reg} + \text{Array reg})]}{}
\]
\[
\text{Reg used} = \text{Base reg} + \text{Scalar reg} + \text{Array reg}
\]
Introduction & Motivation

- Cache model - predicts the number of cache misses and estimates additional cycles needed to execute an iteration of an inner loop
- Guides loop tiling
- Finds the optimal loop tiling size

\[ TLB_{\text{miss}} = \text{Num\_array\_ref} - \text{TLB\_entries}, \text{if}(\text{Num\_array\_ref} - \text{TLB\_entries} > 0) \]
\[ TLB_c = TLB_{\text{miss\_penalty}} \times TLB_{\text{miss}} \]
\[ Cache_c = \sum_{i}^{\text{Levels}} (\text{Clean\_footprint}_i \times \text{Clean\_penalty}_i + \text{Dirty\_footprint}_i \times \text{Dirty\_penalty}_i) \]
Introduction & Motivation

- Parallel model – decides loop level that is the best candidate for parallelization
- Evaluates the cost involved in parallelizing the loop
- Used in auto-parallelization phase

\[
\text{Total}_c = \text{Machine}_c + \text{TLB}_c + \text{Cache}_c + \text{Loop\_overhead}_c + \text{Parallel\_overhead}_c
\]

\[
\text{Machine}_c = \text{Machine\_per\_iter}_c \times \frac{\text{Num\_loop\_iter}}{\text{Num\_threads}}
\]

\[
\text{Loop\_overhead}_c = \text{Loop\_overhead\_per\_iter}_c \times \frac{\text{Num\_loop\_iter}}{\text{Num\_threads}}
\]

\[
\text{Parallel\_overhead}_c = \text{Parallel\_startup}_c + \text{Parallel\_const\_factor}_c \times \text{Num\_threads}
\]
Introduction & Motivation

- **State – of – Art**
  - Optimize single CPU performance, not considering shared resource contention
  - Limited use of models for compiler optimizations and transformations
  - All false sharing detection techniques implemented at runtime
False Sharing

- Processors maintain data consistency via cache coherency
  - Data sharing is at cache line granularity
- A store to a single data invalidates the whole copy of a cache line
- Successive read suffers a cache miss
  - Reload entire cache line
False Sharing

- Processors maintain data consistency via cache coherency
  - Data sharing is at cache line granularity
- A store to a single data invalidates the whole copy of a cache line
- Successive read suffers a cache miss
  - Reload entire cache line
Effects of False Sharing

False sharing is a performance degrading data access pattern that can arise in systems with distributed, coherent caches.

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td>Unoptimized</td>
<td>0.503</td>
</tr>
<tr>
<td>Optimized</td>
<td>0.503</td>
</tr>
</tbody>
</table>
## False Sharing: Monitoring Results

- **Cache line invalidation measurements**

<table>
<thead>
<tr>
<th>Program name</th>
<th>1-thread</th>
<th>2-threads</th>
<th>4-threads</th>
<th>8-threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>13</td>
<td><strong>7,820,000</strong></td>
<td><strong>16,532,800</strong></td>
<td><strong>5,959,190</strong></td>
</tr>
<tr>
<td>kmeans</td>
<td>383</td>
<td>28,590</td>
<td>47,541</td>
<td>54,345</td>
</tr>
<tr>
<td>linear_regression</td>
<td>9</td>
<td><strong>417,225,000</strong></td>
<td><strong>254,442,000</strong></td>
<td><strong>154,970,000</strong></td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>31,139</td>
<td>31,152</td>
<td>84,227</td>
<td>101,094</td>
</tr>
<tr>
<td>pca</td>
<td>44,517</td>
<td>46,757</td>
<td>80,373</td>
<td>122,288</td>
</tr>
<tr>
<td>reverse_index</td>
<td>4,284</td>
<td>89,466</td>
<td>217,884</td>
<td><strong>590,013</strong></td>
</tr>
<tr>
<td>string_match</td>
<td>82</td>
<td><strong>82,503,000</strong></td>
<td><strong>73,178,800</strong></td>
<td><strong>221,882,000</strong></td>
</tr>
<tr>
<td>word_count</td>
<td>4,877</td>
<td><strong>6,531,793</strong></td>
<td><strong>18,071,086</strong></td>
<td><strong>68,801,742</strong></td>
</tr>
</tbody>
</table>
False Sharing: Data Analysis Results

- Determining the variables that cause misses

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Global/static data</th>
<th>Dynamic data</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>-</td>
<td>main_221</td>
</tr>
<tr>
<td>linear_regression</td>
<td>-</td>
<td>main_155</td>
</tr>
<tr>
<td>reverse_index</td>
<td>use_len</td>
<td>main_519</td>
</tr>
<tr>
<td>string_match</td>
<td>key2_final</td>
<td>string_match_map_266</td>
</tr>
<tr>
<td>word_count</td>
<td>length, use_len, words</td>
<td>-</td>
</tr>
</tbody>
</table>
Runtime Handling of False Sharing

Related Work

- **False Sharing Detection Methods**
  - Cache simulation and Memory tracing (Gunther and Weidendorfer WBIA’09, Marathe and Muller TPDS’07, Martonosi et al Sigmetrics’92)
  - Hardware Performance Counters (Marathe et al. Tech. rep.’06, Wicaksono et al. LCPC’11)
  - Memory Protection (Tongping and Berger OOPSLA’11)
  - Memory Shadowing (Zhao et al.VEE’11)

- **False Sharing Elimination Methods**
  - Tune scheduling parameters (chunk size, chunk stride) (Chow and Sarkar ICPP’97)
  - Compiler transformations (array padding, memory alignment) (Jeremiassen and Eggers PPoPP’94)

- All FS detection methods are applied at runtime, incur overhead
False Sharing Cost Model

- **False Sharing (FS) Modeling**
  - Estimates the performance impact of FS on OpenMP parallel loops at compile – time.

- **Features:**
  - Ability to output the total number of FS cases that will occur during execution of the parallel loop.
  - Ability to analyze the performance impact of FS on a parallel loop as a percentage of execution time.
  - Introduces a linear regression model to reduce the modeling time by approximation without impacting its accuracy.
Methodology

- False Sharing Model needs:
  - # of threads executing the loop
  - Loop boundaries
  - Step sizes
  - Index variables
  - Chunk size (if specified for OpenMP loop)
Methodology

False Sharing Modeling

- Technique is comprised of 4 steps
  - Obtain array references made in the innermost loop of a loop nest
  - Generate a cache line ownership list for each thread
  - Apply a stack distance analysis to cache state of each thread
  - Detect false sharing
Methodology – Step 1

- Obtain array references made in the innermost loop of a loop nest
  - Array base name
  - Array indices
  - Memory offsets for arrays with structured data types
Methodology – Step 2

- Generate a cache line ownership list

<table>
<thead>
<tr>
<th>Thread0</th>
<th>Thread1</th>
<th>Thread7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration_1: cacheline_a_1_w, cacheline_b_1_r</td>
<td>Iteration_1: cacheline_a_1_w, cacheline_b_2_r</td>
<td>Iteration_1: cacheline_a_1_w, cacheline_b_8_r</td>
</tr>
</tbody>
</table>

- Assumption: all array variables are cache aligned
Methodology – Step 3

- Apply a stack distance analysis
- Simulate fully associative cache
  - impossible to know corresponding cache line in a set at compile time
  - modeling the fully associative cache is mostly valid especially for caches with high level of associativity

Methodology – Step 4

- Detect False Sharing
  - Perform 1 to All comparison
    - $\varphi(cs_k, cl_i) = \begin{cases} 1, & \text{if } (cl_i \in cs_k \text{ and } cs_k^{cl_i} = W) \\ 0, & \text{otherwise} \end{cases}$
    - do other cache states contain my cache line?

- Perform comparison for each thread’s new cache line ownership list at each iteration until all iterations in one chunk are evaluated

$$false_{\_sharing}_{iter} = \sum_{j=0}^{k-1} \sum_{i=0}^{n} \varphi(cs_j, cl_i) \times \text{mask}(cs_j, cl_i)$$

$$\text{mask}(cs_j, cl_i) = \begin{cases} 0, & \text{if } (cl_i \in \text{CLOL}_{cs}^{cs_j}) \\ 1, & \text{otherwise} \end{cases}$$

- Perform steps 2-4 until all iterations are finished
Methodology - Prediction with linear regression

- Full model is expensive when # iterations becomes large
- Prediction with Linear Regression
  - Predict the total false sharing cases by evaluating much lower number of iterations in much less time
Methodology - Prediction with linear regression

- First $n$ iterations: $x = \{x_1, .. x_n\}$ where $n<<N$
- False sharing cases in $n$: $y = \{y_1, .. y_n\}$

- Prediction can be modeled as $\hat{y} = ax + b$

- **Least Square Solution:** $f(a, b) = \|ax + b - y\|_2$
  - We want $a, b = \arg \min_{a,b} f(a, b) = (ax + b - y)^T (ax + b - y)$

- Differentiate the function: $\frac{\partial f}{\partial a} = 0, \frac{\partial f}{\partial b} = 0$

- Then, $b = \sum_{i=0}^{n-1} y_i - \frac{a}{n} \sum_{i=0}^{n-1} x_i$  
  $a = \sum_{i=0}^{n-1} x_i y_i / \sum_{i=0}^{n-1} (x_i)^2$

- Total number of iterations: $x_{\text{max}}$
- Predict $y_{\text{max}}$ as: $y_{\text{max}} = ax_{\text{max}} + b$
Implementation and Experiments

- Implemented in OpenUH compiler
- Separate IR pass to collect memory load/store details, loop details
- No modification to compiler’s IR
- Can be implemented in similar approach in other compilers
Evaluation

- Memory access dominates the total execution
- Accuracy evaluation of full false sharing model
  - Compare the percentages of measured and modeled FS overhead costs on the total loop execution time.

\[
\frac{T_{fs\_measured} - T_{nfs\_measured}}{T_{fs\_measured}} \approx \frac{T_{fs\_modeled} - T_{nfs\_modeled}}{T_{fs\_modeled}} \approx \frac{N_{fs\_modeled} - N_{nfs\_modeled}}{N_{fs\_modeled}}
\]

- Efficiency of false sharing prediction with linear regression
  - Compare the number of false sharing cases estimated by full false sharing model and the prediction with linear regression model
Experiments

- **Architecture:**
  - Four 2.2 GHz 12-core processors (48 cores in total).
  - Dedicated L1 and L2 caches, 64Kb and 512KB per core.
  - L3 cache of 10240KB shared among 12 cores.
  - Cache line size for all caches, 64 bytes.
## Experimental Results – Heat Diffusion

<table>
<thead>
<tr>
<th># of threads</th>
<th>Measured Time with chunk size=1 FS case (sec)</th>
<th>Measured Time with chunk size=64 non-FS case (sec)</th>
<th>Measured FS effect on execution time (%)</th>
<th>Modeled FS cases effect (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.3593</td>
<td>0.2901</td>
<td>19.2%</td>
<td>6.9%</td>
</tr>
<tr>
<td>4</td>
<td>0.2263</td>
<td>0.1646</td>
<td>27.2%</td>
<td>6.9%</td>
</tr>
<tr>
<td>8</td>
<td>0.1639</td>
<td>0.156</td>
<td>4.8%</td>
<td>6.9%</td>
</tr>
<tr>
<td>16</td>
<td>0.6586</td>
<td>0.6205</td>
<td>5.7%</td>
<td>7.0%</td>
</tr>
<tr>
<td>24</td>
<td>1.0049</td>
<td>0.9564</td>
<td>4.8%</td>
<td>7.1%</td>
</tr>
<tr>
<td>32</td>
<td>1.4671</td>
<td>1.3608</td>
<td>7.2%</td>
<td>7.2%</td>
</tr>
<tr>
<td>40</td>
<td>1.8455</td>
<td>1.6130</td>
<td>12.5%</td>
<td>7.2%</td>
</tr>
<tr>
<td>48</td>
<td>2.247</td>
<td>2.1501</td>
<td>4.3%</td>
<td>7.2%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of threads</th>
<th>Pred. # of FS cases chunk size=1 (chunk run=20)</th>
<th>Pred. # of FS cases chunk size=64 (chunk run=20)</th>
<th>Pred. FS cases effect</th>
<th>Modeled # of FS cases chunk size=1</th>
<th>Modeled # of FS cases chunk size=64</th>
<th>Modeled FS cases effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>91,991K</td>
<td>1,595K</td>
<td>6.8%</td>
<td>94,421K</td>
<td>2,107K</td>
<td>6.9%</td>
</tr>
<tr>
<td>4</td>
<td>92,979K</td>
<td>1,625K</td>
<td>6.8%</td>
<td>94,446K</td>
<td>2,145K</td>
<td>6.9%</td>
</tr>
<tr>
<td>8</td>
<td>93,498K</td>
<td>1,702K</td>
<td>6.8%</td>
<td>94,458K</td>
<td>2,070K</td>
<td>6.9%</td>
</tr>
<tr>
<td>16</td>
<td>93,990K</td>
<td>1,724K</td>
<td>6.9%</td>
<td>96,043K</td>
<td>1,888K</td>
<td>7.0%</td>
</tr>
<tr>
<td>24</td>
<td>94,155K</td>
<td>1,609K</td>
<td>6.9%</td>
<td>96,938K</td>
<td>1,699K</td>
<td>7.1%</td>
</tr>
<tr>
<td>32</td>
<td>93,986K</td>
<td>1,456K</td>
<td>6.9%</td>
<td>97,159K</td>
<td>1,509K</td>
<td>7.2%</td>
</tr>
<tr>
<td>40</td>
<td>94,286K</td>
<td>1,826K</td>
<td>7.0%</td>
<td>97,730K</td>
<td>1,889K</td>
<td>7.2%</td>
</tr>
<tr>
<td>48</td>
<td>94,319K</td>
<td>1,107K</td>
<td>7.0%</td>
<td>97,935K</td>
<td>1,126K</td>
<td>7.2%</td>
</tr>
</tbody>
</table>

### Heat Diffusion

![Heat Diffusion Graph](image_url)
# Experimental Results - FFT

<table>
<thead>
<tr>
<th># of threads</th>
<th>Measured Time with chunk size=1 FS case (sec)</th>
<th>Measured Time with chunk size=16 non-FS case (sec)</th>
<th>Measured FS effect on execution time (%)</th>
<th>Modeled FS cases effect (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.0978</td>
<td>1.762</td>
<td>15.9%</td>
<td>32.0%</td>
</tr>
<tr>
<td>4</td>
<td>1.762</td>
<td>0.9618</td>
<td>45.4%</td>
<td>31.6%</td>
</tr>
<tr>
<td>8</td>
<td>0.8976</td>
<td>0.6033</td>
<td>32.7%</td>
<td>31.5%</td>
</tr>
<tr>
<td>16</td>
<td>0.599</td>
<td>0.3688</td>
<td>38.4%</td>
<td>33.2%</td>
</tr>
<tr>
<td>24</td>
<td>0.5041</td>
<td>0.3163</td>
<td>37.2%</td>
<td>32.8%</td>
</tr>
<tr>
<td>32</td>
<td>0.4727</td>
<td>0.2827</td>
<td>40.1%</td>
<td>35.6%</td>
</tr>
<tr>
<td>40</td>
<td>0.4792</td>
<td>0.2669</td>
<td>44.3%</td>
<td>36.7%</td>
</tr>
<tr>
<td>48</td>
<td>0.4664</td>
<td>0.279</td>
<td>40.1%</td>
<td>35.8%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of threads</th>
<th>Pred. # of FS cases chunk size=1 (chunk run=50)</th>
<th>Pred. # of FS cases chunk size=16 (chunk run=50)</th>
<th>Pred. FS cases effect</th>
<th>Modeled # of FS cases chunk size=1</th>
<th>Modeled # of FS cases chunk size=16</th>
<th>Modeled FS cases effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>52,233K</td>
<td>26,468K</td>
<td>32.4%</td>
<td>53,058K</td>
<td>27,358K</td>
<td>32.0%</td>
</tr>
<tr>
<td>4</td>
<td>52,697K</td>
<td>26,491K</td>
<td>32.8%</td>
<td>53,088K</td>
<td>27,702K</td>
<td>31.6%</td>
</tr>
<tr>
<td>8</td>
<td>52,928K</td>
<td>26,612K</td>
<td>32.8%</td>
<td>53,311K</td>
<td>27,882K</td>
<td>31.5%</td>
</tr>
<tr>
<td>16</td>
<td>52,936K</td>
<td>26,526K</td>
<td>32.9%</td>
<td>54,411K</td>
<td>27,257K</td>
<td>33.2%</td>
</tr>
<tr>
<td>24</td>
<td>52,967K</td>
<td>27,475K</td>
<td>31.8%</td>
<td>54,956K</td>
<td>28,003K</td>
<td>32.8%</td>
</tr>
<tr>
<td>32</td>
<td>52,983K</td>
<td>25,523K</td>
<td>34.2%</td>
<td>55,245K</td>
<td>25,865K</td>
<td>35.6%</td>
</tr>
<tr>
<td>40</td>
<td>53,077K</td>
<td>24,895K</td>
<td>35.1%</td>
<td>55,510K</td>
<td>25,154K</td>
<td>36.7%</td>
</tr>
<tr>
<td>48</td>
<td>52,998K</td>
<td>25,649K</td>
<td>34.1%</td>
<td>55,542K</td>
<td>25,878K</td>
<td>35.8%</td>
</tr>
</tbody>
</table>

**FFT**

![FFT](chart.png)

- Measured
- Predicted
- Modeled

**Number of Threads**

**False Sharing Effect %**

- 0
- 10
- 20
- 30
- 40
- 50
Summary

- False Sharing Cost Model is useful:
  - Assist compiler in optimizing code in high-level loop transformation, low-level instruction scheduling and code generation.
  - Guide traditional loop transformations.
  - Assist in generating efficient code.
Thank You