Analytical Bounds for Optimal Tile Size Selection

Jun Shirako[1], Kamal Sharma[1], Naznin Fauzia[2], Louis-Noël Pouchet[2], J. Ramanujam[3], P. Sadayappan[2], and Vivek Sarkar[1]

[1] Rice University {shirako,kamal.g.sharma,vsarkar}@rice.edu
[2] The Ohio State University {fauzia,pouchet,saday}@cse.ohio-state.edu
[3] Louisiana State University jxr@ece.lsu.edu

Abstract. In this paper, we introduce a novel approach to guide tile size selection by employing analytical models to limit empirical search within a subspace of the full search space. Two analytical models are used together: 1) an existing conservative model, based on the data footprint of a tile, which ignores intra-tile cache block replacement, and 2) an aggressive new model that assumes optimal cache block replacement within a tile. Experimental results on multiple platforms demonstrate the practical effectiveness of the approach by reducing the search space for the optimal tile size by 1,307× to 11,879× for an Intel Core-2-Quad system; 358× to 1,978× for an Intel Nehalem system; and 45× to 1,142× for an IBM Power7 system. The execution of rectangularly tiled code tuned by a search of the subspace identified by our model achieves speed-ups of up to 1.40× (Intel Core-2 Quad), 1.28× (Nehalem) and 1.19× (Power 7) relative to the best possible square tile sizes on these different processor architectures. We also demonstrate the integration of the analytical bounds with existing search optimization algorithms. Our approach not only reduces the total search time from Nelder-Mead Simplex and Parallel Rank Ordering methods by factors of up to 4.95× and 4.33×, respectively, but also finds better tile sizes that yield higher performance in tuned tiled code.

1 Introduction

Modern computer systems utilize multi-level memory hierarchies in which the latency of data access from higher levels are orders of magnitude higher than the time required to perform arithmetic operations. *Loop Tiling* [7, 17, 23, 29, 35, 36] is a classical technique to enhance data reuse in memory hierarchy levels close to the processor. Recent advances have made it possible to automatically generate parametrically tiled code, even for imperfectly nested loops [2, 15, 18, 24]. It is well known that the choice of tile sizes has a significant effect on performance, but the *effective selection of optimized tile sizes* remains an open problem that has become ever more challenging as processor memory hierarchies increase in complexity and depth.

Past work has pursued two main types of approaches for tile size selection, *analytical* and *empirical*. In analytical approaches, a compiler selects tile sizes based on static analysis of loop nests and known characteristics of the memory hierarchy. Although several analytical techniques for tile size selection have been proposed in the literature [8, 10, 13, 16, 19, 26–28], none has been demonstrated to be sufficiently effective for use in practice. As a result, the gap between the performance delivered by the best known tile sizes and those selected by an analytical approach has continued to widen, thereby diminishing the utility of past analytical approaches.

Empirical approaches to tile size optimization treat the loop nest as a black box, and perform empirical *auto-tuning* for a given architecture [4, 31–33] by actually executing the tiled code for a range of different tile sizes. The highly successful ATLAS (Automatically Tuned Linear Algebra Software) system [33] uses empirical tuning at library installation time to find the best tile sizes for different problem sizes on the target machine. One of the most challenging issues in empirical approaches for tile size selection is the enormous search space that must be explored when tiling multiple loops. As shown in many domains (e.g., by Goto and van de Geijn [14] for linear algebra and by Datta et al. [11] for stencil codes), the optimal tile has different tile sizes in different dimensions. The experimental results in this paper support this observation; compared with the best "square" tile, i.e., equal tile sizes along all dimensions, the best non-square tile showed performance speedups of up to 1.40, 1.28, and 1.19 on three different platforms (Xeon, Nehalem, and Power7, respectively). Though many empirical tuning systems attempt to reduce the search space by only examining square tiles, our results reaffirm the importance of including non-square tiles in the search space.

Hybrid approaches to tile size selection that combine analytical models and empirical search have also been pursued [9, 37]. For example, Chen et al. [9] introduced a framework that combines the use of compiler models and search heuristics to perform auto-tuning. However, we are not aware of any hybrid approach that has been demonstrated to be both broadly applicable and effective in practice. In this paper, we develop an analytical approach that is both broadly applicable as well as effectively usable in conjunction with various empirical search strategies for auto tuning.

Since the search spaces for tile size selection increase explosively for multidimensional, non-square and multi-level tiling, an effective approach to prune the search space is critical. Furthermore, while expensive empirical tuning is feasible for libraries such as BLAS that are tuned once per machine and reused across applications, tiled user codes usually require empirical search to be done much more rapidly since the search needs to be performed on all the time-consuming loop nests in the application.

In this paper, we introduce a novel approach using analytical bounds to limit the search space with empirical tuning for square and non-square tiling. As shown in Section 6, the proposed approach to pruning the search space is complementary to, and can be combined with, existing empirical search strategies; e.g., the analytical bounds can be integrated with existing auto-tuning frameworks such as ATLAS [33]. Experimental results show that our approach can reduce the search space by up to four orders of magnitude. Reduction factors of up to 11,879, 1,978, and 1,142 were realized on a Xeon, Nehalem, and Power7, respectively, for the loop nests that we studied.

Our approach employs a pair of analytical models to prune the search space — a conservative model that underestimates the number of iterations in an optimal tile (DL), and an optimistic model that overestimates the number of iterations in an optimal tile (ML). DL (Distinct Lines) [12], a conservative model from past work, models the required cache capacity for a tile as its total data footprint. Under this model, any tiles with a data footprint larger than the cache size are discarded, since they may incur capacity misses during execution.However, this is a pessimistic assumption for many applications, especially applications with streaming data accesses. We therefore introduce an optimistic analytical model, ML (Minimum working set Lines), that assumes an ideal intra-tile cache block replacement. Because DL and ML respectively provide lower and upper bounds for tile sizes, we can use them to bound tile size search space for empirical tuning. Our experiments show that this bounded search space still contains optimal tile sizes, despite reductions of up to four orders of magnitude in the size of the search space.

The paper is organized as follows. Section 2 reinforces the motivation for this work via a case study that highlights some of the challenges arising from modern memory hierarchies. In Section 3, we provide background on parametric tiling and the DL model from past work. Section 4 introduces the new ML model for single-level tiling. Section 5 elaborates on how the DL and ML models can be used to bound the search space for empirical tuning. Section 6 presents experimental results on three platforms using a number of benchmarks to demonstrate the effectiveness of the approach. Optimal tile sizes were always found within the reduced search space. Related work is discussed in Section 7, and we conclude in Section 8.

2 Motivation and Case Study

Past work on performance models for tile size selection were usually geared towards minimizing capacity and conflict misses for the first level of cache [10, 16, 34]. In this section, we illustrate the impact of higher levels of data cache and Translation Lookaside Buffer (TLB) on tile size selection. As a motivating example, we provide a detailed analysis of the execution of a tiled matrix-multiply kernel. Figure 1 is a sample code from [10] that uses the IKJ loop order.

```
// inter-tile loops
for ii = 1 to N, Ti
  for kk = 1 to N, Tk
    for jj = 1 to N, Tj
      // intra-tile loops
      for i = ii to min(ii+Ti,N)
        for k = kk to min(kk+Tk,N)
          for j = jj to min(jj+Tj,N)
             C[i][j] += A[i][k]*B[k][j];
```

Fig. 1: Tiled Matrix Multiply (IKJ loop order)

Tiling is critical in order to increase data locality in this case: *Ti*, *Tj* and *Tk* must be selected such that the data accessed during the computation of a tile fits entirely within the first level cache in order to avoid capacity misses. Reuse analysis suggests that we set *Ti* to *N*, in order to obtain full temporal reuse of the matrix `B` along the `i` loop [10]. This solution is motivated by the fact that no element of `B` will be used in two different tiles, an apparently ideal solution in terms of L1 cache misses (provided *Tj* and *Tk* are selected adequately). Furthermore, setting $Ti = N$ allows us to explore a two-dimensional search space $Tk \times Tj$ instead of a three-dimensional search space $Ti \times Tk \times Tj$, thereby significantly reducing the search space for optimal tile sizes.

This solution focuses only on minimizing L1 cache misses. To illustrate the deficiency of a Level 1 cache-centric approach, we examine performance variation on an

Intel *Xeon* (E7330 2.40 GHz processor with 32KB L1 cache, 3MB L2 cache, 16 entry TLB1, and 256 entry TLB2 (4KB page size)) for a problem size of 3000$\times$3000. After performing an exhaustive empirical search over tile sizes, we found the optimal tile size on this machine to be $(Ti, Tk, Tj) = (60, 10, 120)$. Note that this optimal point has unequal tile sizes in different dimensions because each dimension has a different data reuse patterns on arrays, and a large tile size is needed along the vectorized dimension (innermost tile size *Tj*) for effective vectorization. To illustrate the performance impact of the values of *Ti*, we fix $Tk = 10$ and $Tj = 120$, and plot four metrics — execution time, L1 data cache misses (L1_DCM), L2 data cache misses (L2_DCM), and L2 TLB misses (TLBM_DM) — for different values for *Ti* in Figure 2. Since the absolute values of these metrics are incomparable, the graph plots use standard min-max normalization to convert each metric to a value in the range $0 \ldots 1$. The normalized value for each metric is computed as the ratio, $(x - \min)/(\max - \min)$, where x, min and max are respectively the absolute, minimum, and maximum value of that metric for different values for *Ti*.

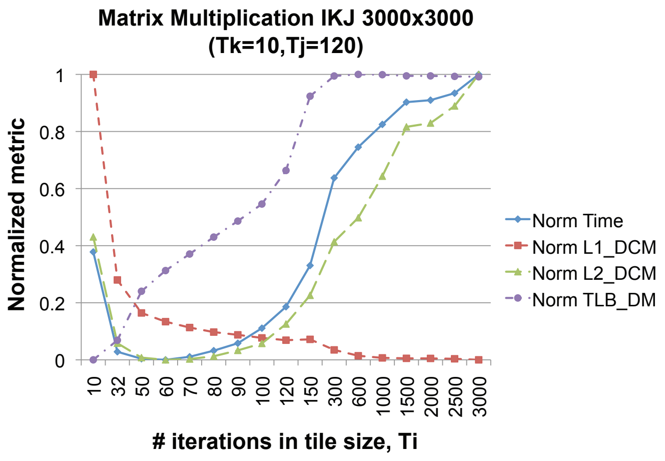

Fig. 2: Normalized Metrics for Matrix Multiplication with an IKJ Loop

First, we observe that L1 misses decrease as *Ti* increases, as expected. However, the graph clearly shows that the optimal tile size does not occur at $Ti = 3000$. At $Ti > 90$, we see an upward trend in execution time, in contrast to what is suggested by considering just the L1 cache. As the cache footprint of the computation increases, TLB2 misses and L2 misses increase, eventually leading to substantial degradation in execution time.

To better understand this effect, one must also consider the virtual-to-physical address translation for this machine. A lookup for address translation is performed at both the TLB levels. When an entry is not found in eiher TLB, a radix tree page walk is performed, with higher (leftmost) bits fetched from the MMU Cache and lower (rightmost) bits from the L2 data cache. When address translation bits are not found in the L2 cache, it causes DRAM accesses, leading to significant stalls for an application. Thus, increasing an application's footprint causes significant pressure on the data cache and TLB. This page walk behavior is not just restricted to Intel architectures. AMD's Page Walking Cache and PowerPC's Hashed Table approach exhibit a similar characteristic, where address translation requires traversal of data caches in the case of TLB misses [1,3].

These observations led us to rethink the tile size selection model. While it is important to optimize for L1 data cache to enable higher reuse, one also needs to consider the reuse effect at higher levels of the memory hierarchy. This leads to considering multiple memory hierarchy levels when searching the $Ti \times Tk \times Tj$ space to find the best tile size for even a single level of tiling. A key contribution of this paper is the development of analytical models to bound the space of candidate tile sizes that take into account multi-level data caches and TLBs. The results in Section 6 demonstrate the following: (1) our analytical models significantly reduced the search space size, while preserving the optimal points in the pruned space; (2) in most cases, the optimal points have non-square tile sizes with performance improvements of up to 40% over square tiles; (3) the developed model also effectively finds optimized tile sizes for parallel tiled code.

3 Background

3.1 DL: Distinct Lines

The DL (Distinct Lines) model was designed to estimate the number of distinct cache lines accessed in a loop-nest [12, 27]. Consider a reference to a contiguously allocated m-dimensional array, A, enclosed in n perfectly nested loops, with index variables $i_1, \cdots, i_n$:

$A\,(f_1(i_1,\cdots,i_n),\cdots,f_m(i_1,\cdots,i_n)) \quad (Fortran)$

$A[f_m(i_1,\cdots,i_n)]\cdots[f_1(i_1,\cdots,i_n)] \quad (C)$,

where $f_j(i_1,\cdots,i_n)$ is an affine function. An exact analysis to compute DL is only performed for array references in which all coefficients are compile-time constants (i.e., for affine references). An upper bound for the number of distinct lines accessed by a single array reference [12] with one-dimensional subscript expression $f(i_1,\cdots,i_n)$ is

$$DL(f) \leq \min\left(\frac{(f^{hi}-f^{lo})}{g}+1, \left\lceil\frac{(f^{hi}-f^{lo})}{L}\right\rceil+1\right),$$

where g is the greatest common divisor of the coefficients of the enclosing loop indices in f, and L is the cache line size in units of array element size; f^{hi} and f^{lo} are the maximum and minimum values of the subscript expression f across the entire loop nest. In practice, the relative error of this estimation is small when, as is usually the case, the range $(f^{hi}-f^{lo})$ is much larger than the values of the individual coefficients of f. For a multidimensional array reference $A\,(f_1,\cdots,f_m)$, the upper bound estimate [12] is as follows (with a heuristic assumption that the first dimension of the array has at least L elements).

$$DL(f_1,\cdots,f_m) = DL(f_1) \times \prod_{j=2}^{m}\left(\frac{(f_j^{hi}-f_j^{lo})}{g_j}+1\right).$$

Extensions of this model to account for multiple array accesses in a loop nest have also been developed [12,27]. These DL definitions for an entire loop nest are also applicable to a tile whose loop boundaries are expressed using tile sizes. In such a case, the DL definition is a symbolic function of tile sizes $t_1,\cdots,t_n$ denoted by $DL(t_1,\cdots,t_n)$ [27].

The DL definition can also be applied to any level of cache or TLB by selecting its cache line size or page size as L. However, the DL model ignores possible replacement of cache lines within a tile, and therefore provides only conservative upper bounds for the number of cache lines needed.

3.2 Parametric Tiling

Although production compilers today may have limited tiling capability, there have been significant recent advances in automatic source-to-source transformations for tiling and several systems for parametric tiling have been developed and made publicly available such as TLOG [24], HITLOG [18] and PrimeTile [15]. With such tiled-code generators, it is now possible to generate tiled code for compute-intensive inner kernels (including imperfectly nested loops), that can be tuned to the cache characteristics of the target platform. Thus, just as ATLAS [33] is used in auto-tuning dense linear algebra codes, it becomes feasible to use auto-tuning for user kernels such as stencil-based computations. However, unlike library kernel optimization, exhaustive search of the tile parameter space over several hours to days is generally not attractive for tuning user kernels. This motivates the approach developed in this paper to significantly prune the search space.

4 ML: Minimum Working Set Lines

We now introduce ML (Minimum working set Lines), a new analytical cost model based on the cache capacity required for a tile when intra-tile reuse is taken into account. We define the ML model next and then develop an approach to computing ML for a tile by first constructing a special sub-tile based on analysis of reuse characteristics and then computing the DL value for that sub-tile. Although we mainly focus on cache capacity in this section, the model is directly applicable to TLBs by replacing the cache line size with the page size.

4.1 Operational Definition of ML

The essential idea behind the ML model is to develop an estimate of the minimum cache capacity needed to execute a tile without incurring any capacity misses; this minimum cache capacity can be viewed as the minimum working set size for the tile. Consider a memory access trace of the execution of a single tile, run through an idealized simulation of a fully associative cache. The cache is idealized in that its size is "unbounded" (i.e., any access to a data element in the tile will never lead to a capacity miss) and an optimal replacement policy, where a line in the cache is marked for replacement as soon as the last reference to any data on that line has been issued (through an oracle that can determine all future references of the tile). Before each memory access, the simulator fetches the desired line into the idealized cache if needed. After each memory access, the simulator evicts the cache line if it is the last access (according to the oracle). *ML corresponds to the maximum number of lines (high water mark) held in this idealized cache during the execution of the entire trace for the tile*.

4.2 Model of Computation

In this paper, we focus on the class of affine imperfectly nested loops, where loop bounds and array access expressions are affine functions of the surrounding loop iterators and program constants. For this class of program, it is possible to restructure the code automatically [6, 15] to expose rectangular tiles of parametric size. Assuming that a system such as PrimeTile [15] has already been used to generate parametric rectangularly tiled code, we focus on the problem of tile size optimization for such codes.

The outermost loop inside a tile (i.e., the outermost intra-tile loop) is denoted by $loop_1$, and the innermost intra-tile loop is $loop_n$. Since tiles are rectangular by construction, $loop_i$ ($1 \leq i \leq n$) has the same trip count for any of its executions. Note that the case of partial tiles is handled with prolog/epilog code [15]. The iteration domain of a tile is represented with a tuple $[T_1;\ T_2;\ \cdots;\ T_n]$. A tile is surrounded by the loops iterating on all tiles, i.e., the inter-tile loops. In this paper, we assume a single level of tiling; extension to multi-level tiling is a subject for future work.

4.3 Distance in Tiled Iteration Space

A specific instance of the loop body is identified by an iteration vector, that is, a coordinate in the iteration space, noted $p = (p_1, p_2, \cdots, p_n)$. The distance between two iteration vectors p and p' is expressed as the distance vector $d = (d_1, d_2, \cdots, d_n) = p' - p$. The *scalar distance* between two iteration vectors is the number of instances of the inner-most loop body to be executed (in lexicographic ordering) between these two iteration vectors. For instance, in a tiled loop nest the scalar distance between two consecutive iterations of the innermost loop $loop_n$ is 1, representing the shortest possible scalar distance. It corresponds to the distance vector $(0, \cdots, 0, 1) = (p_1, \cdots, p_{n-1}, p_n + 1) - (p_1, \cdots, p_{n-1}, p_n)$, for any values of $p_1, \cdots, p_n$. It is always possible to find a sub-tile tuple that corresponds to this scalar distance. Here it is $[1;\ 1;\ \cdots;\ 1]$, i.e., the tuple describing an n-dimensional rectangle containing exactly one point. We call this form a sub-tile tuple expression of the tile tuple $[T_1;\ T_2;\ \cdots;\ T_n]$.

Let us now consider the distance vector $(1,2,3) = (p_1+1, p_2+2, p_3+3) - (p_1, p_2, p_3)$. In general, to compute the associated scalar distance, we first compute the scalar distance corresponding to two consecutive iterations for each of the intra-tile loops: 1 for $loop_3$, T_3 for $loop_2$, and T_2T_3 for $loop_1$. The scalar distance is computed by the dot product $(1,2,3).(T_2T_3, T_3, 1)^t = T_2T_3 + 2T_3 + 3$. It is always possible to compute an associated sub-tile tuple expression corresponding to the sub-part of the tile iteration domain bounded by the two iteration points p and p', by combining multiple rectangles defined by individual sub-tile tuples. Here, $[1;\ T_2;\ T_3] + [1;\ 2;\ T_3] + [1;\ 1;\ 3]$ is the associated sub-tile tuple expression.

One iteration of $loop_i$ strides over $size_i$, which is the total number of iteration points within the loop body of $loop_i$. $size_i$ is defined in both scalar and sub-tile tuple expression as follows:

$size_n = 1 = [1;\ 1;\ \cdots;\ 1] \quad (i = n);$

$size_i = \prod_{j=i+1}^{n} T_j = [1;\ \cdots;\ 1;\ T_{i+1};\ \cdots, T_n] \quad (i < n).$

Using size vector $size = (size_1, size_2, \cdots, size_n)$, we define the *scalar distance* of the distance vector $d = (d_1, d_2, \cdots, d_n)$ and its sub-tile tuple expression as follows:

Scalar distance: $size \cdot d = \sum_{i=1}^{n} (d_i \times size_i)$;

Sub-tile tuple: $\sum_{i=1}^{n} ([1;\ \cdots;\ 1;\ d_i;\ T_{i+1};\ \cdots;\ T_n])$.

4.4 Temporal and Spatial Reuse Distance

Temporal and spatial data reuse are expressed using widely used definitions of "reuse distance vectors" (often shortened to "reuse vectors") [34]. A number of previous efforts introduced methods to compute spatial reuse vectors [34] while temporal reuse vectors are computed from standard dependence analysis.

```
for (p_1 = [low1 : low1+T_1-1])
  for (p_2 = [low2 : low2+T_2-1])
    for (p_3 = [low3 : low3+T_3-1])
      B[p_1][p_2][p_3] = A[p_1][p_3] + B[p_1-2][p_2-3][p_3];
```

Fig. 3: Sample Code

For array A in Figure 3, the pair $((p_1, p_2, p_3), (p_1, p_2+1, p_3))$ has temporal reuse with a reuse distance vector of $d1 = (0,1,0)$. The pair $((p_1, p_2, p_3), (p_1, p_2, p_3+1))$ has spatial reuse in array A with a reuse distance vector of $d2 = (0,0,1)$. For array B, the pair $((p_1, p_2, p_3), (p_1+2, p_2+3, p_3))$ has a temporal reuse vector $d3 = (2,3,0)$, and the pair $((p_1, p_2, p_3), (p_1, p_2, p_3+1))$ has a spatial reuse vector $d4 = (0,0,1)$.

In Section 4.3, we defined the *scalar distance* for these reuse distance vectors. For Figure 3, the size vector is $size = (T_2T_3, T_3, 1)$. The scalar distance of temporal reuse vector $d1 = (0,1,0)$ is calculated by $size \cdot d1 = T_3$, and the scalar distance of spatial reuse vector $d2 = (0,0,1)$ is $size \cdot d2 = 1$. They are also represented as sub-tile tuple expressions: $[1;\ 1;\ T_3]$ and $[1;\ 1;\ 1]$, respectively. Finally, we call the largest scalar distance Maximum Reuse Distance, or MRD; the MRD is defined on a per-array basis. For example, the MRD for array A in Figure 3 is $[1;\ 1;\ T_3]$, and MRD for array B is $[2;\ T_2;\ T_3] + [1;\ 3;\ T_3]$ due to $d3$.

The approximation of reuse distance vectors for non-uniform reuse patterns is still an open question. As described in Section 5, ML is used to derive the upper bounds of tile sizes and conservative approximation may make the tile size boundaries smaller than optimal points. Therefore, we ignore non-uniform reuse so as to estimate tile sizes optimistically, and leave approximation of non-uniform reuse distance to future work.

4.5 Computation of ML

Using Maximum Reuse Distance for array X, we define ML for array X as follows. First, a pair of iteration instances $(p_1, p_2, \cdots, p_n)$ and $(p_1+d_1, p_2+d_2, \cdots, p_n+d_n)$ has the following Maximum Reuse Distance for array X:

$MRD_X = \sum_{i=1}^{n}([1;\ \cdots;\ 1;\ d_i;\ T_{i+1};\ \cdots;\ T_n])$.

In order to exploit all data locality related to array X, the data at $(p_1, p_2, \cdots, p_n)$ must not be removed from the cache memory $(p_1+d_1, p_2+d_2, \cdots, p_n+d_n)$ is accessed. The cache must keep all the distinct cache lines for array X within the distance of MRD_X. Since the MRD is also represented as a sub-tile tuple, the ML for array X is equivalent to the DL of X for the sub-tile tuple defined by MRD_X. Thus we have

$ML_X = DL_X(MRD_X)$.

To compute DL of a sum of sub-tile tuples, we compute the sum of DL of each individual sub-tile tuple. As shown above, the expression of Maximum Reuse Distance MRD_X does not include T_1, therefore, ML_X is also independent of T_1. This is true for any reference: the sub-tile tuple never contains T_1 in its components, as shown in Section 4.3. For instance, MRD_B and ML_B for array B of Figure 3 are as follows.

$MRD_B = [2;\ T_2;\ T_3] + [1;\ 3;\ T_3]$;

$ML_B = DL_B(MRD_B) = DL_B(2, T_2, T_3) + DL_B(1, 3, T_3)$.
ML is defined as the sum of the ML_X values for each array X accessed in the tile:
$ML = \sum_X (ML_X)$.
In order to leverage all intra-tile data locality, we should select tile sizes so that ML is smaller or equal to the number of cache lines of the target cache memory, which is usually level-1 cache.

4.6 Example

Figure 1 from Section 2 shows a single-level tiling example for Matrix Multiplication. We assume an element of array has 8 Bytes, and the cache line size of L1/L2 is 64 Bytes (a cache line contains eight elements). DL is calculated as follows:
$DL = DL_C(Ti, Tk, Tj) + DL_A(Ti, Tk, Tj) + DL_B(Ti, Tk, Tj) = Ti \left\lceil \frac{Tj}{8} \right\rceil + Ti \left\lceil \frac{Tk}{8} \right\rceil + Tk \left\lceil \frac{Tj}{8} \right\rceil$.
Also, the MRD for each array is computed from size vector $size = (TkTj,\ Tj,\ 1)$ and reuse distance vectors. $MRD_C = (TkTj,\ Tj,\ 1) \cdot (0,1,0) = [1;\ 1;\ Tj]$, $MRD_A = (TkTj,\ Tj,\ 1) \cdot (0,1,0) = [1;\ 1;\ Tj]$, and $MRD_B = (TkTj,\ Tj,\ 1) \cdot (1,0,0) = [1;\ Tk;\ Tj]$. Assigning each MRD to the corresponding DL expression, the ML for single-level tiling is computed as
$ML = DL_C(1, 1, Tj) + DL_A(1, 1, Tj) + DL_B(1, Tk, Tj) = \left\lceil \frac{Tj}{8} \right\rceil + 1 + Tk \left\lceil \frac{Tj}{8} \right\rceil$.

5 Bounding the Search Space by using DL and ML

This section presents how our DL/ML model bounds a tiling search space. As discussed in Section 4, ML is used for optimistic cache and TLB capacity constraints for intra-tile data reuse and gives the upper boundaries for estimated tile sizes. In contrast, DL is used for conservative constraints, and gives the lower boundaries. These lower and upper boundaries drastically reduce the search space for single and multi-level tiling. Furthermore, DL, which represents the number of distinct lines within a tile, can be used as a capacity constraint for inter-tile data reuse on higher levels of cache/TLB. The extension of our approach to multi-level tiling is the subject for future work.

5.1 Capacity Constraint for Intra-tile Reuse

Section 4.5 shows ML for single-level tiling can be dependent on tile sizes T_2, T_3, ..., T_n and is independent on T_1 while DL can depend on all tile sizes T_1, T_2, ..., T_n. CS_1 represents the number of cache lines or TLB entries at level-1 cache or TLB memory. All tile sizes within the lower boundaries due to DL and upper boundaries due to ML satisfy the following constraints.
$DL(T_1, T_2, \cdots, T_n) \geq CS_1, \quad ML(T_2, T_3, \cdots, T_n) \leq CS_1$.
We have two bounded regions according to cache and TLB. In our approach, we consider the union of both the regions as candidates for optimal tile sizes, e.g., a point that is within the DL/ML region due to cache but outside the region due to TLB is also a candidate.

5.2 Capacity Constraint for Inter-tile Reuse

Although Section 5.1 shows the boundaries to maximize intra-tile data reuse of level-1 tile, the outermost tile size T_1 is actually not bounded above by the ML constraint. As discussed in Section 2, this corresponds to traditional single-level tiling to fit within single-level cache, where the outermost loop is not tiled [8, 10, 20]. However, the outermost tile size affects inter-tile reuse on higher levels of cache/TLB, and too large tile size would harm the inter-tile data locality and even the overall performance. Using DL definition, we define an additional capacity constraint in order to preserve inter-tile data reuse on level-k ($k > 1$) cache/TLB as follows:

$DL(T_1, T_2, \cdots, T_n) \le CS_k$.

This inequality, which ensures that whole distinct lines within the tile can be kept on level-k cache/TLB and guarantees the inter-tile data reuse, bounds the outermost tile size T_1. It is a subject for future work to select the suitable k according to the target system. In the experiments in Section 6, we select the highest level of cache/TLB as k.

5.3 Empirical Search within Bounded Search Space for Single-level Tiling

Described in previous section, DL/ML capacity constraints for single-level tile consist of the following three conditions.

$DL(T_1, T_2, \cdots, T_n) \ge CS_1$ (lower boundary for intra-tile reuse);

$ML(T_2, T_3, \cdots, T_n) \le CS_1$ (upper boundary for intra-tile reuse);

$DL(T_1, T_2, \cdots, T_n) \le CS_k$ (upper boundary for inter-tile reuse).

Empirical search finds the optimal tile sizes for $T_1, T_2, \cdots, T_n$ that minimizes the objective metrics such as execution time.

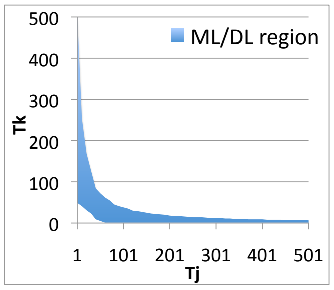

Fig. 4: Search Space for Matrix Multiplication for $Ti = 60$

Let us calculate the search space of Figure 1, which is a single-level tiling example of Matrix Multiplication. We assume the same experimental platform and program size as Section 2 and Section 4.6; L1/L2 cache contains 512/49152 lines respectively. [4] The capacity constraints: $DL = Ti\left\lceil\frac{Tj}{8}\right\rceil + Ti\left\lceil\frac{Tk}{8}\right\rceil + Tk\left\lceil\frac{Tj}{8}\right\rceil \ge 512$,

$ML = \left\lceil\frac{Tj}{8}\right\rceil + 1 + Tk\left\lceil\frac{Tj}{8}\right\rceil \le 512$, and $DL = Ti\left\lceil\frac{Tj}{8}\right\rceil + Ti\left\lceil\frac{Tk}{8}\right\rceil + Tk\left\lceil\frac{Tj}{8}\right\rceil \le 49152$.

Figure 4 shows the bounded 2-D search space for Tk and Tj when Ti is 60, which is much smaller than the original 2-D search space 3000×3000, and the optimal tile size $Ti = 60$, $Tk = 10$, and $Tj = 120$ is found within the bounded region.

[4] We omit details on TLB constraints due to space limitations.

5.4 Compiler Pass for Bounded Search Space

Figure 5 shows the compiler framework to implement the DL-ML bounded search space algorithm. This implementation requires standard compiler tools, such as dependence vector computation and array index expression extraction, readily available in most modern compilers. This is the only program-specific data required to compute the DL-ML equations. Plugging the additional machine-specific information about the different cache level sizes and associated line sizes results in a bounded search space of candidate tile sizes, which is drastically smaller than the original set of candidates. Using these bounds, a tile size tuning framework explores only a fraction of points in the original search space, thereby considerably reducing the tuning overhead.

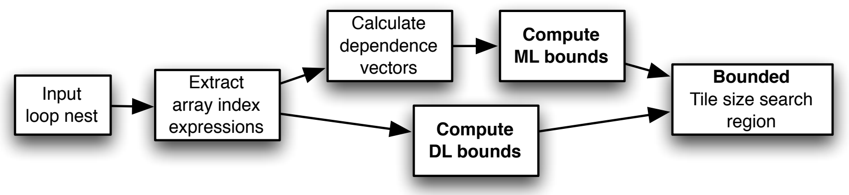
Input loop nest
Extract array index expressions
Calculate dependence vectors
Compute ML bounds
Compute DL bounds
Bounded Tile size search region

Fig. 5: Compiler Implementation of DL-ML Bounding

6 Experimental Results

An experimental assessment was performed on three Linux-based systems: an Intel Core i7 920 running at 2.66 GHz with shared L3 cache (labeled *Nehalem*), an IBM Power 7 running at 3.55 GHz (*Power7*), and an Intel Xeon E7330 running at 2.40GHz with shared L2 cache (*Xeon*). Previous work has used published cache capacity data from manufacturers in analytical models for cache performance. However, due to factors such as page table entries, OS processes, etc., the full capacity of higher level caches may not be actually available for use by the application. We report in Table 1 the effective capacities for the cache and TLB — the published capacity Spec versus the effective capacity Effective that was observed using micro-benchmarks for hardware characterization [25]. It may be seen that the effective capacity may be as low as half the documented size, which can affect the DL/ML capacity constraints. In our experiments, we used the effective capacities. The impact of using published capacities instead of effective capacities is studied in Section 6.2.

Table 1: Cache characteristics of the architectures considered

	L1		L2		L3		Line Size	TLB1	TLB2	Page Size
	Spec.	Effective.	Spec.	Effective.	Spec.	Effective.		Entries	Entries	
Nehalem	32kB	32kB	256kB	256kB	8MB	5.2MB	64B	64	512	4kB
Power7	32kB	32kB	256kB	256kB	32MB	18.4MB	128B	64	512	64kB
Xeon	32kB	32kB	3MB	1.5MB	N/A	-	64B	16	256	4kB

We studied five benchmarks using double precision floating point arithmetic. matmult is a standard matrix-multiply kernel: $C = A.B$; dsyrk is a symmetric rank 1 computation: $C = \alpha.A.A^T + \beta.C$; and dtrmm is a triangular in-place matrix multiplication: $B = \alpha.A.B$ (A is triangular). We also considered a representative 9-point two-dimensional stencil computation, 2d-jacobi, and a 2D Finite Difference Time Domain method, 2d-fdtd. Parametrically tiled code for each benchmark was generated using the publicly available PrimeTile code generator [15] after any necessary preprocessing such as skewing [6] to ensure that rectangular tiling of the loops was legal. For all tested versions, including the original code, the same compiler optimization flags were used: for *Nehalem* and *Xeon*, we used Intel ICC 11.0 with option `-O3`; for *Power7*, we used IBM XLC 10.1 with option `-O3`.

6.1 Performance Distribution of Different Tile Sizes

For each benchmark, in the case of single-level tiling, we conducted an extensive set of experiments, for a subset of tile sizes for each loop ranging from 1 to the loop length, in steps of 10 (approximately). Figure 6a plots the data for matmult (size 3000×3000) for the three considered architectures. A point (x,y) in this cumulative plot indicates that x% of the tile combinations achieved normalized performance greater than or equal to y, where normalization is with respect to the best performing case among all runs and performance is inversely proportional to execution time.

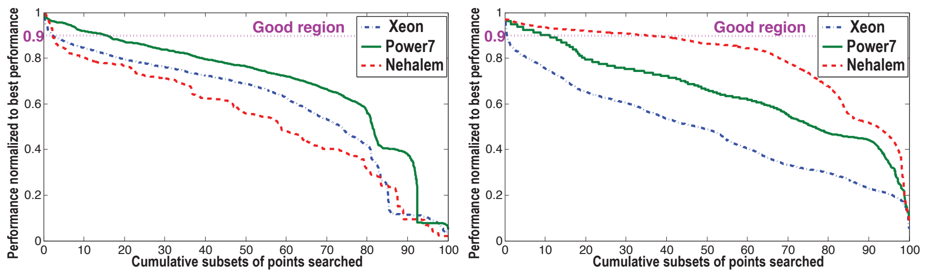

Fig. 6: Performance Distribution for (a) matmult-3000x3000 and (b) 2d-jacobi-50x4000x4000 on *Nehalem*, *Xeon*, and *Power7*

It may be observed that only a small fraction of the tile combinations achieve very good performance — for example, on the *Nehalem*, only 2% of the tile size configurations achieve more than 90% of the maximal performance. Also, there is a very large variation in performance between the best and worst tile size choices, up to a factor of 10. The performance distribution also varies for different targets — for *Power7*, over 20% of the cases provide good performance. Further, we have also observed that the points with good performance are not uniformly distributed in the search space but are clustered in clouds. This highlights the complexity of the search problem when using a blind random search — convergence towards an optimal point may require sampling of a significant fraction of the search space.

Figure 6b shows a similar analysis for the 2d-jacobi benchmark. For the target machines, we observe quite a different trend compared to matmult: about 55% of the tile sizes achieve 90% or more of the maximal performance for *Nehalem*, while this ratio significantly decreases for the two other architectures, down to 1% for *Xeon*.

6.2 Search Space Reduction by DL/ML Model

To assess the effectiveness of search space pruning by use of the DL/ML model, Figures 7-11 show the bounded search region superposed with a marking of all tile choices that achieve over 95% of the maximal performance on *Nehalem* and *Power7*. [5] In each 3-D space, the x, y, and z axes show tile size values for the outer loop, middle loop and inner loop respectively. These tile choices are called "best" points in this section. The surface in each 3-dimensional plot represents the DL/ML upper boundary for single-level tiling, considering intra-tile reuse for level-1 cache and TLB, and inter-tile reuse on the highest level of cache and TLB, as described in Section 5. In order to enhance viewability, the figures do not show the lower DL boundaries, since they fall below the best points.

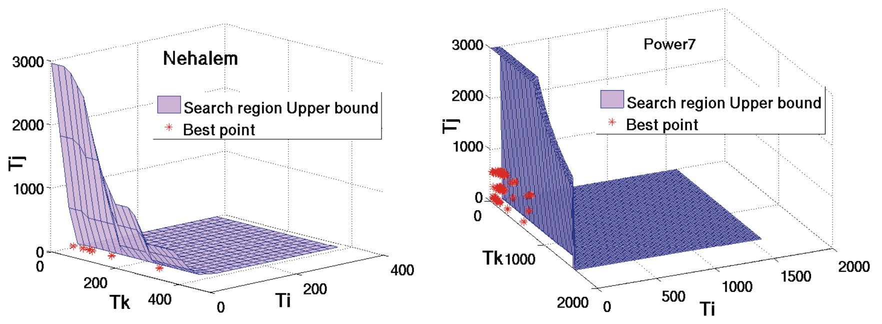

Fig. 7: Best tile sizes (within 95% and more of the optimal) for matmult-3000x3000 with k-i-j loop ordering on 7a *Nehalem* and 7b *Power7*.

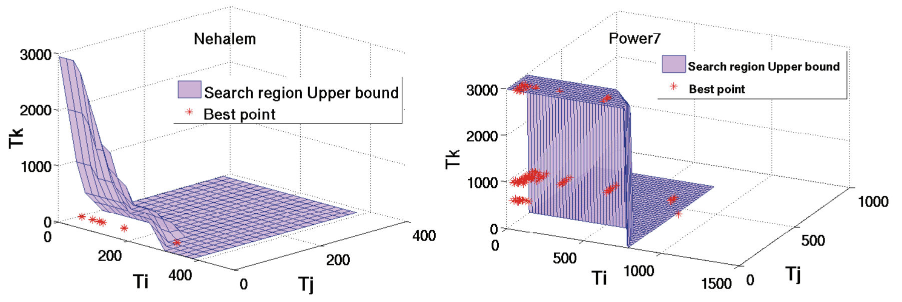

Fig. 8: Best tile sizes (within 95% and more of the optimal) for dsyrk-3000x3000 with i-j-k loop ordering on 8a *Nehalem* and 8b *Power7*.

For all the plots in Figures 7-11, we see that although a small number of points lie outside the bounded search space, the vast majority of best points lie inside it. The density of good solutions in the space is thus very much larger than in the non-pruned space. For all the benchmarks, we found that an optimal tile was within the bounded search region. Figures 7-11 also show that the best tile sizes are relatively smaller on

[5] Data for *Xeon* are not included due to space limitations.

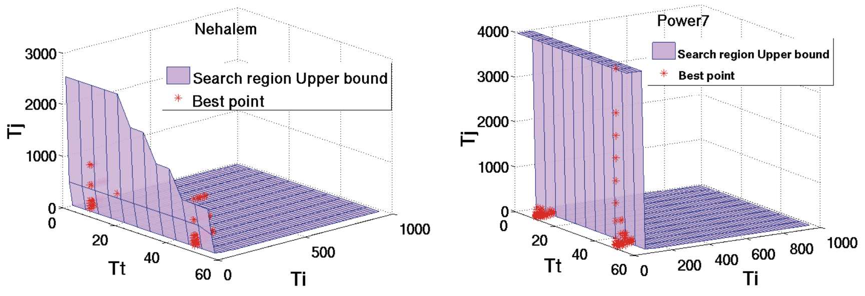

Fig. 9: Best tile sizes (within 95% and more of the optimal) for 2d-jacobi-50x4000x4000 with *t*-*i*-*j* loop ordering on 9a *Nehalem* and 9b *Power7*.

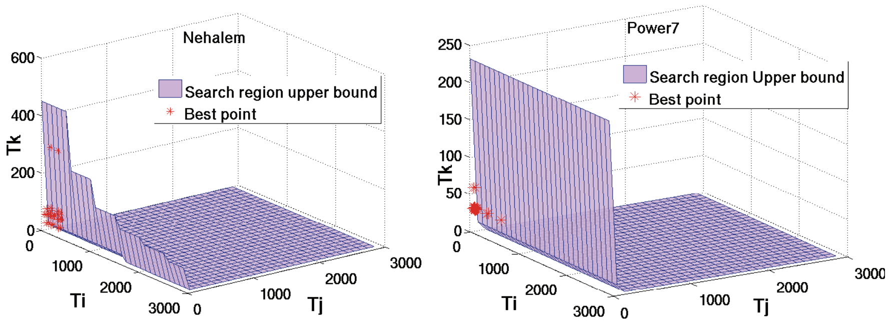

Fig. 10: Best tile sizes (within 95% and more of the optimal) for dtrmm-3000x3000 with *i*-*j*-*k* loop ordering on 10a *Nehalem* and 10b *Power7*.

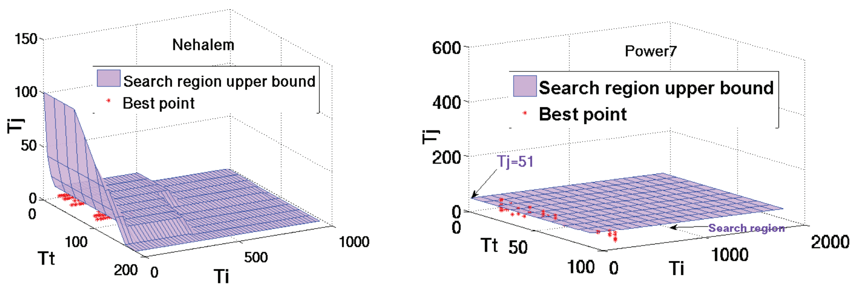

Fig. 11: Best tile sizes (within 95% and more of the optimal) for 2d-fdtd-100x2000x2000 with *t*-*i*-*j* loop ordering on 11a *Nehalem* and 11b *Power7*.

Nehalem, and larger on *Power7*. For example, the best points in dsyrk are within the region of (`Ti` $\leq$ 400, `Tj` $\leq$ 50, `Tk` $\leq$ 100) on both *Nehalem* and *Xeon*. However, the best points on *Power7* are distributed much more broadly, up to the maximum size of 3000. This trend pertains to the impact of the level-1 TLB size on each system; the small (4KB) page size on *Nehalem* and *Xeon* causes the best tile sizes to be small, while the large (64KB) page size on *Power7* allows much larger tiles without causing severe TLB misses. These differences are directly reflected in the upper boundary of the DL/ML model, which covers a larger region for *Power7* than the other machines. As discussed in Section 5.2, the outermost tile size (x axis) is bounded above only by the inter-tile reuse constraints due to the highest level of cache and TLB. It is obvious that the outermost tile size boundaries also contribute to search space reduction in Figures 7-11.

Table 2 shows the ratio of the space considered in the DL/ML range for the three architectures, compared to the full space of tile sizes. This ratio corresponds to the minimal acceleration factor for an exhaustive or random empirical search compared to using the full space. The factor is much lower for *Power7* due to the larger page size, as explained above. In order to assess the impact of using published versus effective capacities, we repeated our analysis also using the published capacity for the highest level of cache instead of the effective capacity. We found the reduction in search space by use of the ML model was virtually identical with use of published size or effective size (Table 2), with one exception: for matmult with 3000×3000 size on *Power7*, the reduction rate decreased from 93.79 to 84.01. This is because the constraint due to the highest level of cache did not affect the search space boundaries for the evaluated benchmarks and platforms except for Power7/matmult.

Table 2: Search space reduction factor across different architectures

	Problem Size	Xeon	Power 7	Nehalem
matmul	600x600	81.12	1.46	21.90
	3000x3000	8710.81	93.79	1856.49
dsyrk	1000x1000	492.24	2.04	91.62
	3000x3000	11879.67	83.99	1978.26
dtrmm	600x600	41.37	2.31	32.74
	3000x3000	2565.00	1142.23	1238.24
2d-jacobi	50x4000	3102.90	76.43	693.45
2d-fdtd	100x2000x2000	1307.19	45.55	358.74

6.3 Summary of Experiments

1-Level Tiling We summarize our experiments for 1-level tiling in Table 3. We report, for each benchmark and each architecture, the execution time (in seconds) of the original, untiled code in the Untiled Time column. DL reports the tile sizes and its execution time as obtained by the purely analytical approach using the DL model [12]; Best Square Tile reports the tile sizes and execution time obtained by an exhaustive empirical search only for square tile sizes; and Best DL/ML is obtained by an exhaustive empirical search in the DL/ML range. The Best DL/ML point was also the globally optimal point in the whole search space for all programs/platforms. We observed that the optimal points represent non-square tile sizes for all cases. For efficient vectorization on all three platforms, the vectorized dimension should correspond to a sufficiently large tile size. Furthermore, the different temporal/spatial data reuse pattern along different dimensions contributes to the unequal sizes of tiles in the different dimensions for the optimal choices.

Empirical search using DL/ML model This section demonstrates the integration of the analytical bounds with existing search optimization algorithms, the Nelder-Mead Simplex method [22] and the Parallel Rank Ordering (PRO) method [30]. In order to

Table 3: 1-level Tiling Results (Time in seconds, N: Nehalem, P: Power7, X: Xeon)

	Untiled	DL		Best Square Tile		Best DL/ML		Impr. by DL/ML	
	Time	Tile Size	Time	Tile size	Time	Tile size	Time	vs. DL	vs. Sq.
matmult-N	33.25	(40, 40, 30)	16.40	(80,80,80)	17.27	(150, 30, 80)	13.48	1.22×	1.28×
matmult-P	25.46	(50, 30, 20)	13.90	(80,80,80)	12.28	(90, 10, 120)	10.60	1.31×	1.15×
matmult-X	153.66	(40, 40, 30)	29.51	(50,50,50)	23.98	(100, 20, 120)	18.35	1.60×	1.31×
dsyrk-N	25.39	(30, 40, 40)	15.47	(80,80,80)	15.54	(30, 30, 90)	12.50	1.23×	1.24×
dsyrk-P	23.32	(40, 30, 30)	15.10	(300,300,300)	10.86	(60, 10, 1000)	9.16	1.64×	1.19×
dsyrk-X	84.89	(30, 40, 40)	26.08	(120,120,120)	25.44	(100, 30, 80)	18.19	1.43×	1.40×
dtrmm-N	142.42	(40, 40, 30)	19.20	(60,60,60)	18.87	(150, 30, 60)	18.20	1.05×	1.04×
dtrmm-P	62.74	(30, 50, 20)	14.60	(60,60,60)	13.06	(600, 30, 32)	11.96	1.22×	1.09×
dtrmm-X	114.70	(40, 40, 30)	28.98	(120,120,120)	29.13	(30, 10, 120)	23.49	1.23×	1.24×
2d-jacobi-N	2.43	(10, 40, 10)	2.60	(50,50,50)	2.24	(10, 8, 150)	2.16	1.20×	1.04×
2d-jacobi-P	2.10	(10, 40, 10)	2.09	(10,50,50)	1.31	(10, 40, 120)	1.19	1.76×	1.10×
2d-jacobi-X	8.75	(10, 40, 10)	2.77	(10,8,8)	2.81	(50, 40, 20)	2.54	1.09×	1.11×
2d-fdtd-N	15.35	(10, 60, 8)	2.41	(50, 8, 8)	2.35	(50, 50, 8)	2.26	1.07×	1.04×
2d-fdtd-P	9.56	(10, 40, 1)	6.90	(50,70,70)	2.11	(40,70,40)	2.09	3.30×	1.01×
2d-fdtd-X	16.42	(10, 60, 8)	4.47	(100,40,40)	4.22	(50,100,8)	4.01	1.11×	1.05×

handle boundary constraints due to the DL/ML model, we used the extended version of the PRO algorithm introduced in the Active Harmony framework [32]. The same extension to handle boundaries was employed in our implementation of the Simplex method, and its stopping criteria are based on the work by Luersen [21]. Regarding initial simplex selection for our Simplex search implementation, we used a model-driven approach based on the DL model for square tiling. The square tile size tuple, $T1 = T2 = T3$, which satisfies the DL capacity constraint is selected as one vertex of the initial simplex. Other tree vertices were chosen so as to form a regular triangular pyramid. Note that all the studied kernel loops are triply nested and the simplex always has four vertices. The initial simplex is bounded by the upper and lower tile sizes in addition to the DL/ML bounds.

Table 4 shows the total execution time for the whole empirical tuning, the best tile size found by each approach, and its execution time. The DL/ML bounds significantly reduced the total tuning time by a factor of 1.02 to 4.95 on *Nehalem*, 1.33 to 2.48 on *Power7*, and 2.95 to 4.66 on *Xeon*. Furthermore, the Simplex and PRO methods using DL/ML boundary constraints found better tile sizes than the cases without DL/ML bounds, except for the simplex method on *Nehalem*. The tile size search space contains various local optimal points, and these empirical search approaches not using the boundary constraints got stuck at local optima far from the global optimal point. Note that these search methods can take arbitrary tile sizes in the search space, and hence found slightly better tile sizes in some cases than Table 3, which shows the result of scanning the search space with strided access.

Parallel execution of tiled code Table 5 reports the best tile sizes found by an exhaustive empirical search using DL/ML bounds when the outer-most tiling loop is parallelized with OpenMP `parallel for` directives. It shows the speedup with respect to the untiled sequential execution when running each program with all cores (parallel) and when running with a single core (sequential, same as Table 3). Although the performance with parallelization is not always better than sequential, the best tile sizes for

Table 4: Empirical Search Results for 1-level Tiling

	Without DL/ML Bounds		With DL/ML Bounds	
	Total [sec]	Best Size / Time [sec]	Total [sec]	Best Size / Time [sec]
matmult-nehalem-simplex	3173.36	(17, 120, 1369) / 13.86	640.98	(36, 56, 64) / 14.71
matmult-nehalem-pro	1294.88	(52, 344, 2270) / 15.64	380.73	(36, 80, 29) / 15.24
matmult-power7-simplex	940.81	(114,1142,858) / 11.4	709.22	(22,82,117) / 11.32
matmult-power7-pro	691.01	(172,1784,2989) / 11.39	442.26	(28,72,126) / 10.58
matmult-xeon-simplex	4268.52	(98,1257,1258) / 21.69	1039.69	(35,56,57) / 19.52
matmult-xeon-pro	2453.03	(97,904,1315) / 21.81	831.73	(31,64,56) / 19.22
2d-jacobi-nehalem-simplex	88.84	(42, 465, 498) / 2.25	26.48	(34,15,64) / 2.32
2d-jacobi-nehalem-pro	51.09	(29, 2001, 2000) / 2.41	50.33	(25,10,627) / 2.2
2d-jacobi-power7-simplex	96.95	(50,37,92) / 1.15	54.94	(50,28,116) / 1.14
2d-jacobi-power7-pro	83.98	(25,8,3495) / 1.61	33.81	(10,53,84) / 1.17
2d-jacobi-xeon-simplex	351.52	(50,40,16) / 2.49	75.49	(50,33,16) / 2.49
2d-jacobi-xeon-pro	248.12	(26,1976,2098) / 8.85	57.34	(10,12,21) / 2.75

parallelized benchmarks also lie in the region bounded by the proposed DL/ML model except for dtrmm and jacobi-2d on *Xeon*, whose parallel performance is lower than sequential performance. This performance degradation results from inefficient data distribution, which may also cause unexpected effects on tile size selection.

Table 5: Parallel 1-level Tiling Results

	Optimal Point (parallel)		Optimal Point (sequential)	
	Tile Size	Speedup vs. Untiled Seq.	Tile Size	Speedup vs. Untiled Seq.
matmult-nehalem (8 Threads)	(80, 10, 120)	9.39×	(150, 30, 80)	2.47×
matmult-power7 (32 Threads)	(100, 1, 300)	15.24×	(90, 10, 120)	2.40×
matmult-xeon (8 Threads)	(150, 32, 80)	57.12×	(100, 20, 120)	8.37×
dsyrk-nehalem (8 Threads)	(150, 30, 120)	0.86×	(30, 30, 90)	2.03×
dsyrk-power7 (32 Threads)	(32, 70, 300)	13.79×	(60, 10, 1000)	2.54×
dsyrk-xeon (8 Threads)	(30, 10, 90)	3.64×	(100, 30, 80)	4.66×
dtrmm-nehalem (8 Threads)	(32, 50, 32)	0.93×	(150, 30, 60)	7.83×
dtrmm-power7 (32 Threads)	(10, 30, 100)	1.90×	(600, 30, 32)	5.25×
dtrmm-xeon (8 Threads)	(1, 1, 30)	1.69×	(30, 10, 120)	4.88×
2d-jacobi-nehalem (8 Threads)	(10, 50, 120)	1.77×	(10, 8, 150)	1.13×
2d-jacobi-power7 (32 Threads)	(10, 32, 120)	2.12×	(10, 40, 120)	1.76×
2d-jacobi-xeon (8 Threads)	(10, 40, 600)	3.06×	(50, 40, 20)	3.44×
2d-fdtd-nehalem (8 Threads)	(30, 80, 8)	14.08×	(50, 50, 8)	6.79×
2d-fdtd-power7 (32 Threads)	(10, 80, 8)	15.17×	(40, 70, 40)	4.57×
2d-fdtd-xeon (8 Threads)	(10, 60, 8)	11.99×	(50, 100, 8)	4.09×

7 Related Work

Exploiting data locality is a key issue in achieving high levels of performance and tiling has been widely used to improve data locality in loop nests. Nevertheless, the choice of tile sizes greatly influences the realized performance. Wolf and Lam [34] were the first to provide precise definitions of reuse and locality and develop transformations to improve locality. Ferrante el al. [12], Wolf and Lam [34], and Bodin et al. [5] were among the earliest to develop cache estimation techniques designed for data locality optimizations. Several authors proposed techniques for selecting tile sizes aimed at reducing self-interference misses [10, 20]. Ghosh et al. [13] developed cache miss equations to find sizes of the largest tiles that eliminate self-interference, while fitting in cache. Chame and Moon [8] developed techniques to minimize the sum of the capacity and cross-interference misses while avoiding self-interference misses. Rivera and Tseng [26] developed padding techniques to reduce interference misses and studied the effect of multi-level caches on data locality optimizations. Hsu and Kremer [16] presented a comprehensive comparative study of tile size selection algorithms. To the best of our knowledge, all of these techniques find a single tile size for each loop that is being tiled. Recently, Yuki et al. [38] have explored the automatic creation of cubic tile size models. In contrast, we have demonstrated (see Table 3) that the best performance is often realized only for rectangular tiles.

Search-based techniques for finding tile sizes (and unroll factors) have received much attention in performance optimization [4, 19, 31–33]. The ATLAS system employs extensive empirical tuning to find the best tile sizes for different problem sizes in the BLAS library; tuning is done once at installation. Unfortunately such an approach is not suited for general tiled codes, as the search process is tuned for dense linear algebra codes only. Only square tile sizes are considered, which significantly hampers the performance of a variety of codes (such as stencil codes) that require rectangular tiles for best performance. Furthermore, ATLAS currently includes a simplistic model where tile sizes are searched as to not exceed the square root of the L1 cache size. Our analytical bounds offer a significantly higher accuracy, capturing both intra- and inter-tile reuse at various cache level.

Kisuki et al. [19] have used different techniques such as genetic algorithms and simulated annealing to manage the size of the search space. Tiwari et al. [32] note: "a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations." The Active Harmony project [31, 32] uses several different algorithms to reduce the size of the search space such as the Nelder-Mead simplex algorithm. In contrast to these approaches, we use a pair of analytical models — a conservative model that overestimates the number of cache lines by ignoring lifetimes and an aggressive model that underestimates the number of cache lines — each leading to different sets of tile sizes, which are used to bound the search space. With our technique, any of the algorithms from [19, 31, 32] can be used to further reduce the search time.

8 Conclusion

In this paper we developed a novel approach to analytically bound the search space for tile size selection based on two models, a conservative model (DL) that ignores intra-tile cache block replacement and a new aggressive model (ML) that assumes optimal replacement. We described how empirical search can be restricted (pruned) by the two models (DL and ML). Search space reductions ranging from $45\times$ - $11,879\times$ were obtained by using this pruning technique for five benchmarks on three different platforms. Our experimental results for single-level tiling on different benchmarks show that almost all tile sizes that deliver 95% or more of the optimal performance fall between the ML and DL bounds used in our approach. Furthermore, we demonstrated the integration of the analytical bounds with existing search optimization algorithms, and the experimental results show that the total search time was reduced by factors ranging from 1.02× to 4.95×. The experiments for parallel execution show that our DL/ML model is also effective for tile size selection of parallelized programs. Taken together, these experimental results make a convincing case of the effectiveness of our new approach to model-driven empirical search for tile sizes.

For future work, we propose to extend our approach to multi-level tiling, and also to leverage correlation studies to identify which levels of the memory hierarchy are most closely tied to performance and compute DL and ML bounds for those hierarchy levels.

Acknowledgments We thank the reviewers for their feedback and suggestions to improve the presentation of the paper, and we are grateful to Jill Delsigne for her assistance with proof-reading the final version of this paper. This work was supported in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127, the U.S. National Science Foundation through awards 0811457, 0811781, 0926687 and 0926688, and by the U.S. Army through contract W911NF-10-1-0004. The opinions and findings in this document do not necessarily reflect the views of the United States Government, Rice University, Ohio State University or Louisiana State University.

References

1. T. W. Barr, A. L. Cox, and S. Rixner. Translation caching: skip, don't walk (the page table). In *ISCA '10*, pages 48–59, New York, NY, USA, 2010. ACM.
2. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *CGO*, pages 200–209, 2010.
3. R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS XIII*, pages 26–35, 2008.
4. J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. ICS*, pages 340–347, 1997.
5. F. Bodin, W. Jalby, D. Windheiser, and C. Eisenbeis. A quantitative algorithm for data locality optimization. In *Code Generation*, pages 119–145, 1991.
6. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *PLDI*, 2008.
7. P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17(1):33–51, 1994.
8. J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *ICS*, pages 492–499, 1999.
9. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO'05*, 2005.
10. S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI*, pages 279–290, 1995.
11. K. Datta. Auto-tuning stencil codes for cache-based multicore platforms. Technical report, University of California, Berkeley, Dec. 2009.
12. J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness. *Proc. LCPC 91*, 589:328–343, 1991.
13. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM TOPLAS*, 21(4):703–746, 1999.

14. K. Goto and R. A. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1), July 2008.
15. A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proc. ICS*, 2009.
16. C. Hsu and U. Kremer. A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279–294, 2004.
17. F. Irigoin and R. Triolet. Supernode partitioning. In *ACM POPL*, pages 319–329, 1988.
18. D. Kim, L. Renganarayanan, M. Strout, and S. Rajopadhye. Multi-level tiling: 'm' for the price of one. In *SC*, 2007.
19. P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.
20. M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th ACM ASPLOS*, pages 63–74, 1991.
21. M. Luersen, R. L. Riche, and F. Guyon. A constrained, globalized, and bounded nelder-mead method for engineering optimization. *Structural and Multidisciplinary Optimization*, 27(1-2):43–54, 2004.
22. J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7(4):308–313, 1965.
23. J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *JPDC*, 16(2):108–230, 1992.
24. L. Renganarayana, D. Kim, S. Rajopadhye, and M. Strout. Parameterized tiled loops for free. In *PLDI*, pages 405–414, 2007.
25. Resource Characterization in the PACE Project. `http://www.pace.rice.edu/Content.aspx?id=41`.
26. G. Rivera and C. Tseng. Locality optimizations for multi-level caches. In *SC*, 1999.
27. V. Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM J. Res. & Dev.*, 41(3), May 1997.
28. V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *IEEE ISPASS*, 2000.
29. R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Tech. Report 90.38, RIACS, NASA Ames Research Center, 1990.
30. V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel parameter tuning for applications with performance variability. In *Proc. Supercomputing '05*, 2005.
31. C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: towards automated performance tuning. In *SC*, pages 1–11, 2002.
32. A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. Scalable autotuning framework for compiler optimization. In *IPDPS '09*, 2009.
33. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
34. M. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91*, pages 30–44, 1991.
35. M. Wolfe. More iteration space tiling. In *Proc. Supercomputing*, pages 655–664, 1989.
36. J. Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
37. K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *International Conference on Supercomputing*, 2005.
38. T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. Eichenberger, and K. O'Brien. Automatic creation of tile size selection models. In *CGO*, pages 190–199, 2010.