Revisiting Loop Tiling for Datacenters: Live and Let Live

Jiacheng Zhao
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zhaojiacheng@ict.ac.cn

Huimin Cui
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
Beijing, China
cuihm@ict.ac.cn

Yalin Zhang
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
zhangyalin2014@ict.ac.cn

Jingling Xue
School of Computer Science and Engineering, University of New South Wales
Sydney, NSW 2052, Australia
jingling@cse.unsw.edu.au

Xiaobing Feng
SKL Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
fxb@ict.ac.cn

ABSTRACT
As DNNs gain popularity in modern datacenters, it becomes imperative to revisit compiler optimizations for DNNs in a co-location scenario. Loop tiling turns out to be the most significant compiler optimization, since DNNs typically apply a series of matrix computations iteratively to a massive amount of data.

We introduce a reuse-pattern-centric approach to obtaining a peer-aware TSS (Tile Size Selection) model for a matrix-based application A. Our key insight is that the co-running cache behavior of A (once tiled) can be determined by its data reuse patterns, together with the cache pressure exerted by its co-running peers, without actually the need for analyzing the code of its co-runners. Compared with static tiling (that determines a tile size for A statically without considering its co-running peers), our peer-aware tiling enables compilers to generate either faster peer-aware efficient code for A (by optimizing the performance of A) or faster peer-aware nice code for A (by optimizing the performance of its co-runners). In addition, our peer-aware tiling also enables library developers to improve the performance of library routines (more effectively than static tiling).

CCS CONCEPTS
• Software and its engineering → Compilers;

KEYWORDS
Tile Size Selection, Quality of Service, Reuse Distance, Loop Tiling, Parametric Tiling, Co-location, Cross-core Interference, Multi-core Processors, Code Generation, Shared Resource Contention

 ACM Reference Format:
Jiacheng Zhao, Huimin Cui, Yalin Zhang, Jingling Xue, and Xiaobing Feng. 2018.
Revisiting Loop Tiling for Datacenters: Live and Let Live. In ICS ’18:
International Conference on Supercomputing, June 12–15, 2018, Beijing, China.

1 INTRODUCTION
Deep neural networks (DNNs) have gained popularity in a variety of applications such as speech recognition, computer vision, and bioinformatics, prompting their deployment across datacenters [20]. A datacenter has two categories of applications [13, 26, 42], user-facing latency-sensitive applications, e.g., web search, with strict QoS needs, and batch applications, e.g., those for training DNNs, with medium QoS needs. A datacenter usually selects some batch applications to co-locate with latency-sensitive applications on CMP servers, thereby increasing hardware resource utilization. However, this is done only when the strict QoS requirements of latency-sensitive applications are guaranteed [42].

Therefore, it becomes imperative to revisit compiler optimizations for DNNs in a co-location scenario. In general, DNN-based applications, especially in convolutional layers and fully connected layers, are compute-intensive, as they typically apply a series of matrix computations iteratively to a massive amount of data. For this reason, loop tiling turns out to be the most significant compiler optimization. In this paper, we introduce peer-aware loop tiling that enables tiling a DNN-based application by optimizing the performance of either itself or its co-running peers. This allows the QoS needs of two types of applications to be met judiciously.

The key challenge faced lies in TSS (Tile Size Selection), which aims to keep the frequently used data in a program at a given level of cache targeted. A great deal of research has been done on TSS for solo executions [3–5, 9, 31, 32, 44, 48, 54], by assuming that the program runs on an unloaded machine, without being aware of its co-runners.
For a co-located application $\mathcal{A}$, loop tiling that is applied in a traditional compiler is ineffective for two reasons. First, $\mathcal{A}$ can no longer enjoy exclusively all the underlying hardware resources. Worse still, the compiler does not even know the amount of shared resources, e.g., the portion of the shared cache available when $\mathcal{A}$ starts executing. Second, loop tiling is now applied to achieve a different goal. We will need to tile $\mathcal{A}$ by making it run not only efficiently but also nicely [41]. To make $\mathcal{A}$ efficient by itself, the compiler would like to keep the data that are frequently used by $\mathcal{A}$ in the shared cache as in the solo execution of $\mathcal{A}$. To make $\mathcal{A}$ nice to its co-runners, the compiler should ensure that $\mathcal{A}$ does not snatch the shared resources (e.g., the portion of the shared cache) that are supposed to be used by its co-running peers so as to violate their QoS requirements.

Tang et al. [41] discuss how to compile for niceness, by inserting nop instructions to throttle down the memory access rates of batch applications. Srinivas et al. [37] introduce reactive loop tiling, which generates multiple tiled code versions with different tile sizes for a given loop nest and dynamically selects the best version at runtime. However, reactive tiling makes two assumptions. First, a TSS model is assumed to be available so that tile sizes can be selected. Second, the OS/hardware supports cache partitioning so that different applications run in disjoint parts of the shared cache.

In this paper, we focus on developing a TSS model that is well suited to a co-location scenario for modern datacenters without requiring any special OS/hardware support. This brings two benefits. First, it is difficult to pre-determine a cache partitioning scheme between co-running applications, due to a combinational explosion of application co-locations, which is caused by the sheer number of workloads to be considered on a large number of processor cores. Second, it is also difficult to make cache reservation for latency-sensitive applications, due to their workload fluctuations, since they may need to run on more cores at peak times but less otherwise.

For a matrix-based application $\mathcal{A}$, our key insight is that the co-running cache behavior of its tiled version, $T_\mathcal{A}$, can be determined by analyzing its data reuse patterns alone without actually the need for analyzing the code of its co-running peers. Specifically, the co-running cache behavior of $T_\mathcal{A}$ is simply an aggregation of the co-running cache behaviors of its individual data reuse patterns. This insight makes it possible to develop a peer-aware TSS model $M(T_\mathcal{A})$ for $T_\mathcal{A}$, analytically, by characterizing its shared cache (LLC) miss count as a function of the cache pressure exerted to $T_\mathcal{A}$ by its co-running peers. In particular, $M(T_\mathcal{A})$ is built efficiently offline by using a small amount of profiling data.

We have developed a peer-aware loop tiling approach that inserts instrumentation code just before $T_\mathcal{A}$ to monitor the cache pressure exerted by its co-running peers and then selects an appropriate tile size based on $M(T_\mathcal{A})$ to enable $T_\mathcal{A}$ to run afterwards. Currently, we allow a tile size to be selected by optimizing one of the two optimization objectives. To maximize the performance of $T_\mathcal{A}$, we will select a tile size that minimizes the shared cache miss count for $T_\mathcal{A}$ in order to keep its frequently used data in the shared cache. To maximize the performance of $T_\mathcal{A}$’s co-running peers (at its own expense), we will select a tile size that minimizes the shared cache miss frequency of $T_\mathcal{A}$ in order to minimize the shared cache interference to the co-running peers.

Peer-aware loop tiling can be easily deployed in modern datacenters for DNN-based applications. Just before each matrix computation starts, its tile size can be selected according to a user-annotated optimization objective and the cache pressure detected in real time. This paper makes the following contributions:

- We introduce a reuse-pattern-centric approach for modeling the co-running cache behavior of a tiled matrix-based application as an aggregation of the co-running cache behaviors of its data reuse patterns.
- We introduce an approach for building analytically a peer-aware TSS model for a tiled application, without the need for analyzing its co-running peers. For a given tile size, the model predicts the application’s shared cache miss count as a function of the cache pressure exerted by the co-running peers.
- For a tiled matrix-based application, we observe that its optimal tile size is a non-monotone function of the cache pressure exerted by the co-running peers due to the combined effect of the application’s multiple data reuse patterns. Our model explains the reasons behind this non-intuitive observation.
- We introduce a peer-aware tiling approach that enables compiling a matrix-based application for efficiency or niceness in a co-location scenario. For the three representative benchmarks evaluated in a co-location scenario, our approach can predict their LLC miss counts with high accuracy on real machines (with an average error of 5.0% only). We have compared peer-aware tiling with static tiling on two Xeon machines (Westmere-based and Sandy Bridge-based). The Westere-based machine, peer-aware tiling can reduce their co-location performance slowdowns by 34.1%, 40.3% and 36.9%, on average, respectively. For a latency-sensitive co-running application, memcached [17], peer-aware tiling decreases its performance slowdown by 31.1% for the 95th percentile latency. For the Sandy Bridge-based machine, peer-aware is similarly more effective than static tiling.
- Our TSS model can also be applied to the development of library routines for matrix-based computations. By applying it to ATLAS, we achieve a performance increase of 8.9% in a co-location scenario.

The rest of the paper is organized as follows. Section 2 motivates this work. Section 3 presents our analytical model for peer-aware TSS. Section 4 describes our loop-tiling framework. Section 5 evaluates this work and analyzes our experimental results. Section 6 discusses the related work. Finally, Section 7 concludes.

2 MOTIVATION

Our motivating example is GEMM, a well-known kernel of General Matrix Multiplication. We present some results to demonstrate the limitations of static tiling. The experimental platform used is an Intel 2.40GHz six-core processor, with the details given in Section 6.

Figure 1 gives the GEMM code with two-level tiling. The inner-level tiling (with $IB$ as tile size) is for private caches while the outer-level (with $OB$ as tile size) is for the shared cache. In our experiments, $IB$ is fixed as 56 and how to select $OB$ is the focus of this paper, which is set as 840 for solo executions by hand-tuning. For simplicity, we focus on square tiles in both cases.
//Tiled code.
void gemm(int N)
{ int i, j, k, ii, jj, kk, iii, jjj, kkk;
  int OB, IB;
  //Tiling for shared cache.
  for (iii=0; iii<N; iii+=OB)
    for (jjj=0; jjj<N; jjj+=OB)
      for (kkk=0; kkk<N; kkk+=OB)
        for (ii=iii; ii<min(iii+OB,N); ii+=IB)
          for (jj=jjj; jj<min(jjj+OB, N); jj+=IB)
            for (kk=kkk; kk<min(kkk+OB,N); kk+=IB)
              //Tiling for private cache.
              for (i=ii; i<min(ii+IB, N); i++)
                for (j=jj; j<min(jj+IB, N); j++)
                  for (k=kk; k<min(kk+IB, N); k++)
                    C[i][j]+=A[i][k]*B[k][j];
}

Figure 1: The GEMM kernel code after two-level tiling.

Figure 2: Performance of GEMM when co-running with 10 workloads (with 0 for its solo-run).

2.1 Behaviors of Tiled Code When Co-located
We have manually generated 10 distinct STREAM kernels [30]. Each kernel has a working set of 200MB but with different data-fetching speeds, so that the 10 kernels exhibit different cache pressures to GEMM.

Figure 2 shows the performance variations of GEMM when executed alone and co-running with the 10 workloads, with the horizontal axis representing the workloads (with 0 for solo-run) and the vertical axes representing GEMM’s execution times (against the left one) and GEMM’s LLC miss counts (against the right one). There is a good correlation between the execution times and their corresponding LLC miss counts.

2.2 Limitations of Static Tiling
We discuss the limitations of static tiling for GEMM by continuing to use the same 10 co-running workloads. We select three tile sizes for GEMM, where $OB \in \{448, 616, 840\}$, and examine their performance variations in solo-run and when co-running with the 10 workloads. Figure 3 shows the results, with the horizontal axis representing the workloads and the vertical axis representing the LLC cache miss counts of GEMM. All the LLC cache miss counts are normalized to the tile size of 840.

Figure 3 uses red triangles to highlight the optimal tile sizes for all the workloads. For solo execution, the optimal tile size is statically fixed as 840. However, as shown in Figure 3, 840 is not always optimal in a co-location scenario. In particular, it is optimal only for workloads #4 and #5. The tile size 616 is optimal for workloads #1, #6 and #7. Finally, 448 is the optimal tile size for workloads #2, #3, and #8 – #10.

2.3 Behaviors of Co-runners
To demonstrate the interference of GEMM to a co-located latency-sensitive application, we co-run memcached with GEMM twice, once tiled with size $OB = 448$ and once tiled with size $OB = 840$. Note that memcached is a distributed memory caching system, which provides an in-memory key-value store for small chunks of arbitrary data and speeds up dynamic database-driven websites [17], as discussed in Section 5.

Figure 4 shows the 95th percentile latency of memcached for 100 repeated runs, with the horizontal axis representing the ID of a run and the y-axis representing the 95th percentile latency. The runs are not stable initially but become stabilized later. When memcached runs alone, the average 95th percentile latency for the 100 runs is 225µs. When co-running with GEMM (tiled with 448), the average 95th percentile latency increases to 256µs. When co-running with GEMM (tiled with 840), the average 95th percentile latency increases to 265µs. As demonstrated in [13], low tail latency is significantly important in datacenters.

2.4 Summary
In a co-location scenario, static tiling does not usually provide the optimal tile size for a tiled application. In addition, there does not
exist a fixed tile size that can always provide optimal performance. Finally, different tile sizes lead to different amounts of interference to the co-runners of the tiled application.

3 PEER-AWARE TILE SIZE SELECTION

This section presents our peer-aware TSS approach. We first discuss how to quantify the co-running cache behavior of a tiled matrix-based application in a reuse-pattern-centric manner (Section 3.1). We then develop a peer-aware TSS model (Section 3.2). Finally, we use GEMM to verify its accuracy (Section 3.3).

For simplicity, we ignore inclusion victims, i.e., we assume that the number of requests issued to the shared (LLC) cache remains unchanged in solo and co-located scenarios. In Section 3.3, we will see that this assumption does not affect the accuracy of our model.

3.1 Reuse-Pattern-Centric Analysis

Most matrix computations contain more than one reuse pattern. We take the tiled GEMM code given in Figure 1 as an example. Figure 5 shows how its data are reused in the shared cache. Each small square corresponds to a tile in the private cache. We are not concerned with the data reuse inside the private caches. The tiled GEMM code contains three reuse patterns, denoted as \( R_A, R_B, R_C \), with each pattern identified by a (data) reuse block highlighted in yellow. For each matrix \( X \in \{ A, B, C \} \), \( n(\text{Reu} X) \) represents the (reuse) size of a reuse block for \( X, rd(\text{Reu} X) \) the reuse distance of an element in the reuse block, and \( rc(\text{Reu} X) \) the number of times that such an element is reused.

Let us see how the reuse distance of \( R_A \) is computed. Between two successive accesses of an element for \( R_A \), the data fetched into the shared cache comprise one reuse block \( R_A \) of size \( IB \times OB \), one column of size \( OB \times IB \) from one reuse block of \( R_B \), and one small tile of size \( IB \times IB \) from one reuse block \( R_C \). Thus, the reuse distance for (an element in) \( R_A \) is found as:

\[
rd(\text{Reu} X) = 2 \times OB + IB + 1B \times IB
\]

The reuse distances for \( R_B \) and \( R_C \) are found similarly.

Now we analyze what happens when another application \( L \) is co-located with the tiled GEMM code. If \( L \) fetches data at a fixed speed that fills the shared cache before \( R_C \) is reused, then \( R_C \) will be evicted from cache due to its largest reuse distance and the LRU replacement policy used. If \( L \) has a fixed data-fetching speed throughout, we can expect that for each element \( c \) in \( R_C \), its reuse distance ends up being increased by the same amount, i.e., a fixed number of \( L \)'s data elements fetched into the cache between two successive accesses of \( c \). Therefore, there exists a certain cache pressure that turns all cache hits of \( R_C \) into cache misses.

As shown in Figure 5, \( R_C \) has \( n(\text{Reu} C) \) data elements with each element accessed \( rc(\text{Reu} C) \) times. Therefore, the additional cache miss count caused to \( R_C \) (due to the shared cache interference from \( L \)) is:

\[
\Delta(\text{Reu} C) = n(\text{Reu} C) \times rc(\text{Reu} C) = \frac{c^3}{OB}
\]

Similarly, if \( L \) fetches data at a higher speed that fills the shared cache before \( R_B \) is reused, not only \( R_C \) but also \( R_B \) will be evicted from cache, making all the accesses to \( R_B \) and \( R_C \) cache misses. The same applies to \( R_A \).

**Observation 3.1.** Consider a shared cache that uses LRU as its replacement policy. Let an application \( L \) be co-located with a tiled matrix application, which consists of \( m \) reuse patterns \( R_1, \ldots, R_m \). As \( L \) increases its data-fetching speed, their reuse blocks will be evicted from the shared cache in decreasing order of their reuse distances. Furthermore, if a reuse block for \( R_i \) is being evicted from the shared cache, then a reuse block for \( R_j \) that has a larger reuse distance, where \( rd(R_j) > rd(R_i) \), has already been evicted from the shared cache.

3.2 Creating a Peer-Aware TSS Model

Observation 1 leads to the development of a peer-aware TSS model. For a matrix-based application \( A \), we continue to denote its tiled version by \( T_A \).

We first discuss how to characterize the data reuse patterns for \( T_A \) (Section 3.2.1). We then model the co-running cache behaviors of its reuse patterns individually (Section 3.2.2). Finally, we obtain a peer-aware TSS model by aggregating all such individual co-running cache behaviors obtained (Section 3.2.3).

It is important to emphasize that our model is parameterized (often implicitly) by the tile size of \( T_A \). The key novelty here is that our model is developed to enable the performance of either \( T_A \) or its co-runners to be optimized, without having to analyze the code of the co-runners. Based on the cache pressure exerted to \( T_A \) by its co-runners just before its execution, the optimal tile size for \( T_A \) is determined from a set of candidate tile sizes, depending on whether we optimize \( T_A \) (for efficiency) or its co-runners (for niceness).

3.2.1 Characterizing Data Reuse Patterns. A reuse pattern \( R_i \) is identified by:

- \( n(R_i) \): The size of a reuse block for \( R_i \), 
- \( rd(R_i) \): The reuse distance of an element in \( R_i \), and 
- \( rc(R_i) \): The number of times that \( R_i \) is reused.

For the tiled GEMM code given in Figure 1, Figure 5 lists the three reuse patterns, \( R_A, R_B \) and \( R_C \).

For convenience, given a reuse pattern \( R_i \), we define:

\[
nord(R_i) = rd(R_i) - n(R_i)
\]

which represents the number of distinct elements outside \( R_i \) that are accessed between two successive accesses of an element in \( R_i \).
Revisiting Loop Tiling for Datacenters: Live and Let Live

ICS ’18, June 12–15, 2018, Beijing, China

Table 1: Parameters used for modeling a reuse pattern.

<table>
<thead>
<tr>
<th>Category</th>
<th>Notation</th>
<th>Tile Size</th>
<th>Remarks</th>
<th>How Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_A$ (Solo-Run)</td>
<td>$hit_{solo}$</td>
<td>$ti$</td>
<td>cache hit count for $ti$ in solo-run</td>
<td>offline</td>
</tr>
<tr>
<td>$T_A$ (Solo-Run)</td>
<td>$miss_{solo}$</td>
<td>$ti$</td>
<td>cache miss count for $ti$ in solo-run</td>
<td></td>
</tr>
<tr>
<td>$T_A$ (Solo-Run)</td>
<td>$t_{solo}$</td>
<td>$ti$</td>
<td>execution time for $ti$ in solo-run</td>
<td></td>
</tr>
<tr>
<td>$R$ (Reuse Pattern)</td>
<td>$rd(R)$</td>
<td>$ti$</td>
<td>reuse distance of $R$ for $ti$</td>
<td>compiler</td>
</tr>
<tr>
<td>$R$ (Reuse Pattern)</td>
<td>$n(R)$</td>
<td>$ti$</td>
<td>size of of $R$ for $ti$</td>
<td></td>
</tr>
<tr>
<td>$R$ (Reuse Pattern)</td>
<td>$rc(R)$</td>
<td>$ti$</td>
<td>reuse count of $R$ for $ti$</td>
<td></td>
</tr>
<tr>
<td>$F$ (Cache Flusher)</td>
<td>$p$</td>
<td>-</td>
<td>L2LinesInRate of $F$, i.e., its cache pressure</td>
<td>online</td>
</tr>
<tr>
<td>(Platform)</td>
<td>$\epsilon_{mem}$</td>
<td>-</td>
<td>memory access latency</td>
<td>constant</td>
</tr>
<tr>
<td>(Platform)</td>
<td>$\epsilon_{LLC}$</td>
<td>-</td>
<td>LLC access latency</td>
<td></td>
</tr>
<tr>
<td>(Platform)</td>
<td>$C$</td>
<td>-</td>
<td>LLC cache size</td>
<td></td>
</tr>
<tr>
<td>(Platform)</td>
<td>$\sigma$</td>
<td>-</td>
<td>LLC cache ways</td>
<td></td>
</tr>
</tbody>
</table>

Within $rd(R_i)$, two type of data are accessed: those in $R_i$ and those outside $R_i$.

By definition, $nord(R_i)$ gives the worst case cache miss count at the critical state just before $R_i$ is evicted from the shared cache. This is the point at which $R_i$ is still used but all elements outside $R_i$ are not.

3.2.2 Modeling an Individual Reuse Pattern. When modeling the shared cache misses for an individual reuse pattern $R$ of $T_A$, we use a cache flusher $F$ as its co-runner. We first assume that the shared cache is fully-associative with an LRU replacement policy and then adjust our model for a real machine.

Model Parameters. Table 1 lists a total of 12 parameters used. The top seven depend on a tile size $ti$ given but are not qualified by $ti$ to avoid cluttering. These parameters can be divided into the following four categories:

- **Behavior of $T_A$ in solo-run.** There are three parameters, recording $T_A$’s cache hit count, cache miss count, and execution time in solo-run. We use offline profiling to collect these data for a given tile size $ti$.
- **Reuse pattern $R$.** There are three parameters for characterizing $R$ (Section 3.2.1), which are calculated for a size $ti$ parametrically by compiler analysis.
- **$F$’s cache pressure.** There is just one parameter, the L2LinesInRate of $F$, used for representing $F$’s cache pressure exerted to $T_A$. This can be obtained by using lightweight online profiling with little overhead.
- **Hardware parameters.** There are four platform-specific parameters used: the average memory access latency, the LLC access latency, the cache capacity, and cache associativity (referred to as ways).

**Modeling Cache Sharing.** According to our observation in 3.1, the additional cache miss count that $R$ suffers when $T_A$ is co-running with $F$ can be estimated by using a step function:

$$\Delta(R) = hit(R) \times \sigma(R, F, C) \quad (2)$$

where $hit(R)$ is the number of cache hits corresponding to $R$’s accesses in solo-run, which will be modeled below, and $\sigma$ is the growth factor of $R$’s cache miss count:

$$\sigma(R, F, C) = \begin{cases} 1 & f_p(R) + f_p(F) > C \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Therefore, when the combined footprint of $R$ and $F$, i.e., $f_p(R) + f_p(F)$ exceeds the cache capacity $C$, all the cache hits of $R$ in solo-run are turned into misses.

**Modeling the Footprints $f_p(R)$ and $f_p(F)$.** To compute a footprint, we need to work with the physical time. We define the elapsed time between $R$’s two successive reuses as $t_R$ when $T_A$ is in solo-run. Thus, $f_p(R)$ and $f_p(F)$ represent the footprints of $R$ and $F$ during the time period $t_R$, respectively. By definition, $f_p(R)$ is simply its reuse distance and $f_p(F)$ can be estimated as the amount of data that are fetched at its data-fetching speed $p$ during $t_R$:

$$f_p(R) = rd(R) \quad (4)$$

$$f_p(F) = p \times t_R$$

Now we need to estimate $t_R$, the physical time of $R$’s two successive reuses when co-located. For simplicity, we use the physical time of $T_A$’s solo-execution.

Let us consider first when $R$ is a reuse pattern with the largest reuse distance in $T_A$, e.g., $R_C$ in GEMM. Since $t_R$ is a fragment of $T_A$’s execution, we assume that the same miss frequency, which can be computed as $miss_{solo}/t_{solo}$, occurs throughout. Thus, we have:

$$t_R = miss_{f_{p}}(R)/miss_{frequency} = miss_{solo}(R)/(miss_{solo}/t_{solo}) \quad (5)$$

where $miss_{f_{p}}(R)$ is $R$’s cache miss count between $R$’s two successive accesses in solo-execution, which is given by $nord(R)$, and $t_{solo}$ and $miss_{solo}$ are the execution time and miss count of $T_A$ listed in Table 1.

Next, we consider the case when $R$ is a reuse pattern that does not have the largest reuse distance. According to Observation 1, if $R$ is being evicted from the shared cache, all data reuse blocks with larger reuse distances have already been evicted from cache, with all their corresponding cache hits turned into cache misses. Therefore, $t_R$ given in (5) needs to be modified, since $miss_{solo}$ and $t_{solo}$ used there are no longer accurate.

In this case, $T_A$’s cache miss count turns out to be:

$$miss' = miss_{solo} + \sum_{rd(R_i)>rd(R)} hit(R_i)$$

and the corresponding execution time of $T_A$ becomes:

$$t' = t_{solo} + \epsilon_{mem} - \epsilon_{LLC} \times (\sum_{rd(R_i)>rd(R)} hit(R_i)) \quad (6)$$

As a result, the physical time $t_R$ has been refined to:

$$t_R = miss_{f_{p}}(R)/miss_{frequency} = miss_{solo}(R)/(miss' / t') = \frac{miss_{solo} \cdot t_{solo} \cdot \epsilon_{mem} \cdot \epsilon_{LLC} \cdot \sum_{rd(R_i)>rd(R)} hit(R_i))}{miss_{solo} \cdot \sum_{rd(R_i)>rd(R)} hit(R_i)} \quad (7)$$
where \( c_{mem} \) and \( c_{LLC} \) are the average access latencies for memory and LLC, respectively, given in Table 1.

**Modeling Hit Count \( \text{hit}(R) \).** Theoretically, we can estimate \( \text{hit}(R) \) as \( n(R) \times r_c(R) \), at some loss of accuracy. In this paper, however, we use the profiling data of \( T_A \) to improve accuracy, by distributing the cache hits for all the reuse patterns proportionally according to their reuse counts:

\[
\text{hit}(R) = \frac{n(R) \times r_c(R)}{\sum_i n(R_i) \times r_c(R_j)} \times \text{hit}_{\text{so}a} 
\]

(8)

**Adjusting for Real Machines.** Examining (2), we estimate \( \Delta(R) \) by using a cache with a capacity \( C \) and a growth factor of \( R \)'s cache miss count as a step function in (3). On a real machine, there are two differences. First, the cache is not fully associative, reducing the effective cache size available. Second, the cache is not completely LRU, rendering the growth factor not to be a step but continuous.

Below, we adjust our model accordingly. First, we adopt the following effective cache size, following [36]:

\[
eC = \frac{C}{\#\text{ways} \times (2 + 2)}
\]

(9)

where \#\text{ways} represents the cache associativity.

Second, we adjust the growth function into a continuous function. By (2) – (7), we obtain the following upper bound \( p^* \) for the pressure exerted to \( R \) by \( F \):

\[
p^* = \max(0, (C - rd(R))/t_R)
\]

To make adjustments on a real machine, we use the effective cache size \( eC \) to replace \( C \) in (10) to obtain a lower bound \( p^* \) for the pressure exerted to \( R \) by \( F \). We leverage a \( \text{tanh} \) function to connect \((p^*, 0)\) and \((p^*, \text{hit}(R))\) smoothly over the range \([p^*, p^*]\):

\[
\Delta(R) = \text{hit}(R) \times \frac{1}{2} \left( \text{tanh}(p - \frac{p^* + p^*}{2}) \times \frac{6}{p^* - p^*} + 1 \right)
\]

(11)

Specifically, we obtain this continuous function in three steps. First, we change the output of \( \text{tanh} \) from \((-1, 1)\) to \((0, \text{hit}(R))\) by applying a vertical shift (factor 1) and a vertical stretch (factor \( \frac{1}{2} \)). Second, we transform its symmetric point from \((0, \text{hit}(R))/2\) to \((p^* + p^*)/2, \text{hit}(R))/2\) by performing a horizontal stretch (factor \( p^* + p^* \)) to \((p^* + p^*)/2, \text{hit}(R))/2\). Finally, we perform a horizontal stretch (factor \( 6/(p^* - p^*) \)) in order to obtain 0 when the pressure is \( p^* \) and \( \text{hit}(R) \) when the pressure is at \( p^* \).

**Predicting Cache Misses for One Reuse Pattern.** We can obtain our cache miss prediction model for a given reuse pattern \( R \) under a given tile size (not shown explicitly) when \( T_A \) co-runs with a cache flusher \( F \):

\[
\Delta(R) = \left\{
\begin{array}{ll}
0, & p < p^*
\\
\text{hit}(R) \times \frac{1}{2} \left( \text{tanh}(p - \frac{p^* + p^*}{2}) \times \frac{6}{p^* - p^*} + 1 \right), & p \in [p^*, p^*]
\\
p^* = \max(0, (C - rd(R))/t_R)
\\
\end{array}
\right.
\]

(12)

Here, \( t_R \) can be estimated by (5) if \( R \) is a reuse pattern with the largest reuse distance and (7) otherwise. In addition, \( \text{hit}(R_i) \) can be estimated by (8).

### 3.2.3 Obtaining \( M(T_A) \) for All Reuse Patterns

Based on (12), we can obtain the additional cache misses introduced by all the reuse patterns of \( T_A \) in the presence of a co-located application \( F \) in decreasing order of their reuse distances. Summarizing over all these reuse patterns, we obtain \( M(T_A) \), a peer-aware TSS model, for estimating the total number of cache misses for \( T_A \) when \( T_A \) co-runs with \( F \):

\[
\text{miss}(T_A) = \sum_i \Delta(R_i) + \text{miss}_{\text{so}a}(T_A)
\]

(13)

### 3.3 Verifying the Peer-Aware TSS Model

We use GEMM tiled with \( OB = 616 \) to verify our peer-aware TSS model, as shown in Figure 7.

**Figure 6: Rendering \( \Delta(R) \) for real machines.**

Figure 6 illustrates \( p^* \) and \( p^* \) for the rendered function \( \Delta(R) \). In \([p^*, p^*]\), the dotted blue line shows the ideal cache miss count for \( R \) and the solid blue line shows the estimated behavior by the rendered function.

**Figure 7: Model verification for GEMM (OB = 616).**

The solid red line shows the predicted cache miss count, by applying (12) to \( R_C, R_B \) and \( R_A \) successively. In particular, the three dotted boxes highlight the models obtained for the three patterns. The dotted blue line gives the measured cache miss count when GEMM co-runs with different workloads, with each workload consisting of up to 5 distinct STREAM kernels.

### 4 PEER-AWARE TILING FRAMEWORK

Based on our peer-aware TSS model, we can apply peer-aware tiling to a matrix-based application in the presence of co-located applications. In datacenters, DNN-based matrix computations iteratively apply a series of matrix computations, with each matrix computation being not extremely large. There are no performance
benefits to dynamically change their tile sizes during program execution. Therefore, our peer-aware loop tiling framework determines the tile size for a loop nest in real time just before it starts its execution.

The cache pressure detector is implemented as a helper program to communicate with a tiled application via a socket. In particular, the detector serves as the socket server, waiting for messages from the application. We leverage the profiling technology commonly deployed in modern datacenters [34], which continuously profiles the whole system and applies performance analysis, with negligible overhead. Our detector periodically reads the performance counter L2LinesIn for each co-runner and computes the L2LineslnRate for a preset interval, i.e., 1 second. When a request get_required_ts() is inserted to select the optimal tile size for a loop nest T_A, according to its user annotation. There are two cases. Let TS be the set of candidate tile sizes. If T_A is annotated as //@peer-aware(efficiency)*/, we optimize the performance of T_A by selecting the tile size in TS that minimizes its total cache miss count miss(T_A) in (13). If T_A is annotated as //@peer-aware(niceness)*/, we optimize the performance of T_A’s co-runners by selecting the tile size in TS that minimizes

\[ \text{miss}_{\text{seq}}(T_A) = \text{miss}(T_A) / t_{\text{std-time}} \]

where \( t_{\text{std-time}} = t_{\text{elo}} + (C_{\text{mem}} - C_{\text{LLC}}) \times \text{miss}(T_A) \) derived based on (6). Intuitively, this leads to the niceness of T_A towards its co-runners as the overall memory bandwidth tends to be minimized [41].

5 EVALUATION

The objective is to show that peer-aware tiling is more effective than static tiling in guiding (1) compilers to optimize the performance of a tiled matrix-based application or its co-runners and (2) library developers to optimize the performance of library routines.

Platforms. The main platform used is a two-socket server, with each socket containing a Westmere-based Intel 2.40GHz six-core Xeon E5645 with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, and a shared 12MB L3 cache. To measure the effectiveness of our approach accurately, we disable SMT (Simultaneous Multithreading), DVFS (Dynamic Voltage and Frequency Scaling) and hardware prefetchers. Another Intel platform is also discussed briefly.

Benchmarks. The three matrix-based applications are selected from Pluto [4] and Caffe [23]: matmul (GEMM), tmm (triangular matrix multiplication), and conv (a kernel used in a convolutional layer of DNNs). For these applications, we have considered various matrix sizes ranging from 2000 to 5000 and obtain similar results. We report our results by choosing 3300 as a representative size.

Figure 8 depicts the framework. The static component applies parametric tiling and inserts instrumentation code for a tiled application, enabling the dynamic component to detect the cache pressure exerted to the application by its co-runners and determine the optimal tile size at runtime based our peer-aware TSS model. In particular, we use PrimeTile [18] to tile loops with parametric tile sizes (rather than compile-time constants).

The following source-code annotation is provided:

//@peer-aware(efficiency/niceness)*/

Users can annotate a loop nest that need to be tiled in a peer-aware manner, with the parameter specifying an objective to be optimized. Our framework inserts instrumentation code for each annotated loop nest to serve two purposes at runtime: obtaining the co-runners’ cache pressure from the runtime detector and determining the optimal tile size according to the peer-aware TSS model. Figure 9 is the source code after parametric tiling and instrumentation for GEMM.

The cache pressure detector is implemented as a helper process to communicate with a tiled application via a socket. In particular, the detector serves as the socket server, waiting for messages from the application. We leverage the profiling technology commonly deployed in modern datacenters [34], which continuously profiles the whole system and applies performance analysis, with negligible overhead. Our detector periodically reads the performance counter L2LinesIn for each co-runner and computes the L2LineslnRate for a preset interval, i.e., 1 second. When a request get_required_ts() is inserted to select the optimal tile size for a loop nest T_A, according to its user annotation. There are two cases. Let TS be the set of candidate tile sizes. If T_A is annotated as //@peer-aware(efficiency)*/, we optimize the performance of T_A by selecting the tile size in TS that minimizes its total cache miss count miss(T_A) in (13). If T_A is annotated as //@peer-aware(niceness)*/, we optimize the performance of T_A’s co-runners by selecting the tile size in TS that minimizes

\[ \text{miss}_{\text{seq}}(T_A) = \text{miss}(T_A) / t_{\text{std-time}} \]

where \( t_{\text{std-time}} = t_{\text{elo}} + (C_{\text{mem}} - C_{\text{LLC}}) \times \text{miss}(T_A) \) derived based on (6). Intuitively, this leads to the niceness of T_A towards its co-runners as the overall memory bandwidth tends to be minimized [41].

5 EVALUATION

The objective is to show that peer-aware tiling is more effective than static tiling in guiding (1) compilers to optimize the performance of a tiled matrix-based application or its co-runners and (2) library developers to optimize the performance of library routines.

Platforms. The main platform used is a two-socket server, with each socket containing a Westmere-based Intel 2.40GHz six-core Xeon E5645 with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, and a shared 12MB L3 cache. To measure the effectiveness of our approach accurately, we disable SMT (Simultaneous Multithreading), DVFS (Dynamic Voltage and Frequency Scaling) and hardware prefetchers. Another Intel platform is also discussed briefly.

Benchmarks. The three matrix-based applications are selected from Pluto [4] and Caffe [23]: matmul (GEMM), tmm (triangular matrix multiplication), and conv (a kernel used in a convolutional layer of DNNs). For these applications, we have considered various matrix sizes ranging from 2000 to 5000 and obtain similar results. We report our results by choosing 3300 as a representative size.

Figure 8 depicts the framework. The static component applies parametric tiling and inserts instrumentation code for a tiled application, enabling the dynamic component to detect the cache pressure exerted to the application by its co-runners and determine the optimal tile size at runtime based our peer-aware TSS model. In particular, we use PrimeTile [18] to tile loops with parametric tile sizes (rather than compile-time constants).

The following source-code annotation is provided:

//@peer-aware(efficiency/niceness)*/

Users can annotate a loop nest that need to be tiled in a peer-aware manner, with the parameter specifying an objective to be optimized. Our framework inserts instrumentation code for each annotated loop nest to serve two purposes at runtime: obtaining the co-runners’ cache pressure from the runtime detector and determining the optimal tile size according to the peer-aware TSS model. Figure 9 is the source code after parametric tiling and instrumentation for GEMM.

The cache pressure detector is implemented as a helper process to communicate with a tiled application via a socket. In particular, the detector serves as the socket server, waiting for messages from the application. We leverage the profiling technology commonly deployed in modern datacenters [34], which continuously profiles the whole system and applies performance analysis, with negligible overhead. Our detector periodically reads the performance counter L2LinesIn for each co-runner and computes the L2LineslnRate for a preset interval, i.e., 1 second. When a request get_required_ts() is inserted to select the optimal tile size for a loop nest T_A, according to its user annotation. There are two cases. Let TS be the set of candidate tile sizes. If T_A is annotated as //@peer-aware(efficiency)*/, we optimize the performance of T_A by selecting the tile size in TS that minimizes its total cache miss count miss(T_A) in (13). If T_A is annotated as //@peer-aware(niceness)*/, we optimize the performance of T_A’s co-runners by selecting the tile size in TS that minimizes

\[ \text{miss}_{\text{seq}}(T_A) = \text{miss}(T_A) / t_{\text{std-time}} \]

where \( t_{\text{std-time}} = t_{\text{elo}} + (C_{\text{mem}} - C_{\text{LLC}}) \times \text{miss}(T_A) \) derived based on (6). Intuitively, this leads to the niceness of T_A towards its co-runners as the overall memory bandwidth tends to be minimized [41].

5 EVALUATION

The objective is to show that peer-aware tiling is more effective than static tiling in guiding (1) compilers to optimize the performance of a tiled matrix-based application or its co-runners and (2) library developers to optimize the performance of library routines.

Platforms. The main platform used is a two-socket server, with each socket containing a Westmere-based Intel 2.40GHz six-core Xeon E5645 with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, and a shared 12MB L3 cache. To measure the effectiveness of our approach accurately, we disable SMT (Simultaneous Multithreading), DVFS (Dynamic Voltage and Frequency Scaling) and hardware prefetchers. Another Intel platform is also discussed briefly.

Benchmarks. The three matrix-based applications are selected from Pluto [4] and Caffe [23]: matmul (GEMM), tmm (triangular matrix multiplication), and conv (a kernel used in a convolutional layer of DNNs). For these applications, we have considered various matrix sizes ranging from 2000 to 5000 and obtain similar results. We report our results by choosing 3300 as a representative size.

For conv, we take the fourth convolutional layer from AlexNet [25] as conv4. The number of images is 32, the number of input channels is 384, the number of filters is 1024, the size of an input image is 13 + 13, and the size of the convolution kernel is 3 + 3.

The co-runners for these matrix computations are randomly generated workloads from SPEC CPU 2006 and STREAM [30], with each workload containing up to 5 applications. In addition, we also use a widely used distributed memory caching system, memcached [17], as a latency-sensitive co-running application.

Loop Tiling. For each matrix-based application, we apply two-level loop tiling, with the inner tile size fixed as 56. We did not use the default 32 from Pluto [4], because 56 leads to better performance.

For static tiling, the outer tile size is set to 840 by hand-tuning. The tiled code is generated by Pluto.

For peer-aware tiling, the outer tile size search space TS starts from 280 and ends at 840 with a step of 56. Starting with 280, we ensure that the working set still exceeds the private L2 cache so
that the shared L3 cache is used. We will not look beyond 840 since the optimal tile size in a co-location scenario should be smaller.

Finally, all programs are compiled by GCC -O3.

5.1 Optimizing Tiled Matrix Computations

We show that peer-aware tiling is more effective than static tiling in optimizing the performance of a tiled matrix-based application. We also analyze the accuracy of our peer-aware TSS model in allowing optimal or near-optimal tile sizes to be selected.

5.1.1 Overall Performance. For each tiled matrix-based application, we co-run it with 50 randomly generated workloads from SPEC2006 and STREAM [30]. Figure 10 shows the overall performance for GEMM. The horizontal axis represents the ID of a workload and the vertical axis the performance degradation when co-located, which is computed by \((T_{\text{co-run}} - T_{\text{sole}})/(T_{\text{sole}})\), where \(T_{\text{sole}}\) (\(T_{\text{co-run}}\)) is the execution time of GEMM in solo (co-run) execution. There are three bars for each workload, with the leftmost one for static tiling, the middle one for peer-aware tiling, and the rightmost one for the oracle, i.e., optimal tiling. For static tiling, the performance slowdowns range from 9.2% to 39.9% with an average of 22.9%. For peer-aware tiling, the performance slowdowns range from 8.1% to 23.5% with an average of 15.1%. Therefore, peer-aware tiling is more effective than static tiling in reducing the average co-location slowdown (i.e., interference) for GEMM by 34.1% (i.e., from 22.9% to 15.1%). Finally, peer-aware tiling is on par with oracle tiling, for which the performance slowdowns range from 7.0% to 23.5% with an average of 14.8%.

For peer-aware tiling, its performance gain comes from the reduced LLC misses, since our model can accurately predict the LLC miss count of GEMM when co-running with different workloads. This allows us to select the tile size leading to the minimal cache miss count by (13). Figure 11 shows the accuracy of our prediction for 10 representative workloads (to save space) from Figure 10, with an average error of 3.9% (across the 50 workloads).

Figure 12 (Figure 13) shows the overall performance and the prediction accuracy for tmm (conv). We again plot the results for only 10 representative workloads to save space but use the results of all the 50 workloads in computing the average performance. Compared with static tiling, peer-aware tiling reduces the average co-location performance slowdown for tmm by 40.3% (i.e., from 18.1% to 10.8%) and for conv by 36.9% (i.e., from 17.9% to 11.3%). For both applications, peer-aware tiling is on par with oracle tiling in terms of the overall performance achieved. Finally, our model predicts the LLC cache misses quite accurately, with an average error of 3.5% and 7.5% for tmm and conv, respectively.

5.1.2 Selected Optimal Tile Sizes. Figure 14 compares the predicted optimal tile sizes (represented by the thick blue lines) and real optimal ones (represented by the orange triangles) for GEMM when co-running with the same 50 workloads given in Figure 10. For the 50 co-running workloads, the predicted tile sizes are close to their corresponding real optimal ones: identical for 24 workloads, differing by 56 for 24 workloads, and differing by 112 for 2 workloads. Note that our outer tile size is an integral multiple of the inner tile size, i.e., 56, implying that the predicted and real optimal tile sizes differ by at most two step sizes.

Similarly, the predicted and real optimal tile sizes are also close for tmm and conv. For tmm, both are identical for 30 workloads, differ by 56 for 13 workloads, and differ by 112 for 7 workloads. For conv, both are identical for 26 workloads, differ by 56 for 16 workloads, differ by 112 for 7 workloads, and differ by 168 for 1 workload.

5.2 Optimizing Co-located Applications

We show that peer-aware tiling is also more effective than static tiling in optimizing the performance of the co-runners of a tiled matrix-based application. Our co-runners are instances of memcached [17], whose QoS is sensitive to cache contention [26]. To avoid interference from network I/O and focus on cache contention, memcached server and mutillate [26] load tester run on different sockets of a common node, which is similar to the case when both run on different nodes, simulating the common scenario in a data-center. Specifically, we use one socket to run the memcached server, another socket to generate the client loads for the server using mutillate and force them to only access socket-local memory resources. Two configurations are considered:

- **Conf-1.** On one socket, two cores are reserved for the memcached server with two threads and the other four cores for running GEMM (one instance per core). The mutillate load tester occupies all cores on the other socket and generates 180K requests per second (RPS).

- **Conf-2.** On one socket, four cores are reserved for the memcached server with four threads and the other two cores for running GEMM (with one instance per core). The mutillate load tester occupies all cores on the other socket and generates 360K requests per second.

5.2.1 Request Latency. For each configuration, we compare solo-run, peer-aware tiling, and static tiling in terms of the CDF (Cumulative Distribution Function) for the request latency required in each case. For solo-run, we start the memcached server to run alone and send the requests from the client. For static tiling, we co-locate memcached with GEMM with its tile size found by static tiling. For peer-aware tiling, we co-locate memcached also with GEMM with its tile size found by peer-aware tiling.

Figure 15 shows the CDF of the request latency for memcached under **Conf-1**, with the three curves corresponding to solo-run, static tiling and peer-aware tiling. For static tiling, the optimal tile size found is 840. For peer-aware tiling, the optimal (nice) tile size found for optimizing the performance of memcached is 280. For each curve, the 95th percentile request latency is annotated with a dotted line. Static tiling has increased memcached’s 95th percentile latency from 228µs to 292µs, causing a performance slowdown of 28.0% over solo-run. In contrast, peer-aware tiling has increased memcached’s 95th percentile latency from 228µs to 271µs, representing a performance slowdown of only 18.8% over solo-run. As a result, peer-aware tiling is more effective than static tiling in reducing the co-location performance slowdown, with a reduction factor of 32.9%.

Let us consider **Conf-2**. For static tiling, the optimal tile size remains to be 840. For peer-aware tiling, the optimal tile size selected is 504. Static tiling has increased memcached’s 95th percentile latency from 720µs to 823µs (a slowdown of 14.3% over solo-run). With peer-aware tiling, we have reduced the 95th percentile latency to 770µs (a slowdown of 6.9% only over solo-run), and consequently, the co-location interference by 51.7%.
5.2.2 Selected Nice Tile Sizes.

When GEMM co-runs with memcached, we select a so-called nice tile size for GEMM that minimizes \( \text{miss}_{\text{req}}(T) \) given in (14) in order to optimize the performance of memcached. GEMM is said to be nice to memcached as GEMM does not snatch the portion of the shared cache supposed to be used by memcached.

For Conf-1, the cache pressure from memcached is \( p = 0.39 \). All the tile sizes in our search space exhibit similar LLC cache miss counts. However, the tile size 280 has the lowest LLC miss frequency according to (14) and thus selected as the optimal (nice) tile size.

For Conf-2, the cache pressure from memcached is \( p = 0.73 \). The tile size 504 has the lowest LLC miss frequency by (14) and thus selected as the nice tile size.

5.3 Optimizing Library Routines

We show that peer-aware tiling is superior to static tiling in enabling library developers to optimize the performance of matrix-based library routines. Our application is ATLAS [43], for which two-level tiling is also applied. The inner tiling is 3-D (tiled for the \( i, j \) and \( k \) loops with a tile size of 60 \( \times \) 60 \( \times \) 60) and the outer tiling is 1-D (tiled for the \( k \) loop with a tile size of 480).
We co-run ATLAS with 10 randomly generated workloads, again from SPEC2006 and STREAM. Figure 16 compares the three approaches, identified by ATLAS, static tiling, and peer-aware tiling (all normalized to the performance of ATLAS in solo-run).

![Figure 16: Performance of ATLAS, static tiling and peer-aware tiling (normalized to ATLAS in its solo-run).](image)

ATLAS represents the case when it runs in a co-location scenario. For static tiling, we have applied 3-D outer tiling to ATLAS and selected the (same fixed) optimal tile size $960 \times 960 \times 960$ (for all workloads) by hand-tuning. Static tiling outperforms ATLAS in every co-location scenario evaluated. For peer-aware tiling, the same 3-D outer tiling is applied except that the optimal tile sizes for different co-running workloads are selected at runtime by our peer-aware TSS model for the purposes of optimizing the performance of ATLAS. On average, peer-aware tiling outperforms ATLAS and static tiling by 8.9% and 7.1%, respectively.

### 5.4 Accuracy of Peer-Aware TSS

We use GEMM to demonstrate the accuracy of our peer-aware TSS model. We focus on optimizing the performance of GEMM. The case for optimizing the performance of its co-runners is analyzed similarly.

Figure 17 displays the curves of the predicted and real cache miss counts under different cache pressures for three tile sizes, 448, 616, and 840. We have omitted the other tile sizes in order to avoid cluttering. A total of 50 STREAMING workloads are used. For each tile size, the predicted curve is calculated by using (13) for each workload while the real curve is generated by co-running GEMM with each workload. Both curves are close under different cache pressures.

One interesting observation is that GEMM’s optimal tile size is a non-monotone function of the cache pressure exerted by its co-runners. This is because the co-running cache behavior of GEMM is the aggregation of the co-running cache behaviors of its reuse patterns.

Consider Figure 17 again. Let $p$ be the the cache pressure from the co-runners of GEMM. If $p < 0.01$, the tile size 840 introduces the minimal cache misses according to (13) and is thus optimal. However, if $p$ lies in $[0.01, 0.1]$, the tile size 840 will introduce the maximal cache misses, causing GEMM to exhibit the worst performance. If $p$ lies in $[0.1, 0.2]$, the tile size 840 is neither the best nor the worst. If $p$ lies in $[0.22, 0.53]$, the tile size 840 introduces the minimal cache misses again according to (13) and is thus optimal. Finally, if $p > 0.53$, the tile size 840 is the worst again.

![Figure 17: Predicted and real LLC misses for 3 tile sizes.](image)

### 5.5 Evaluation on a Second Platform

We discuss briefly our evaluation on an Intel 2.00 GHz Sandy Bridge-based six-core Xeon E5-2620 with a private 32KB L1 D-cache, a private 32KB L1 I-cache, a private 256KB L2 cache, and a shared 15MB L3 cache.

We focus only on Gemm for this platform. Figure 18 (for Sandy Bridge) is an analogue of Figure 10 (for Westmere). As the shared L3 cache is larger this time, the performance slowdowns for all the three approaches are smaller. However, peer-aware tiling, which is on par with oracle tiling (“Optimal”), remains more effective than static tiling in reducing the average performance slowdown by 43.3% (i.e., from 12.0% to 6.8%).

Figure 19 is an analogue of Figure 11 for GEMM, with an average error of 3.2% (across the 50 workloads).

### 6 RELATED WORK

There has been a lot of work on applying loop tiling to improve locality and parallelism [3–5, 9, 31, 32, 44, 46–50]. Earlier, compiler researchers rely on analytical models for tile size selection by characterizing the performance of a tiled loop nest as a function of its tile size.

Recently, model-driven empirical search methods are often used for tile size selection. In [14, 43], empirical tile size selection is applied to ATLAS, which generates multiple versions of the tiled loop nest, runs them on the actual platforms, and automatically selects the optimal tile size on the platforms [14, 43]. In [52], analytical modeling and empirical searching are compared, and observed that analytical models can achieve comparable performance to that of code generated by empirical optimizers [52]. In [8, 53], analytical models are used to prune the search space for empirical optimization and guide their empirical tile size selection. Nowadays, some machine learning techniques are considered to select tile sizes [28, 33, 54]. Moreover, pattern-based compilation methods [10–12] are also proposed to conduct semantic-specific optimizations.

On multicore processors, researchers have studied the cache behavior of co-running applications and applied cache partitioning to partition the shared cache among such co-runners. Chandra et al. [6] investigate the impact of cache sharing on simultaneously running threads. Mars et al. [29, 51] and Zhao et al. [56, 57] concentrate on quantitative analysis of performance interference incurred by resource contention in memory subsystem. Stone et al. [38] consider how to partition the shared cache among the competing processes. Zhang et al. [27] propose a cache partitioning strategy...
implemented in operation systems. Chang et al. [7] apply cooperative cache partitioning to adapt multiple time-sharing partitions among simultaneously running threads. Suh et al. [40] propose a dynamic cache partitioning approach for any partition granularity. Some researchers also estimate the effects of cache interference on application performance at runtime [15, 16, 39, 45]. He et al. [21] proposed an approach to estimating application slowdowns due to contention between heterogeneous platforms, such as CPU and GPU.

In recent years, parameterized loop tiling has also been used to facilitate iterative compilation and auto-tuning. Renganarayan et al. [35] introduce a parameterized tiled loop generator for perfectly nested loops. Kim et al. [24] extends it to support multi-level tiling. Hartono et al. introduce PrimeTile to generate parameterized sequential tiled code for imperfectly nested loops [18] and DynTile to generate parameterized tiled code for parallel execution [19]. Baskaran et al. [2] present PTile for wavefront parallel tiled execution.

Nowadays, researchers have paid increasing attention to compilation technologies in co-running environments, from two aspects: contentiousness and sensitivity. For contentiousness, Tang et al. introduce a static transformation to throttle down the memory access rate of the contentious regions in low priority applications [41] and a static/dynamic compilation approach to adaptively manipulating the contentiousness at runtime [42].

In the case of sensitivity, Bao and Ding [1] apply defensive tiling to private caches, by performing tile size selection under a given defensiveness, in order to obtain robust performance in co-running environments. Jain et al. [22] propose a continuous shape shifting framework, ShapeShifter, to reshape iteration spaces and pinpoint near-optimal loop tiling configurations. Srinivas et al. [37] introduce reactive loop tiling, which generates multiple tiled code versions with different tile sizes for a given loop nest and dynamically selects the best version at runtime. ShapeShifter and reactive loop tiling are the closest related but orthogonal to our work. These two earlier frameworks focus on code rewriting at runtime according to the monitored co-runners. In contrast, our work focuses on building a peer-aware TSS model analytically by taking a reuse-pattern-centric approach. Furthermore, our model can be used to guide compilers to optimize not only a tiled application but also its co-runners and library developers to optimize the performance of matrix-based library routines.

7 CONCLUSION

We have introduced a reuse-pattern-centric approach for quantifying the co-running cache behavior of a matrix-based application, allowing us to develop a peer-aware TSS model for the application without the need to analyze its co-runners. Based on this model, we have developed a peer-aware loop tiling approach that can guide compilers to optimize the performance of a tiled matrix-based application or its co-runners. In addition, peer-aware tiling also allows library developers to optimize the performance of matrix-based library routines.

This work has opened up a few opportunities for future research on improving the performance of co-located applications in datacenters. As one future work, we will apply our approach to large-scale DNN applications, including training and inference. In addition, we will also study how to integrate an OS/hardware cache partitioning mechanism with peer-aware tiling.

ACKNOWLEDGMENTS

This work is supported in part by the National Key R&D Program of China (2016YFB1000201 and 2016YFB1000402), the National Natural Science Foundation of China (61521092, 61432016, 61432018, 61332009) and an Australian Research Council grant (RG171010). The authors would like to thank all the anonymous reviewers for their valuable comments and helpful suggestions.