Locality-Aware Mapping and Scheduling for Multicores

Wei Ding  Yuanrui Zhang  Mahmut Kandemir  Jithendra Srinivas  Praveen Yedlapalli
The Pennsylvania State University, University Park, PA 16802, USA
{wzd109, yuz123, kandemir, jsk265, praveen}@cse.psu.edu

Abstract
This paper presents a cache hierarchy-aware code mapping and scheduling strategy for multicore architectures. Our mapping strategy determines a loop iteration-to-core mapping by taking into account application data access patterns and on-chip cache hierarchy. It employs a novel concept called “core vectors” to obtain a mapping matrix which exploits data reuses at different layers of the cache hierarchy based on their reuse distances, with the goal of maximizing data locality at each level, while minimizing data dependences across the cores. Our scheduling strategy on the other hand determines a schedule for the iterations assigned to each core, with the goal of reducing data reuse distances across the cores for dependence-free loop nests. Our experimental evaluation shows that the proposed mapping scheme reduces miss rates at all levels of caches and application execution time significantly, and when supported by scheduling, the reduction in cache miss rates and execution time become much larger.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Compilers, Optimization

General Terms Languages

Keywords Loop transformation, Cache hierarchy, Multicore

1. Introduction
Multicore Machines and Software Support. To take full advantage of emerging multicore architectures, there are at least two types of help needed from the software side. First, software should exploit thread level parallelism, and a recent body of research [14, 24–26] explores this avenue. Second, software should exploit unique architectural features of target multicores such as on-chip caches, fast on-chip communication, and limited off-chip bandwidth. Unfortunately, there are very few code optimization schemes proposed to date that can take full advantage of on-chip caches hosted by multicores. In addition, most existing schemes are straightforward extensions of data locality optimization strategies originally developed in the context of uniprocessor architectures, and consequently, they do not capture important multicore features such as shared on-chip caches.

Our Goal and Contributions. Emerging multicore architectures differ from prior multiprocessor based systems in that they typically employ on-chip cache hierarchies where subsets of caches are shared by subsets of cores (see Figure 1). This cache sharing can result in opportunities as well as problems in a multithreaded execution, depending on how compatible the data access and sharing patterns exhibited by threads are with the physical cache sharing imposed by the underlying architecture. Investigating this relationship and exploiting it to improve application performance are the main objectives of this paper. Specifically, our goal is to automatically transform application programs for achieving better performance on on-chip cache hierarchies of emerging multicore architectures. We enlist help from an optimizing compiler to implement our automatic code optimization. Our contributions can be summarized as follows.

• We propose a code parallelization/mapping strategy for multicore architectures. This strategy employs a novel, cache hierarchy-aware core tagging scheme and uses this tagging scheme to ensure that most important and frequent data reuses are exploited at the higher levels of the on-chip cache hierarchy (i.e., in caches that are closer to cores). A unique characteristic of this strategy is that it explicitly takes into account the on-chip cache hierarchy of the target multicore.

• We propose a code scheduling strategy for multicores. A distinguishing feature of this strategy is that it determines a customized schedule for each core in the target architecture considering how this core shares on-chip caches in the system with other cores. Both our mapping and scheduling strategies are implemented in a linear algebraic framework. As a result, they can easily be integrated with many existing high-level compiler optimizations.

• We present the implementation details of our mapping and scheduling strategies and carry out an experimental evaluation. Our experiments with eight different multithreaded codes running on an Intel multicore machine reveal that the
proposed parallelization/mapping scheme improves L1, L2 and L3 cache miss rates by, respectively, 13.5%, 16.6% and 14.9% on average, resulting in an average execution time improvement of 14.1%. Further, when supported by cache hierarchy-aware scheduling, these savings jump to 24.7%, 21%, 18.7% and 18% for L1 miss rate, L2 miss rate, L3 miss rate and execution time, respectively.

2. Data Locality Problem in Multicores

One of the important characteristics of emerging multicore machines is that they have a variety of on-chip cache hierarchies. Figure 1 illustrates three sample multicore architectures whose cache topologies are different from each other. Consequently, a given multithreaded application can have very different performance characteristics in these architectures. This motivates for a cache hierarchy-aware code parallelization and optimization strategy. Let us now briefly explain how code mapping and code scheduling can help us achieve high degrees of constructive sharing and avoid destructive interferences. In the context of data intensive applications, code mapping assigns each loop iteration to a core, which achieves a parallelization of a loop nest at the same time. If two loop iterations share data, it is better (from a constructive sharing perspective) to map them to cores that share a cache in some layer in the on-chip hierarchy. Further, the more intense the sharing is (i.e., more frequently it occurs), the more important to exploit the resulting reuse at higher layers of the cache hierarchy, i.e., layers close to cores. On the other hand, if two iterations do not share data, it is better (from a destructive interference perspective) to assign them to cores that do not share any cache in the system (so that they do not displace each other’s data). If this is not done, they can conflict in shared caches. While careful mapping of loop iterations to cores can improve chances for constructive sharing and minimize chances for destructive interferences, a mapping can only dictate a placement of iterations (computations); it does not impose any order in execution of the iterations assigned to each core. Thus, mapping cannot guarantee that the data involved in the reuse will be caught in that shared cache when the reuse actually takes place. This is where scheduling comes into the picture. Our goal in scheduling is to ensure that, if two iterations mapped to two different cores access the same data element (or data block), it is better that they access that data element/block in close proximity in time so that the second use (reuse) can catch the data while it is still in shared cache. Therefore, if applied carefully, mapping and scheduling can cooperatively maximize constructive sharing and minimize destructive interferences.

3. Overview of Our Approach

Figure 2 illustrates the high-level view of our proposed compiler-based approach. For each loop nest, we apply our cache topology-aware parallelization and locality optimization scheme, which consists of two steps, namely, mapping and scheduling. Mapping determines how loop iterations are distributed across the cores, while scheduling decides the execution order of loop iterations assigned to each core. Both these steps take into account intra-core and inter-core data access and reuse patterns as well as the on-chip cache hierarchy of the target multicore architecture.

In particular, the mapping component further contains two sub-steps: iteration-to-virtual core mapping and virtual core-to-physical core mapping. The former one employs a concept of virtual cores to represent a “virtual architecture” that has the same number of levels of caches as the target (physical) architecture, but infinite number of cores and cache components at each level, and determines an iteration-to-core mapping such that data reuses with different reuse distances (represented by reuse vectors in our work) can be exploited at different layers of the cache hierarchy. The core idea behind this step is that, data reuses with shorter reuse distances should be exploited in the higher level caches that have faster accesses, since they occur more frequently during program execution and usually take place in a shorter period of time, whereas data reuses with longer reuse distances can be exploited in the lower levels of caches that have larger capacities. In other words, data reuses with different reuse distances are expected to be converted into locality at different layers of the cache hierarchy that have different characteristics (in terms of access latency and capacity). Note that, this can only be achieved through a cache topology-aware computation mapping. Once the iteration-to-virtual core mapping is done, the second sub-step employs a folding function to map infinite virtual cores to limited physical cores by considering the data dependences across the cores. As a result, a loop nest is parallelized and all its iterations are properly assigned to cores. Note that, our mapping strategy also tries to maximize the utilization of all (physical) cores available; that is, it is oriented toward maximizing data locality as well as parallelism.

![Figure 1. Three different multicore architectures with shared on-chip caches. Ovals represent cores and rectangles represent caches. In each figure, the cache closest to a core is L1. The numbers attached to the caches indicate their IDs.](image)
After computation-to-core mapping, the scheduling component takes the iterations assigned to physical cores and determines an execution order for them, with the goal of tuning the timing of data sharing (reuse) among the cores so that data reuse can be converted into locality during program execution. Both our mapping and scheduling strategies are formulated and implemented in a linear algebraic framework; therefore, they can easily be integrated with many previously-proposed high-level parallelism and data locality optimizations.

4. Program Representation

Our work targets loop and data intensive applications. In these applications, an l-level loop nest represents an l-dimensional iteration space, and each point in this space (i.e., an iteration) can be denoted by an iteration vector \( \vec{i} = (i_0, i_1, ..., i_{l-2}, i_{l-1})^T \), where \( i_j \) (\( 0 \leq j \leq l \)) can take values from \([L_j, U_j]\), \( L_j \) and \( U_j \) being, respectively, the lower and upper bounds of the \( j \)th loop from the outermost. The original execution order of loop iterations obbeys lexicographical ordering\(^1\) (denoted using \( \prec \)). The iteration space of a loop nest can also be viewed as a bounded polyhedron [29]. If two iterations \( \vec{i}_1 \) and \( \vec{i}_2 \) access the same data element, we say that these two iterations exhibit temporal data reuse [30]. Similarly, if they access data elements that are mapped to the same cache line (block) boundary in the memory space, they are said to have spatial data reuse [30]. In either case, assuming that \( \vec{i}_1 \prec \vec{i}_2 \), \( \vec{r} = \vec{i}_2 - \vec{i}_1 \) is called the reuse vector [30], which represents the reuse distance between two accesses to the same data element. Moreover, if at least one of these two iterations is a write operation to the data element, this reuse vector is also called a dependence vector [1]. Note that a dependence vector can be seen as a special type of reuse vector, which enforces a specific execution order between the two iterations involved.

Reuse vectors can be computed based on pairs of array index expressions. In cases where two references to an array are uniformly generated; that is, references whose array index expressions differ in at most a constant term, e.g., \( A[i,j+1] \) and \( A[i,j] \), the reuse vector is constant, e.g., \((0,1)^T\). However, in cases where two references to an array are non-uniformly generated, e.g., \( A[i+j,j] \) and \( A[i,j] \), the reuse vector is a symbolic expression containing loop iterator variables, e.g., \((j,0)^T\). Note that, a reuse vector computed based on array index expressions usually denotes the reuse distance(s) for a set of (multiple) loop iteration pairs, irrespective of whether the reuse vector is constant or not, since array index expressions are expressed in terms of loop iterator variables (i.e., \( i_j \), where \( 0 \leq j \leq l \)) bounded by loop bounds, and as loop iterators vary, different loop iterations can be enumerated. Consequently, each reuse vector is in fact implicitly associated with a domain that specifies the loop iterations involved in the reuses represented by this vector. For example, if \( 0 \leq i \leq 1 \) and \( 0 \leq j \leq 1 \), the domain of the constant reuse vector \((0,1)^T\) generated from references \( A[i,j+1] \) and \( A[i,j] \) and the domain of the non-constant reuse vector \((j,0)^T\) generated from references \( A[i+j,j] \) and \( A[i,j] \) are \( \{(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T\} \).

Reuse vector is the primary abstraction used for data locality optimization. For a given loop nest, we use \( R = (\vec{r}_0, \vec{r}_1, ..., \vec{r}_{s-1}) \), called the reuse matrix, to represent all its reuse vectors, where each column \( \vec{r}_i \) is a reuse vector identified in this loop nest. Clearly, from a performance point of view, we prefer small reuse vectors over large ones (in a lexicographic sense on their absolute values). This is because small reuse vectors indicate shorter reuse distances, which means we have higher chances for catching the reused data in the cache.

5. Architectural Abstraction

5.1 Core Vectors

The primary architectural abstraction used by our compiler-based approach is built on a novel concept called core vector, which is used to represent (tag) each core in a multicore architecture with multiple levels of caches. To assign core vectors to cores, we first enumerate all the caches and cores from left to right at each layer of the hierarchy, as shown in Figure 1, and then represent each core using a vector that is composed of the core ID followed by IDs of all the caches it accesses in a top-down order (i.e., from the fastest/smallest cache to the slowest/largest). For example, in Figure 1(c), the core vectors for core 0 and core 1 are \( \vec{c}_0 = (0,0,0,0)^T \) and \( \vec{c}_1 = (1,1,0,0)^T \), respectively. If an architecture has \( s \) levels in its cache hierarchy (including cores as a separate layer), the core vector for core \( k \) \((0 \leq k < n)\), where \( n \) is the total number of cores, would be \( \vec{c}_k = (c_{k_0}, c_{k_1}, ..., c_{k_{s-1}})^T \), where \( c_{k_0} \) is the ID of core \( k \) and \( c_{k_1}, ..., c_{k_{s-1}} \) are IDs of the caches that core \( k \) accesses at each level. Clearly, this numbering strategy can easily be extended to architectures with larger number of cores and cache layers.

The motivation behind the use of core vectors is that, when two iterations (or threads or computations) are assigned to two cores with core vectors \( \vec{c}_i \) and \( \vec{c}_j \) \((0 \leq i,j < n)\), respectively, by comparing the two core vectors, one can

\(^1\)Let \( \vec{x} = (x_1, x_2, ..., x_n) \) and \( \vec{y} = (y_1, y_2, ..., y_n) \) be two vectors in an n-dimensional space. \( \vec{x} \) is said to be lexicographically smaller than \( \vec{y} \), denoted by \( \vec{x} \prec \vec{y} \) if \( x_1 < y_1 \) or both \( x_1 = y_1 \) and \( x_2, ..., x_n < y_2, ..., y_n \).
determine how these two iterations share cores and caches at different layers of a hierarchy, and therefore further decide whether the data reuse between these two iterations could potentially be exploited in the cache hierarchy, and if so, at what level. For example, if two iterations are mapped to two cores \(c_0\) and \(c_2\) in Figure 1(c) with core vectors \((0, 0, 0, 0)^T\) and \((1, 1, 0, 0)^T\), the data reuse between these two iterations can be exploited in the L2 and L3 caches connected to these two cores, since the two core vectors have the same L2 and L3 IDs. As another example, if two iterations are mapped to two cores \(c_0\) and \(c_4\) that have core vectors \((0, 0, 0, 0)^T\) and \((4, 4, 2, 1)^T\), the data reuse between these two iterations cannot be exploited in any cache in the hierarchy, since the two core vectors have no common entries, which means the corresponding two cores do not share cache at all. We now formally define the core difference vector to capture the sharing patterns of computations (iterations) mapped to two cores.

**Definition 5.1.** Let \(\vec{c}_i\) and \(\vec{c}_j\), \(0 < i, j < n\), \(\vec{c}_i, \vec{c}_j \in Z^n\), be two core vectors, where \(n\) is the total number of cores, and \(s\) is the total number of levels in a cache hierarchy (including cores). The core difference vector \(\vec{d}_{i,j}\) is defined as \(\vec{d}_{i,j} = (d_0, d_1, ..., d_{s-1})^T\), where \(d_t\) \((0 \leq t < s)\) is calculated as:

\[
d_t = \begin{cases} 
0, & \text{if } c_{i,t} = c_{j,t}; \\
1, & \text{otherwise}.
\end{cases}
\]

Note that, cache hierarchy in current multicore architectures is usually a “tree-like” structure, where the shared off-chip main memory can be seen as the root, caches as internal nodes, and cores as leaves. This means that, if cores \(i\) and \(j\) share a cache at level \(t\) \((1 \leq t < s)\), they will share caches at levels \(t+1, t+2, \cdots, s-1\) as well; consequently, after the first occurrence of a zero value, say, \(d_t = 0\) is detected, all the values from \(d_{t+1}\) to \(d_{s-1}\) will be 0 in the core difference vector \(\vec{d}_{i,j}\). However, our model is general enough to capture other (possibly, asymmetric) cache connections as well.

### 5.2 Virtual Cores

Given a target architecture with \(s\) levels of hierarchy for cores and caches, all core vectors collectively define an \(s\)-dimensional core space. However, it should be noted that certain vectors such as \((0, 1, 0, 0)^T\) are not legal core vectors, since they do not have any corresponding points in the core space. Therefore, the core space for a multicore architecture with a “tree-like” on-chip cache hierarchy is not continuous.

To determine a mapping for all iterations in a loop nest, our strategy is to first construct a continuous linear core space that consists of virtual core vectors (where each virtual core vector represents a virtual core), then use a mapping matrix \(M\) to map each loop iteration to a virtual core vector in this space, and finally employ a folding function to map multiple virtual core vectors to fewer, legal virtual core vectors that correspond to physical cores in the target architecture. The virtual cores represent a virtual architecture which has the same number of levels of cache hierarchy as the target architecture; however, it has an infinite number of cores and caches at all levels, and it allows a core or a cache at level \(t\) \((0 \leq t < s)\) to be connected with any cache at level \(t+1\) \((0 \leq t < s-1)\). As a result, a virtual architecture has a general “graph-like” structure; and any “tree-like” on-chip hierarchy can be considered as a special case of this virtual architecture. In other words, a virtual architecture is a large template onto which many physical architectures with the same number of cache layers can be mapped.

Virtual cores serve two purposes in our approach. Firstly, they provide a continuous linear space for applying parallelization/mapping, which makes our approach easier to formulate. If virtual cores were not there, in our mathematical formulations (explained later), we would have to deal with constant bounds (e.g., number of cores), which would make implementation harder. Virtual cores help us separate the linear algebraic analysis from the mapping problem. Secondly, once we determine our mapping matrix for a virtual architecture (i.e., in terms of virtual core vectors), we can use them for any physical architecture that fits the template and customize only the folding function for each physical architecture.

### 6. Mapping

Our mapping component consists of two steps, namely, the iteration-to-virtual core mapping and the virtual core-to-physical core mapping. In this section, we will discuss the details of these steps.

#### 6.1 Iteration-to-Virtual Core Mapping

Given a target cache hierarchy (including cores), our iteration-to-virtual core mapping employs a mapping matrix to map each iteration of a loop nest to a virtual core, such that existing data reuses can be exploited at different layers of cache hierarchy based on their reuse distances.

#### 6.1.1 The Mapping Matrix

Recall that data reuse can be represented by reuse vectors in a loop nest, as explained in Section 4. If two iterations \(\vec{r}_1\) and \(\vec{r}_2\) exhibit a data reuse between them with a reuse vector of \(\vec{r} = \vec{r}_2 - \vec{r}_1\) (assuming \(\vec{r}_2\) is lexicographically greater than \(\vec{r}_1\)) and we want to exploit their data sharing at level \(k\) \((0 < k < s)\) in the cache hierarchy of a target (physical) architecture, then we should map these two iterations to two physical cores with core vectors \(\vec{c}_1\) and \(\vec{c}_2\), whose core difference vector \(\vec{d}_{i,j}\) has the following format:

\[
d_{i,j} = \begin{pmatrix} 1 & 1 & \cdots & 1 & 0 & 0 & 0 & 0 \end{pmatrix}^T. 
\]
That is, the two physical cores should share some cache at level \( k \) (and below),\(^2\) so that the data reuse between these two iterations has a chance to get converted into locality at level \( k \). Note however that this does not guarantee that the reuse targeted will certainly be exploited at level \( k \) or the data element in question will definitely be caught at level \( k \) at the time of reuse. It depends on the times at which these iterations (\( \vec{t}_1 \) and \( \vec{t}_2 \)) are executed, i.e., on their scheduling. Mapping only ensures that \( \vec{t}_1 \) and \( \vec{t}_2 \) with data reuse are mapped to cores that share a cache between them, and it creates an opportunity for exploiting the reuse at level \( k \).

Further, since \( d_{i,j} = f(\vec{t}_2) - f(\vec{t}_1) \), we have \( d_{i,j} = M. \vec{t}_2 - M. \vec{t}_1 = M. (\vec{t}_2 - \vec{t}_1) = M. \vec{r} \). The expression above can be re-written as:

\[
M. \vec{r} = \begin{pmatrix} 1 & 1 & \cdots & 1 & 0 & 0 & 0 & 0 \\ \end{pmatrix}^T (1) \]

This tells us that, we can use reuse vectors of a loop nest to decide the mapping matrix \( M \).

Note that, as mentioned in Section 2, we would like to exploit data reuses with shorter reuse distances (represented by shorter reuse vectors) at higher layers of the cache hierarchy since they occur more frequently and hence take better advantage of low access latency provided by the caches at higher layers. We also prefer to exploit data reuses with longer reuse distances (represented by longer reuse vectors) at lower layers of caches since such reuses can be caught in the caches with larger capacities when taking place. Therefore, given \( q \) reuse vectors in a loop nest, denoted by the reuse matrix \( R = (\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{q-1}) \), we first sort them in a lexicographically-ascending order based on their absolute values, e.g., \( r_0 \preceq r_1 \preceq \cdots \preceq r_{q-1} \) (since the absolute value of a reuse vector represents its reuse distance), then divide them into groups denoted by \( g_i \) \((1 \leq i < s)\) according to the layers of cache hierarchy using separator vectors \( \vec{d}_i \) \((1 \leq i < s - 1)\), and finally map them to different cache layers using Eq.(1) by determining a proper mapping matrix \( M = (\vec{m}_0, \vec{m}_1, \vec{m}_2, \ldots, \vec{m}_{s-1})^T \), where \( \vec{m}_i \) \((0 \leq i < s)\) is a row vector of \( M \). This idea is illustrated in Figure 3 for an \( s \)-layer cache hierarchy, where all the reuses lexicographically smaller than or equal to \( \vec{d}_i \) are expected to be exploited at the L1 cache layer, and all the reuses that are larger than \( \vec{d}_i \) and smaller than or equal to \( \vec{d}_{i+1} \) are expected to be exploited at the L2 cache layer, and so on, so that the performance of the on-chip cache hierarchy can be maximized. The separator vectors \( \vec{d}_i \) \((1 \leq i < s - 1)\) are selected based on the cache capacity at each layer, as will be discussed shortly. It is to be noted at this point that, we do not show \( \vec{m}_0 \) in Figure 3, which corresponds to the core layer (i.e., layer 0) in the hierarchy, since we decide it by using a group of dependence vectors that are chosen from all available reuse vectors (recall that dependence vector is a special type of reuse vector), with the goal of minimizing synchronization overhead across the cores. In this case, Eq.(1) can be re-written as:

\[
s \text{ entries } \quad M. \vec{d} = (0 \quad 0 \quad 0 \cdots 0)^T, \quad (2)\]

where \( d \) is a dependence vector. To summarize, our general approach in the mapping step is to determine a mapping matrix \( M \) based on the cache hierarchy and a classification of reuse vectors, and then use this matrix to map all iterations to cores.

Observe that, Eq.(1) and Eq.(2) represent an iteration-to-physical core mapping, since their right-hand side is the format for a physical core difference vector \( d_{i,j} \). Recall that, cache hierarchy in real multicore architectures is usually a “tree-like” structure, and if the \( k^{th} \) \((0 \leq k < s)\) entry in \( d_{i,j} \) is 0, all the entries following it are 0. However, the virtual architecture constructed based on a given target (physical) architecture allows arbitrary connections between the adjacent layers in the hierarchy, which means that, in exploiting the data sharing (or data dependence) between two iterations at layer \( k \), where \( 0 \leq k < s \), we only need to care about the \( k^{th} \) entry in their virtual core difference vector and ensure that it is 0. Therefore, we have the following equation for iteration-to-physical core mapping:

\[
s \text{ entries } \quad M. \vec{u} = (X \cdots X \quad 0 \quad X \cdots X)^T, \quad (3)\]

where \( \vec{u} \) denotes either a reuse vector \( \vec{r} \) or a data dependence vector \( \vec{d} \).

In addition, since \( M = (\vec{m}_0, \vec{m}_1, \vec{m}_2, \ldots, \vec{m}_{s-1})^T \), we have \( \vec{m}_0 \cdot \vec{d} = 0 \) if we want to map the two iterations involved in certain data dependence represented by the dependence vector \( \vec{d} \) onto the same core, and \( \vec{m}_k \cdot \vec{r} = 0 \) \((0 < k < s)\) if we want to exploit certain data reuse represented by the reuse vector \( \vec{r} \) between two iterations in a shared cache at level \( k \), e.g., \( \vec{r} \in \vec{g}_k \). It can be noticed that, this allows us to solve each row vector in matrix \( M \) independently and makes our mathematical formulation simpler.

Note also that, we still need to obtain a mapping between iterations and physical cores ultimately, which should conform to the format of the core difference vector in Eq.(1) and Eq.(2). Although our iteration-to-virtual core mapping step does not guarantee this, our virtual core-to-physical core mapping will ensure it through a folding process, as discussed in Section 6.2.

6.1.2 Determining the Mapping Matrix

In the following discussion, we present the technical details of how to determine a mapping matrix \( M \) for an \( l \)-level loop

\[^2\text{We assume that caches are inclusive and they have a “tree-like” structure in the target architecture, which means that, if a data element is found in a cache at level } \text{k, it can also be found in the cache that connects to this cache at level } \text{k + 1.}\]
nest and an s-level cache hierarchy. In order to achieve a one-to-one/onto (many-to-one) mapping from the iteration space to the virtual core space, we only need \( \min(l, s) \) number of row vectors in \( M \), as proved by the following lemma. Note that, to have a valid mapping, the mapping should be either one-to-one or onto.

**Lemma 6.1.** Given a cache hierarchy of \( s \) layers (including cores) and an \( l \)-dimensional loop nest, to achieve a one-to-one or onto iteration-to-virtual core mapping, only \( \min(l, s) \) number of row vectors are required in matrix \( M \).

**Proof:** First, the number of columns in \( M \) is \( l \), since our iteration-to-virtual core mapping is determined by \( M \). 

For \( i=1 \), and \( i \) is an \( l \)-dimensional vector. Second, in the case that we have \( s < l \), we can always find \( s \) linearly independent row vectors to construct an \( s \times l \) mapping matrix \( M \). Let the linear transformation \( T : Z^l \rightarrow Z^s \) be defined as \( T(\vec{i}) = M \vec{i} \). Since the rows of \( M \) are linear independent, the columns of \( M \) span \( Z^s \); therefore, \( T \) is an onto (many-to-one) transformation (mapping). In the case that we have \( s > l \), we can find \( l \) linearly independent row vectors to construct \( M \), and the columns of \( M \) are also linearly independent, which implies that \( T \) is a one-to-one mapping. Therefore, we can always use \( \min(l, s) \) number of row vectors to construct matrix \( M \).

Our approach to determining a mapping matrix \( M \) includes two steps: (i) reuse vector grouping, and (ii) row vector determination.

**Step 1: Reuse Vector Grouping**

The first step is to divide all the reuse vectors in the loop nest into groups \( (g_i, 1 \leq i < s', \text{ where } s' = \min(l, s)) \) based on the target cache hierarchy and cache capacities (in case we have \( s' < s \), we use only \( s' \) top levels of the hierarchy to decide the grouping). Suppose that, we have \( q \) reuse vectors, represented by the reuse matrix \( \vec{R} = (\vec{r}_0, \vec{r}_1, ..., \vec{r}_{s'-1}) \), and each is associated with a domain that specifies the iteration pairs involved in its reuse. To group them, we first sort them in a lexicographically-ascending order based on their absolute values. Recall however that a reuse vector may be constant (uniform reuse) or non-constant (non-uniform reuse), and we cannot perform sorting directly if there are non-constant vectors. To solve this problem, we first enumerate all distinct values for each non-constant vector using its domain information (see Section 4), i.e., convert each non-constant vector into a set of constant vectors, and then sort them along with the constant vectors.

Let us assume that, after sorting, we have \( \vec{r}_{0} \leq \vec{r}_{1} \leq \cdots \leq \vec{r}_{s'-1} \), where \( \vec{r}_i \) (\( 0 \leq i < p \)) is either an original constant vector from \( R \) or a constant vector converted from a non-constant one, and \( p \) is the total number of (distinct) constant vectors. Now, we need to select separator vectors in order to partition these reuse vectors into groups. We determine our separator vectors using the cache capacities at different layers of the cache hierarchy. Let \( C_k \) denote the capacity of a level-\( k \) cache, \( \vec{d}_k = (d_{0k}, d_{1k}, ..., d_{lk})^T \) represent the target separator vector for determining group \( k \), where \( 1 \leq k < s'-1 \), as depicted in Figure 3. \( B_i \) \((0 \leq i < l) \) denote the length (i.e., total number of iterations) of the \( i \)-th loop in the loop nest, and \( E \) represent the size of all data elements accessed by one iteration. Then, the separator vector \( \vec{d}_k \) should satisfy the following constraint:

\[
E \times (i_0 \times B_1 + i_1 \times B_2 + \cdots + i_{l-2} \times B_{l-1} + i_{l-1}) = C_k.
\]

Although this equation may have many solutions, we take the lexicographically smallest solution as the value for \( \vec{d}_k \). The meaning of \( \vec{d}_k \) determined by this equation is that, the number of data elements accessed (estimated by \( E \times (i_0 \times B_1 + i_1 \times B_2 + \cdots + i_{l-2} \times B_{l-1} + i_{l-1}) \)) between any two iterations involved in a data reuse represented by a reuse vector \( \vec{r}_i \), which is smaller than \( \vec{d}_k \), should be less than the size of a level-\( k \) cache, so that, at the time of the reuse, the reused data element has a chance to be caught in that cache.

**Step 2: Row Vector Determination**

After grouping the reuse vectors, we next determine the row vectors for matrix \( M \). We divide rows of \( M \) into two groups, \( \vec{m}_0 \) and \( \vec{m}_k \) \((1 \leq k < s') \), and use a slightly different strategy for each group. It should be noted that, for each row vector \( \vec{m}_k \) \((0 \leq k < s') \), and each iteration vector \( \vec{i}, \vec{m}_k, \vec{i} \) must be positive to ensure \( \vec{i} \) being mapped to valid virtual core space. Therefore, we have \( \vec{m}_k, \vec{i} \geq 0 \). This is the constraint that any row vector of \( M \) must satisfy.

**Case 1. Determining \( \vec{m}_0 \).**

In this case, we select a set of dependence vectors to determine \( \vec{m}_0 \) based on Eq.(3), since we want the iterations involved in these data dependences to be mapped to the same core, which can help reduce the costs of inter-core synchronization. According to Eq.(3), we have \( \vec{m}_0, \vec{d} = 0 \), where \( \vec{d} \) is a dependency vector. Note that, to determine \( \vec{m}_0 \) in this equation, we can select at most \((l-1)\) linearly independent dependence vectors. Since there might be more than \((l-1)\) dependence vectors, our strategy is to select \((l-1)\) lexicographically smallest ones from the sorted sequence \( \vec{r}_{0} \leq \vec{r}_{1} \leq \cdots \leq \vec{r}_{s'-1} \) obtained in the first step. Assuming that \( \vec{d}_1, \vec{d}_2, \ldots, \vec{d}_{l-1} \) are the selected ones, then we have the following equality constraint:

\[
\vec{m}_0, \vec{d}_1 = 0; \quad \vec{m}_0, \vec{d}_2 = 0; \quad \cdots; \quad \vec{m}_0, \vec{d}_{l-1} = 0. \quad (4)
\]

For the remaining dependence vectors \( \vec{d} \) that are not selected or used in Eq.(4), we should have the following condition satisfied to ensure the legality of the transformation
(called the legality constraint):
\[ \vec{m}_0, \vec{r}^T \geq 0. \]  

Also, for all reuse vectors \( r^T \) that are not used in Eq.(4) (including the non-selected dependence vectors \( \vec{d}^T \)), we have the following locality constraint:
\[ |\vec{m}_0, r^T| \leq q_0, \]  

with the goal of minimizing \( q_0 \). The motivation for setting up this last constraint is that, even if the iterations involved in the reuse represented by the reuse vector \( r^T \) cannot be mapped to the same virtual core (\( |\vec{m}_0, r^T| \neq 0 \)), we still want to minimize the distance between the two (different) virtual cores they are mapped to, i.e., we want to minimize the virtual-core mapping distance \( |\vec{m}_0, r^T| \). On the one hand, a smaller virtual-core mapping distance can lead to a smaller physical-core mapping distance (i.e., the distance between physical cores onto which the iterations involved in the reuse are mapped) because of our folding function (explained later); on the other hand, in real architectures, cache hierarchy is usually a “tree-like” structure, and the closer the two physical cores, more likely that they share a cache at a higher level in the hierarchy, and therefore, mapping iterations to two nearby physical cores usually leads to better locality.

At this point, we have a system of constraints: Eq.(4), Eq.(5) and Eq.(6). We now employ the integer Fourier-Motzkin Elimination (FME)\(^3\) to find the minimal \( q_0 \), under which we select the lexicographically smallest vector as the solution for \( \vec{m}_0 \). Note that, there may be less than \( (l - 1) \) dependence vectors, or no dependence vector at all in the loop nest, which means that we have fewer (or no) equalities in Eq.(4), in which case we use the integer FME to find a solution based on constraints Eq.(5) and Eq.(6), plus whatever equalities provided by Eq.(4).

If the integer FME does not return a non-trivial solution (i.e., if it returns only the trivial solution of 0), each time, we drop the constraint in Eq.(4) that involves the largest (in a lexicographic sense) dependence vector, and add a new constraint for this dependence vector based on Eq.(5). Then, we attempt to solve the resulting relaxed system of constraints using the integer FME. If there is still no solution, we drop the constraints involving the largest reuse or dependence vector each time, and try to solve the remaining constraints until no non-trivial solution is found.

\(^3\)Fourier Motzkin Elimination (FME) [29] is a mathematical algorithm for eliminating variables from a system of linear inequalities. Elimination of variables, \( V \), from a system of relations (here, linear inequalities) involves creating another system of the same kind, but without the variables \( V \), such that both systems have the same solutions over the remaining variables. Let us consider a system \( S \) of \( n \) inequalities with \( r \) variables, \( x_r \) through \( x_r \), with \( x_r \) being the variable which we want to eliminate. Each linear inequality that involves \( x_r \) can be re-written as \( \sum_{k=1}^{r-1} a_k, x_k \leq \sum_{k=1}^{r-1} b_k, x_k \), which is equivalent to \( \sum_{k=1}^{r-1} b_k, x_k \leq \sum_{k=1}^{r-1} a_k, x_k \). This effectively eliminates \( x_r \).

Figure 4. (a) An illustration of iteration-to-physical core mapping for a two-level loop nest with a sample architecture. (b) An illustration of customized scheduling for each core. The direction in each block denotes the iteration execution order.

**Case 2. Determining \( \vec{m}_k \) (1 ≤ \( k \) ≤ \( s' \)).**

Similar to Case 1, to determine the row vector \( \vec{m}_k \) for \( M \) based on Eq.(3), we select at most \((l - 1)\) linearly independent reuse vectors from group \( g_k \) (1 ≤ \( k \) < \( s' \)), which are lexicographically smallest. Suppose that \( r_1', r_2', \ldots, r_{l-1}' \) are the selected reuse vectors. Then, we have the following equality constraint:
\[ \vec{m}_k, r_1' = 0; \quad \vec{m}_k, r_2' = 0; \quad \ldots; \quad \vec{m}_k, r_{l-1}' = 0. \]  

This means that, we want the iterations involved in these data reuses to be mapped to the same shared cache at layer \( k \) in the hierarchy, in order to improve data locality. Note that, any reuse vector or dependence vector can be selected and used only once in the equalities captured by Eq.(4) and Eq.(7). For every dependence vector \( \vec{d} \) in \( R \), we also have the following legality constraint:
\[ \vec{m}_k, \vec{d} \geq 0, \]  

and for each reuse vector \( r^T \) in \( R \) that are not used in Eq.(7), we have the following locality constraint to minimize the mapping distance at layer \( k \) (with a similar reason for Case 1):
\[ |\vec{m}_k, r^T| \leq q_k. \]  

At this point, we use the integer FME to solve Eq.(7), Eq.(8) and Eq.(9) by finding the minimum \( q_k \). Note also that \( q_k \) may be or may not be the same as \( q_t \), where 0 ≤ \( t \) < \( k \) already found.

In addition, we need to ensure that \( \vec{m}_k \) is linearly independent from all the row vectors that have already been determined so far, i.e., \( \vec{m}_t \) (0 ≤ \( t \) < \( k \)). Therefore, we perform a check whenever a solution \( \vec{m}_k \) is returned. Let \( M' \) be the matrix that consists of the row vectors determined already, and \( M'_\perp \) be the sub-space orthogonal to \( M' \), where \( M'_\perp = I - M'^T, (M', M'^T)^{-1}, M' \). Then, \( \vec{m}_k \) should satisfy the following condition: \( M'_\perp \vec{m}_k \neq 0 \).

6.2 Virtual Core-to-Physical Core Mapping

Given an on-chip cache hierarchy of \( s \) layers and a mapping matrix \( M = (\vec{m}_0 \quad \vec{m}_1 \quad \cdots \quad \vec{m}_{s'-1})^T \), where \( s' =
min(l, s), our goal now is to determine a folding function $F$, denoted as $F = (f_0 \ f_1 \ \cdots \ f_{s'-1})^T$. As mentioned earlier, the main functionality of a folding function is to map each virtual core to a physical core (in the form of core vector). Recall that the virtual core $\vec{c}$ to which an iteration $\vec{i}$ gets mapped is given by $\vec{c} = M. \vec{i}$. Let $c_0$ denote the number of cores that are connected to an $L1$ cache in the target architecture and $c_i$ denote the number of caches at layer $i$ that are connected to the same cache at layer $i + 1$, where $1 \leq i \leq s' - 1$. Each entry $f_j$ ($0 \leq j \leq s' - 1$) satisfies:

$$f_j = (\bar{m}_{ij} \cdot \vec{i}_j / T_j) \mod c_j,$$

where $T_j$ is the block size in terms of the number of virtual cores. After applying $f_j$, every $T_j$ virtual cores are given the same cache ID at layer $j$ when we have $1 \leq j \leq s' - 1$, or the same core ID when $j = 0$. Note that, this mapping represents a block-wise distribution of virtual cores to physical cores. Once the folding function $F$ is determined, we map each iteration to a physical core. Figure 4(a) illustrates the iteration-to-physical core mapping for a two-level loop nest, for the sample multicore architecture shown in Figure 1(a).

### 6.3 Code Generation

In this section, we briefly go over our code generation strategy. Our iteration-to-virtual core mapping (characterized by the mapping matrix $M$) can be easily embedded into existing affine [6] or linear [18] loop transformation frameworks.

Given an $l$-dimensional loop nest, an affine loop transformation finds $l$ linearly independent $1 \times l$ integer vectors (called hyperplane vectors), and a constant offset vector to generate the transformed loop nest. The process of finding the hyperplane vectors and and the offset vector can be separated. Recall that, based on Lemma 6.1, our matrix $M$ only has $\min(l, s)$ number of $1 \times l$ row vectors. These vectors can be used as $\min(l, s)$ hyperplane vectors. We can then apply any existing affine transformation framework to find the remaining $l - \min(l, s)$ hyperplane vectors, as well as the offset vector to generate the transformed code.

In comparison, a typical linear loop transformation employs an $l \times l$ transformation matrix $T$ to map an $l \times 1$ iteration vector $\vec{i}$ to a new iteration vector $\vec{i}'$ by $\vec{i}' = T. \vec{i}$. The array reference $\bar{r}$, where $\bar{r} = A.\vec{i} + \vec{o}$, is changed accordingly to $\bar{r}' = T.\vec{i}' + \vec{o}$. As can be observed, our mapping matrix $M$ can be considered as one part of the transformation matrix. As in the affine transformation case, the remaining $l - \min(l, s)$ row vectors of $T$ can be determined by using any existing linear loop transformation scheme. As an example, Figure 5(b) shows the linear transformed code after applying our mapping matrix. Our current implementation adopts this linear loop transformation.

After performing either of the above transformations, we employ loop tiling/blocking to implement the virtual core-to-physical core mapping (based on our folding function). An important characteristic of our approach is that the tiled/blocked version of the code is unrolled based on the granularity of tiles/blocks. Specifically, we first partition the outermost $\min(l, s)$ loops of the transformed $l$-dimensional loop nest (obtained in the affine/linear loop transformation) into blocks, which corresponds to our folding function determined in Section 6.2. This will generate an $l + \min(l, s)$ tiled loop nest, where the newly added outermost $\min(l, s)$ loops iterate over the blocks. The tiled version for the code in Figure 5(b) is illustrated in Figure 5(c). Next, we unroll the formed blocks within the outermost $\min(l, s)$ loops up to the number of cores in the target architecture, and bind each block to a specific physical core. The unrolled code of our running example is illustrated in Figure 5(d).

### 7. Scheduling for Dependence-Free Loops

As explained in Section 6.3, we can employ existing linear/affine loop transformation techniques to determine the value of important transformation parameters (e.g., the offset vector and remaining row vectors of the transformation matrix), and therefore perform code generation. By doing this, the generated code actually already indicates an iteration execution order, i.e., how iterations are scheduled in the given loop nest. If the input loop nest is not dependence-free, this generated code is our final version. However, if the input loop nest is dependence-free, we propose a novel loop scheduling scheme to further improve the performance of the shared caches in the system. This can be achieved by reducing the reuse distance between two iterations that are mapped to two different cores that share an on-chip cache. In other words, our goal is to come up with a schedule for each core such that, if two cores share a data block, they share it within a short period of time, thereby increasing chances of catching the reused data block in the shared cache. A distinguishing characteristic of our proposed approach compared to previous loop transformation and scheduling theory, is that we determine a customized schedule for each core yet these schedules are adjusted with respect to each other to reduce inter-core data reuse distances. An example of our scheduling is illustrated in Figure 4(b), where the arrows in each tile/block indicate the iteration execution order. If the $i$ loop is the innermost loop, then, taking the top-left block as an example, the iterations in this block will be executed in a left-right, top-down fashion. Figure 5(e) illustrates the generated code in our scheduling scheme.

The implementation of our loop scheduling is quite easy. Since the loop nest is dependence-free, we simply reverse the loop indexing order based on the physical core that each block is mapping to. For example, in Figure 4(b), we first fix the loop indexing order for the first tile/block (which is the top-left one). Since iterations in this block are executed from left to right (under the same $j$ loop), the iterations in the block on its left (mapped to core 3) should be executed from right to left. Therefore, we reverse its indexing order.
for loop $i$ by changing $(i = ii, i <= min(ii + t, n), i + +)$ to $(i = min(ii + t, n), i >= 0; i = - -)$.

8. Experimental Evaluation

1. Setup and Applications

We performed all our experiments on an Intel Dunnington multicore machine. The relevant details of this architecture are presented in Table 1. In our experiments, all 12 cores are used. Dunnington has a cache hierarchy similar to the three-layer architecture shown in Figure 1(c), except that it has 12 L1s, 6 L2s, and 2 L3s. Each L3 is connected to 3 L2s, and each L2 is connected to 2 L1s. The applications used in our study are listed in Table 2. It shows cache statistics and parallel execution times under the default (original) version (explained below). The first six applications are frequently used in data-intensive/scientific computing; and the last two codes are taken from a computer game development toolkit. Among these eight applications, lu, mggrid, cg, and fma3d have data dependences in the main loop nests, and the rest four applications are used the dependence-free version in our experiment. The total sizes of the datasets manipulated by these applications vary between 24.4MB and 58.2MB. Most of the results presented in the following subsections are percentage improvements (in cache misses or execution cycles) over the Default Version (defined below). Most of our experiments are performed using four different versions explained below.

- **Default.** In this version, iterations of a parallel loop nest are distributed in a block fashion across cores and each core executes the iterations assigned to it in their original execution sequence. In a sense, this version represents a straightforward mapping strategy combined with the default execution order for iterations mapped to each core.

- **Default+.** This is similar to the default version except that, after mapping, computations mapped to each core are optimized using loop permutation and loop tiling. More details on this version will be given later in this section.

- **Mapping.** This is our proposed parallelization/mapping strategy. In our experiments, we evaluated three variants of this version:

  - **L3.** This version aims at optimizing only for the L3 cache.
  - **L2+L3.** This is a version in which our approach targets only the L2 and L3 layers.
  - **L1+L2+L3.** This represents a version where our approach targets all three layers in the target on-chip cache hierarchy.

Note that, while this version optimizes mapping, for the iterations mapped to each core, it maintains their execution order in Default+.

- **Mapping+Scheduling.** This version first uses our parallelization/mapping scheme and then our scheduling strategy, explained in Section 7. Note that, in this version, we also apply our scheduling scheme to those loop nests containing data dependences (e.g., lu, mggrid, cg, and fma3d). Although this will affect the correctness of the program, the reason for us to do so is to illustrate how much performance improvement can be brought by scheduling.

We used the SUIF compiler infrastructure [2] (as a source-to-source translator) to implement these versions. All these versions have been compiled (after the source-

![Figure 5. Example.](image-url)
to-source translation) with the native Intel compiler with -O3. Consequently, even the default version incorporates low level (single core centric) code optimizations. The different versions also execute the same set of iterations in parallel. Therefore, performance differences among different versions are only due to different mapping and scheduling strategies they adopt.

2. Results

Our first set of results are presented in Figure 6 and plot the percentage L1 miss, L2 miss, L3 miss and execution time improvements brought by versions L3, L2+L3, and L1+L2+L3 over the default version. There are two important observations one can make from these results. First, all optimized versions generate better results than the default version. Second, and more importantly, we can see that optimizing for the entire on-chip cache hierarchy is very important. Specifically, average execution time improvements brought by the L3, L2+L3, and L1+L2+L3 versions over the default version are, respectively, 5.4%, 9.3%, and 14.1%. This means, for example, that optimizing for L3 alone does not guarantee very good performance at the L2 and L1 layers.

Figure 7 plots the performance of three different versions. The first group of bars (named Default+) represents a version that uses loop tiling implemented in SUIF and block distribution (mapping) of iterations over cores (as in the case of the default version). The second group of bars (named Mapping) is the L1+L2+L3 variant explained earlier (reproduced here for comparison). Finally, the last group of bars show the results obtained when using the Mapping+Scheduling version (i.e., the L1+L2+L3 version followed by our scheduling scheme). Our first observation is that using conventional data locality optimizations (instead of the hierarchy and sharing aware mapping and scheduling) generates about 7.3% improvement in execution times on average. This is much lower compared to the average execution time improvement of 14.1% produced by the Mapping version. In other words, careful mapping of iterations to cores is very important in multicore. The second observation is that the Mapping+Scheduling version achieves an average execution time improvement of about 18%, which means careful scheduling of loop iterations (assigned to cores) considering the “timing” of data sharing can bring further benefits over optimized mapping alone.

### Table 2. Benchmarks used in our study.

<table>
<thead>
<tr>
<th>Appl.</th>
<th>Brief Description</th>
<th>Misses (%)</th>
<th>Exec Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mxm</td>
<td>Matrix-matrix multiplication</td>
<td>37.1</td>
<td>6.8</td>
</tr>
<tr>
<td>rbo</td>
<td>Red-black overrelaxation</td>
<td>27.7</td>
<td>19.2</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition</td>
<td>18.9</td>
<td>26.3</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multigrid solver (SPECOMP [4])</td>
<td>16.4</td>
<td>20.8</td>
</tr>
<tr>
<td>cg</td>
<td>Conjugate gradient (NAS [5])</td>
<td>8.8</td>
<td>13.2</td>
</tr>
<tr>
<td>fma3d</td>
<td>Finite element crash simulation (SPECOMP [4])</td>
<td>9.5</td>
<td>9.6</td>
</tr>
<tr>
<td>rcomm</td>
<td>Rendering through predefined objects like sprites</td>
<td>37.1</td>
<td>6.6</td>
</tr>
<tr>
<td>draw 1.1</td>
<td>Vector drawing library</td>
<td>26.8</td>
<td>19.2</td>
</tr>
<tr>
<td>draw 1.1</td>
<td>Vector drawing library</td>
<td>26.8</td>
<td>19.2</td>
</tr>
</tbody>
</table>
3. Comparison against Hierarchical Tiling and Polyhedral Approach

In our next set of experiments, we compare our approach against two alternate schemes. The first of these is a hierarchical tiling approach where an iteration space is tiled multiple times (hierarchically) considering the underlying cache topology. For example, when targeting the Dunnington architecture, the loop nest is tiled three times; the first tiling is based on L3 cache capacity; the second one is based on L2 capacity; and the last one is based on L1 capacity. After this tiling, the outer tiles (i.e., those based on L1) are distributed across cores. The reason why we perform experiments with this version is to see whether a simple hierarchical tiling strategy generates similar results to our mapping scheme. The second scheme that we test is based on a recent study [13] which implements a polyhedral approach for code mapping in multicores. From the results in Figure 8, one can observe that our scheme (Mapping+Scheduling) generates better results than hierarchical tiling for all application programs tested. This is because it takes into account not just cache capacities (as in the case of hierarchical tiling) but also cache topology, which in turn leads to better exploitation of data reuse across cores. As compared to the polyhedral approach, our scheme generates better results in six of our eight applications, resulting in an additional execution time improvement of 7% on average in these applications. In four of the applications we generate better results than [13], mainly because the code transformations determined by the two approaches were different. In the remaining two cases we perform better, the main reason is the high overheads involved with the code generated using the polyhedral model. Compared to our linear algebraic model, a polyhedral model has two main drawbacks. First, it generally has higher overheads at runtime due to the large number of checks, branch operations, and complex loop bound computations. Second, transformations applied to a statement necessarily touch all instances of that statement, leading to both code bloat and increased loop overhead. For example, when the approach in [13] is applied to the codes in our experimental suite, we observed about a 3x increase in code size, which may not be acceptable in certain environments. The polyhedral model generated slightly better results (around 1%) in only two benchmarks (mgrid and draw 1.1), due to the fact that it produced a different transformation than ours. Overall, the results plotted in Figures 6, 7, and 8 clearly underline the importance of cache hierarchy-aware data locality optimization for multicores.

4. Impact of Separator Vectors

Figure 9 plots the improvements in execution times (over the default version) with different selection vectors. On the x-axis of this graph, (0, 0) corresponds to the separator vectors determined by the strategy in Section 6.1. (−, 0) corresponds to moving the first separator vector toward left (i.e., reducing the number of reuses to be exploited in L1 layer and increasing the number of reuses to be exploited in L2 layer), whereas (+, 0) corresponds to moving the first separator vector toward right (i.e., increasing the number of reuses to be exploited in L1 and reducing the number of reuses to be exploited in L2 layer). In both the cases, the second separator vector is kept the same. Similarly, (0, −) and (0, +) correspond to moving the second separator vector toward left and right, respectively. One can see from these results that both (−, 0) and (+, 0) generate lower performance than (0, 0), though for different reasons. (0, 0) utilizes the available L1 space very well and trying to increase the number of reuses to be exploited in L1 layer (i.e., (+, 0)) puts more pressure in the cache and, as a result, L1 caches overflow, resulting in performance degradation. On the other hand, (−, 0) leads to underutilization of the L1 cache space. The same observation also explains why we have relatively lower performance when (0, +) or (0, −) is adopted.

9. Related Work

Prior work on data locality optimizations are mostly in the context of single core machines, and include linear loop transformations [23, 30], loop tiling [15, 18], and data layout optimizations [7]. Since most of these efforts target single core architectures (i.e., private caches), they are in a sense complementary to our approach. However, interactions between our locality oriented mapping/scheduling strategy and conventional locality optimizations require further study. Prior code and data mapping strategies – developed in the context of traditional parallel systems – include [3, 7, 10, 11, 20]. Prior parallelization, scheduling, and mapping strategies outside the multicore domain also include [8, 9, 19, 22]. The main goal behind most of these studies is to exploit data-computation affinity, that is, to collocate computation and data carefully to improve locality of accesses. Our target architecture (multicores with shared cache hierarchies) and optimization goal (maximizing on-chip cache locality for shared data) are different from the targets (mostly distributed memory architectures with a couple of studies on SMPs) and goals (maximizing the number of local data accesses) of these prior studies. Consequently, our proposed algorithms are different from theirs.

Emergence of multicore architectures renewed interest in code parallelization [6, 16, 25, 27]. However, these parallelization strategies do not consider the target architecture explicitly. Locality-oriented efforts targeting multicore machines include [31] and [21]. Sarkar and Tullsen propose a data-cache aware compilation to find a layout for data objects which minimizes inter-object conflict misses [28]. Zhang et al study reference affinity and present a heuristic model for data locality [32]. Kandemir et al [12] explore a code optimization strategy targeting only caches shared by all cores. In comparison, our strategies can work with any on-chip multi-layer cache hierarchy. Kandemir et al also propose a code mapping strategy based on polyhedral model
[13]. In our experimental analysis, we compared our approach against [13]. Kurzak et al [17] present a scheduling strategy targeting dense linear algebra operations. In comparison, we propose a linear algebra based optimization framework that can handle both mapping and scheduling problem. In addition, the uniqueness of our approach is that we determine a cache topology-aware mapping based on reuse distances, rather than the amount of data shared as [12]. Our approach works better in practice, since even if two iterations share a large amount of data, their reuse distance may still be long, in which case, mapping them to the two cores that share a cache at a very high level (e.g., L1) may not help in exploiting the data reuse between them.

10. Concluding Remarks

The goal of this work is to maximize constructive sharing and minimize destructive interferences, in the context of multicores, through careful mapping and scheduling of loop iterations considering on-chip cache topology. We implemented our mapping and scheduling schemes by extending a compiler infrastructure and tested their impact using a commercial multicore machine and eight application programs. The experimental results collected show that our mapping scheme improves the L1, L2 and L3 cache miss rates by, respectively, 13.5%, 16.6% and 14.9% on average, resulting in an average execution time improvement of 14.1%. Our scheduling support takes these improvements to higher values.

References