TurboTiling: Leveraging Prefetching to Boost Performance of Tiled Codes

Sanyam Mehta*
Department of Computer Science and Engineering
University of Minnesota
Twin Cities, MN USA
sanyam.mehta@gmail.com

Rajat Garg
Department of Computer Science and Engineering
University of Minnesota
Twin Cities, MN USA
gargx093@umn.edu

Nishad Trivedi
Department of Computer Science and Engineering
University of Minnesota
Twin Cities, MN USA
triv0025@umn.edu

Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota
Twin Cities, MN USA
yew@cs.umn.edu

ABSTRACT

Loop tiling or blocking improves temporal locality by dividing the problem domain into tiles and then repeatedly accessing the data within a tile. While this reduces reuse, it also leads to an often ignored side-effect: breaking the streaming data access pattern. As a result, tiled codes are unable to exploit the sophisticated hardware prefetchers in present-day processors to extract extra performance.

In this work, we propose a tiling algorithm to leverage prefetching to boost the performance of tiled codes. To achieve this, we propose to tile for the last-level cache as opposed to tiling for higher levels of cache as generally recommended. This approach not only exposes streaming access patterns in the tiled code that are amenable for prefetching, but also allows for a reduction in the off-chip traffic to memory (and therefore, better scaling with the number of cores). As a result, although we tile for the last level cache, we effectively access the data in the higher levels of cache because the data is prefetched in time for computation. To achieve this, we propose an algorithm to select a tile size that aims to maximize data reuse and minimize conflict misses in the shared last-level cache in modern multi-core processors. We find that the combined effect of tiling for the last-level cache and effective hardware prefetching gives significant improvement over existing tiling algorithms that target higher level L1/L2 caches and do not leverage the hardware prefetchers. When run on an Intel 8-core machine using different problem sizes, it achieves an average improvement of 27% and 48% for smaller and larger problem sizes, respectively, over the best tile sizes selected by state-of-the-art algorithms.

*This author is now at Cray Inc.

ICS '16, June 01-June 03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926288

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Optimization

Keywords

Loop tiling; Prefetching; Multi-core

1. INTRODUCTION

Loop tiling [40, 22] is a well known compiler optimization that allows better data reuse in higher levels of the memory hierarchy. In order to achieve this reuse, the tiled code re-iterates over the data in a single tile (or block) before the program execution moves on to the next tile. As a result, loop tiling may disrupt the streaming data access patterns suitable for hardware prefetchers. The tiled code thus does not benefit as much from the powerful hardware prefetchers at different levels of cache in modern processors when it does not take data prefetching into consideration.

For any tiled code, selection of an adequate tile size is a complex, but crucial task. A good tile size ensures that reusable data is not evicted from the cache due to capacity or conflict misses. Traditionally, tile size selection algorithms [26, 11, 9] only target data reuse in the higher-level caches such as L1 cache. Recently, *tss* [23] also considered cache set-associativity and the interaction with the SIMD unit, but still targets L1 or L2 cache. As a result of targeting higher levels of cache, the existing approaches achieve limited data reuse, resulting in a large number of costly off-chip memory accesses. Upon parallelization on multiple cores, the problem is exacerbated because each core is now competing for the scarce off-chip memory bandwidth.

In this paper, we propose a novel tile size selection algorithm that overcomes the aforementioned drawbacks. Our algorithm TurboTile Selection (TTS) chooses tile sizes by targeting data reuse primarily in the last level cache (LLC) which is the largest cache on modern processors. This approach greatly reduces the off-chip memory accesses. More importantly, tiling for LLC enables the selection of larger tile sizes that can include many contiguous cache lines. This facilitates streaming access patterns that are suitable for hardware prefetchers. The tiled code thus leverages the hardware prefetchers at different levels of cache to bring data all the way to

the L1 cache. This compensates for the potential problem of a high miss rate incurred due to reduced data reuse in the L1 cache. In addition, the tiled code with large tiles leads to better vectorization and less pronounced conflict misses in the cache. Our experimental results show that the transformed program, although tiled for the LLC, gives a higher performance than the program tiled for higher-level caches on modern processors with data prefetchers.

Tiling for data reuse in LLC leads to other important advantages. For example, as the data reuse reduces the number of off-chip memory accesses, the tiled code scales better when more on-chip cores are used. In addition, the LLC is often large enough (up to 20MB on Intel Ivy Bridge) to hold multiple rows of an array in a tile. Therefore, the code can be left untiled in one dimension (except for very large problem sizes). The problem of tile size selection is thus reduced to choosing tile factors in 2 dimensions instead of 3 dimensions.

The resultant tile selection algorithm is also much simpler and, most importantly, more adaptable to different host processors and problem sizes.

In summary, our work makes the following contributions.

1. We propose an effective algorithm for tile size selection that takes hardware data prefetchers into consideration. It primarily targets data reuse in the LLC. Performance improvement primarily stems from (1) reducing costly off-chip memory accesses due to tiling for the LLC and (2) leveraging hardware data prefetchers to make the data available in the L1 cache even though the code is tiled for the LLC.

2. We present an analytical model that estimates the reduction in the total cache misses (TCM) in the LLC and the resulting reduction in total off-chip memory accesses.

3. We also present an analytical model for effective load balancing across different on-chip cores in a multi-core environment. This model helps to decide whether to tile for the LLC or higher-level caches for a given problem size.

4. The resultant tiled code is thus faster and more scalable than those produced by other state-of-the-art algorithms that have not considered data prefetching. Also, resultant alleviation of self-interference misses leads to a simpler and more robust tile size selection algorithm.

5. Our algorithm, Turbo-Tile Selection, is implemented in the PLuTo polyhedral compiler. In a multi-core scenario, it shows 27% and 48% performance improvements for smaller and larger problem sizes, respectively, over the best tile size selected by the state-of-the-art algorithm (tss) [23] that tiles for smaller L1-L2 caches.

Although we propose an analytical model for choosing suitable tile sizes, our model can also be used to prune the search space for auto-tuning frameworks.

The rest of the paper is organized as follows. Section 2 motivates the problem by using Intel's performance monitoring tool, VTune [30], to study the effect of hardware prefetching on tiling targeting the last level cache. Section 3 provides relevant background on tiling and hardware prefetching. Section 4 discusses the multiple factors that determine effective tile sizes. Our algorithm, TTS, and its applicability is discussed in Section 5. In Section 6, we describe our experimental setup, and present and discuss our results. Section 7 sheds more light on the basic premise of TTS and its usability. Section 8 discusses related work, and the conclusions from the work are presented in Section 9.

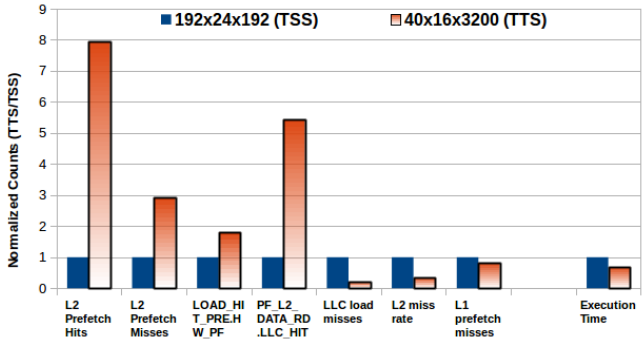

Figure 1: Normalized values for tiled *dsyrk* that targets L3 cache (tile size is 40x16x3200). The baseline is the best L1-L2 tiled code by tss (tile size is 192x24x192). Problem size is 3200.

2. MOTIVATION

In this section, we show that tiling for LLC not only gives an opportunity for exploiting streaming access patterns through hardware prefetchers, but also reduces the number of off-chip memory accesses. For this purpose, we use performance counters in VTune to study two different 3D tile sizes for a Level-3 BLAS [15] kernel, *dsyrk*, with problem size of 3200. These measurements were carried out on an Intel Xeon processor (E5-2650 v2, 2.60GHz) with eight Ivy Bridge cores. The cores have a 32KB private L1 cache, 256KB private L2 cache, and 20MB shared L3 cache. The processor is equipped with an effective mid-level cache (MLC) streamer hardware prefetcher that prefetches data to L2 cache from L3 cache/memory, and a Data Cache Unit (DCU) prefetcher at L1 cache, which prefetches data from L2 cache to L1 cache. As a result, the data could potentially be brought from the L3 (LLC) all the way to the L1 cache through coordinated prefetching at the L1 and L2 caches, effectively hiding L3 latency. The results of the experiment are shown in Figure 1.

In Figure 1, the first tile size chosen (192x24x192) is the best tile size given by tss, which aims at making effective data reuse in L1 and L2 caches. The second tile size (40x16x3200) is chosen to exploit data reuse in L2 and L3 caches. Note that the innermost tile factor is equal to the problem size of 3200. We choose this tile factor to expose streaming access patterns of array references to hardware prefetchers.

In Figure 1, seven different performance counters (related to the caches) and the overall execution times are used as the x-axis for the two tiled codes. On the y-axis, we plot the normalized values (TTS/tss) of these counters.

We make the following observations from the figure.

1. The first two counters reveal the *prefetch hits* and *prefetch misses* for MLC hardware prefetcher. From Figure 1, we observe that the prefetching activity of the MLC hardware prefetcher increases drastically in the code tiled for L3 cache as compared to the code tiled for L1-L2 caches.

2. The third counter *LOAD_HIT_PRE.HW_PF* shows the effectiveness of the DCU prefetcher at L1 cache. It counts the *number of load requests made at L1 cache* that incur a hit on one of the cache lines prefetched by the DCU prefetcher. An increase in hits for the tiled code is because of the timely prefetching of data from L2 cache to L1 cache. This is also supported by the reduced *L1 prefetch misses* (seventh counter), which validates that the data indeed enjoys L1 accesses.

3. The fourth counter *PF_L2_DATA_RD.LLC_HIT* in Figure 1 shows how often *prefetch read requests* from MLC hardware prefetcher to L3 cache result in hits. The upward trend in the hits reflects that if we tile for L3 cache, data can be quickly prefetched to L2 cache because most of it is already in L3 cache. This is further supported by reduced *L2 miss rate* (sixth counter).

4. The fifth counter shows the *LLC load misses*. A decrease in the LLC misses reflects the reduced number of off-chip memory accesses. Thus, contention for *off-chip memory bandwidth* can also be reduced when we tile for L3 cache in a multi-core environment.

5. Lastly and most importantly, the reduced execution time (last counter) of the tiled code on 8 cores proves that our approach of tiling for L3 cache and exploiting streaming access patterns through hardware prefetchers is quite effective.

In the case of tiling for L3 cache, observations (2) and (3) prove that hardware prefetchers (both the MLC prefetcher and the DCU prefetcher) indeed help significantly in bringing data from L3 cache to L2 cache and subsequently to L1 cache. Further, observation (4) confirms the reduced conflict misses in the form of reduced off-chip memory accesses. Thus, the program code enjoys data reuse in L1 cache, and the code tiled for L3 cache runs faster than the code tiled for L1-L2 caches. In order to confirm the role of prefetcher, we conducted another experiment where we turned the L1 and L2 hardware prefetchers off, and we observe that while performance of tss worsens by only 1%, the performance of TTS worsens by 10%. Since there was no change in the two tests apart from the prefetcher, this corroborates the benefit to TTS from better prefetching.

3. BACKGROUND

In this section, we explain some of the concepts that are relevant to our work on the interaction between hardware prefetching and tiling.

In the 3D tiled *matmul* code of Figure 2 (b), the innermost i, k, j loops (note that loops j and k are inter-changed to support vectorization) correspond to *intra*-tile loops (shown in black), which iterate through all the elements of a given tile. The outermost iT, jT and kT loops correspond to *inter*-tile loops (shown in red), which iterate from one tile to another. This terminology will be used in the subsequent sections.

3.1 Data Reuse and Tiling

Data reuse and *data locality* are two main concepts that are closely related to the purpose of tiling an array for a particular level of cache. The tile size should be chosen such that a *cache line* can be reused in successive iterations before being evicted from the cache. There are two types of data reuse that are important for tile size selection:

1. **Self-Temporal reuse**: This type of data reuse exists when the *same* array references are reused in different iterations of a loop. For example, in Figure 2 (c), one tile-row of array C[i][j] has self-temporal reuse in *intra*-tile loop k. Similarly, for *inter*-tile loop kT, the entire tile of array C has self-temporal reuse, and the entire tile of array B has self-temporal reuse in *intra*-tile loop i.

2. **Self-Spatial reuse**: This constitutes the reuse happening on adjacent data elements in the *same cache line* between different iterations of a loop. For example, in the innermost *intra*-tile loop j, arrays C and B have self-spatial reuse.

Loop tiling is aimed at improving the *self-temporal reuse* of data. It decreases the *reuse distance* of data (i.e. the period between two accesses of the same data) before the data is evicted from the cache due to capacity or conflict misses. In other words, tile sizes have to be chosen while keeping in mind the structure of caches, so as to minimize the total number of capacity and conflict misses.

Capacity misses occur when the working set size is larger than the cache size. For example, in the case of the Turbo-Tiled *matmul* in Figure 2 (c), for *intra*-tile loop i, its working set will consist of one whole tile of array B, a single row each of array C and A. Thus, working set size in i loop is (K*N+K+N), which is dominated by the elements of array B (K*N elements). Most capacity misses can be avoided if we can select a tile size that keeps the working set size (i.e. K*N+K+N) to be smaller than the effective cache size.

Conflict misses occur when multiple data items map to the same set that exceed the number of ways (set-associativity) in the set, resulting in collisions. This can take place when there are non-contiguous memory accesses of too many different rows within a large tile. Conflict misses are further classified as *self-interference* and *cross-interference*.

Self-interference misses occur when the elements of the *same* array cause collisions. Self-interference misses are pronounced for pathological problem sizes (a power of 2). These occur because the cache size is also a power of 2, thereby increasing the probability of 2 contiguous rows of an array mapping to the same set. Thus, self-interference misses are pronounced in case of pathological problem sizes. On the other hand, *cross-interference* misses occur when elements of different arrays cause collisions in the cache.

3.2 Hardware Prefetching on modern multi-core processors

Hardware prefetchers analyze the patterns of memory accesses and speculatively load data into the cache for future memory accesses. These data access patterns are classified in two broad categories: *streaming* and *striding*. Modern processors employ multiple streaming and striding hardware prefetchers sitting at different levels of caches to bring data from the next level of memory hierarchy. On Ivy Bridge, the streamer hardware prefetcher sitting on the L1 cache can prefetch only 1 cache line ahead while the more powerful MLC streamer at the L2 cache can prefetch up to 20 cache lines ahead with a prefetch degree of 2 (two lines at a time).

4. DETAILS OF THE APPROACH

To explain our approach to effective tile size selection, we use the 3D-tiled *matmul* code of Figure 2 (b) as an example. *Although we demonstrate our analysis for matmul kernel, our analysis holds true for other kernels which have reuse in one of the loop nests (even for many array references). Thus, we use the conclusions drawn out of the analysis to implement the Turbo-Tile Selection (TTS) algorithm in Section 5.*

In the *matmul* code of Figure 2 (b), I, J, K are the *tile factors* for *inter*-tile loops iT, jT and kT respectively; i, j, k are the loop indices for the corresponding *intra*-tile loops. We can further observe that each tile of array B is *reusable in consecutive iterations of loop i*, and each tile of array C is *reusable in consecutive iterations of loop kT*. In TTS, instead of reusing these tiles of arrays B and C in higher level caches (L1, L2), we propose to reuse them in lower-level caches (L2, L3). This allows us to choose larger tile sizes, which helps in reducing the off-chip memory accesses as explained in Sections 4.1 and 4.2.

Furthermore, since the outermost *inter*-tile iT loop (Figure 2 (b)) is also the parallel loop, its iterations can be scheduled among multiple cores. Our technique presents an analytical model for effec-

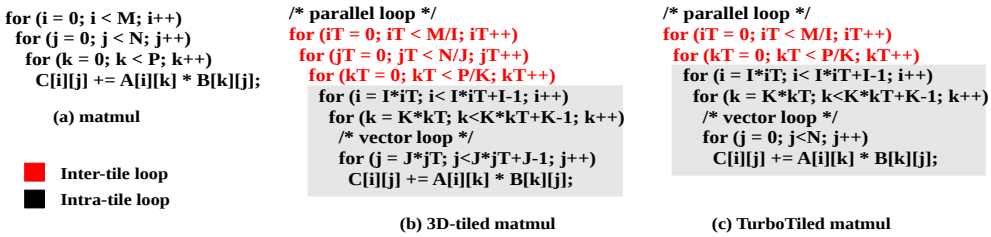

Figure 2: Source code of matrix multiplication (*matmul*), before and after 3D tiling. *Inter*-tile loops step across different tiles of the arrays. *Intra*-tile loops step across different array elements in a tile.

tive load balancing across different on-chip cores (which share L3 cache). This model is presented in Section 4.3.

We apply our approach to iterative stencils as well. Iterative stencils benefit from 2D tiling as tiling enables the code to fully exploit group reuse in larger caches. *Group reuse* takes place when different array references (group) access the same memory locations. Factors that determine the selection of effective tile sizes for iterative stencils are presented in Section 4.4.

4.1 Data Reuse in LLC

To facilitate our tile size selection strategy, we use the metric *Total Cache Misses* (TCM). TCM is defined as the total number of misses that include cold, conflict and capacity misses incurred in a particular level of cache. However, to simplify our estimation and to facilitate our analysis, we assume that data elements, once brought into the cache, will not be evicted by accesses of other arrays and data. That is, there will only be negligible conflict or capacity misses. Cold misses are the most dominant component of TCM. Our experimental results show that this assumption is reasonable. This is because our tiling algorithm aims to select the appropriate tile sizes, so that their corresponding working set can fit in a particular level of cache to avoid capacity misses. Our algorithm also takes the set-associativity into account to minimize all possible conflict misses. All these considerations effectively reduce most of the conflict and capacity misses. Hence, our main focus in estimating TCM will only be on cold misses. They provide an excellent metric for our tile size selection.

Here, the analysis is shown for rectangular matrices C[M][N], A[M][P] and B[P][N]. We first analyze cold misses as seen by a single core for a tile, whose size is denoted as a *tuple (I,J,K)*. The number of cold misses for 1 iteration of the kT loop (shown below) should thus be the total cold misses to bring 1 tile each of arrays C, A and B. They have I*J, I*K and K*J elements, respectively. Here, CLS is the number of data elements contained in one *cache line*.

$$I * \lceil \frac{J}{CLS} \rceil + I * \lceil \frac{K}{CLS} \rceil + K * \lceil \frac{J}{CLS} \rceil \quad (1)$$

We can observe that a tile of array C can be perfectly reused throughout the kT loop. However, for arrays A and B, we need to bring in an additional P/K tiles in subsequent iterations of the kT loop. Thus, the estimated total cold misses for the entire kT loop on simplification are as follows.

$$\frac{I * J + I * P + J * P}{CLS} \quad (2)$$

Now, assuming that we parallelize the outermost loop over r cores, we obtain a total of $\frac{M*N}{I*J*r}$ iterations in loops iT and jT. Thus, the *total number of cache misses* (TCM) seen by 1 core in LLC cache on simplification are as follows.

$$\left(\frac{1}{I} + \frac{1}{P} + \frac{1}{J}\right) * \frac{M * N * P}{r * CLS} \quad (3)$$

For the common case when the problem size is not very large, we leave the innermost loop (loop j) untiled as shown in Figure 2(c). We find this to be feasible given the large LLCs in modern multicores and particularly helpful since this exposes long streams to the prefetcher. Thus, with J=N, we obtain from Equation 3 above that the LLC misses, or in other words, the total number of off-chip memory accesses, can be minimized by choosing a large I. This is further facilitated by our decision of choosing to tile for the larger LLC instead of L1 or L2 cache.

As a result of choosing a large J (= N, in the general case) and a large I, we find that the working set in the LLC is dominated by array C. Based on this observation, our TTS algorithm assumes that all but 1 ways of the LLC are occupied by array C. Note that sharing of the LLC by all cores is taken into account. TTS then models the filling of the data in the LLC for array C for a given tile size. It thus chooses the largest I such that no (self-)interference misses result.

4.2 Data Reuse in multiple levels of cache

As discussed in Section 3.1, data reuse in matmul-like kernels exists at two loop levels - a tile of array B could be reused in outermost intra-tile loop i and a tile of array C could be reused in the innermost inter-tile loop kT. In the previous section, we derived that the reuse of array C in LLC can be maximized by choosing a large I. A similar analysis as above could be applied for the L2 cache and we obtain that the total cache misses are minimized when we choose a large K. Our TTS algorithm is thus applied with the corresponding parameters for the L2 cache to model the filling of data accessed in a tile of array B and the largest K could be obtained before interference is set in. The only difference is that in the case of L2 cache, we assume that although array B dominates the working set, it still occupies only 75% of the ways since the L2 is small and is shared by program data and instructions. We thus see that TTS can be applied to multiple levels of cache (L2 and L3 in this case) to obtain the maximum tile sizes that do not cause pronounced evictions through interference.

4.3 The Effect of Problem Size and Load Balancing on Tile Factors

Our discussions so far have been geared toward large problem sizes. For smaller problem sizes, we cannot exploit data reuse of array C in L3 cache effectively due to ineffective load balancing among available cores. This is explained as follows.

The reusable data of array C is I*N (i.e. its tile size). If the problem size N is very small, we need to increase I in order to effectively exploit data reuse in the allotted portion of L3 cache. With a large value of I, the number of iterations in the outermost *inter*-tile

```
for (k=2, N-1)
  for(j=2,N-1)
    for(i=2,N-1)
      A(k,j,i) = C*(B(k,j,i-1) + B(k,j,i+1)
                  +B(k,j-1,i)  + B(k,j+1,i)
                  +B(k-1,j,i) + B(k+1,j,i)
```

(a)

```
for(jT = 2,N-1,Tj)
  for(iT = 2,N-1,Ti)
    for (k=2, N-1)
      for(j=jT , min(jT+Tj-1,N-1))
        for(i=iT , min(iT+Ti-1,N-1))
          A(k,j,i) = C*(B(k,j,i-1) + B(k,j,i+1)
                      +B(k,j-1,i)  + B(k,j+1,i)
                      +B(k-1,j,i) + B(k+1,j,i)
```

(b)

```
for(jT = 2,N-1,Tj)
  for (k=2, N-1)
    for(j=jT , min(jT+Tj-1,N-1))
      for(i=2,N-1)
        A(k,j,i) = C*(B(k,j,i-1) + B(k,j,i+1)
                    +B(k,j-1,i)  + B(k,j+1,i)
                    +B(k-1,j,i) + B(k+1,j,i)
```

(c)

Figure 3: (a) Original 3D jacobi stencil; (b) 2D tiled 3D jacobi stencil; (c)Turbo-Tiled 3D jacobi stencil

iT loop ($\frac{M}{I}$) becomes small. This will lead to significant reduction in the amount of coarse-grained parallelism for iT loop. This can exacerbate the problem of uneven distribution of load among available execution cores.

Therefore, for smaller problem sizes, we keep the tile factor I small to minimize workload imbalance among cores. To determine this inflection point, we perform the following analysis.

As mentioned earlier, we assign W ways of L3 cache per core to a tile of array C with I*N data elements. The total number of data elements that can occupy W ways is as follows.

$$W * nSets * CLS \tag{4}$$

In Equation 4, *nSets* is the number of sets in the L3 cache that can be calculated as $\frac{cSize_{L3}}{lSize*n_{L3}}$. Here, $cSize_{L3}$, *lSize* and n_{L3} are the L3 cache size, line size and set-associativity, respectively. CLS is the number of data elements which can be accommodated in a single *cache line*, which in terms of the cache parameters is equal to $\frac{lSize}{size(DataType)}$. The number of ways W assigned to each core is $(\lfloor\frac{n_{L3}}{r}\rfloor - 1)$ [1].

Putting these values in Equation 4 and equating it to *I*N*, we have the following equation after simplification.

$$I * N = (\lfloor\frac{n_{L3}}{r}\rfloor - 1) * \frac{cSize_{L3}}{n_{L3} * size(Datatype)} \tag{5}$$

For optimum load balancing, we define a metric *granularity* to represent the amount of workload distributed to each of the r cores. Since the number of loop iterations for outermost *intra*-tile loop iT is $\lfloor\frac{M}{I}\rfloor$, *granularity* is calculated as follows.

$$granularity = \frac{M}{I * r} > 2 \tag{6}$$

To exploit data reuse of array C in L3 cache, we set the *granularity* of Equation 6 to be greater than 2. This is in accordance with our empirical observations, i.e. when an application is run on r cores, each core must be assigned at least 2 iterations of the outermost *inter*-tile iT loop for effective load distribution among cores.

By calculating the value of I from Equation 5 and substituting in *granularity* of Equation 6 , we get

$$M * N > \frac{2 * r * (\lfloor\frac{n_{L3}}{r}\rfloor - 1) * cSize_{L3}}{n_{L3} * size(DataType)} \tag{7}$$

Thus, we observe that we require a larger problem size when we are using greater number of cores to prevent load imbalance while benefitting from tiling and prefetching. This is in line with our choice of tiling for the last level cache for larger problem sizes. Note that we choose a smaller I for smaller problem sizes, which minimizes the amount of reuse in the LLC but yields better performance due to better load distribution.

[1]Array reference C occupies all but 1 ways of the number of ways assigned to a core, and we assign equal number of ways to each core (as discussed in Section 4.2). The case when $r \geq n_{L3}$ is discussed in Section 5.1

4.4 LLC Reuse for stencil codes

For iterative stencil codes like *3D Jacobi* in Figure 3 (a), Rivera et al. [32] show that exploiting group reuse by (2D) tiling for a suitable cache size can significantly improve performance. Group reuse in a 3D stencil is explained as follows.

As shown in Figure 4, 3D Jacobi uses a 6-point stencil in 3 dimensions. In a single iteration of the innermost loop, it accesses 4 elements in the array plane k and 1 element each in array planes k+1 and k-1. As the program iterates through the innermost i loop, it sweeps 3 columns along the i dimension in the array plane k and 1 column each in array plane k+1 and k-1. Therefore, across the entire i loop, a 6-point stencil accesses five columns in three adjacent planes at the same time. Consequently, across the middle loop (j), three entire adjacent array planes are accessed. Thus, for preserving group reuse across the outermost loop (k), these adjacent 3 planes have to fit in a particular level of cache. For a 256KB L2 cache, problem sizes that can fit into the cache are limited to 128x128 (assuming square problem sizes) for a 4-byte single-precision floating-point datatype. If we have problem sizes larger than this working set, group reuse is lost as it results in cache overflow. Thus, the need for tiling arises in the case of larger problem sizes.

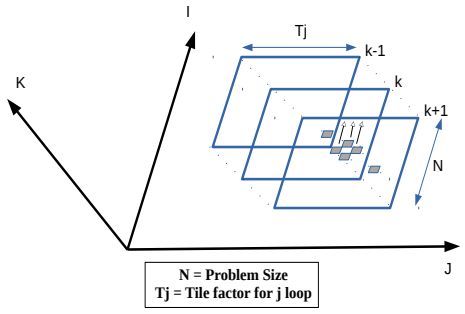

Figure 4: Access pattern of Turbo-Tiled code of 3D stencil

Group reuse can be preserved if we tile the innermost two loops. Therefore, i and j loops are tiled with tile factors Ti and Tj. These tile factors determine working set size which has 3*Ti*Tj data elements that correspond to the 3 I-J planes accessed in 1 iteration of the outermost k loop. It is therefore sufficient for the cache to hold 3*Ti*Tj elements of data.

In order to exploit the use of the hardware prefetchers, we keep this data in the L3 cache which is achieved by keeping the tile factor Ti equal to the problem size. This allows us to exploit streaming access patterns to prefetch arrays A and B all the way from the L3 cache to the L1 cache. The resultant code is simpler (Figure 3 (c)) than the 2D-tiled code (Figure 3 (b)).

5. TURBO-TILE SELECTION ALGORITHM

Based on our analysis of the *matmul* kernel for rectangular matrices of problem size *tuple (M,N,P)* in Section 4, we propose the Turbo-Tile Selection (TTS) algorithm (Algorithm 1). TTS uses the *ComputeTileFactor* routine (Algorithm 2) to decide the tile factors I and K for improving reuse of arrays B and C in L2 cache and LLC (L3), respectively. As described in Section 4.3, for smaller problem sizes, exploiting data reuse in assigned portion of L3 cache has an adverse affect on multi-core performance. We make this switch based on the problem size in accordance with the analysis done in Section 4.3. Moreover, we keep the tile factor K similar for all problem sizes, i.e. tiling array B for L2 cache. The *ComputeTileFactor* subroutine returns the values of K and I, and TTS outputs a *tuple (I,K,N)* (as discussed in Section 4). In case of fractional *granularity*, the value of tile factor I is rounded off to a value corresponding to an integral *granularity* for effective load balancing. We also make sure that if the problem size is too big (which results in a small I, i.e. less than 4), then we resort to tiling for the higher level caches as in tss since TTS may yield very thin tiles. This situation, however, is rare as the LLC is generally large enough to accommodate a few rows unless the problem size is too large.

ALGORITHM 1: Turbo-Tile Selection

INPUT:
Size of L2, LLC caches: $cSize_{L2}, cSize_{LLC}$
Set-associativity of L2, LLC caches: n_{L2}, n_{LLC}
Size of cache line: $lSize$
Problem Size: MxNxP
No. of arrays with self-temporal reuse in outermost intra-tile loop: $s1$
No. of arrays with self-temporal reuse in innermost inter-tile loop: $s2$
No. of execution cores: r
Effective set-associativity of L2 : n_e $\left(= \lfloor \frac{n_{L2}}{s1} - 1 \rfloor\right)$
Effective no. of ways of LLC per Core: W $\left(= \lfloor \frac{n_{LLC}}{r*s2} \rfloor - 1\right)$
if $M * N > \frac{2*r*(\lfloor \frac{n_{LLC}}{r} \rfloor - 1)*cSize_{LLC}}{n_{LLC}*size(DataType)}$ **then**
  $I = ComputeTileFactor(cSize_{LLC}, lSize, n_{LLC}, M, N, W)$
  if $I < 4$ **then**
    Fall back to tss to choose the tile size tuple (I,J,K)
  else
    $granularity = \lfloor \frac{M}{I*r} \rfloor$
    if $granularity < \frac{M}{I*r}$ **then**
      while $\frac{M}{granularity} \neq \lfloor \frac{M}{granularity} \rfloor$ **do**
        granularity++
      $I = \frac{M}{granularity*r}$
    $K = ComputeTileFactor(cSize_{L2}, lSize, n_{L2}, P, N, n_e))$
else
  $I = 4$
OUTPUT:(I,K,N)

The *ComputeTileFactor* routine emulates the behavior of incoming data with respect to a particular level of cache. Emulation of the number of filled ways of a set is tracked by the counter fillCache[S]. The algorithm increments this counter whenever a cache line accessed within a tile maps to the set S. This counter is increased until it becomes equal to the effective number of ways assigned to an array for each core. The emulation is stopped at this point because interference sets in at that point. The value of fillCache[S] at this point is used as the height of the tile. TTS sets the tile width to be the problem size N along the leading dimension of arrays B and C.

As discussed earlier, the working set in the L2 and L3 caches is dominated by tiles of arrays B and C, respectively. Therefore, in Algorithm 1, we assign W as $\left(= \lfloor \frac{n_{L3}}{r*s2} \rfloor - 1\right)$ and n_e $\left(= \lfloor \frac{n_{L2}}{s1} - 1 \rfloor\right)$ to accommodate arrays C and B in L3 and L2 caches, respectively.

ALGORITHM 2: ComputeTileFactor

INPUT:
Size of cache: $cSize$
Size of cache line: $lSize$
Set-associativity of cache: n
Problem Size along non-leading dimension: M
Problem Size along leading dimension: N
Effective number of ways per array per Core: W
Begin:
No. of elements in a cache line: CLS $\left(= \frac{lSize}{size(DataType)}\right)$
No. of sets in cache: $nSets$ $\left(= \frac{cSize}{lSize*n}\right)$
$w = \frac{N}{CLS}$ {/* Fixing the width of tile to problem size */}
$r = 0$
repeat {/* iterates over rows in a tile */}
  $nset \leftarrow \left(\frac{r*M}{CLS}\right) \% nSets$
  for $c = 0$ to w **do** {/* brings a tile-row into cache */}
    if $fillCache[nset + c] = W$ **then**
      $h = r$ {/* tile height is maximum rows in tile */}
      return {/* interference begins at this point */}
    else
      $fillCache[nset + c] + +$
  $r + +$
until $r = M$
OUTPUT: (h)

We thus compute the tile factors I and K through a call to *ComputeTileFactor* with the corresponding parameters. Also, note that we assume no data sharing among cores to be conservative. Finally, since TTS drastically reduces the number of tile sizes evaluated (and does not involve an actual run of the tiled codes), it returns its estimate of the tile size in less than a second in all cases. This is significant improvement (in compile-time) over search-based strategies.

5.1 Algorithm's Applicability

While discussing the applicability of the TTS algorithm, we consider the types of programs that can benefit from the algorithm. We also consider its robustness in a multi-core environment and its adaptability to a range of problem sizes in this section.

TTS employs the *ComputeTileFactor* routine to exploit data reuse in one of the loops in the tiled code to obtain the corresponding tile factor. Thus, codes containing reuse in any one of the loop nests can use this routine to find a suitable tile factor. More importantly, if a program has data reuse in more than one array at different levels of nested loops, it can utilize our algorithm more effectively at different levels of caches. In this study, we are exploiting data reuse in the innermost *inter*-tile loop and outer most *intra*-tile loop. The loop order of *intra*-tile loops in the PLuTo generated tiled code also determines whether we can exploit data reuse in both the *inter*-tile and the *intra*-tile loop independently or not.

For example, if the innermost *inter*-tile and innermost *intra*-tile loops are the same (as in *strsm*), data reuse in innermost *inter*-tile loop conflicts with both data reuse in outermost *intra*-tile loop and vectorization. However, by selecting the tile factor that corresponds to the innermost *intra*-tile loop to be problem size, we reap all of the advantages: data reuse in *inter*-tile loop jT (instead of kT loop, as it will not exist), vectorization and data reuse in the outermost *intra*-tile loop. Thus, any program that uses rectangular tiles (but not loop skewing) can take advantage of our algorithm. Some examples of such kernels are BLAS, data mining kernels, iterative stencils and image processing kernels. It is important to note that although TTS

does not adequately handle time-tiled stencils, it does find good tile sizes for the larger stencil codes as discussed in Section 4.4. For time-tiled stencils, there have been multiple non-conventional tiling strategies proposed recently [19, 3, 36, 35] that are considerably better in terms of scalability than conventional approaches that introduce pipelined parallelism. These newly proposed strategies introduce their own peculiarities in tile size selection. For example, the diamond tiling proposed in [3] requires tiles to be square or else there is severe load imbalance. It is for this reason that we do not attempt to deal with time-tiled stencils in this work.

TTS exploits scalability. The most significant feature of TTS is that it is versatile for any number of cores in the system. It gives a tile factor that aims to minimize conflict misses in the shared L3 cache. Since set-associativity of LLC in current micro-architectures is very high, TTS exploits this fact effectively by assigning separate ways to different cores.

Applicable to a wide range of problem sizes. TTS not only gives good results for large problem sizes, but it also outperforms tss for small problem sizes. As discussed in section 4.3, for smaller problem sizes like 200 and 400, we cannot avail the effective data reuse of array C in L3 cache. In spite of this constraint, we still benefit from better vectorization and prefetching through the DCU prefetcher at the L1 cache to prefetch array B from the L2 cache to the L1 cache.

Applicable to many cores and many array references. To determine tile factor I, TTS depends on the effective number of ways ($\lfloor \frac{n_{L3}}{r*s2} \rfloor - 1$) assigned to each core. Tile factor K depends on effective associativity calculated as $n_e = \lfloor \frac{n_{L2}}{s1} - 1 \rfloor$. Thus, if the number of cores or the number of array references are much larger than the set-associativity of the respective caches, we can no longer assign an individual way to each core/array. In such a case, we emulate our algorithm by multiplying the set-associativity by a factor *d* (generally, 2 is sufficient). We also reduce the number of available sets by *d* to keep the total cache size constant in our estimation. This approach effectively assigns a portion of a single way to a core, and the emulation can no longer guarantee a tile factor that gives minimal conflict misses. Even so, this improvisation still gives a reasonably good approximation. This is attributed to keeping the cache size the same to minimize capacity misses, and since the number of sets remains a power of 2, it still accounts for the conflict misses reasonably well.

5.2 The Framework

We have implemented the TTS algorithm using the PLuTo [8] framework. PLuTo is a source-to-source compiler based on the polyhedral model. It optimizes programs for locality and parallelism simultaneously.

PLuTo takes an input of the initial source code and generates *statement-wise affine* transformations, which include various loop transformations such as permutation, reversal, skewing, relative shifting and tiling. The next step in PLuTo is to specify the tile sizes provided by the *Polyhedral Tile Specifier* [8]. We intercept this step by providing the TTS generated tile sizes to the *Polyhedral Tile Specifier*. Subsequently, the enhanced transformations that include the tile sizes are provided to CLooG [5] to generate the tiled code. This tiled code, generated by CLooG can be compiled by any production compiler such as icc or gcc. The integrated framework is shown in Figure 5.

The TTS algorithm needs 4 types of parameters as input: (1) L2, L3 cache parameters (cache size, set-associativity and line size), (2) Programmer specified parameters (number of cores (r)), (3) Source code specific parameters (problem sizes and datatype), (4) Program specific parameters (loop order, number of arrays with

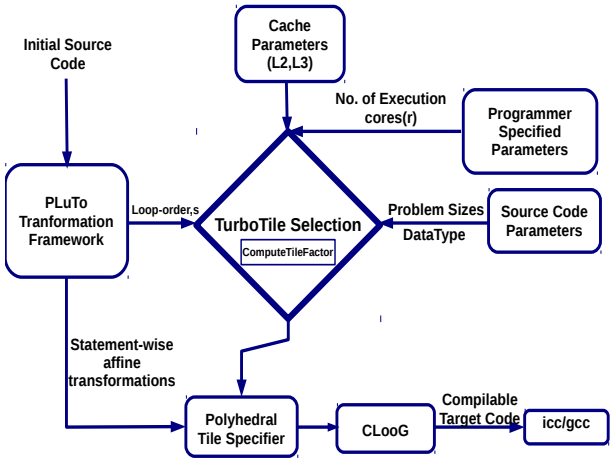

Figure 5: Framework

self-temporal reuse (s)). The information about the number of arrays with self-temporal reuse in respective loops is obtained from PLuTo by extracting the access matrices of each array and the loop order. In particular, we determine that an array a is reused in loop i if i does not appear in the subscript of that array access since this indicates that we access the same elements of a in different iterations of loop i. The loop order of the *intra*-tile loops is determined by PLuTo so as to favor vectorization. This information is needed in order to determine the correct order of tile sizes. All architectural cache-specific parameters are obtained from files located at */sys/devices/system/cpu/cpu0/cache/* in Linux and by the system call *GetLogicalProcessorInformation* in Windows.

6. EXPERIMENTAL SETUP AND RESULTS

We tested our algorithm on an Intel Xeon CPU E5-2650 v2 processor with 8 cores running at 2.60GHz based on the Intel Ivy Bridge micro-architecture, and an AMD Opteron (tm) Processor 2376 with 4 cores running at 2.30 MHz with K10 Shanghai micro-architecture. Intel processor has an 8-way 32 KB L1 cache, 8-way 256KB L2 cache and a 20-way 20MB L3 cache while the AMD processor has a 2-way 64KB L1 cache, 16-way 512KB L2 cache and a 48-way 6MB L3 cache. We choose the effective L3 cache capacity as listed in [12]. For our experiments, we use 9 kernels from PluTo and PolyBench [27] and we have evaluated 1 kernel from SPEC OMP2012. We have tested our algorithm for 6 different problem sizes, one of which is a power of 2. Problem sizes that are a power of 2 are termed as *pathological* because they can cause pronounced conflict misses for certain tile sizes (as described in Section 3.1). We choose a large range of problem sizes in particular to show the versatility of our algorithm for both smaller and larger problem sizes. In the case of smaller problem sizes, the working set gets reduced to fit in L1 or L2 cache. Thus, it becomes difficult to exploit data reuse in L3 cache. For this reason, the TTS algorithm resorts to exploiting data reuse only in L2 cache while still setting the innermost tile factor to be the problem size. It leverages streaming access patterns through hardware prefetchers at either L2/L3 or L1 cache for bigger and smaller problem sizes, respectively.

For generating the tiled code, we use PLuTo (version 0.9.0) with the '–tile' and '–parallel' options. For compiling the programs, we use the Intel C Compiler (icc version 15.0.1). For multi-core runs, we use the KMP_AFFINITY environment variable to pin 1 thread on each of the cores. We also use a chunk size of 1 with static scheduling (for parallelizing the iterations of outermost *inter*-tile

Bench	Scheme	Problem Size = 200		Problem Size = 400		Problem Size = 800		Problem Size = 1024		Problem Size = 1600		Problem Size=3200	
		Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.
dsyrk	TTS	4x192x200	1.8	4x96x400	13.5	4x64x800	52	4x48x1024	63.4	100x32x1600	52.1	40x16x3200	82.2
	tss	24x48x128	1.9	48x12x400	11.9	100x24x208	37.5	6x48x992	56.6	212x32x192	32.9	192x24x192	53
	Searched Tile	32x192x128	2.6	32x384x96	13.5	32x224x32	44.8	32x512x64	55.7	32x480x32	71	32x384x64	77.5
dsyr2k	TTS	4x96x200	4.1	4x48x400	36.13	4x32x800	60.3	4x24x1024	76.8	100x16x1600	62.1	40x8x3200	88.4
	tss	24x24x128	3.6	48x6x400	29.95	100x12x208	45.4	6x24x992	69.7	212x16x192	61.8	192x12x192	72
	Searched Tile	32x96x160	4.2	32x256x32	22.9	32x448x32	62.3	32x992x32	67.1	32x448x32	84.3	32x448x32	83
matmul	TTS	4x192x200	11.8	4x96x400	63.1	4x64x800	158	4x48x1024	146	100x32x1600	163.8	40x16x3200	166.7
	tss	24x48x128	11.6	48x12x400	60.8	100x24x208	154.4	6x48x992	137.67	212x32x192	164.6	192x24x192	143.3
	Searched Tile	32x192x128	9.5	32x288x96	60	64x512x64	113.9	160x512x32	122.6	224x256x64	156.7	224x384x32	150.7
strsm	TTS	24x24x200	1.1	4x20x400	9.3	4x64x800	20	6x48x1024	28	100x32x1600	50.2	40x16x3200	71.7
	tss	200x48x128	1.1	400x12x400	5.5	800x24x208	10.9	1024x48x992	21.5	1600x32x192	16	3200x24x192	16.2
	Searched Tile	32x128x96	2.3	32x384x384	13.6	64x736x736	31.1	64x992x992	46.6	32x2848x2592	58.9	32x3104x3104	57.9
tmm	TTS	4x192x200	2	4x96x400	8.5	4x64x800	24.7	4x48x1024	11.3	100x32x1600	30.6	40x16x3200	31.7
	tss	24x48x128	1.6	48x12x400	6.3	100x24x208	26.2	6x48x992	10.6	212x32x192	29.4	192x24x192	24.7
	Searched Tile	160x128x128	1.6	32x288x64	9.4	32x224x32	21.2	128x512x32	28.4	224x480x32	29.5	32x384x64	30.2
trisolv	TTS	24x24x200	.7	4x20x400	6.3	4x64x800	40.7	4x48x1024	43.4	100x32x1600	53.4	40x16x3200	71.6
	tss	24x48x128	.7	48x12x400	3.4	100x24x208	26.7	6x48x992	48.2	212x32x192	43.4	192x24x192	66.8
	Searched Tile	192x160x128	2.1	192x32x128	12.7	288x64x32	36.3	896x96x32	41.7	704x64x32	61.4	224x160x64	55.8
corcol	TTS	4x192x200	3.3	4x96x400	18.4	4x64x800	47.2	4x48x1024	57.4	100x32x1600	53.1	40x16x3200	70
	tss	24x48x128	1.4	48x12x400	15.9	100x24x208	33.9	6x48x992	51.8	212x32x192	38	192x24x192	48.7
	Searched Tile	32x192x96	2	32x256x32	12.5	32x416x32	28.2	32x608x32	45.3	64x416x32	56.7	128x384x32	57
covcol	TTS	4x192x200	2.1	4x96x400	10	4x64x800	31.3	4x48x1024	37.5	100x32x1600	34.7	40x16x3200	45.5
	tss	24x48x128	1.5	48x12x400	9.3	100x24x208	14.4	6x48x992	24	212x32x192	24	192x24x192	25.8
	Searched Tile	32x64x160	1.3	32x256x32	7.9	32x608x32	26.4	32x736x32	30.5	64x384x32	36.4	96x352x32	34

Table 1: Performance comparison on 8 cores (in GFLOPS) on Intel Xeon

loop) to maximize *granularity*. Furthermore, we use the '–openmp' compiler option with ICC to obtain results on multiple cores. It is important to note that since the tiled code is complicated enough, *icc* did not perform any further optimizations to favor some tiled codes versus others. We then compare our results with tss that tiles for L1-L2 caches. We also make sure that for smaller problem sizes, the value of tile factor I chosen by tss results in equal load distribution among the execution cores.

6.1 Performance of TTS Algorithm in a multi-core Environment

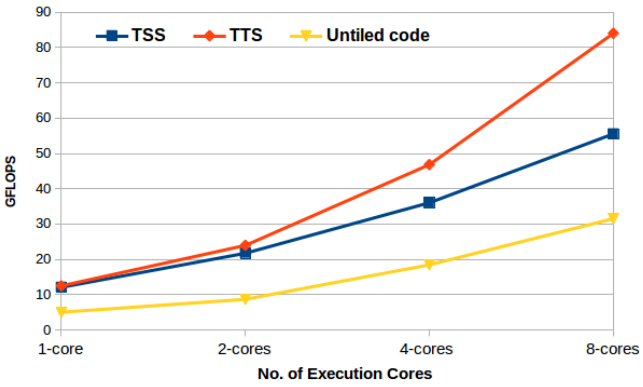

Figure 6: Scalability comparison for dsyrk (N=3200)

We show the performance results for 8-cores in Table 1 in GFLOPS. We also show the *"Searched Tile"* performance that is obtained by exhaustively running all combinations of tiled codes with the tile factor ranging from 32 to $\sqrt{\frac{L3CacheSize}{r*size(DataType)}}$ at gaps of 32. Note that 32 is equal to 2 cache lines for the float datatype; we choose 2 cache lines instead of 1 in order to prune the large search space. We choose the upper limit of the search space after considering the maximum amount of data that can reside in the L3 cache in a multi-core processor. We adopt this semi-exhaustive heuristic based search method for the *"Searched Tile"* because a similar approach is adopted by the popular auto-tuning framework ATLAS [39]. In other words, we intend this "Searched Tile" to represent heuristic based auto-tuning approaches (note that the chosen heuristic tends towards square problem sizes although the heuristic could be made intelligent by incorporating the effect of prefetching). For a valid comparison, we verify that CLooG [6], the code generator used by PLuTo, does not cause vastly different codes generated for different tile sizes to hamper comparison. We see, however, that in some cases, there is load imbalance but that mainly happens for the smaller problem sizes.

Figure 6 compares the scalability of our algorithm with tss and untiled code. The results are shown for BLAS kernel *dsyrk* with problem size of 3200. We can observe the increasing performance gap between the L1-L2 tiled code and TTS code for an increasing number of cores. This increasing gap indicates the combined effect of data reuse in the L3 cache and timely prefetching of data to the L1 cache (shown by VTune counters in Section 2.1). Furthermore, we can make the following inferences from Figure 6.

1. When a program code is executed on 8 cores, contention for off-chip bandwidth increases due to a larger number of cores trying to access the off-chip memory simultaneously. When tiling for L3 cache, due to reduction in off-chip memory accesses (as shown in Section 2.1), contention for off-chip bandwidth is minimized. This is reflected in high scalability performance of L3 tiled code in Figure 6.

2. Moreover, we can conclude from the scalability comparison that there is no disadvantage from not tiling for the faster L1-L2 caches. Since tiling for L3 cache exposes streaming access patterns in array references to hardware prefetchers, it results in timely prefetching of data from L3 cache to L1 cache. Thus, L3 tiled code gets the best of both worlds, i.e. advantage of increased data reuse in the larger L3 cache and effective data accesses in L1 cache with the help of prefetchers.

We can observe from Table 1 that for larger problem sizes (1600 and 3200), TTS performs better than tss by approximately 60% for the *dsyrk* benchmark, while the average improvements in the

Bench	Scheme	Problem Size = 200		Problem Size = 400		Problem Size = 800		Problem Size = 1024		Problem Size = 1600		Problem Size=3200	
		Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.	Tile Size	Perf.
dsyrk	TTS	4x192x200	9.4	4x96x400	10.35	4x64x800	11.7	4x48x1024	11.1	100x32x1600	11.3	40x16x3200	12.4
	tss	24x48x128	8	48x12x400	11.4	100x24x208	11.7	6x48x992	10.6	212x32x192	9.6	192x24x192	12.1
	Searched Tile	64x128x32	8.5	64x224x32	11	32x416x32	11.6	960x512x32	11.9	32x544x32	12.2	256x384x64	12.1
dsyr2k	TTS	4x96x200	13.1	4x48x400	13.9	4x32x800	13.5	4x24x1024	14.2	100x16x1600	6.3	40x8x3200	15.8
	tss	24x24x128	13	48x6x400	15.6	100x12x208	16	6x24x992	12.6	212x16x192	6.3	192x12x192	14.6
	Searched Tile	64x128x32	12.8	384x320x32	14.2	512x416x32	14.7	896x544x32	12.8	96x320x32	14.9	192x128x288	15.8
matmul	TTS	4x192x200	23.6	4x96x400	25.5	4x64x800	26	4x48x1024	23.5	100x32x1600	25.1	40x16x3200	23.5
	tss	24x48x128	23.4	48x12x400	27.6	100x24x208	27.4	6x48x992	22.9	212x32x192	26.8	192x24x192	26.4
	Searched Tile	128x160x32	24.2	384x32x384	27.7	448x32x704	24.7	160x512x32	24.8	96x160x32	27.9	256x32x352	26.6
strsm	TTS	24x24x200	8.2	4x20x400	10.6	4x64x800	10.5	6x48x1024	10.4	100x32x1600	12.2	40x16x3200	12.6
	tss	200x48x128	3.6	400x12x400	10.5	800x24x208	3.1	1024x48x992	9.8	1600x32x192	2.7	3200x24x192	2.6
	Searched Tile	128x32x192	6.5	64x96x384	8.3	128x160x32	28.3	960x32x992	9.4	2080x32x2080	12.3	32x32x384	2.9
tmm	TTS	4x192x200	9.2	4x96x400	11	4x64x800	10	4x48x1024	11.3	100x32x1600	12.2	40x16x3200	11.7
	tss	24x48x128	8.7	48x12x400	6.3	100x24x208	11.9	6x48x992	10.6	212x32x192	11.9	192x24x192	11.8
	Searched Tile	96x128x32	8.7	96x224x32	11.4	64x608x32	11.8	448x544x32	11.6	128x480x32	12	224x256x32	11.9
trisolv	TTS	24x24x200	10.9	4x20x400	11.8	4x64x800	10.9	4x48x1024	11	100x32x1600	12	40x16x3200	11.4
	tss	24x48x128	10	48x12x400	12.5	100x24x208	12.2	6x48x992	10.8	212x32x192	13.1	192x24x192	13
	Searched Tile	128x160x32	10.4	224x192x32	11.5	160x256x32	13.2	512x864x32	12.4	160x64x32	13.4	224x288x32	13.2
corcol	TTS	4x192x200	7.2	4x96x400	8.8	4x64x800	10.9	4x48x1024	9.9	100x32x1600	10.5	40x16x3200	10.5
	tss	24x48x128	6.5	48x12x400	9.2	100x24x208	10.7	6x48x992	9.5	212x32x192	10.1	192x24x192	10.4
	Searched Tile	128x128x64	7.2	96x224x32	9.8	64x640x32	10.5	32x576x32	10.6	192x544x32	11	160x384x32	10.7
covcol	TTS	4x192x200	2.1	4x96x400	10	4x64x800	6.7	4x48x1024	6.7	100x32x1600	7.2	40x16x3200	6.9
	tss	24x48x128	1.5	48x12x400	9.3	100x24x208	6.7	6x48x992	6.3	212x32x192	6.8	192x24x192	6.5
	Searched Tile	64x128x32	4.9	32x224x32	6.7	32x416x32	7.1	32x544x32	7.3	32x544x32	7.4	160x384x32	7.1

Table 2: Performance comparison on 1 core (in GFLOPS) on Intel Xeon

cases of *dsyr2k* and *matmul* benchmarks are 12% and 8%, respectively. In these benchmarks, the number of memory accesses in the loop body are high as compared to the computation time of the loop body. This puts a lot of pressure on hardware prefetchers to prefetch more data in a less amount of time. Therefore, many triggered prefetches remain *in-flight* and the required data does not reach the L1 cache in time. This is reflected in the increase of *L1 data cache prefetch misses* in case of *matmul* and *dsyr2k* by 120% and 95% , respectively, over *dsyrk*. This is because the DCU prefetcher at L1 cache prefetches only 1 cache line ahead. If the DCU prefetcher could prefetch two cache lines ahead, data would have arrived in the L1 cache more timely and benchmarks like *dsyr2k* and *matmul* could also have seen significant performance improvement.

As explained in Section 5.1, in a 3D-tiled *strsm* benchmark, achieving data reuse in innermost *inter*-tile loop (kT) will conflict with vectorization. Therefore, tss favors vectorization over data reuse in kT loop by leaving the outermost *intra*-tile loop untiled. On the other hand, we keep the innermost *intra*-tile loop untiled to leverage self-temporal locality in innermost *inter*-tile loop. Thus, *strsm* gives an improvement of more than 2.5x for TTS, as compared to tss, on 8 cores.

We show the performance of TTS for 6 square problem sizes (3200, 1600, 1024, 800, 400, 200) with float datatype. Among them, 3200 and 1600 are large enough that they can get good performance by tiling array C of Figure 2 (c) in L3 cache. As the problem size reduces, the effect of load imbalance becomes more pronounced. Thus, we have to sacrifice data reuse of array C in L3 cache. This threshold is estimated according to Equation 7. The threshold value comes out to be around 1414 for the float datatype. Therefore, we choose the switching point to be between 1600 and 1024. For the double datatype, this threshold comes out to be around 1000, and so we switch between 1024 and 800. This is in accordance with our experimental observations. Thus, our strategy of choosing tile sizes proves to be very robust for different problem sizes and datatypes.

We also show 1-core performance results (in GFLOPS) in Table 2 for TTS and compare it with tss and the *"Searched Tile"*. We can observe that 1-core TTS results for all benchmarks and problem sizes are comparable to tss and *"Searched Tile"* indicating that, despite its design for a large number of cores, TTS performs equally well to tss even for a single core.

When run on the AMD Opteron processor with 4 cores, TTS shows an average improvement of 29% and 62% for smaller and larger problem sizes, respectively, over tss. Thus, TTS performs well on AMD processor also, whose cache-hierarchy is sufficiently different from Intel processor.

We don't compare our approach with tools like ATLAS or MKL because these tools perform additional optimizations like loop unrolling, pipeline scheduling and inserting inline assembly (to get best vector performance, etc.). Therefore, such a comparison would not be an apple-to-apple comparison.

6.2 Results of TurboTiling on Iterative Stencils

Benchmark	Problem Size	Scheme	Tile Size	8-core Performance
mgrid	Class B	Untiled	-	91.38
		TTS	32x256	106.28
advect3D	300	Untiled	-	4.95
		TTS	16x300	7.97

Table 3: Performance comparison on 8 cores for stencils (in GFLOPS)

As discussed in Section 4.4, for iterative stencils, TTS tiles for j loop of stencil code and leaves the i loop untiled. In this way, the tiled code reaps all the advantages: group reuse in L3 cache, vectorization and streaming access patterns through prefetchers. For example, in the case of *advect3D*, the group reuse is at a distance of 5. TTS tries to fit 5*Ti*Tj elements of data from five I-J planes in L3 cache, where Ti is set to the problem size N. Since in *advect3D*, data reuse of 8 arrays takes place in L3 cache, there are too many array references and cores which implies that we can no longer assign an individual way to each array. Therefore, we adopt the improvisation mentioned in Section 5.1. For example, for a 20-way L3 cache on a 8-core Ivy Bridge, TTS assigns one-eighth of $\lfloor \frac{20}{8} \rfloor - 1$ ways to each array of *advect3D*. Improvement in the

performance of *advect3D* shows that such an improvisation indeed holds good when there are too many arrays in the application program. A similar approach is adopted for *mgrid*. We compare our approach with untiled code in Table 3 and 4 for 8-cores and 1-core, respectively, over the respective untiled codes. Our approach gives a speed-up of 50% and 55% for *advect3D*, and 16% and 10% for *mgrid* on 8-cores and 1-core, respectively.

Benchmark	Problem Size	Scheme	Tile Size	1-core Performance
mgrid	Class C	Untiled	-	2.35
		TTS	64x512	2.59
advect3D	500	Untiled	-	10.31
		TTS	132x500	16.05

Table 4: Performance comparison on 1 core for stencils (in GFLOPS)

7. DISCUSSION

In this section, we discuss some additional points relevant to TurboTiling that better establish the basic premise of the proposed algorithm and its general applicability to the problem of tile size selection when compared to existing tools.

7.1 Precise cause for performance improvement - Prefetching or LLC reuse?

The cause for the performance improvement seen with TTS over tss is not just better reuse in the last level cache or efficient use of hardware prefetching - it is both better reuse and prefetching.

Better LLC reuse helps TTS since the improvement with TTS increases significantly over tss for larger number of threads (as seen in Figure 6). This is due to reuse in L3 and therefore, smaller off-chip accesses, which is particularly helpful with more threads. Prefetching is not the cause for better scalability of TTS since prefetching neither reduces the number of memory accesses nor the latency of each access - it only hides the latency in the same way for 1 thread and 8 threads.

In order to demonstrate the benefit from just prefetching, we extend tss (that currently reuses data in L1 and L2) to tss+L3 that reuses the data in L1 and L3 just like TTS. TTS outperforms tss+L3 by 7% and 18% on 1 and 8 threads (on Intel Xeon, for N=3200, the largest problem size), respectively. The results were similar for dsyrk. We made sure that tss+L3 does not cause load imbalance for a fair comparison (note that tss by default does not do load balancing and performs worse). This clearly demonstrates the improvement precisely earned via better prefetching due to leaving the innermost loop untiled. Our discussion on a mini-test in Section 2 also points to the benefit from prefetching.

7.2 Is TTS a fully compile-time tile size selection algorithm?

TTS as is, relies on the user to provide information about the problem size and datatype (and also the number of cores for multi-threaded codes), and is therefore not *fully* compile-time. However, like other popular libraries such as ATLAS, MKL, etc., we could insert conditional code in the kernels to handle the different cases (of problem sizes, datatypes and number of cores). This will make TTS a truly compile-time approach, but will lead to sub-optimal performance in some cases (as in popular libraries) since we cannot optimize for all cases in the source code.

8. RELATED WORK

There has been significant research done on the problem of tile size selection, hardware prefetching and software prefetching [2, 20], and a combination of the two [21, 24]. A lot of effort has been spent on hardware and software prefetching aimed at tuning prefetch distance by inserting software prefetches and minimizing the overhead of software prefetching. But, we do not know of a work which has exploited streaming patterns of data accesses through hardware prefetchers for tile size selection and data reuse in shared caches. As far as choosing tile sizes for shared caches is concerned, Bao et al. [4] have proposed a static technique that works in multi-programming environments to minimize interference among programs running on different cores. The authors in [33] and [1] have studied how prefetching affects tiling. [33] only considers software prefetching concluding that software prefetching does not benefit tiling. [1] shows that prefetching helps in tolerating conflict misses, but only for the L1 cache. Hyper-blocking [25] eliminates interference conflict misses by doing copy optimization in contiguous memory. Hyper-blocking can also benefit from data prefetching but this technique has overhead of copying and requires complex address calculations. Moreover, none of the production compilers automate data copying.

Tile size selection has been a very widely studied problem. Various analytical frameworks [26, 11, 18, 34, 38] exist that consider interactions between application programs and the host architecture to determine effective tile factors. However, there have been many programs for which these models have not proved to be useful. For example, some works [26, 11, 18] take into account only self-interference misses when considering the impact of problem size and cache geometries on effective tile sizes. They ignore cross-interference misses. On the other hand, [11] and [9] have included the effect of cross-interference misses, but they only consider direct-mapped caches with probabilistic models that result in sub-optimal results for pathological problem sizes. Rivera et al. [31] use array padding to eliminate this problem, but this work neither accounts for set-associativity of caches, nor for data reuse in multiple levels of cache.

Another domain through which tile size selection problem can be approached is *auto-tuning*. This approach involves an extensive search [7, 17, 39, 28] in a huge space of tile sizes at runtime. This usually require a large amount of time and resources to reach the best performing tile sizes. The most popular work in this area is the ATLAS library generator that produces best codes for BLAS libraries by doing extremely time-consuming search for all the available tile sizes. There have been many other approaches to improve the efficiency of auto-tuning frameworks. [37] have combined CHiLL [10] framework with Active Harmony [13] to reduce the search space. [41] have made the auto-tuning approach more flexible by applying tranformation-aware algorithms to modify the search space in the POET [42] framework. Recently, Ding et al. [14] have handled the complicated problem of input-sensitivity to deal with large optimization and complex input spaces. But, in spite of these modifications, the search space remains very large in existing auto-tuning frameworks. The authors in [29] have proposed a neural network approach that tries to capture the complex interaction between different parts of micro-architectures such as caches and hardware prefetchers. This approach has to run a training set that includes traversing a large search space.

Recently, Mehta et al. have proposed tss [23, 16] that considers data reuse in the L1 and L2 cache. However, tss does not consider the impact of hardware prefetchers or data reuse in the larger shared L3 cache and is therefore not scalable with increasing cores. TTS leads to scalable tiled codes.

9. CONCLUSION

In this work, we present a novel approach to leverage hardware prefetchers when tiling loops in modern multi-core processors with a large shared last level cache (LLC). Using this approach, instead of tiling for the higher-level caches like most existing tiling approaches, we tile for the LLC. This allows larger tile sizes to enhance data reuse in the LLC and expose streaming access patterns to hardware prefetchers that can bring data all the way to the L1 cache. Tiling for the LLC not only results in maximum data reuse to reduce off-chip memory accesses, but also simplifies the problem of tile size selection. We have developed a simpler and more effective tiling algorithm (TTS) based on this approach. Experimental results show that our strategy of combining tiling for the LLC and leveraging hardware prefetchers performs significantly better than state-of-the-art tiling algorithms. Our analysis also shows that the TTS algorithm performs particularly well for large problem sizes on multiple cores. This will be highly beneficial for scientific and big data applications in a high performance computing environment. Moreover, our algorithm can be readily integrated into most production compilers.

References

[1] E. Athanasaki, N. Koziris, and P. Tsanakas. A tile size selection analysis for blocked array layouts. In *INTERACT-2005. 9th Annual Workshop*, pages 70–80.

[2] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In ICS '01, pages 486–500.

[3] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In SC '12, pages 1–11, 2012.

[4] B. Bao and C. Ding. Defensive loop tiling for shared cache. In CGO '13, pages 1–11.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In PACT '04, pages 7–16.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '13*, pages 7–16, Juan-les-Pins, France, September 2004.

[7] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In ICS '97, pages 340–347.

[8] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. In L. Hendren, editor, *In CC'08*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. 2008.

[9] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In ICS '99, pages 492–499.

[10] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[11] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. *In PLDI'95*, 30(6):279–290.

[12] K. Cooper and J. Sandoval. Portable Techniques to Find Effective Memory Hierarchy Parameters. Technical report, 2011.

[13] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In SC '02, pages 1–11.

[14] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. P. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *In PLDI'15*, pages 379–390.

[15] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *In TOMS'90*, 16(1):1–17.

[16] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi- and many-core memory systems through microbenchmarking. *ACM Trans. Archit. Code Optim.*, 11(4):55:1–55:26, Jan. 2015.

[17] M. Frigo. A fast fourier transform compiler. In PLDI '99, pages 169–180.

[18] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In ICS '97, pages 317–324.

[19] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In ICS '12, pages 311–320, New York, NY, USA, 2012. ACM.

[20] D. Kim, S. S.-w. Liao, P. H. Wang, J. d. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In CGO '04.

[21] J. Lee, H. Kim, and R. Vuduc. When prefetching works, when it doesn't, and why. *In TACO'12*, 9(1):2:1–2:29.

[22] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In PPoPP '01, pages 103–112.

[23] S. Mehta, G. Beeraka, and P.-C. Yew. Tile size selection revisited. *In TACO'13*, 10(4):35:1–35:27.

[24] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew. Multi-stage coordinated prefetching for present-day processors. In ICS '14, pages 73–82.

[25] S. Moon and R. H. Saavedra. Hyperblocking: A data reorganization method to eliminate cache conflicts in tiled loop nests. Technical report, Conflicts in Tiled Loop Nests, USC-CS-98-671, USC Computer Science, 1998.

[26] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In ASPLOS-V'92, pages 62–73.

[27] L.-N. Pouchet. Polybench Benchmark Suite. Available at http://www\-roc.inria.fr/~pouchet/software/polybench/.

[28] A. Qasem, K. Kennedy, and J. M. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *In SC'06*, 36(2):183–196.

[29] M. Rahman, L.-N. Pouchet, and P. Sadayappan. Neural network assisted tile size selection. In *IWAPT '2010*.

[30] J. Reinders. *VTune performance analyzer essentials*.

[31] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In CC '99, pages 168–182.

[32] G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In SC'00.

[33] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *IPPS '96*, pages 39–45.

[34] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In CC'12, pages 101–121.

[35] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache accurate time skewing in iterative stencil computations. In ICPP '11, pages 571–581, Sept 2011.

[36] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In SPAA '11, pages 117–128, 2011.

[37] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In IPDPS '09, pages 1–12.

[38] F. G. Van Zee and R. A. van de Geijn. Blis: A framework for rapidly instantiating blas functionality. *TOMS'15*, 41(3):14:1–14:33.

[39] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *In Parallel Computing*, 27:3 – 35.

[40] M. Wolfe. More iteration space tiling. In SC '89, pages 655–664.

[41] Q. Yi and J. Guo. Extensive parameterization and tuning of architecture-sensitive optimizations. In *ICCS'11*, pages 2156–2165.

[42] Q. Yi, K. Seymour, H. You, R. W. Vuduc, and D. J. Quinlan. POET: parameterized optimizations for empirical tuning. In *IPDPS'07*, pages 1–8.