Revisiting loop tiling for datacenters: Live and Let Live

Jiacheng Zhao, Huimin Cui, Yalin Zhang, Jingling Xue, Xiaobing Feng
Institute of Computing Technology, Chinese Academy of Sciences
Work in conjunction with Prof. Jingling Xue, UNSW, Australia
ICS’18, Beijing, China, Jun 15, 2018
Compiler optimization meets datacenter

• Compiler optimization
  ▪ Try to utilize hardware resources efficiently
    – Multi-level parallelism for ILP and TLP
    – Instruction scheduling for pipeline
    – Loop tiling for cache
  ▪ Common property:
    – Exclusively enjoy all resources of a target machine
    – Especially architectural shared resources

• Datacenter era:
  ▪ Mix workloads to attain high utilization
Compiler optimization meets datacenter

• Loop (cache) tiling:
  - Locality optimization for matrix computations
  - Tile data for different levels of cache
  - Always tile for entire cache

Uncontended

Tiled data

Co-located

Apps

Tiled data
Compiler optimization meets datacenter

- Loop (cache) tiling:
  - Locality optimization for matrix computations
  - Tile data for target levels of cache
  - Always tile for entire cache

Tiled application suffered from co-location
Compiler optimization meets co-location

- Loop (cache) tiling:
  - Locality optimization for matrix computations
  - Tile data for target levels of cache
  - Always tile for entire cache

Tile size really matters
Compiler optimization meets co-location

- **Loop (cache) tiling:**
  - Locality optimization for matrix computations
  - Tile data for target levels of cache
  - Always tile for entire cache

- **Summary:**
  - Static best tile size can not deliver optimal performance when co-running
  - No single tile size can always perform best with different co-runners

- **Problem:**
  - How to select tile size in a co-location scenario?
Our Goals

• Constructing a peer-aware analytical TSS model
  ▪ Previous TSS model:
    – Cache
    – Reuse
  ▪ TSS Model
    – Take co-runners into consideration
  ▪ Work for real machines
  ▪ Without special os/hardware support
A tiled GEMM: $C = A \times B$

- **Two level 3-D tiling**
  - **Inner level**: for private cache (IB), fixed in our work
  - **Outer level**: for shared cache (OB)
A tiled GEMM: $C = A \times B$

- Three reuse patterns and reused data
  - $R_C$, $R_B$, $R_A$,
  - Descending reuse distance
Our Intuition

- Tiled application ($T_A$) + Cache flusher ($F$)
  - $T_A$: Multiple reuse patterns
  - $F$: Tunable pressure on shared cache
- Consider the two consecutive access of $R_C$
- What will happen as $F$ gradually increases its speed of fetching data from shared cache?
Our Intuition

Physical Time

Uncontended

Cache

\( rd(R_C) < C \)

\( rd(R_B) < C \)
Our Intuition

**Uncontended**

Content with \( F \) Very **slow** fetching speed

**Physical Time**

Data fetched by \( F \)
Our Intuition

Uncontended

Content with $F$ Very slow fetching speed

Content with $F$ Faster fetching speed

Physical Time

Data fetched by $F$

Reuse of $R_C$ disappeared
Our Intuition

Uncontended

Content with **Faster** fetching speed

Content with **Fastest** fetching speed

Data fetched by **F**

Reuse of $R_C$ disappeared

Reuse of $R_B$ disappeared

Physical Time
Key Observation

- Cache miss as a function of data fetching speed
- Ideal Model: A step function
Key Observation

- Reuse blocks of $T_A$ will be evicted from shared cache in decreasing order of their reuse distances, as $F$ increases its data-fetching speed
  - $R_C, R_B, R_A$ in order
Key Observation

• If $R_i$ is going to be evicted, all other reuse blocks with a larger reuse distance have already been evicted
Reuse-pattern-centric approach

- Focusing on each reuse pattern
- Two important model parameters:
  - The number of cache misses if one reuse pattern disappears in shared cache
  - The breakpoint “speed” after which a reuse pattern disappears in shared cache
Outline

- Introduction and Backgrounds
- Our Key Observation
- Our Approach: Peer-aware tiling
- Experimental Results
- Conclusion
Our Approach: Peer-aware tiling

• Modeling each individual reuse pattern
  ▪ Co-runners modeled as aggregate cache pressure, $p$
    – No need to analyze all co-running peers
  ▪ Decouple cache misses into two parts:
    – Cache miss in solo run: $miss_{solo}$
    – Extra cache miss when co-running: $\Delta(R)$
      » Reuse data are evicted as their reuse distance is larger than cache size
      » Modeled as a function of cache pressure

• Aggregate all reuse patterns
  ▪ Simply add $\Delta(R)$ from all reuse patterns
Our Approach: Reuse-pattern-centric modeling

- Modeling individual reuse pattern:
  - Profiling + Compiler analysis approach
  - Model parameters:
    - For $T_A$: $hit_{solo}$, $miss_{solo}$, $t_{solo}$
      » Obtained by offline profiling
    - For each reuse pattern: $rd(R)$, $n(R)$, $rc(R)$
      » Obtained through compiler analysis
  - Platform constants:
    » Memory/LLC access latency
    » Shared cache association
    » Cache size ($C$)
  - Cache flusher: $p$
    » L2LinesInRate
    » Cache Pressure
Reuse-pattern-centric modeling: Overall

- Basically, we have
  \[ \Delta(R) = \text{hit}(R) \times \sigma(R, F, C) \]
- where
  \[ \sigma(R, F, C) = \begin{cases} 
  1 & f p(R) + f p(F) > C \\ 
  0 & \text{otherwise} 
  \end{cases} \]
Estimating $hit(R)$

- Ideally, $hit(R)$ can be calculated from $n(R) \times rc(R)$
- A more precise approach: distribute hits in solo run over all reuse patterns

\[
hit(R) = \frac{n(R) \times rc(R)}{\sum_j n(R_j) \times rc(R_j)} \times hit_{solo}
\]
Estimating \( p^* \)

\[
\sigma(R, F, C) = \begin{cases} 
1 & \text{if } fp(R) + fp(F) > C \\
0 & \text{otherwise}
\end{cases}
\]

- We have:
  \[
  fp(R) = rd(R)
  \]
  \[
  fp(F) = p^* t_R
  \]

- \( t_R \) is the time duration of two consecutive accesses of \( R \)

\( \text{Physical time, not logical time} \)
Physical Time

Data fetched by $F$

Uncontended

Content with $F$

Very slow fetching speed

Content with $F$

Faster fetching speed

Reuse of $R_C$ disappeared

$t_R$
Estimating $p^*$

For the outer-most reuse pattern, e.g. $R_C$
- $t_R$ can be estimated using $t_{solo}$

For other reuse patterns, e.g. $R_B, R_A$
- Estimating execution time when all reuse blocks with larger reuse distance have been evicted, details in paper

$f_p(F) = p \times t_R$
Estimating $p^*$

- We can obtain:

$$p^* = \frac{C - rd(R)}{t_R}$$
Adjustment for real machines

**Ideal**

![Graph showingIdeal](image)

**Real**

![Graph showing Real](image)
Adjustment for real machines

- Cache is not fully-associative
  - Cache is “smaller”:
    - Effective cache size ($eC$)
    - $eC = \frac{c}{\#ways} \left(\frac{\#ways}{2} + 2\right)$
  - Based on [Sen+, SIGMETRICS ’13]
- Step function to continuous function
- A lower bound: $\widehat{p}^*$
  - Corresponding to $eC$
  - $p^*$ as upper bound
- A $\text{tanh}$ function for simulation
  - Inspired by machine learning
Put it all together

- Modeling individual reuse pattern:
  - Extra cache miss for a reuse pattern $R$ when co-running with a cache flusher of pressure $p$

$$\Delta(R) = \begin{cases} 
  0 & \text{hit}(R) \\
  \frac{1}{2} \left( \tanh \left( p - \frac{p^* + \tilde{p}^*}{2} \right) \right) & \tilde{p}^* \leq p \leq p^* \\
  \text{hit}(R) & p > p^*
\end{cases}$$

- Aggregate all reuse patterns:

$$\text{miss}(T_A) = \sum_i \Delta(R_i) + \text{miss}_{solo}(T_A)$$
Our Approach: A Peer-aware tiling framework

• Leverage our model to support two optimization objectives:
  ▪ Efficiency: Live
    – Reduce performance slowdown suffered by $T_A$
    – Tile size minimizing shared cache miss count of $T_A$

$$miss(T_A) = \sum_i \Delta(R_i) + miss_{solo}(T_A)$$
Our Approach: A Peer-aware tiling framework

- Leverage our model to support two optimization objectives:
  - **Efficiency: Live**
    - Reduce performance slowdown *suffered* of $T_A$
    - Tile size minimizing shared cache miss count of $T_A$
  - **Niceness: Let Live**
    - Reduce performance slowdown *incurred* by $T_A$
    - Tile size minimizing shared cache miss frequency of $T_A$
      » Actually minimizing memory bandwidth consumption

\[
\text{miss}_\text{freq}(T_A) = \frac{\text{miss}(T_A)}{t_{\text{est}_\text{d}\_\text{time}}}
\]
Our Approach: Tiling for different objectives

- Leverage our model to support two optimization objectives:
  - Efficiency: Live
    - Reduce performance slowdown *suffered* of $T_A$
    - Tile size minimizing shared cache miss count of $T_A$
  - Niceness: Let Live
    - Reduce performance slowdown *incurred* by $T_A$
    - Tile size minimizing shared cache miss frequency of $T_A$
      » Actually minimizing memory bandwidth consumption
- We focus on TSS model
- Parametric tiling by PrimeTile [Hartono+ , ICS’09]
Outline

- Introduction and Backgrounds
- Our Key Observation
- Our Approach: Peer-aware loop tiling
- Experimental Results
- Conclusion
Evaluation Setup: Methodology

• **Evaluating efficiency: Live**
  - Optimizing performance of tiled matrix computations
  - Reducing performance slowdown *suffered*
  - LLC miss count reduction/prediction
  - Accuracy of our prediction model
  - Key observation of optimal tile size
  - Improve performance of library routines

• **Evaluation niceness: Let live**
  - Optimizing performance of co-located applications
  - Reducing performance slowdown *incurred*
Evaluation Setup: Apps & Platforms

- **Applications:**
  - *Matmul (GEMM) & tmm* from Pluto
  - *conv* from Caffe
  - *memcached* as a latency-sensitive application with *mutilate* being load generator and latency tester
  - *SPEC* and *STREAM* as co-runners

- **Platforms:**
  - Westmere-based Intel six-core Xeon E5645 (main)

- **Loop tiling:**
  - Two-level loop tiling
    - Inner tile size: fixed as 56
    - Outer tile size: 840 (static best, found by hand-tuning)
      - Search space for our approach: [280, 840] with a step of 56 (inner tile size)
Efficiency: Overall Performance of GEMM

- Optimizing performance of GEMM when co-locating

- Performance slowdown suffered is reduced by 34.1%, from 22.9% to 15.1% (14.8% for optimal tiling)
Efficiency: LLC Miss Reduction

• Benefits come from cache miss reduced

• LLC miss count is reduced by 24%, from 8.24x to 6.24x
Efficiency: LLC Miss Prediction

• Precise prediction LLC Miss of GEMM when co-locating

• Prediction error: average 3.9%
Efficiency: Accuracy of Prediction Model

- Prediction models for three representative tile sizes
Efficiency: Optimal Tile Size

- Optimal tile size is a non-monotone function of co-runner’s cache pressure

- Aggregate behavior of different reuse patterns
Efficiency: Optimizing Library Routines

- Improving performance of ATLAS when co-running
  - Three configurations
    - **ATLAS**: default implementation
    - **Static**: ATLAS + hand-tuning outer tiling (fixed tile size)
    - **Peer-aware**: Static + our approach (tile sized determined at runtime)
  - One application
    - *cblas_dgemm*
Efficiency: Optimizing Library Routines

• **8.9%** improvement over ATLAS
• **7.1%** improvement over Static
Niceness: Optimizing co-running peers

- Scenario:
  - memcached and GEMM co-located together
  - GEMM’s optimization objective is set to “niceness”

- 95th latency: Solo (228 us)
  - Static (292 us), 28% slowdown
  - Peer-aware (271 us), 18.8% slowdown, a reduction factor of 32.9%
Outline

- Introduction and Backgrounds
- Our Key Observation
- Our Approach: Peer-aware tiling
- Experimental Results
- Conclusion
Conclusion

• A reuse-pattern-centric approach for modeling co-running cache behavior of tiled matrix computations
• An approach for analytically building above model
• A peer-aware tiling framework supporting two optimization objectives: efficiency and niceness, based on our analytical model
Thank you.