# Cache Topology Aware Computation Mapping for Multicores *

Mahmut Kandemir†, Taylan Yemliha‡, SaiPrashanth Muralidhara†,
Shekhar Srikantaiah†, Mary Jane Irwin†, Yuanrui Zhang†

†The Pennsylvania State University, University Park, PA. {kandemir, smuralid, srikanta, mji, yuazhang}@cse.psu.edu
‡Syracuse University, Syracuse, NY. {tyemliha}@syr.edu

## Abstract

The main contribution of this paper is a compiler based, cache topology aware code optimization scheme for emerging multicore systems. This scheme distributes the iterations of a loop to be executed in parallel across the cores of a target multicore machine and schedules the iterations assigned to each core. Our goal is to improve the utilization of the on-chip multi-layer cache hierarchy and to maximize overall application performance. We evaluate our cache topology aware approach using a set of twelve applications and three different commercial multicore machines. In addition, to study some of our experimental parameters in detail and to explore future multicore machines (with higher core counts and deeper on-chip cache hierarchies), we also conduct a simulation based study. The results collected from our experiments with three Intel multicore machines show that the proposed compiler-based approach is very effective in enhancing performance. In addition, our simulation results indicate that optimizing for the on-chip cache hierarchy will be even more important in future multicores with increasing numbers of cores and cache levels.

***Categories and Subject Descriptors*** B.3.2 [*Memory Structures*]: Design Styles – Cache memory; C.4 [*Performance of Systems*]: Design Studies

***General Terms*** Management, Design, Performance, Experimentation, Algorithms

***Keywords*** Cache, Multi-level, Multicore, Topology-aware, Compiler

## 1. Introduction

Running into the power wall has forced processor designers to look to other than clock-frequency scaling to continue the scaling in processor performance that we have come to expect over the last few decades. Since Moore Law continues to provide a doubling in the number of transistors in every technology generation (every two to three years), one way to double the performance without increasing the clock frequency is to double the number of processors (cores) on the chip. The first (non-embedded) dual-core architecture was delivered in 2001 and now quad-core architectures are becoming increasingly common. As the microprocessor world experiences this transition from out-of-order execution and clock-frequency scaling to scaling through the placement of multiple processor cores on one chip, programming these multicore machines and optimizing software for them is emerging as a very challenging task. It is clear that, without proper application software and system software support that can take advantage of multiple cores on the same chip, we will not be able to extract the expected performance benefits from these machines. Unfortunately, developing reusable and portable software for emerging multicore architectures is not easy as the architectures of these machines (even those produced by the same chip manufacturer) can be very different from one another.

Consider for example three sample multicore architectures, shown in Figure 1, from Intel: Harpertown, Nehalem, and Dunnington. Each of these architectures has two sockets; Harpertown and Nehalem have 8 cores and Dunnington has 12 cores. We see that these three architectures have very different on-chip cache hierarchies. For example, while Harpertown has only two levels of on-chip caches (L1 and L2), the other two machines have three levels of on-chip caches (L1, L2, and L3). Also, in Nehalem, each core has a private L2, whereas in Dunnington each L2 is shared by a pair of cores. Considering the fact that the on-chip cache hierarchy plays a very important role in determining the overall performance of a multi-threaded application [8, 11, 21, 32, 33, 38], one can expect that a given parallel code can have dramatically different behaviors on these three machines. As a result, if a multi-threaded code written and optimized for one multicore architecture is to be ported to another, either one must accept the resulting performance degradation or one must perform extensive code re-tuning and re-optimization by hand to customize the code for the new target architecture. Clearly, neither of these options is very desirable from a programmer's viewpoint.

In this work, we explore a third option – enlisting the compiler's help in customizing a program for a target multicore on-chip cache hierarchy. Traditionally, optimizing compilers have three layers: front-end (which implements scanning, parsing and intermediate representation construction), middle-end (which mostly implements architecture-independent optimizations such as code restructuring and data layout reorganization), and back-end (which implements architecture-dependent optimizations such as instruction scheduling and register allocation).[1] However, we believe that, with multicore architectures, the middle-end layer of an optimizing

* This research is supported in part by NSF grants CNS #0720645, CCF #0811687, OCI #0821527, CCF #0702519, and CNS #0720749, and a grant from Microsoft Corporation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PLDI'10,* June 5–10, 2010, Toronto, Ontario, Canada.

Copyright © 2010 ACM 978-1-4503-0019-3/10/06. . . $10.00

[1] We should mention that a few optimizing compilers that target uniprocessor systems consider cache capacity during the optimizations in the middle layer.

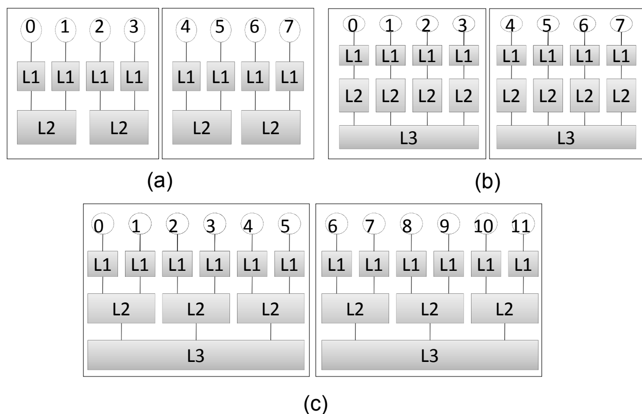



**Figure 1.** Multicore cache architectures: (a) Harpertown (b) Nehalem (c) Dunnington. In each figure, ovals denote processors (cores) and L1, L2 and L3 represent on-chip caches. Typically, L2 has higher data access latency than L1, and L3 has higher access latency than L2. Each of these architectures has two sockets (denoted by rectangles), each holding half of the cores and half of the cache components.

**Figure 2.** Normalized parallel execution times for a multi-threaded application (*galgel*) on different Intel multicore architectures.

compiler should consider the on-chip cache hierarchy as well; that is, the cache hierarchy of the target multicore architecture should be *exposed* to the middle-end layer of the compiler. While we present a detailed experimental analysis later in the paper, Figure 2 illustrates the normalized parallel execution times for a multi-threaded application (named *galgel,* which is a fluid dynamics code that implements an analysis of oscillatory instability) on our three Intel multicore machines, shown in Figure 1. The three bars in each group correspond, from left to right, to the code versions generated while targeting the on-chip hierarchies of Harpertown, Nehalem and Dunnington, respectively. For example, the first bar in the second group gives the execution time of the Harpertown version of the code when run on Nehalem. The results for each group of bars are normalized with respect to the best-performing version, and the Dunnington version is executed using 8 threads (1 thread per core) when ported to the other machines. We observe from these results that, on each multicore machine, the version that has been specialized for that machine (by considering the on-chip cache hierarchy) generates the best result. Exploiting the on-chip cache hierarchy is critical for achieving the best performance on a given multicore machine. As an example, if one were to run code optimized for Harpertown on Nehalem, one would see a performance hit of 26%. In the remainder of this paper, we present a compiler based, cache topology aware code optimization strategy that customizes the data access pattern of an application for the on-chip cache hierarchy of a target multicore architecture.

Our target application domain is array/loop intensive programs. These programs cover at least two important classes of applications. First, many scientific and engineering applications operate on large data arrays using nested loops. Examples include applications for climate modeling, astrophysics, computational chemistry, bioinformatics, and nuclear structure exploration [20]. Second, many data-intensive embedded applications such as multimedia codes operate on large data arrays that represent signals using structured loop nests [14]. The important point is that these array/loop intensive programs (from both the scientific/engineering domain and the embedded world) can potentially take advantage of emerging multicore systems very well, as their parallelization has been thoroughly studied and many loop level code/data restructuring techniques (e.g., loop reordering, iteration space tiling, data layout optimizations) have been developed to maximize their parallelism.[2] What is missing, however, is the *data locality* aspect, i.e., how one can maximize data locality for a given on-chip cache hierarchy. Unfortunately, existing data locality (cache performance) optimization techniques – in the context of both single processor machines and discrete multi-processor machines – do not specifically address *shared on-chip caches* that are now almost ubiquitous across the spectrum of multicore machines.

We make three main contributions in this paper:

• We present a cache topology aware loop iteration distribution scheme for multicore systems. This strategy uses polyhedral arithmetic to distribute the iterations of a parallel loop across multiple cores such that the performance of the on-chip cache hierarchy is maximized. We describe our compiler algorithm and give an example to illustrate how it operates in practice.

• We explain how this strategy can be extended to handle data dependencies (i.e., if we want to execute in parallel the iterations of a loop nest that have loop-carried dependencies) and to exploit intra-core data locality.

• We implemented in a compiler and evaluated our proposed scheme using a set of twelve applications and three commercial multicore machines. In addition, to study some of our experimental parameters in detail and to explore future multicores (with larger numbers of cores and deeper on-chip cache hierarchies), we also present results from a simulation based study.

The results collected from our experiments with three Intel multicore machines show that the proposed compiler-based approach is very effective in enhancing the performance of on-chip cache hierarchies of multicores, and as a result, significant overall performance improvements are possible. For example, our loop distribution scheme generated 16%, 17% and 21% improvements in execution cycles, on average, over a state-of-the-art data locality optimization strategy, when targeting the Harpertown, Nehalem and Dunnington systems, respectively. Our results also indicate that optimizing for the on-chip cache hierarchy will be even more important in future multicores with increasing numbers of cores and cache levels.

The next section explains the problem of optimizing data locality in shared on-chip caches of multicores. The details of our proposed iteration distribution and scheduling schemes are presented in Section 3. Section 4 presents an evaluation of the proposed schemes using both real multicore machines and a simulation framework. Section 5 discusses related work and the paper is concluded in Section 6 with a summary of our major observations.

[2] The main parallelization strategy for these applications is *loop-level parallelism* in which iterations of a given (parallel) loop are distributed across processors.

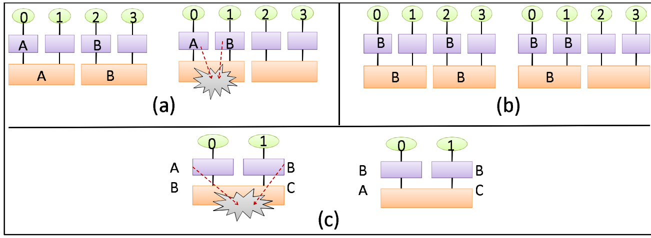



**Figure 3.** Motivation for distributing loop iterations and reorganizing loop iterations. A, B and C denote different data blocks. (a) The case with iterations that do not share data. (b) The case with iterations that share data. (c) Impact of local scheduling. In the original schedule (left), core 0 accesses first A and then B, and core 1 accesses first B and then C. It is assumed that A and B conflict in the shared cache, and therefore, access to B by core 0 will result in a cache miss. The revised schedule on the right fixes this problem.

## 2. The Problem of Enhancing the Performance of On-Chip Hierarchy

In multicore machines, the cores on the same chip can share an on-chip cache space. How they share this cache space can make a difference in runtime performance as cache sharing across different cores can be constructive or destructive [13]. In the former case, concurrently scheduled threads share data through the cache and/or their access patterns are such that they do not displace each other's data from the cache. In the latter case, however, accesses from concurrently scheduled threads displace each other's data. This occurs frequently when multiple applications are executed at the same time on the same multicore machine (as different applications do not share memory-resident data), but it can also happen when data accesses from the threads that belong to the same application (and share data between them) conflict in the shared cache [37].

We now focus on distributing and scheduling the iterations of a parallel loop across multiple cores. We say that "two cores have affinity at cache L" if both have access to that cache. For example, in the Dunnington architecture in Figure 1(c), cores 0 and 1 have affinity at the first (leftmost) L2 cache as both are connected to it (as well as the L3 cache they share). Consider two cores that have affinity at cache L. The chances for experiencing destructive interactions in the cache space are higher if these cores do not share data, as this will typically increase the number of distinct data elements that compete for the limited cache space. Therefore, if two iterations (that belong to a parallel loop) do not share data, it is better to assign those iterations to cores that do not have affinity at any cache (if possible). Figure 3(a) illustrates this scenario (A and B represent two different data blocks). The left figure in (a) shows the case when the iterations that access A and B are assigned to two cores with no affinity, whereas the right one illustrates a potential conflict if the iterations that do not share data are assigned to cores with cache affinity. On the other hand, if two iterations do share data, it is better to assign them to cores that have affinity at some cache (see the right part of Figure 3(b)). If we do not do so (see the left part of Figure 3(b)), this leads to data replication across multiple on-chip caches at the same layer, which in turn reduces the effective on-chip cache capacity, and ultimately hurts overall performance.

The two scenarios illustrated in Figures 3(a) and (b) provide motivation for distributing loop iterations across available cores carefully by considering on-chip cache topology. Figure 3(c), on the other hand, provides motivation for reorganizing the loop iterations assigned to a core, i.e., scheduling them. This reorganization (scheduling) has two goals: (i) improving the shared cache locality beyond what can be achieved by loop distribution alone and (ii) improving L1 cache locality. In this particular case, if two data accesses from two cores conflict in the shared cache (e.g., when the access sequence is B[core 1], A[core 0], B[core 0] and C[core 1] and A and B conflict in the shared cache space), the reuse of B by core 0 will not be converted into locality because the second access to B will result in a miss (see the left part of Figure 3(c)). One solution is to change the order of data accesses for this core, as shown on the right part of the figure. That is, the goal should be to ensure that when two cores do share data, they access that data at similar times to increase the chances for exploiting the resulting reuse while the shared data is still in the shared cache space. In addition, for a given core, the successively scheduled computations should reuse data as much as possible to improve private (L1) cache locality.

To summarize, there are two complimentary problems. The first is the distribution of loop iterations across cores, that is, the *loop iteration-to-core assignment problem.* This problem should be attacked considering the data sharing across loop iterations as well as the cache topology of the target multicore system. The cores with affinity should be given iterations that share data (to minimize conflicts in the cache), whereas the cores without affinity should be given iterations that do not share data (to minimize data replication within the on-chip cache space). In this paper, our main goal is to address this assignment problem. Once, this problem is handled, one can also be interested in a second problem, namely, that of *scheduling* iterations assigned to a core to further improve data locality. Clearly, in some cases, a compiler/user can decide to run the iterations of a loop with loop-carried dependencies in parallel and use proper synchronization to ensure correct semantics. In this paper, we also discuss the necessary enhancements to our base approach to handle such cases that involve data dependencies (see Section 3.5.2).

## 3. Cache Hierarchy Aware Loop Iteration Distribution and Scheduling

### 3.1 Assumptions

The input to our approach is a set of loop iterations that are to be distributed over available cores for parallel execution. For now, we restrict ourselves to *fully-parallel loops*, i.e., the loops in which there are no loop-carried dependencies across iterations. As a result, any distribution of the iterations of such a loop is legal (i.e., semantically correct). However, as will be demonstrated later, different distributions can have significantly different data locality (on-chip cache performance) behavior. The reason why we first focus on fully-parallel loops is that, in practice, many commercial compilers execute only such loops in parallel. While there are certainly cases where benefits are possible when the iterations of a loop with dependencies are distributed across available cores and the correctness is ensured through inter-core synchronization, such cases are rare in practice. In fact, in the parallel benchmarks that we have, only 14% of the loops that are executed in parallel have some sort of dependencies across iterations; the rest are fully parallel. Still, we do discuss in this paper how our base approach can be enhanced to handle data dependencies.

### 3.2 Representing Data Elements, Loop Iterations, and Array Accesses

In our compiler framework, we use sets to represent array elements, loop iterations, and mappings between them. Consider for example the code fragment in Figure 4, written in a C-like pseudo-language.

```
int A[0..D1-1][0..D2-1]
...
for(i1=0;i1<Q1;i1++)
 for(i2=2;i2<Q2+2;i2++)
   ...A[i1+1][i2-1]...
```

**Figure 4.** Example code fragment.

```
int B[0..m-1]
...
for(j=2k;j<m-2k+1;j++)
  B[j] = B[j] + B[2k+j]
    + B[j-2k]
```

**Figure 5.** Example code fragment.

The iteration space $\mathcal{K}$ of this loop nest can be expressed as follows:

$$\mathcal{K} = \{(i_1, i_2) | (0 \leq i_1 \leq Q_1 - 1) \wedge (2 \leq i_2 \leq Q_2 + 1)\},$$

where $\wedge$ denotes the "and" operator. We use $\vec{I} = (i_1\ i_2)^T$ to denote the iteration vector for this loop nest. The set of values $\vec{I}$ can assume, i.e., iteration space $\mathcal{K}$, is the set of all combinations of the values of the loop indices. Similarly, the set

$$\mathcal{D} = \{(d_1, d_2) | (0 \leq d_1 \leq D_1 - 1) \wedge (0 \leq d_2 \leq D_2 - 1)\}$$

defines the data elements declared in the code fragment. Further, the array reference in the loop body can be expressed as:

$$\mathcal{R} \ = \ \{(i_1, i_2) \rightarrow (d_1, d_2) | (i_1, i_2) \in \mathcal{K} \wedge (d_1, d_2) \in \mathcal{D} \wedge (d_1 = i_1 + 1) \wedge (d_2 = i_2 - 1)\},$$

where $\rightarrow$ represents a mapping from the iteration space to the data space. Note that, while the first line of this expression enforces the data and loop indices to be within the array and loop bounds, respectively, the second line captures the relationship between these indices. We use $\mathcal{R}(\vec{I})$ to indicate the array element accessed by iteration $\vec{I}$ using reference $\mathcal{R}$ (the fragment in Figure 4 has only one reference, which is $A[i_1 + 1][i_2 - 1]$).

These sets can be viewed as a *polyhedral model* in which the objects of interest (loop iterations and data elements) are represented as integer valued points in various regions of different spaces and the mappings (such as $\mathcal{R}$) are used to connect these spaces. Two of the widely used polyhedral tools are the Omega Library [23] and Polylib [3]. In this work, we use the Omega Library to capture the iteration and data sets and the mappings between them as well as to generate output code. However, its choice is orthogonal to the main focus of this work, i.e., if desired, it can be replaced by another polyhedral tool.

### 3.3 Tagging Iterations

We assume that data manipulated by an application are partitioned into equal-sized *blocks*, and use $\beta_0$, $\beta_1$, $\beta_2$, $\cdots$ $\beta_{n-1}$ to denote these blocks, assuming a total of $n$ data blocks. We want to emphasize that (i) this partitioning is a logical one, i.e., the data are not physically divided into blocks, (ii) blocks do not cross array boundaries, i.e., each array starts with a new block, (iii) blocks are numbered sequentially in some order (though the exact ordering scheme used is not very important, one would expect two consecutive blocks of an array to be assigned consecutive numbers, and the number of the first block of the next array is going to be larger by 1 than the last block of the current array), and (iv) all blocks collectively cover all the data elements accessed by the loop nest being optimized.

Similar to data arrays, iterations of a loop nest are partitioned into *groups*. We use notation $\xi^{\Delta}$ to indicate an iteration group $\xi$ with *tag* $\Delta$. This tag is determined based on the *data block access pattern* exhibited by the iteration group. Specifically, if $\Delta$ is $\delta_0\delta_1 \cdots \delta_{n-2}\delta_{n-1}$, for each $j$ between 0 and $n-1$, $\delta_j$ is 1 if all iterations $\vec{I} \in \xi^{\Delta}$ access a data element $\vec{d}$ that belongs to $\beta_j$; otherwise, $\delta_j$ is set to 0. Therefore, the tag associated with an iteration group captures the set of data blocks accessed by the iterations in that group as well as those that are not accessed. For instance, assuming $n = 4$, $\xi^{1100}$ represents an iteration group that accesses the first two data blocks and does not access the last two data blocks. As another example, $\xi^{1000}$ represents an iteration group that accesses only the elements of the first data block. Note that $\xi^{1100}$ and $\xi^{1000}$ do not share any iteration between themselves, as all the iterations in the former one access the second data block, whereas none of the iterations in the latter one accesses the second data block. Based on this discussion, we can say that two different iteration groups do not share any iterations between them, and all iteration groups collectively cover the entire iteration set of the loop nest being distributed, that is,

$$\mathcal{K} = \bigcup_{\Delta} \xi^{\Delta}.$$

### 3.4 Going from Data Blocks to Iteration Groups

We now discuss how we obtain iteration groups from data blocks and generate code that enumerates the iterations in an iteration group. This code generation capability is important because, once we distribute loop iterations across available processor cores and reorganize (schedule) them, we need to generate code for each core.[3]

Let us focus, without loss of generality, on $\xi^{\Delta}$, where $\Delta = \delta_0\delta_1 \cdots \delta_{i-1}\delta_i\delta_{i+1} \cdots \delta_{n-2}\delta_{n-1} = 11 \cdots 100 \cdots 00$, that is, the first $i$ bits are 1 and the rest are 0. Then, assuming that there are $R$ references in the loop nest ($\mathcal{R}_0, \cdots, \mathcal{R}_{R-1}$), we can write:

$$\xi^{\Delta} = \{\vec{I} | \forall q, 0 \leq q \leq (i-1)[\exists r, 0 \leq r \leq R-1 \text{ s. t. } \mathcal{R}_r(\vec{I}) \in \beta_q] \text{ and } \neg\exists q', i \leq q' \leq (n-1)[\exists r, 0 \leq r \leq R-1 \text{ s. t. } \mathcal{R}_r(\vec{I}) \in \beta_{q'}]\}.$$

The first line of this expression captures the iterations that access all data blocks $\beta_q$ where $0 \leq q \leq (i-1)$, while the second line indicates that none of these iterations access any data block $\beta_{q'}$ where $i \leq q' \leq (n-1)$. In our approach, the iteration distribution across the cores is carried out at an *iteration group granularity*. That is, each core is assigned a number of iteration groups. Therefore, once we determine the set of iteration groups assigned to a core, we need to generate code that enumerates the iterations in those groups. Note that, for a given $\xi^{\Delta}$, the Omega Library can be used for generating code for it. Specifically, the $codegen(.)$ utility provided by the Omega Library helps us generate the code (typically in form of a set of nested loops) that enumerates the iterations in $\xi^{\Delta}$ (and we repeat this for all iteration groups based on the scheduling determined for the core).

### 3.5 Algorithms

In this section, we first describe our iteration distribution scheme for the fully parallel case, i.e., when the loops to be executed in parallel do not have any loop-carried dependencies. We then discuss the case with data dependencies, and explain how data locality can be further improved through intra-core scheduling of iteration blocks.

#### 3.5.1 Iteration Distribution Algorithm

Our loop distribution algorithm is shown Figure 6. This algorithm takes the set of iterations to be distributed and the cache topology of the target multicore machine as the input, and produces iteration groups/clusters to be scheduled on each core such that cache sharing among the iteration groups is maximized.

In the initialization step, the iterations of the loop nest are grouped into *iteration groups* based on the similarity of their tags. Recall that a tag of an iteration is a *signature* of the data blocks accessed by that iteration, and that a set of iterations are clustered into

[3] Note that, the iteration groups allocated to a core and their scheduling defines the *thread* that will be executed on that core.

an iteration group if they have the same tag. After this initialization, we build a *graph* with these iteration groups being nodes. An edge between two nodes of this graph has a weight equal to the number of common “1”s between the tags of the two nodes. Therefore, the weight of an edge between any two nodes is an indicator of the degree of data block sharing (sort of affinity) between the iterations of the two nodes.

We then cluster the nodes (iteration groups) hierarchically according to the *cache hierarchy tree.* The cache hierarchy tree is a tree representation of the on-chip cache hierarchy with the last level cache as the root. By considering the cache hierarchy tree in clustering the iteration groups, our scheme customizes the clustering for the cache topology of the target multicore architecture. We start at the root and cluster the iteration groups, level by level, until we reach the leaf level. To emphasize, the graph mentioned above captures the degree of data sharing between iteration groups, whereas the cache hierarchy tree represents the target cache topology.

In clustering for a given level in the cache hierarchy, we consider the bitwise sum of the tags of all the nodes in the cluster as the *tag of the cluster*. We compute the dot product (denoted using “•” in the algorithm) of the tags of two clusters as the qualitative measure of affinity and use that as a measure in our clustering algorithm. The dot product measures the degree of data block sharing between any two clusters and hence is used as a qualitative measure for clustering. The number of clusters formed at each level is the same as the number of child nodes in the cache hierarchy tree. If the number of clusters formed is less than the number of child nodes, we go on splitting the clusters (i.e., form smaller clusters) until the number of clusters is equal to the number of child nodes.

After the clustering step, we are left with the required number of iteration group clusters for the current level of the cache hierarchy. Our goal in the next step is load balancing, i.e., balancing the sizes of the iteration group clusters. In this context, size of a cluster is the total number of iterations present in the iteration group cluster – the sum of the sizes of all the iteration groups present in the cluster. For the load balancing step, we use a greedy approach to balance the number of iterations[4] across the clusters. Using a balance threshold (which is the maximum tolerable imbalance across the iteration counts of different cores), we compute an upper and a lower limit for the size of each cluster. Note that the balance threshold is a tunable parameter. We evict iteration groups from the largest sized cluster to the smallest in each step. While doing so, we ensure that the dot product of the tag of the evicted iteration group and that of the recipient cluster is maximized. Also, we only evict if, after the eviction, the donor cluster size does not drop below the lower limit of the cluster size and the recipient cluster size does not go above the upper limit. If no such eligible iteration group is found for eviction, we break an iteration group according to the balance threshold requirements and evict that iteration group. We repeat this step until all the iteration group clusters are within the limits of the balance threshold.

Our algorithm repeats the clustering and load balancing steps at each level of the cache hierarchy tree. After all the levels of the cache hierarchy tree are considered, the number of clusters we have is equal to the number of cores in the target multicore architecture. Note that, by performing the clustering and load balancing at each level of the cache hierarchy, our approach considers the data block sharing at each level of the on-chip cache and optimizing the same at each cache level.

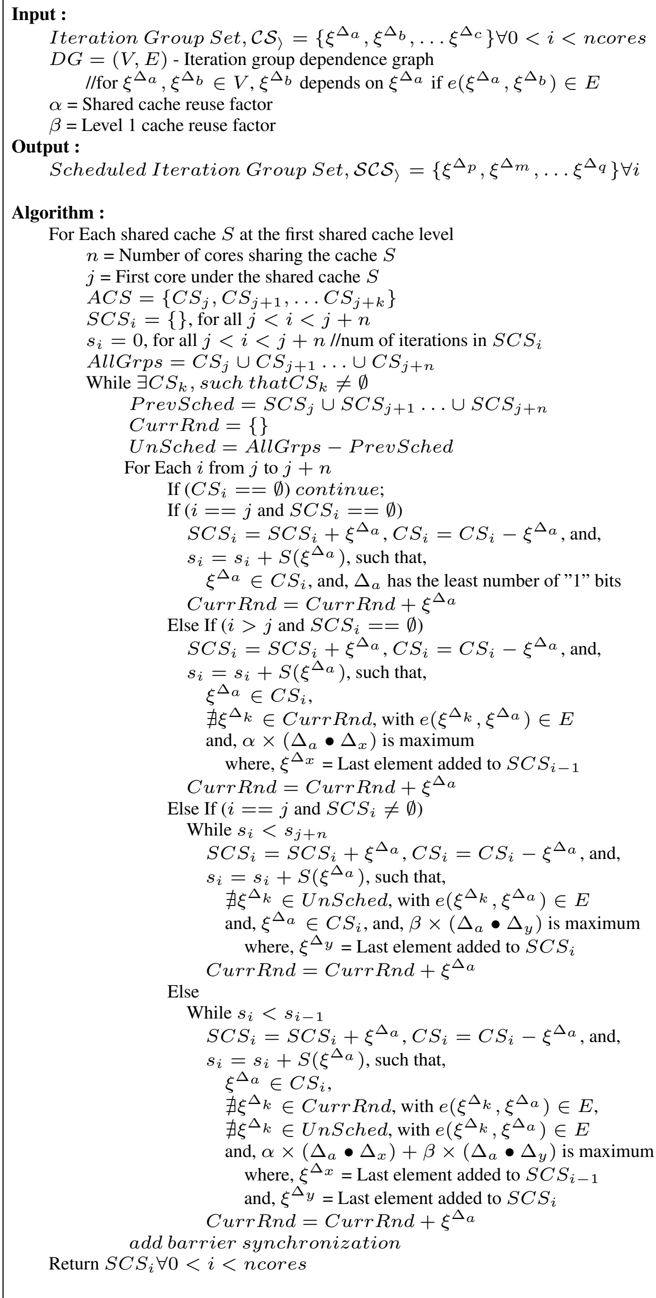
[4] Although iteration count is not the ideal metric for load balancing, it performs much better than not balancing at all, and is implementable within a compiler.

**Input :**
$Loop\ Iteration\ Set, \mathcal{IS} = \{\psi_0, \psi_1, \ldots \psi_m\}$
$\mathcal{IS}$ is the set of all iterations in the loop nest
$Data\ Block\ Set, \mathcal{DS} = \{\beta_0, \beta_1, \ldots \beta_{r-1}\}$
$\mathcal{DS}$ is the set of all equal sized data element blocks accessed by the loop nest
$Architecture\ Description, \mathcal{A} = \{T, N\}$
$\mathcal{T}$ is the cache hierarchy tree with the last level cache as the root node and $N$ is the number of cores
/*Off chip memory is treated as the root if there are more than one last level caches*/
$BalanceThreshold$ = Maximum tolerable imbalance in iteration counts
**Output :**
$Iteration\ Group\ Set, \mathcal{CS} = \{c_0, c_1, \ldots c_k\}$
$c_i = \{\psi_j, \psi_l, \ldots \psi_z\}$, $N$ is the number of cores
**Algorithm**

**Initialization :**
Initialize tags:
Assign a tag $\Delta_j = \delta_0\delta_1 \ldots \delta_{r-1}$ to iteration $\psi_i$
where, $\delta_k = 1$ if $\psi_i$ accesses data block $\beta_k$
Iteration Group $\xi^{\Delta} = \{\psi_k,\ such\ that\ \psi_k\ has\ tag\ \Delta\}$
/*As there are $r$ data blocks, there are $2^r$ tags and consequently, $2^r$ possible iteration groups*/
Size of the iteration group, $\xi^{\Delta}$, $S(\xi^{\Delta}) = |\xi^{\Delta}|$
Build Graph:
Build a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$,
$\mathcal{V} = \{\xi^{\Delta_1}, \xi^{\Delta_2}, \ldots \xi^{\Delta_{2^r}}\}$
$\mathcal{E} = \{(\xi^{\Delta_i}, \xi^{\Delta_j})\ such\ that\ \omega(\xi^{\Delta_i}, \xi^{\Delta_j}) = number\ of\ "1"\ bits\ in\ \Delta_i \wedge \Delta_j\}$

**Hierarchical Iteration Distribution:**
$HierLevel$ = root of the cache hierarchy tree, $T$
$NumClusters$ = degree of nodes at level " $HierLevel$ "
Cluster Set, $\mathcal{CS} = \{cs\}, cs = \{\{c\} \forall\ c \in V\}$
$While\ HierLevel\ \neq\ leaf\ level$:
New cluster set, $\mathcal{NCS} = \{\}$
For Each cluster $cs_i \in \mathcal{CS}$:
$\mathcal{CS} = \mathcal{CS} - cs_i$
$Totaliterations$ = total number of iterations in $cs_i$
**Clustering:**
While$(|cs_i| > NumClusters)$ :
For each cluster $c^{\alpha_p} \in cs_i$,
$c^{\alpha_p} = \{\xi^{\Delta_a}, \xi^{\Delta_b}, \ldots \xi^{\Delta_c}\}$
$\alpha_p = BitwiseSum(\Delta_a, \Delta_b, \ldots \Delta_c)$
$S(c^{\alpha_p}) = |\xi^{\Delta_a}| + |\xi^{\Delta_b}| + \ldots + |\xi^{\Delta_c}|$
Select and merge $c^{\alpha_p}$ and $c^{\alpha_q}$ in to a new cluster $c^{\alpha_{new}}$
$such\ that, \alpha_p \bullet \alpha_q (dotproduct)\ is\ maximized$
$\therefore c^{\alpha_{new}} = c^{\alpha_p} \cup c^{\alpha_q}$
If $(|cs_i| < NumClusters)$ :
//if current number of clusters < required number at this level
While$(|cs_i| \neq NumClusters)$ :
Select $c^{\alpha_q} \in cs_i$, such that $S(c^{\alpha_q})$ is maximum
Break $c^{\alpha_q}$ into two clusters
**Load balancing:**
After clustering, $cs_i = \{c^{\alpha_1} \ldots c^{\alpha_{NumClusters}}\}$
/*Use greedy approach to balance cluster sizes*/
$UpLimit = \frac{Totaliterations}{NumClusters} + BalanceThreshold$
$LowLimit = \frac{Totaliterations}{NumCusters} - BalanceThreshold$
While $\exists\ c^{\alpha_p} \in cs_i$, such that $S(c^{\alpha_p}) > UpLimit$:
Select $c^{\alpha_q} \in cs_i$ such that,
$S(c^{\alpha_q}) < LowLimit$
Evict some $\xi^{\Delta_a}\ from\ c^{\alpha_p}\ to\ c^{\alpha_q}$ such that,
$LowLimit < S(c^{\alpha_p}) < UpLimit$
$LowLimit < S(c^{\alpha_q}) < UpLimit$
and, $\Delta_a \bullet \alpha_q\ is\ maximum$
If no such node exists, split $\xi^{\Delta_a}$ such that
$S(c^{\alpha_p})$ and $S(c^{\alpha_q})$ are with in limits and evict as described above
For Each $c^{\alpha_p} \in cs_i$:
$\mathcal{NCS} = \mathcal{NCS} + \{\{\xi^{\Delta_a}\} \forall \xi^{\Delta_a} \in c^{\alpha_p}\}$
$\mathcal{CS} = \mathcal{NCS}$
$Hierlevel = Hierlevel + 1$,
Update $NumClusters$ to the degree of nodes at "$Hierlevel$"
After $h = log_2 N$ hierarchical levels, $\mathcal{CS} = \{c_0, c_1, \ldots c_k\}$
where, $N$ is the number of cores
**Return** $\mathcal{CS}$

**Figure 6.** Iteration distribution algorithm.



**Input :**

$Iteration\ Group\ Set, \mathcal{CS}_i = \{\xi^{\Delta_a}, \xi^{\Delta_b}, \ldots \xi^{\Delta_c}\} \forall 0 < i < ncores$

$DG = (V, E)$ - Iteration group dependence graph

//for $\xi^{\Delta_a}, \xi^{\Delta_b} \in V$, $\xi^{\Delta_b}$ depends on $\xi^{\Delta_a}$ if $e(\xi^{\Delta_a}, \xi^{\Delta_b}) \in E$

$\alpha$ = Shared cache reuse factor

$\beta$ = Level 1 cache reuse factor

**Output :**

$Scheduled\ Iteration\ Group\ Set, \mathcal{SCS}_i = \{\xi^{\Delta_p}, \xi^{\Delta_m}, \ldots \xi^{\Delta_q}\} \forall i$

**Algorithm :**

For Each shared cache $S$ at the first shared cache level
- $n$ = Number of cores sharing the cache $S$
- $j$ = First core under the shared cache $S$
- $ACS = \{CS_j, CS_{j+1}, \ldots CS_{j+k}\}$
- $SCS_i = \{\}$, for all $j < i < j+n$
- $s_i = 0$, for all $j < i < j+n$ //num of iterations in $SCS_i$
- $AllGrps = CS_j \cup CS_{j+1} \ldots \cup CS_{j+n}$
- While $\exists CS_k, such\ that CS_k \neq \emptyset$
  - $PrevSched = SCS_j \cup SCS_{j+1} \ldots \cup SCS_{j+n}$
  - $CurrRnd = \{\}$
  - $UnSched = AllGrps - PrevSched$
  - For Each $i$ from $j$ to $j+n$
    - If $(CS_i == \emptyset)$ $continue$;
    - If $(i == j$ and $SCS_i == \emptyset)$
      - $SCS_i = SCS_i + \xi^{\Delta_a}, CS_i = CS_i - \xi^{\Delta_a}$, and,
      - $s_i = s_i + S(\xi^{\Delta_a})$, such that,
        - $\xi^{\Delta_a} \in CS_i$, and, $\Delta_a$ has the least number of "1" bits
      - $CurrRnd = CurrRnd + \xi^{\Delta_a}$
    - Else If $(i > j$ and $SCS_i == \emptyset)$
      - $SCS_i = SCS_i + \xi^{\Delta_a}, CS_i = CS_i - \xi^{\Delta_a}$, and,
      - $s_i = s_i + S(\xi^{\Delta_a})$, such that,
        - $\xi^{\Delta_a} \in CS_i$,
        - $\nexists \xi^{\Delta_k} \in CurrRnd$, with $e(\xi^{\Delta_k}, \xi^{\Delta_a}) \in E$
        - and, $\alpha \times (\Delta_a \bullet \Delta_x)$ is maximum
          - where, $\xi^{\Delta_x}$ = Last element added to $SCS_{i-1}$
      - $CurrRnd = CurrRnd + \xi^{\Delta_a}$
    - Else If $(i == j$ and $SCS_i \neq \emptyset)$
      - While $s_i < s_{j+n}$
        - $SCS_i = SCS_i + \xi^{\Delta_a}, CS_i = CS_i - \xi^{\Delta_a}$, and,
        - $s_i = s_i + S(\xi^{\Delta_a})$, such that,
          - $\nexists \xi^{\Delta_k} \in UnSched$, with $e(\xi^{\Delta_k}, \xi^{\Delta_a}) \in E$
          - and, $\xi^{\Delta_a} \in CS_i$, and, $\beta \times (\Delta_a \bullet \Delta_y)$ is maximum
            - where, $\xi^{\Delta_y}$ = Last element added to $SCS_i$
        - $CurrRnd = CurrRnd + \xi^{\Delta_a}$
    - Else
      - While $s_i < s_{i-1}$
        - $SCS_i = SCS_i + \xi^{\Delta_a}, CS_i = CS_i - \xi^{\Delta_a}$, and,
        - $s_i = s_i + S(\xi^{\Delta_a})$, such that,
          - $\xi^{\Delta_a} \in CS_i$,
          - $\nexists \xi^{\Delta_k} \in CurrRnd$, with $e(\xi^{\Delta_k}, \xi^{\Delta_a}) \in E$,
          - $\nexists \xi^{\Delta_k} \in UnSched$, with $e(\xi^{\Delta_k}, \xi^{\Delta_a}) \in E$
          - and, $\alpha \times (\Delta_a \bullet \Delta_x) + \beta \times (\Delta_a \bullet \Delta_y)$ is maximum
            - where, $\xi^{\Delta_x}$ = Last element added to $SCS_{i-1}$
            - and, $\xi^{\Delta_y}$ = Last element added to $SCS_i$
        - $CurrRnd = CurrRnd + \xi^{\Delta_a}$
  - $add\ barrier\ synchronization$

Return $SCS_i \forall 0 < i < ncores$

**Figure 7.** Dependence-aware local iteration scheduling algorithm. This algorithm is invoked after the one in Figure 6.

### 3.5.2 Handling Data Dependencies

In the cache topology aware loop iteration distribution algorithm described in the previous section, we restricted ourselves to fully-parallel loops, i.e., loops in which there are no loop-carried dependencies across iterations. In cases where distributing the loop iterations with dependencies could bring high benefits, our algorithm can be extended by distributing the loop iterations with dependencies across available cores and ensuring correctness through inter-core synchronization.

There are at least two ways of extending our approach to handle loops with dependencies. First, we can ensure that the clustering algorithm clusters all iteration groups with loop carried dependencies together in a single cluster. This can be achieved by associating an infinite edge weight between iteration groups that have dependencies between them. This ensures correctness without the need for inter-core synchronization. However, conservatively clustering all dependent iteration groups together may reduce the benefits of parallelism in iteration group execution, and therefore, this approach may not be very effective when we have large number of dependencies. Alternatively, the clustering algorithm can treat loop carried dependencies between iteration groups as normal data block sharing. Therefore, in the presence of dependencies across loop iterations, the data sharing resulting from these dependencies is accounted for by the edge weights used to quantify the sharing of data between the iteration groups containing the respective iterations (provided they are different). Therefore, we can use the same global cache hierarchy aware loop iteration distribution algorithm described above (given in Figure 6) to improve data sharing. However, to ensure correctness, the dependencies can be detected during the local reorganization step (explained shortly) and corresponding inter-core synchronization directives can be inserted to enforce the dependencies. We now elaborate on this second option.

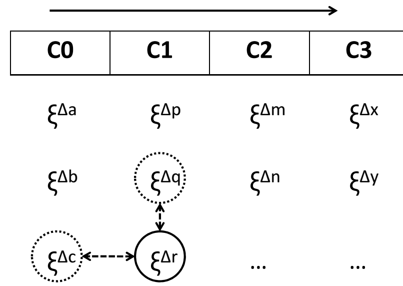



**Figure 8.** Scheduling order of iteration groups for a 4-core machine

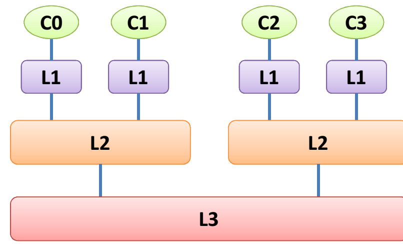
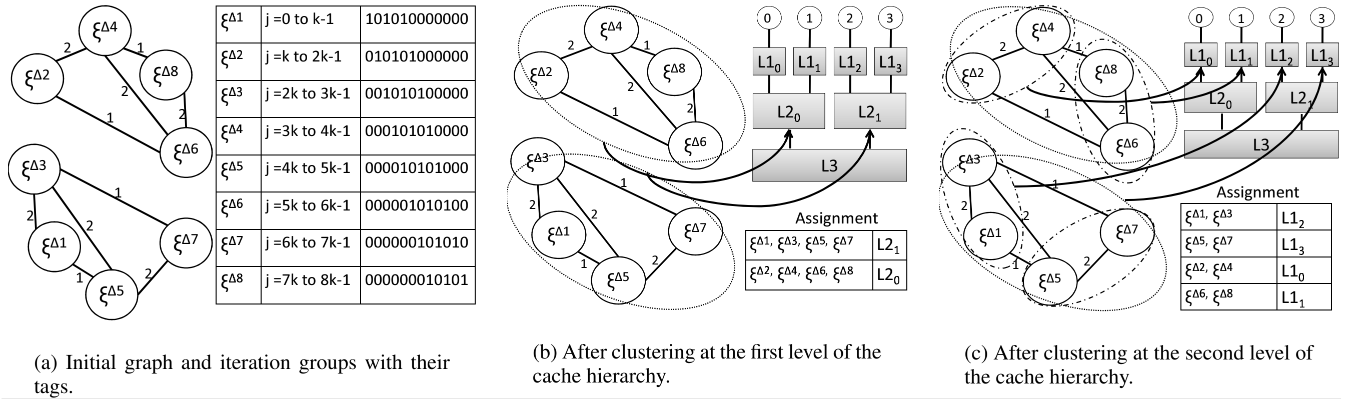



**Figure 9.** Target multicore architecture.

Note that so far we did not discuss how the iteration groups assigned to a core by our iteration distribution algorithm are scheduled (i.e., their order of execution). The important point is that if there are no loop carried dependencies, any scheduling of iterations groups is legal. However, if we have data dependencies, we need to determine a legal schedule that respects all dependencies. In our extension, the cache hierarchy aware loop distribution algorithm clusters the iteration groups as in the case of the dependence-free case. Once the iteration group clusters are computed, a local reorganization algorithm schedules the iteration groups assigned to cores such that the iteration group dependencies are preserved. In order to do this, when the iteration groups are scheduled in time, proper synchronization constructs are inserted. Figure 7 describes this algorithm in detail.

The dependence-aware local reorganization algorithm takes the iteration group set computed by the hierarchical clustering algorithm ($CS$) and the iteration group dependence graph ($DG$) as inputs. The iteration group dependence graph ($DG$) represents dependencies between the iteration groups. It contains an edge from




(a) Initial graph and iteration groups with their tags.

(b) After clustering at the first level of the cache hierarchy.

(c) After clustering at the second level of the cache hierarchy.

**Figure 10.** Example application of our scheme.

node (iteration group), $\xi^{\Delta_a}$ to $\xi^{\Delta_b}$, if at least an iteration in $\xi^{\Delta_a}$ depends on an iteration in $\xi^{\Delta_b}$. Note that an edge in $DG$ can go from an iteration group assigned to one core to an iteration group assigned to another core. Note also that $DG$ can potentially be a cyclic graph, since some iterations present in $\xi^{\Delta_a}$ can depend on iterations in $\xi^{\Delta_b}$, while other iterations present in $\xi^{\Delta_a}$ can be dependent on iterations in $\xi^{\Delta_b}$. We remove all the cycles in the dependence graph by merging the involved nodes and consequently convert the graph to an acyclic graph before proceeding. The algorithm then computes a schedule of iteration groups for each core with inserted synchronization constructs. We schedule the iteration groups starting with the first core. For the first core, we pick any iteration group that belongs to the cluster assigned to the first core and does not depend on any iteration group (i.e., it is scheduleable). We then consider all the iteration groups assigned to the second core and pick the iteration group which is not dependent on the last scheduled group on core one. We repeat this process for the second core till the total number of iterations assigned to the second core is at least as many as that assigned to the first core. When that happens, we move on to the next core and repeat the process. This way, we try to balance the iteration counts across the cores. After one round of scheduling for all the cores, we insert a barrier synchronization construct. Note that, not balancing the iteration counts in the above fashion can hurt performance in presence of barrier synchronization. We then start the second round of scheduling from the first core. In the second and later scheduling rounds, we also ensure that the scheduled iteration group does not depend on any iteration group yet to be scheduled. These scheduling rounds are repeated until all the iteration groups are scheduled. With this process, not only are there no dependencies between iteration groups scheduled in a round, but also iteration groups scheduled at any particular round can only be dependent on the iteration groups of the previous rounds. The dependencies between iteration groups are enforced by the inserted barrier synchronization construct.

### 3.5.3 Improving Cache Block Reuse

While the dependence-aware scheduling described in Section 3.5.2 schedules the iteration groups on cores in a loop-carried dependence-aware manner, it does not consider the data block sharing at different cache levels. The algorithm described in Figure 7 not only tries to perform dependence-aware scheduling, it also tries to improve both local level one cache reuse and the first shared level cache reuse. This local reorganization can be used with both the dependence and dependence-free cases.

The cache block reuse can be improved both at the local level one cache or at the first shared cache level. The local level one cache reuse can be improved by scheduling the iteration groups assigned to a core such that the tags of the contiguously scheduled iteration groups have the least possible Hamming Distance between them. On the other hand, shared level cache reuse can be improved by scheduling iteration groups with minimum Hamming Distance on the cores sharing the cache such that those iteration groups are executed simultaneously on the cores, thereby improving cache performance. Our algorithm exploits data locality in two directions: *horizontal* and *vertical.* Maximizing the dot product with the last scheduled iteration group on the previous core improves data block reuse at the first level of shared cache (horizontal). In comparison, maximizing the dot product with the last scheduled iteration group on the same core improves data block reuse on the local level one cache (vertical). We weigh these dot products with tunable parameters, $\alpha$ and $\beta$, so that the cache level at which the sharing needs to be improved can be customized by assigning suitable values to $\alpha$ and $\beta$. In Figure 8, in the third round of scheduling for core 1, we pick $\xi^{\Delta_r}$ because it maximizes the value $\alpha \times (\Delta_c \bullet \Delta_r) + \beta \times (\Delta_q \bullet \Delta_c)$. Therefore, to schedule $\xi^{\Delta_r}$ (circled) on core 1, we consider its left and upper neighbors (dotted circles) as indicated by the arrows in Figure 8. In this way, our approach takes care of the data reuse in both horizontal and vertical directions.

### 3.5.4 Example

Consider the example loop shown in the Figure 5. We consider a dependence-free case here for simplicity. This loop body has four references with each iteration accessing three elements of the same array. For illustrative purposes, we assume a data block size of $k$ elements. We also assume that the total number of data blocks is twelve, i.e., $m/k = 12$. The cache hierarchy of the target multicore architecture is depicted in Figure 9. The iterations of the loop can be divided into eight iteration groups based on the data blocks accessed. The iteration groups and the initial graph are shown in Figure 10(a).

We now go over our hierarchical loop iteration distribution scheme. The first step is to cluster the loop iteration groups for the L2 cache (as L3 is shared by all cores and is considered the root of the cache hierarchy tree). The graph and the assignment after the first level of clustering and loop iteration distribution are shown in Figure 10(b). Next, the loop iteration distribution is applied to each of the two clusters formed in the previous step. After this second and final level of clustering and load balancing, the iteration

| Core 0 | $\xi^{\Delta 2}, \xi^{\Delta 4}$ |
|---|---|
| Core 1 | $\xi^{\Delta 6}, \xi^{\Delta 8}$ |
| Core 2 | $\xi^{\Delta 1}, \xi^{\Delta 3}$ |
| Core 3 | $\xi^{\Delta 5}, \xi^{\Delta 7}$ |

**Figure 11.** Final assignments and schedule.

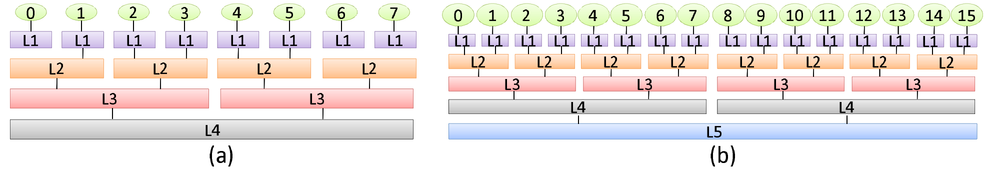



**Figure 12.** Architectures with complex on-chip cache hierarchies: (a) Arch-I and (b) Arch-II.

| | **Harpertown** | **Nehalem** | **Dunnington** |
|---|---|---|---|
| **Number of Cores** | 8 cores (2 sockets) | 8 cores (2 sockets) | 12 cores (2 sockets) |
| **Clock Frequency** | 3.2GHz | 2.9GHz | 2.4GHz |
| **L1** | 32KB, 8-way, 64-byte line, 3 cycle latency | 32KB, 8-way, 64-byte line, 4 cycle latency | 32KB, 8-way, 64-byte line, 4 cycle latency |
| **L2** | 6MB, 24-way, 64-byte line, 15 cycle latency | 256KB, 8-way, 64-byte line, 10 cycle latency | 3MB, 12-way, 64-byte line, 10 cycle latency |
| **L3** | - | 8MB, 16-way, 64-byte line, 30-40 cycle latency | 12MB, 16-way, 64-byte line, 32-40 cycle latency |
| **Off-Chip Latency** | ∼100 ns | ∼60 ns | ∼50 ns |

**Table 1.** Important parameters for our three multicore machines.

clusters are assigned to the cores as shown in Figure 10(c). Finally, the iteration groups assigned to each core are scheduled using the local iteration group scheduling algorithm given in Figure 7. The final iteration group assignments and the corresponding schedule for each core is depicted in Figure 11.

## 4. Experimental Evaluation

### 4.1 Setup

In our experimental evaluation, we used both commercial multicore machines and a simulation framework (for sensitivity experiments). Table 1 gives the important characteristics of the three Intel machines we used: Harpertown, Nehalem, and Dunnington (all illustrated in Figure 1).

The set of applications used in this study is given in Table 2. We experimented with two types of applications, namely, sequential and parallel. For the sequential benchmarks, we executed a parallelism extraction phase before our scheme could be applied. Since the selection of the loop parallelization strategy used is orthogonal to our scheme (which performs iteration distribution across cores), here we only summarize it. This strategy, which is similar to the one discussed in Anderson's thesis [1], identifies (for each loop nest) the outermost loop that does not have any loop-carried dependence and parallelizes it. This tends to exploit coarse grain parallelism and minimize inter-core synchronization. For the parallel benchmarks on the other hand, our strategy is slightly different. Since the loops to execute parallel are already identified, what we need to do is to apply our scheme to distribute (redistribute if necessary) iterations to cores. Our parallel benchmarks are from three sources: *applu*, *galgel* and *equake* are from SpecOMP [16]; *cg* and *sp* are from NAS [4]; and *bodytrack, facesim* and *freqmine* are from the Parsec benchmark suite [7]. Two of our sequential benchmarks, *namd* and *povray*, are from the Spec2006 suite [18]. The remaining two benchmarks, *mesa* and *H.264*, are two serial applications we maintain locally. The data set sizes of these applications varied from 4.6MB to 2.8GB. Unless otherwise stated, the default data block size used in our experiments is 2KB. Our iteration distribution algorithm operates with a given data block size. However, we use a strategy to determine the block size (and consequently the number of blocks). This strategy tries to ensure that the total size of data manipulated by even the "most aggressive" iteration group does not exceed L1 cache capacity. An iteration group becomes most aggressive when it has all 1's in its tag. We first profile the application and determine the size of the total data it manipulates and then, assuming a tag size of $k$ bits (unknown), we determine the amount of data accessed by the most aggressive iteration group. This value is in terms of $k$ and we solve for $k$ such that this value does not exceed L1 capacity. Please note that this sets an upper bound, and any lower value would be good as well.

| **Benchmark** | **Description** | **Time** |
|---|---|---|
| applu | Parabolic/elliptic partial differential equations | 2.4 sec |
| galgel | Fluid dynamics: analysis of oscillatory instability | 4.6 sec |
| equake | Finite element simulation; earthquake modeling | 3.5 sec |
| cg | Estimates the largest Eigenvalue of a sparse matrix | 11.2 sec |
| sp | Solves a synthetic system of nonlinear PDEs | 3.8 sec |
| bodytrack | Body tracking of a person | 6.7 sec |
| facesim | Simulates the motions of a human face | 6.3 sec |
| freqmine | Frequent itemset mining | 8.6 sec |
| H.264 | Video compression: equivalent to MPEG-4 AVC | 5.0 sec |
| mesa | 3D graphics library | 7.9 sec |
| namd | biomolecular systems simulation | 7.1 sec |
| povray | image ray-tracing | 8.0 sec |

**Table 2.** Our applications. The last column gives the execution time when the original application is executed on a single core of the Dunnington machine.

In this work, we compare our scheme against two base cases. The first of these, called *Base,* is the original application code without any modification (except for parallelization when the input application is sequential). The second base case, referred to as *Base+,* represents the state-of-the-art in data locality (cache performance) enhancement. It improves data locality, for each core, by applying a set of well-known locality optimizations, which include loop permutation (changing the order in which loop iterations are executed) and iteration space tiling (also known as blocking, which implements a blocked version of the code to improve temporal reuse in outer loop positions). Please note that Base+ contains a comprehensive set of well-established locality optimizations including linear transformations and tiling. In fact, the linear transformations we used were very similar to those discussed in [43]. However, we found that non-unimodular transformations (such as scaling) did not bring additional benefits over unimodular ones in our benchmarks. To approximate the ideal tile size (blocking factor), we experimented with different tile sizes and selected the one that performed the best. The important point to emphasize is that the set of iterations assigned to each core is the same in both Base and Base+; the only difference is the order used to execute these iterations. Therefore, in a sense, Base+ can be viewed as intra-core locality optimization, or an extension of single core locality optimization to the multicore case (i.e., we apply conventional locality optimization to each core separately). In the rest of this section, we refer to our proposed loop iteration distribution strategy as "Topology Aware". Unless explicitly stated, Topology Aware does not in-

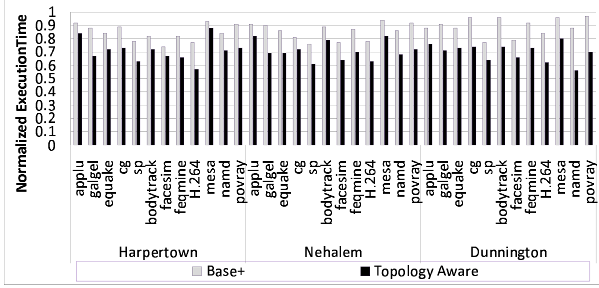



**Figure 13.** Normalized execution time results for Base+ and Topology Aware.

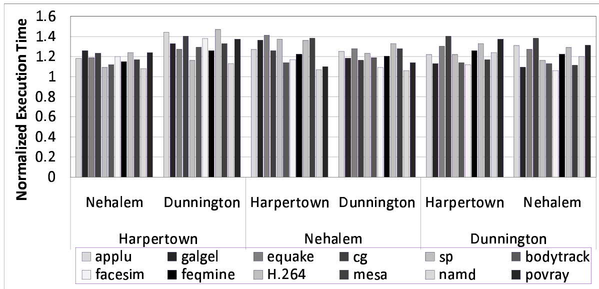



**Figure 14.** Cross comparison among three Intel multicore architectures.

clude the loop scheduling step, which is part of the algorithm in Section 3.5.3. Instead, once the iteration distribution is carried out, the iteration groups assigned to each core are scheduled considering only data dependencies. In order to have a better understanding of their behavior, we study loop distribution and loop scheduling in isolation as well as when they are combined. Note that, in Base, Base+, and Topology Aware, the set of iterations executed in parallel is the same; the only difference is in the way in which these iterations are partitioned across the cores and scheduled within a core.

In our experiments, the load balance threshold mentioned in Section 3.5.1 is set to 10%, and the $\alpha$ and $\beta$ parameters discussed in Section 3.5.3 are both set to 0.5 (i.e., equal weights). When we use Intel machines, the results are obtained using all available cores (8 in the case of Harpertown and Nehalem and 12 in the case of Dunnington). All the versions used in this work have been implemented using Microsoft's Phoenix infrastructure [31]. Specifically, after the input code is analyzed by Phoenix, we build the polyhedral framework and pass it to the Omega Library [23]. The code returned from the Omega Library enumerates the iterations in the iteration groups assigned to the cores and is passed back to the Phoenix intermediate format. In our experiments with the Intel machines, we used the Intel compiler (with the most powerful optimization flag available) as our back-end compiler. In our simulations on the other hand, the back-end compiler we used is gcc (again, with the highest optimization flag).

Before moving to the discussion of our experimental analysis, we want to mention that the increase in compilation times due to our scheme varied between 65% and 94% (depending on the application being compiled) over the compilation that includes a parallelization step (but does not include any data locality optimization). We also observed that the our approach did not have a significant impact on the instruction cache miss rates (less than 1% increase in all the application codes we tested).

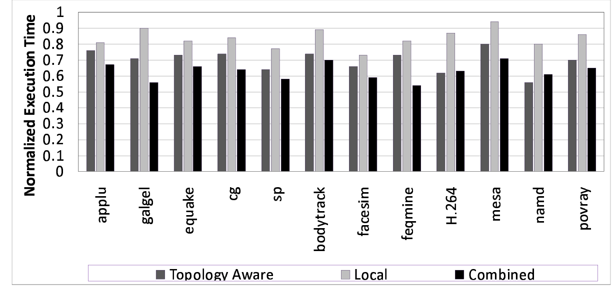



**Figure 15.** Influence of local iteration reorganization (scheduling).

## 4.2 Results

**Goals.** We have four main goals in our experimental evaluation. First, we want to demonstrate that, for a given multicore architecture, our approach generates better results than both Base and Base+. Second, we want to show that a version customized for a specific multicore architecture may not perform well when executed on another multicore architecture. Third, we want to see whether considering the entire cache hierarchy is a must for achieving maximum savings. Fourth, we want to illustrate that our approach is expected to perform well in future multicore systems with larger numbers of cores and deeper on-chip cache hierarchies. All the results presented in this subsection are *normalized* with respect to Base.

**Results on Commercial Machines.** Figure 13 gives the execution cycles for the benchmarks in our three multicore machines, normalized with respect to Base (remember that all different versions use the same back-end compiler). Our main observation from these results is that, for all three multicores and all application programs, our approach generates better results than both Base and Base+, indicating the importance of cache topology aware loop iteration assignment. We also observe that the difference between our scheme and Base+ is higher in Dunnington. This is because Dunnington has a more complex on-chip cache topology, which makes optimizing for data locality even more important. The average performance improvements our topology aware approach brings over Base and Base+ are about 28% and 16%, respectively, in the case of Harpertown. The corresponding savings are 29% and 17% for Nehalem and 30% and 21% for Dunnington. Since Topology Aware, Base, and Base+ have exactly the same set of loop iterations executed in parallel and only differ in how these iterations are distributed and scheduled, this difference across execution times is due to entirely on-chip cache behavior. For example, we observed that, in Dunnington, our approach reduced the L1, L2 and L3 cache misses on average by 18%, 39%, 47%, respectively, over the Base version. The corresponding cache miss reductions over Base+ were 16%, 31% and 37%.

**Cross-Machine Results.** Our next set of results quantify the performance of a multi-threaded code generated for a specific multicore when run on another multicore, and are presented in Figure 14. The first group of bars in this graph corresponds to the execution of the Nehalem version on Harpertown, and the second group represents the execution of the Dunnington version on Harpertown. The remaining groups are interpreted similarly. These results clearly underline the importance of customizing loop iteration distribution to the specific on-chip cache topology. The important point to note here is that, in the Harpertown machine, using the multi-threaded versions generated for Nehalem and Dunnington results, on average, in about 17% and 31% worse performance, respectively,

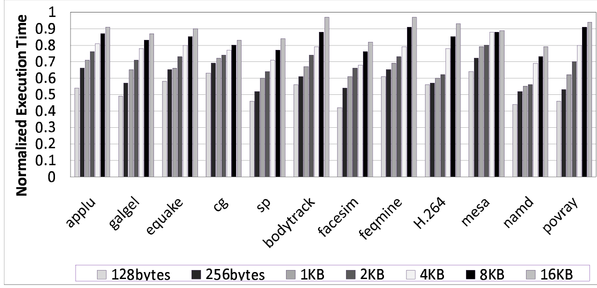



**Figure 16.** Impact of data block size.

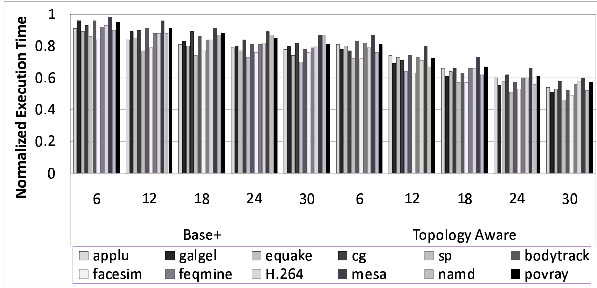



**Figure 17.** Impact of increasing the number of cores (in a Dunnington-like architecture).

than the version customized for Harpertown. Similarly, using the Harpertown and Nehalem versions in Dunnington leads to 24% and 21% average degradation, and using the Harpertown and Dunnington versions in Nehalem results in 25% and 19% performance degradation, respectively, on average. These results combined with those in Figure 13 help us to conclude that, if one is to map an application to a target multicore architecture, it is not a good idea to use a conventional data locality optimization strategy or to simply use the code optimized originally with a different multicore in mind. Instead, the best results are achieved by customizing loop iteration distribution considering the underlying cache topology.

**Impact of Intra-Core Scheduling.** Recall that Section 3.5.3 presented a local iteration reorganization (scheduling) strategy, which can be applied after the global iteration distribution scheme. Figure 15 plots the normalized results, for Dunnington, that summarize the influence of this algorithm in improving overall performance. In this bar-chart, for each application, we have three bars: global loop distribution alone (Topology Aware), local iteration reorganization alone (Local), and combined (when the local reorganization is applied after the global distribution scheme). Note that in Local the iteration distribution across cores is either the default one indicated by the original parallel code or random (in the case of sequential codes parallelized by our loop parallelization step). Two trends can be observed from this plot. First, Local generates similar (slight better) results than Base+ (see Figure 13 for the Base+ results). This is not very surprising as both Local and Base+ improve the locality behavior from an individual core's perspective. While Base+ tries to do that using conventional restructuring techniques, Local takes a more data centric approach and attempts to cluster iterations with similar data block access patterns. In addition, as explained earlier, Local also considers affinity with other cores when performing scheduling for a given core. Second, the best results are achieved when both these schemes are used together. Specifically, this combined scheme obtains an average performance improvement of around 37%. Although not presented here, we also performed experiments with different values for $\alpha$ and $\beta$ (recall

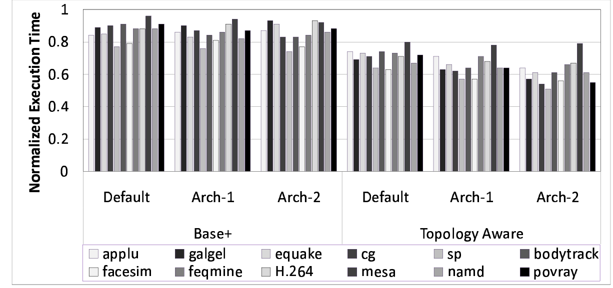



**Figure 18.** Impact of the on-chip cache hierarchy (Default represents the configuration in the commercial Dunnington machine).

that the default value used so far is 0.5 for both). We observed that giving them equal values generated the best results. Specifically, if $\beta$ is too big, the potential locality in the shared caches is missed, and if $\alpha$ is too big, L1 locality starts to suffer.

**Sensitivity Experiments.** In the rest of our experiments, we conduct a sensitivity study with Topology Aware. In these experiments, our starting point is the Dunnington architecture (see Figure 1(c)). We first present the sensitivity of our savings to the data block size. Recall that the default block size used in our experiments was 2KB. One can observe from Figure 16 that, as expected, smaller data block sizes are better as they lead to smaller iteration groups which in turn result in a finer granular clustering by our algorithm (Figure 6). While this result motivates small block sizes, a smaller block size increases overall compilation time. For example, we observed that, as we move from 2KB to 256 bytes, the compilation time increased by more than 80%.

For our experiments described below, we used Simics [28], a simulator that can simulate multiprocessor architectures. We also used GEMS [29], which is a set of modules for Simics that enables detailed (cycle accurate) simulation of different types of multiprocessor systems, including multicores. In Figure 17, we give the normalized results obtained by Topology Aware and Base+ when the number of cores is increased. Recall that, in Dunnington, the total number of cores is 12.[5] We see from these results that the effectiveness of our approach increases as we increase the number of cores (for each experiment, we added a six more cores to the architecture shown in Figure 1(c)). For example, the average performance improvement our approach brings over Base jumps from 29% to 46% as we move from 12 cores (in our default configuration) to 24 cores. The main reason for this is that a higher number of cores typically lead to more sparse data access patterns from the perspective of a single core, and this in turn causes the performance of Base to suffer. Since the results in Figure 17 are normalized with respect to the Base scheme, we see an improvement. These results are important mainly because future multicore machines are expected to accommodate large core counts [9].

Figure 18 illustrates the results from our simulations with deeper on-chip cache topologies shown in Figure 12. It needs to be emphasized that Arch-I is more complex than our default Dunnington architecture (see Figure 1(c)) and Arch-II is more complex than Arch-I. One can observe from Figure 18 that Topology Aware performs better as we have deeper cache hierarchies (the best improvements are obtained with Arch-II). Again, this result is very important, because (considering current trends) one can ex-

[5] It needs to be noted that the 8 core results are slightly different from those presented for Dunnington earlier. This difference is because while the former results are obtained on real machines, the latter results are obtained using simulation.

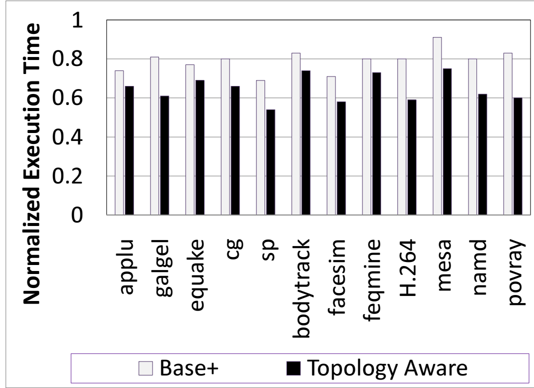



**Figure 19.** Impact of the total data set size / the cumulative on-chip cache capacity.

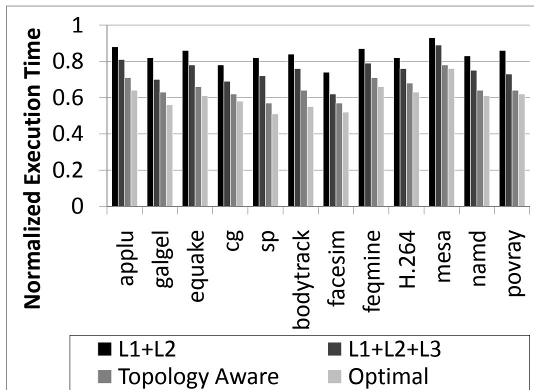



**Figure 20.** Results with different versions and optimal scheme.

pect future multicores to have increasingly deeper on-chip cache hierarchies [9].

Our next set of experiments were designed to explore what happens when the values of the parameter total data set size/cumulative on-chip cache space is increased. Since increasing data set sizes of our applications is not trivial (as it requires understanding the details of the underlying algorithms implemented by the application), we instead changed the on-chip cache capacities. Specifically, maintaining the original Dunnington topology, we cut the capacity of each cache component (L1, L2 and L3) by half. We see from Figure 19 that our scheme performs better with smaller cache capacities. More specifically, the average performance improvements brought by Base+ and Topology Aware are approximately 21% and 33%, respectively. These savings jumped to 29% and 41%, respectively, when loop distribution was combined with loop scheduling (detailed results are omitted due to space concerns).

**Results with Different Versions and Comparison with Optimal Savings.** Focusing now on the architecture in Figure 12(a), we try to answer the question of whether it is really important to consider the entire cache hierarchy. In Figure 20, L1+L2 and L1+L2+L3 refer to two different versions of our scheme (Topology Aware) where only L1 and L2, and only L1, L2 and L3 are considered, respectively. We see that, on average, considering all levels in the hierarchy (L1+L2+L3+L4) bring 21.8% and 12.7% improvement over the L1+L2 and L1+L2+L3 versions, respectively. In other words, for the best results, it is necessary to consider the entire cache hierarchy in distributing loop iterations among cores. Figure 20 also illustrates the results from an optimal scheme. To obtain these results, we determined the ideal iteration group-to-core mapping (for each parallel loop nest) using integer linear programming (which took up to 23 hours in some cases). We see that the performance difference between the mapping generated by our scheme and the optimal mapping is around 7.6%.

## 5. Related Work

There have been several architectural level schemes to optimize shared cache management [44] [10] [5] [19]. Zhang et al propose a scheme that employs a small victim cache to improve the performance of the shared level 2 cache in multicores [44]. Chang and Sohi propose to combine the advantages of both shared and private caches using a unified, cooperative caching scheme [10]. Beckmann et al propose that, by monitoring the workload behavior and using controlled selective cache block replication, performance can be significantly improved [5]. Hsu et al discuss various policies that can be employed in multicore shared cache management [19]. There have also been schemes that partition shared caches of multicores in order to improve the overall throughput in most cases and fairness, QoS in a few cases. These schemes include dynamic cache way partitioning [38] and fairness aware partitioning [25].

Liu et al discuss different organizations for the last level of cache [27]. Youn et al also recognize the importance of a combined approach [40]. Their scheme employs a private L2 cache architecture, while emulating a shared L2 cache through evictions into peer L2 private caches. Speight et al present simple architectural extensions for effective management of L2 and L3 caches in multicores [36]. Kim et al state that large cache designs will be limited by growing wire delays and propose a nonuniform cache architecture to placate the effects of growing wire delays [24]. Beckmann and Wood propose a cache management scheme that incorporates various latency management techniques [6]. They also show that block migration is less effective in commercial workloads than expected.

Prior parallelization, scheduling, and mapping strategies outside the multicore domain include [15, 17, 26, 35]. There have also been compiler based schemes aimed at cache management for multicores. Sarkar and Tullsen propose a data-cache aware compilation to find a layout for data objects which minimizes inter-object conflict misses [34]. Anderson et al propose using sharing rules to avoid data races [2]. Chen et al use a compiler directed approach to increase the idle periods of communication channels by reusing the same set of channels for as many communication messages as possible, there by reducing the power requirement in network-on-chip based multicores [12]. Kandemir et al [22] discuss a multicore mapping strategy that does not customize mapping based on target on-chip cache hierarchy. Zhang et al [45] evaluate the impact of cache sharing on modern parallel workloads. Zhang et al study reference affinity and further present a heuristic model for data locality [41]. Markatos et al propose a loop iteration distribution scheme that balances the workload, minimizes synchronization overhead and also co-locates the iterations with the data they access [42]. Li et al propose to not only balance the workload but also take data locality in to account [43]. In contrast to the above two schemes ([42] and [43]), our proposed scheme not only balances the workload but also takes the cache hierarchy of the underlying multicore architecture into account while deciding the iteration scheduling. Although not presented here due to space concerns, our initial experience with dynamic scheduling schemes like [42] did not generate good results on the Harpertown and Dunnington machines, mostly due to the cost of dynamic iteration distribution.

Our approach differs from the above efforts in that it tries to improve the performance of shared on-chip caches using loop distribution and loop scheduling. In doing so, it takes the target cache topology as input. It is also complementary to many of these prior studies. For example, in a setting where multiple multi-threaded applications exercise the same multicore machine, an OS based scheme can partition shared caches across different applications,

and our scheme can optimize the performance of each application individually.

## 6. Concluding Remarks

In order to take advantage of emerging multicore architectures, we need compilers that can optimize a given application for the target architecture. Unfortunately, current multicore architectures are very different from each other and we expect this dissimilarity to grow wider in future generations of multicores. This makes it very difficult to develop architecture agnostic high level code and data optimizations. The two main contributions of this paper are an iteration distribution algorithm and a loop scheduling algorithm targeting at improving the performance of on-chip cache hierarchies of emerging multicore systems. We implemented our approach using a compiler infrastructure, and performed experiments on real architectures, and also conducted a simulation based study. Our results are encouraging and show that considering on-chip cache topology makes significant difference in performance. Our results also show that the code optimized using our proposed scheme remains within 8% of the schedule generated by an optimal strategy.

## References


[1] J. M. Anderson. Automatic Computation and Data Decomposition for Multiprocessors. *Ph.D Thesis,* Stanford University, March 1997.

[2] Z. R. Anderson et al. Lightweight annotations for controlling sharing in concurrent data structures. *SIGPLAN Not.*, 44(6):98–109, 2009.

[3] R. Bagnara et al. The PARMA polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[4] D. Bailey et al. The NAS Parallel Benchmarks 2.0, NASA. *Technical Report,* 1995.

[5] B. M. Beckmann et al. ASR: Adaptive selective replication for CMP caches. In *Proc. MICRO*, 2006.

[6] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proc. MICRO*, 2004.

[7] C. Bienia et al. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. PACT*, 2008.

[8] R. Bitirgen et al. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proc. MICRO*, 2008.

[9] S. Borkar et al. Platform 2015: Intel processor and platform evolution for the next decade. *Technical Report,* Intel Corporation, 2005.

[10] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proc. ISCA*, 2006.

[11] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. ICS*, 2007.

[12] G. Chen et al. Compiler-directed channel allocation for saving power in on-chip networks. In *In Proc. POPL*, 2006.

[13] S. Chen et al. Scheduling threads for constructive cache sharing on CMPs. In *Proc. SPAA*, 2007.

[14] F. Catthoor et al. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, Boston, 2002.

[15] A. Darte et al. Scheduling the Computations of a loop nest with respect to a given mapping. In *Proc. Europar*, 2000.

[16] SPEC OMP V3.2. http://www.spec.org/omp/

[17] P. Feautrier. Scalable and structured scheduling. *Int. J. Parallel Program.* 34, 5, 2006.

[18] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[19] L. R. Hsu et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. PACT*, 2006.

[20] INCITE Leadership Computing. Technical Report, http://www.er.doe.gov/ ascr/incite/.

[21] R. Iyer et al. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, 2007.

[22] M. Kandemir et al. Optimizing shared cache behavior of chip multiprocessors. In *Proc. MICRO*, 2009.

[23] W. Kelly et al. The Omega Library interface guide. *Technical Report,* University of Maryland, 1995.

[24] C. Kim et al. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGPLAN Not.*, 37(10):211–222, 2002.

[25] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. PACT*, 2004.

[26] A. Legrand et al. Mapping and Load-Balancing Iterative Computations. *IEEE TPDS*, 2004.

[27] C. Liu et al. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proc. HPCA*, 2004.

[28] P. S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[29] M. M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[30] The OPENMP API specification for parallel programming. http://openmp.org/wp/

[31] Phoenix Compiler Infrastructure. *Technical Report,* Microsoft. https://connect.microsoft.com/Phoenix.

[32] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO*, 2006.

[33] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *Proc. PACT*, 2006.

[34] S. Sarkar and D. M. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *Proc. HiPEAC*, 2008.

[35] S.K. Singhai and K.S. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, vol. 40, no. 6, 1997.

[36] E. Speight et al. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(2):346–356, 2005.

[37] S. Srikantaiah et al. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proc. ASPLOS*, 2008.

[38] G. E. Suh et al. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[39] P. Viana et al. Configurable cache subsetting for fast cache tuning. In *Proc. DAC*, 2006.

[40] S. Youn et al. A reusability-aware cache memory sharing technique for high-performance low-power CMPs with private l2 caches. In *Proc. ISLPED*, 2007.

[41] C. Zhang et al. A hierarchical model of data locality. In *Proc. POPL*, 2006.

[42] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. In *Proc. IPDPS*, 1994.

[43] H. Li et al. Locality and Loop Scheduling on NUMA Multiprocessors. In *Proc. ICPP*, 1993.

[44] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proc. ISCA*, 2005.

[45] E. Zhang et al. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?. In *In Proc. PPOPP*, 2010.