Optimizing Shared Cache Behavior of Chip Multiprocessors *

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Sai Prashanth Muralidhara
Pennsylvania State University
smuralid@cse.psu.edu

Sri Hari Krishna Narayanan †
Pennsylvania State University
snarayan@cse.psu.edu

Yuanrui Zhang
Pennsylvania State University
yuazhang@cse.psu.edu

Ozcan Ozturk
Bilkent University
ozturk@cs.bilkent.edu.tr

ABSTRACT

One of the critical problems associated with emerging chip multiprocessors (CMPs) is the management of on-chip shared cache space. Unfortunately, single processor centric data locality optimization schemes may not work well in the CMP case as data accesses from multiple cores can create conflicts in the shared cache space. The main contribution of this paper is a compiler directed code restructuring scheme for enhancing locality of shared data in CMPs. The proposed scheme targets the last level shared cache that exist in many commercial CMPs and has two components, namely, allocation, which determines the set of loop iterations assigned to each core, and scheduling, which determines the order in which the iterations assigned to a core are executed. Our scheme restructures the application code such that the different cores operate on shared data blocks at the same time, to the extent allowed by data dependencies. This helps to reduce reuse distances for the shared data and improves on-chip cache performance. We evaluated our approach using the Splash-2 and Parsec applications through both simulations and experiments on two commercial multi-core machines. Our experimental evaluation indicates that the proposed data locality optimization scheme improves inter-core conflict misses in the shared cache by 67% on average when both allocation and scheduling are used. Also, the execution time improvements we achieve (29% on average) are very close to the optimal savings that could be achieved using a hypothetical scheme.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Compilers*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithm, Performance, Design, Experimentation

*This research is supported in part by NSF grants CNS 0720645, CCF 0811687, CCF 0702519, CNS 0202007 and CNS 0509251, a grant from Microsoft Corporation and support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRC's Focus Center Research Program.

†Currently a post-doctoral researcher at The Argonne National Laboratory, USA.

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.
MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

1. INTRODUCTION

Several factors including increasing complexities of single core architectures, increasing power consumption, verification/ validation costs due to these complexities, and the desire to extract increasing levels of parallelism have led to the emergence of chip multiprocessors (CMPs). All major chip manufacturers today are producing multi-core chips [1, 28, 41, 44, 35, 31, 37, 30, 29]. CMPs will not only be the processors used in laptop and desktop machines, but they will also be the mainstream components for next-generation large-scale parallel machines targeted at high-performance applications. CMPs have the potential to provide several orders of magnitude increase in performance for a wide range of important applications using both thread-level parallelism and instruction-level parallelism on a fabric which enables very fast inter-processor communication.

One of the critical issues in CMPs is the management of the shared on-chip cache space (L2 or L3). This is not a trivial task as data accesses from different processors (cores) can create conflicts in the shared cache, which can in turn reduce overall performance significantly. For example, a data access from one core can displace a data element brought to the shared cache by another core. As a result, when the displaced data element is requested again (a reuse) by any core, we incur a cache miss. While such inter-core conflicts can occur frequently when the involved cores are executing threads that belong to different applications, it has been shown [51] that shared cache conflicts are frequent even across the threads of the same application that are mapped to different cores. Unfortunately, many of the existing OS-level or architecture-level shared cache partitioning schemes [52, 14, 45, 46] tend to increase potential conflicts among the threads that belong to the same application. The main reason for this is the fact that most such schemes partition a given cache space across competing applications by distributing cache-ways. For example, a multi-threaded application can get, say, 4 ways after a 16-way set-associative shared cache is partitioned across concurrently-executing applications. Obviously, fewer ways mean in general a higher number of inter-core conflict misses across the threads of the application, which makes proper management of per-application cache space even more important.

One impact of conflicts in the shared on-chip cache is that the same data element may have to be brought to the cache

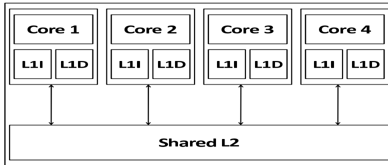

Figure 1: Target CMP architecture.

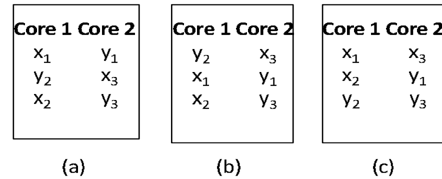

Figure 2: Different cases regarding data reuse exploitation. (a) original case, (b) exploiting only intra-core reuse, (c) exploiting both intra-core and inter-core reuses (result generated by our approach).

more than once by different cores. Clearly, in the ideal case, we want a data element to be brought to the shared cache space only once, i.e., all reuses of the data should take place while it resides in the on-chip cache space. Unfortunately, both limited cache space and conflicts in data access patterns of different threads may not always allow this ideal case to be achieved. Our goal in this paper is to restructure a given application code such that the number of inter-core[1] conflict misses is reduced. In this paper, we say that an "inter-core conflict miss" occurs when a data displaced from the shared cache by a core (e.g., through an access to some other data) is later requested by a different core. Specifically, this paper makes the following contributions:

- For a set of multi-threaded applications executing on shared cache based CMPs, we present a distribution of data reuse distances. We also associate this distribution with application performance (inter-core conflict misses and execution latency). Our experiments with Splash-2 [56] and two Parsec [8] applications show that inter-core conflict misses constitute nearly 51% of total L2 misses on average. We also quantify the maximum potential benefits that could be obtained from a hypothetical scheme that eliminates all inter-core conflict misses in the L2. Our results indicate that eliminating inter-core cache misses completely can reduce parallel execution time by 33% on average.
- We present a compiler based code restructuring scheme oriented toward minimizing the number of inter-core conflict misses. The unique characteristic of this scheme is that, using two complementary steps (*allocation*, which determines the set of loop iterations assigned to each core, and *scheduling*, which determines the order in which the iterations assigned to a core are executed), it restructures the application code such that different cores operate on shared data blocks at the same time, to the extent allowed by data dependencies. This in turn helps us to reduce reuse distances for the shared data, and reduces the number of inter-core conflict misses.
- We quantify the effectiveness of this scheme with regards to improving parallel execution performance and discuss how close it comes to a hypothetical scheme. We also compare the proposed compiler scheme against a state-of-the-art locality-enhancing technique. To test the performance of our scheme and compare it to the state-of-the-art, we performed experiments with a simulation framework (based on SIMICS [49] and GEMS [40]) as well as on two commercial multi-core machines (an AMD quad-core [44] and an Intel six-core [27]). Our experimental evaluation indicates that the proposed data locality optimization scheme reduces inter-core conflict misses in the shared cache by 67% on average when both allocation and scheduling are used. Also, the execution time improvements we achieve (29% on average) are very close to optimal savings.

2. TARGET CMP ARCHITECTURE, APPLICATION DOMAIN, AND OUR GOAL

Our target CMP, shown in Figure 1 for the case of 4 cores, is a shared memory based multi-core architecture, each core having private L1 instruction and data caches, and all cores sharing an on-chip unified L2 cache. We assume that a MESI-like protocol [23] is employed to ensure data coherence across the L1 caches. Note that shared L2 is the last line of defense in this architecture, and therefore maximizing its hit rate is critical for high performance. We would like to mention that our approach is also applicable to the CMP architectures in which L3 is the last line of defense (instead of L2), as in the case of two commercial multi-core machines for which we later report experimental results.

Our application domain is array-based, loop-intensive, multi-threaded applications, frequently used in scientific computing as well as in embedded image/video processing. To evaluate the impact of our approach, we use all the benchmarks from the Splash-2 suite [56] and two array-intensive applications from Parsec [8].

Our goal in this work is to improve data reuse in the last level of cache of the CMP. We achieve this by re-organizing data accesses considering how different cores access the shared data elements, which helps reduce the number of inter-core conflict misses (defined in Section 1). Consider the two-core execution scenario depicted in Figure 2 for illustration purposes, assuming that each of x_1, x_2, x_3, y_1, y_2 and y_3 denotes a set of loop iterations (and each set contains a similar number of iterations). Assume further that x_1, x_2 and x_3 share data elements among them, and similarly, y_1, y_2, and y_3 access common data elements. This figure shows three alternate execution strategies, assuming all are allowable based on data dependencies. In each case, the time is assumed to flow from top to bottom, e.g., in (a), core 1 first executes x_1, then y_2, and finally, x_2.

In (a), we show the execution pattern before any data locality optimization (i.e., the original case). In (b) on the other hand, we illustrate the situation when only intra-core locality optimization is applied. In this case, each core tries to reuse the same data as soon as possible, by scheduling the sets that share data one after another. For example, in core 1, x_1 and x_2 are executed one after another, and similarly, in core 2, y_1 and y_3 are executed successively. However, one can also observe that inter-core data reuse is not very good in this case, e.g., x_1 and x_3 are executed in different steps (and similarly y_2 and y_3). Finally, (c) depicts the case in which *both* intra-core reuse and inter-core reuse are enhanced. In particular, x_1 and x_3 are scheduled together, and y_2 and y_3 are scheduled together. Our proposed approach tries to obtain the scenario in (c) using both allocation (deciding which iteration sets should be assigned to each core) and scheduling (deciding the execution order of these sets in each core). In more general terms, when different cores share the same data element, we want them to access that data element at similar times, so that the shared data can be reused while it is in the shared cache. The longer the distance between two successive accesses (reuses) to the same data, the higher the chances that the second access will incur a miss in the shared cache (i.e., an inter-core conflict miss can displace the data from the shared cache before it gets reused). The results of our experiments presented in Section 4 show that contribution of such misses to overall L2 misses can be very

[1]Since in our experiments we use one thread per core, we use the terms intra-core and intra-thread interchangeably, and similarly, the terms inter-core and inter-thread are used interchangeably.

Cores	8 four-issue processors	
L1 Cache	8 Split I/D, each 32KB, 2-way, 64B line, 3-cycle latency, write-back	
L2 Cache	Unified, 4MB, 32-way, 64B line, 22-cycle latency	
Main Memory	4GB, 300-cycle latency	
Data Blocks	**Size**	32 KB
	Orientation	row-wise

Table 1: System configuration parameters.

Application Name	L1 Miss Rate	L2 Miss Rate	Cycles ($\times 10^6$)
Barnes	6.2%	54.5%	865.3
FMM	16.6%	71.2%	933.6
LU	11.3%	66.8%	702.2
Ocean	18.1%	49.1%	597.5
Radiosity	7.6%	61.3%	880.8
Raytrace	13.9%	42.6%	592.1
Volrend	16.1%	37.2%	631.9
Water	5.3%	57.1%	911.4
Cholesky	9.8%	36.6%	826.5
FFT	18.2%	33.0%	923.7
Radix	22.5%	68.4%	774.2
Freqmine	4.4%	29.4%	628.4
BodyTrack	6.1%	38.7%	711.7

Table 2: Important characteristics of our benchmarks.

high, and therefore, reducing such misses can improve CMP performance significantly.

3. EXPERIMENTAL SETUP

Most of our experiments have been performed using a simulation environment built upon SIMICS [49] and use the timing model from GEMS [40]. SIMICS is a full system simulation platform, capable of simulating multi-core systems and boot and run operating systems and commercial workloads. Our major simulation parameters and their default values are given in Table 3 (the concept of "data block" will be explained later). In this study, we used all the applications from the Splash-2 [56] benchmark suite (written originally using the Pthread Library [48]), and two array-based applications in the Parsec suite [8] (Freqmine, and BodyTrack) whose threads share data (both written in OpenMP). The important characteristics of these benchmarks, under the architectural values listed in Table 3, are presented in Table 3. Since the original dataset sizes of these applications are not very big for typical L2/L3 capacities of modern CMPs, we increased their dataset sizes. For example, in Barnes, instead of working with 16K particles (original input), we worked with 512K particles. Similarly, in Water, we used an input size of 4,096 molecules (instead of 512 molecules of the original case). As a result, the dataset sizes of our applications range between 61MB (Water) and 273MB (FMM). We also note from Table 3 that the L2 performance of these applications is not very good, giving an average miss rate of 49%.

In addition to simulation based analysis, we also used two commercial multi-core machines to evaluate our compiler-based approach. The first commercial platform we used is Quad-Core AMD Opteron [44], in which each of the four cores have a private L2 cache of 512KB and all cores share an on-chip L3 cache of 2MB. The second commercial machine we used, Intel Dunnington, has six 45-nanometer Penryn-class cores integrated onto a single die. Each pair of Penryn cores shares 3MBs of L2 cache, and each of the six cores can access 12MBs of L3 cache. Note that, in both these systems, L3 is the last line of defense. In experiments with AMD and Intel machines, we use compilers provided by vendors with the highest optimization-level. For example, in AMD, we used O3, which includes loop-fusion, loop-interchange, and software-prefetching. In our simulations on the other hand, all the code versions tested exercise the same low-level gcc compiler with O3 and data prefetching is on.

```
for(i1=1;i1<N;i1++)
  for(i2=0;i2<N-1;i2++)
    U[i1][i2] = (V[i1-1][i2]+V[i1][i2+1])/2
```

4. EVALUATION OF ORIGINAL APPLICATIONS

Figure 3 gives a distribution of reuse distances of the original multi-threaded applications without any modification (using the values given in Table 3 and each core executing a single thread of the application). A bar for distance range $[x-y]$ in this graph indicates the fraction of the data reuses that fall between x cycles and y cycles. For example, if a data element is used by a core in cycle z_1 and requested again in cycle z_2 by the same or a different core, this reuse contributes to the bar $[x-y]$ if $x \leq z_2 - z_1 \leq y$. Clearly, from a data locality perspective, we want most reuses to have short distances, so that we can catch the data at the time of reuse in the shared cache. The results in this bar chart show a balanced distribution of reuse distances. To have a better understanding of this distribution, we also collected statistics for intra-core reuses and inter-core reuses separately, and present them in Figures 4 and 5, respectively. For our purposes, *intra-core reuse* corresponds to the reuse of a data element by the same core, while *inter-core reuse* means the reuse of the same element by different cores. We see from Figure 4 that intra-core reuse distances are not very high; in fact, most of intra-core reuses fall between 0 and 5,000 cycles. In contrast, the inter-core reuse distances tend to have very high values: more than 65% of inter-core data reuses have a distance more than 5,000 cycles. These results indicate that, when a core uses a data element, it is very likely that it will reuse it soon. In contrast, when a core uses an element, the next use of the same element by another core is typically not very close.

We also quantified the contribution of inter-core L2 conflict misses to the total L2 misses. The first bar for each application in Figure 6 gives this contribution, which averages in 51%. The second bar in the same figure gives the upper-bound for improvement in execution time if all inter-core conflict misses in L2 could be eliminated using a hypothetical scheme. We see that the average saving in execution cycles when all applications are considered is nearly 33%. Obviously, a realistic scheme cannot achieve all these savings as it may not be able to eliminate all inter-core misses in L2. Still, these results clearly indicate that there is a large scope for potential performance improvement by exploiting data reuse in the shared on-chip cache. We later show that our proposed approach comes close to this theoretical upper-bound.

5. SHARED CACHE AWARE CODE RESTRUCTURING

In the following discussion, we focus on a loop nest which may have multiple loops. Iterations of some of these loops can be fully parallel (i.e., no loop-carried data dependencies [21]), whereas other iterations are sequential.

Our approach optimizes each loop nest in isolation. Consider, as an example, the nested loop on the left. In this loop, (i_1, i_2) combination can take different values. In the discussion below, we use the term "loop iteration" to refer to such a combination. That is, in a multi-loop nest, an iteration is a vector and is represented by $\vec{I}$. Note that in this example all iterations can be executed in parallel. However, if the first right-hand-side reference was $U[i_1-1][i_2]$ (instead of $V[i_1-1][i_2]$), not all combinations (iterations) could be executed in parallel due to data dependencies. We want to emphasize that we are not proposing a new parallelism extraction scheme in this work. Rather, given a parallelization (which may choose to run only fully-parallel loops in parallel, or (in addition) some sequential loops in parallel and

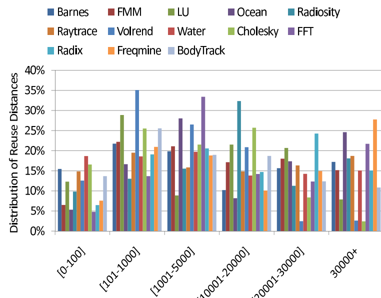

Figure 3: Distribution of all reuse distances.

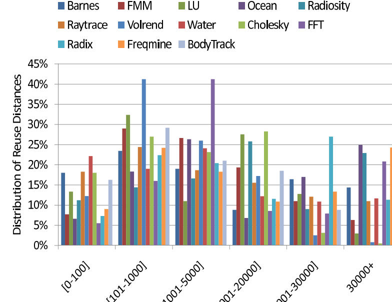

Figure 4: Distribution of intra-core reuse distances.

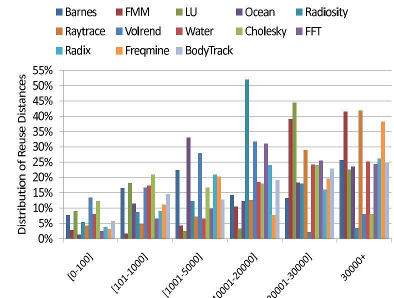

Figure 5: Distribution of inter-core reuse distances.

maintain correctness in this case through synchronization), we describe how to enhance shared L2 (or L3) behavior by distributing iterations to cores and reorganizing their execution order. In this way, we want to achieve both intra-core locality and inter-core locality. More specifically, the only input we get from the prior phases in compilation is the set of loop iterations (some of which may be dependent on others) that will be executed in parallel by available cores.

We distinguish between two related problems in data locality optimization for CMPs. The first problem, which we call *allocation,* determines the processor core to execute a given loop iteration. That is, it distributes the loop iterations across available cores. The second problem, termed as *scheduling,* decides the execution order of loop iterations assigned to the cores. In this paper, we consider two scenarios. In the first scenario, we assume that our approach performs both allocation and scheduling. In the second scenario, we assume that the loop iterations have already been distributed across the cores (i.e., the allocation step has already been performed), and only scheduling is to be carried out.

5.1 Background

In this work, we use sets to represent the data elements manipulated by loops as well as the iterations of the loops. The restructured loops are also represented as sets, from which we generate code. Specifically, these sets contain Presburger formulas [43]. The Presburger formulas are a class of logical formulas built from affine constraints over integer variables, logical connectives ($\vee$, $\wedge$, $\neg$), and quantifiers ($\exists$ and $\forall$). In this work, we employ the Omega Library [42] to manipulate iteration and data sets, which are described using the Presburger formulas. However, individual loop iterations, array (data) elements (i.e., their indices), and mappings between iterations and data are represented using vectors and matrices, which are embedded into our Presburger sets. As an example, for a loop nest where i_1 is the outer loop and i_2 is the inner loop, vector $\vec{I} = (i_1 \quad i_2)^T$ represents different iterations for different values of i_1 and i_2. The set of all iterations in a loop nest is represented using ϕ. Similarly, indices of array elements accessed by loop iterations are represented by matrices and vectors. For example, assuming again that i_1 is the outer loop and i_2 is the inner loop, a reference such as $U[i_1+1][i_2-1]$ is represented using $\xi\vec{I}+\vec{\zeta}$, where ξ is the two-by-two identity matrix and ζ is $(1 \quad -1)^T$. We use $\mathcal{R}$ to represent a reference, which is a mapping from iterations to data. Therefore, for the reference above, we can write $\mathcal{R}(\vec{I}) = (i_1+1 \quad i_2-1)^T$. We use the symbol $\Re$ to represent the set of all references in a loop nest. The set of data elements accessed by a given reference $\mathcal{R}$ can be expressed using the following Presburger set $\{\vec{d} | \ \exists \vec{I} \in \phi : \ \mathcal{R}(\vec{I}) = \vec{d}\}$, which means this set holds all data elements ($\vec{d}$) that can be accessed using $\mathcal{R}$ in some iteration ($\vec{I}$).

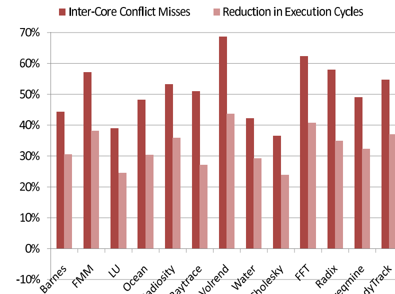

Figure 6: Contribution of inter-core conflict misses to total L2 misses and potential savings.

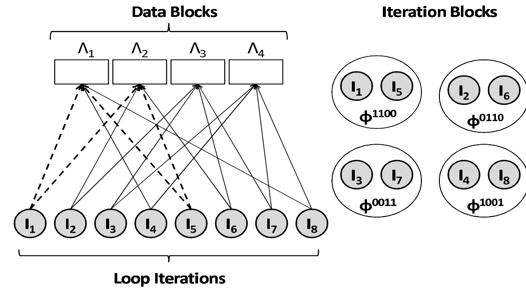

Figure 7: An example access pattern (iteration-to-data mapping) and forming different iteration blocks. We have four data blocks (λ_1 through λ_4), and the iterations that are mapped to the same iteration block access exactly the same set of data blocks.

5.2 Integrated Allocation and Scheduling

Let λ be the set of data (array) elements accessed by the loop nest. We start by dividing this set into d subsets, λ_1, λ_2, $\cdots$, λ_d, called *data blocks.* This is a logical partitioning (i.e., data are not physically divided into blocks) and is only used for code restructuring purposes.[2] Similarly, the set of loop iterations to be allocated and scheduled (ϕ) is partitioned into $2^d - 1$ subsets, called *iteration blocks.* Each of these iteration blocks is assigned a *tag,* which captures the set of data blocks accessed by the iterations in that block. More specifically, $T(\phi)$, the tag of iteration block ϕ, is a d-bit vector,

[2]Data blocks can come from different arrays. For example, in a loop that accesses two arrays, we may have 100 blocks, 40 covering the first array and 60 covering the second one. We assume that all the data blocks are of the same size except maybe those at the end of arrays. Also, our data blocks are row-wise, i.e., a data block of size L holds L consecutive elements from an array.

whose k^{th} bit is set to 1 if ϕ accesses (a data element from) λ_k; otherwise, it is set to 0 (note that all iterations in an iteration block access exactly the same set of data blocks). Therefore, the tag associated with an iteration block summarizes the data access pattern of the iterations it holds. It is also to be noted that in general different iteration blocks can have different number of iterations. In the rest of this section, we use ϕ^T to represent an iteration block with tag T. Consequently, our iteration blocks can be written as ϕ^{T_1}, ϕ^{T_2}, ϕ^{T_3}, $\cdots$, $\phi^{T_{2^d-1}}$. As an example, assuming for illustrative purposes we have 4 data blocks, ϕ^{1100} indicates an iteration block that accesses only the first and second data blocks (see Figure 7). That is, if $\vec{I} \in \phi^{1100}$, this means $\vec{I}$ accesses both of the first two data blocks and does not access the last two data blocks. Note that ϕ^{T_1}, ϕ^{T_2}, ϕ^{T_3}, $\cdots$, $\phi^{T_{2^d-1}}$ are disjoint and they collectively cover the entire set of iterations (ϕ).

An important point to note at this juncture is that, we consider the data block-size as an input to the compiler. We use a simple heuristic to determining block-size. The main potential issue is that the selected size maybe such that the total size of the data blocks accessed by an iteration block maybe larger than shared cache capacity. The maximum number of blocks is accessed when the tag is all 1s. By taking into account this possibility, shared cache capacity and number of array-references in the loop-body, we calculate the maximum block-size that will not allow an iteration group to access data that exceeds the cache capacity. In most of our loop nests, this value turned out to be around 32KB. Therefore we use 32KB as the default block size in this paper.

We now explain how to compute ϕ^T for a given tag T and set of data blocks λ_1, λ_2, $\cdots$, λ_d. Without loss of generality, let us assume that $T = < t_1 t_2 \cdots t_{q-1} t_q t_{q+1} \cdots t_d > = < 11 \cdots 110 \cdots 0 >$, (i.e., the first q entries are 1 and the rest are 0). We can express ϕ^T using the following Presburger set:

$$\phi^T = \{\vec{I} \mid \vec{I} \in \phi \text{ and } \exists \mathcal{R}_1, \mathcal{R}_2, \cdots \mathcal{R}_q, \in \Re : \mathcal{R}_j(\vec{I}) \in \lambda_j \quad (1 \leq j \leq q) \text{ and } \neg(\exists \mathcal{R}_k \in \Re : \quad \mathcal{R}_k(\vec{I}) \in \lambda_k \quad (q+1 \leq k \leq d))\}.$$

We first describe the case when the iterations to be allocated to cores and scheduled have no dependencies, that is, they are fully parallel. We later discuss how our baseline allocation/scheduling scheme can be extended to accommodate data dependencies across iteration blocks.

5.2.1 Dependence Free Case

We are given a set of loop iterations that do not have any dependencies among them, and our goal is to distribute them over available cores and schedule them for minimizing inter-core conflict misses in the shared L2. An important observation we can make is that, more similar the tags of two iterations blocks, more similar their data access patterns. "Similarity" in this context can be captured and represented using the *Hamming Distance,* which is the number of positions for which the corresponding bits are different. So, lower the Hamming Distance between the tags of two different iteration blocks, more similar their data access patterns (i.e., they manipulate similar set of data blocks). Consider, for example, iteration blocks ϕ^{1000} and ϕ^{1100}. The Hamming Distance between their tags is 1. The iterations in the first iteration block access only the first data block, whereas those in the second iteration block access only the first two data blocks. Clearly, there may be data shared between these two groups of iterations. On the other hand, we do not expect iteration blocks ϕ^{0011} and ϕ^{1100} to share any data, as they access disjoint sets of data blocks (their tags have a Hamming Distance of 4).

Based on this observation, we can summarize our integrated allocation-scheduling algorithm as follows. We visit the iteration blocks starting with the one that has the lowest tag value. At each step, we visit an iteration block. When an iteration block is visited, we distribute the iterations in that block across available processor cores. That is, if there are L iterations in an iteration block and P cores in the CMP, the first $\lfloor L/P \rfloor$ iterations are assigned to the first core, the next $\lfloor L/P \rfloor$ iterations are assigned to the second core, and so on (the excess iterations, if any, can be given to the last core). After the iterations of an iteration block are assigned to cores in this fashion, we move to the next iteration block. This next block is selected such that the Hamming Distance between the tag of the current block and the tag of the next block is *minimum* among all possible alternatives. From a given core perspective, the set of iterations allocated for it from step $m+1$ are added to those allocated from step m. In the rest of our discussion, we use $\delta(T_1, T_2)$ to denote the Hamming Distance between tags T_1 and T_2. Note that the iterations assigned to a core form that core's *allocation,* whereas the *scheduling* order of iterations in a core is the order in which iterations are assigned (i.e., corresponding to the steps of the algorithm in Figure 5.2.1, as will be explained shortly).

```
1: divide data into d blocks λ1, λ2, ···, λd
2: compute iteration blocks φ^T1, φ^T2, φ^T3, ···, φ^T(2^d−1)
3: G := sorted set of iteration blocks based on Hamming Distance
4: step := 1
5: while there are nodes in G to be scheduled do
6:   φ^T := next-node(G)
7:   for (p:=1; p≤P; p++) do
8:     allocation(p,step) := share(φ^T,p)
9:   end for
10:  step := step + 1
11:  remove φ^T from G
12: end while
13: for (p:=1; p≤P; p++) do
14:   emit-code(p,allocation(p,1),allocation(p,2),allocation(p,3),···)
15: end for
```

Figure 8: Integrated allocation/scheduling algorithm for the dependence-free case.

It should be observed that some of the iteration blocks may be empty (i.e., there may not be any iteration that accesses a particular subset of data blocks). In any case, our approach tries to minimize the Hamming Distance between the tags of the successively-scheduled iteration blocks. The difference is that, if some iteration blocks are empty, the Hamming Distance between the tags of the successively-scheduled iteration blocks can be more than 1. If on the other hand all iteration blocks have some iterations (at least P iterations, where P being the number of cores), we can always achieve a Hamming Distance of 1 for each core, as we move from one scheduling step to another.

Figure 5.2.1 gives the pseudo-code for the algorithm that implements our scheme. Each neighboring pair of iteration blocks in $\mathcal{G}$ have tags that have the minimum Hamming Distance. In this algorithm, share(ϕ^T,p) returns the subset of iterations in ϕ^T to be allocated for core p, and allocation(.) is the data structure that holds these elements at each step. emit-code(.) generates output code from this data structure (in our implementation, it invokes the code-gen(.) utility of the Omega Library [42], as all the sets are maintained as Presburger sets throughout the entire compilation process)[3]. Finally, next-node(.) returns the next node (iteration block) to be scheduled. As stated above, this node is selected such that the Hamming Distance between the tag of this node and the tag of the previous node is minimum (like Gray Codes).

Note that, as far as the shared L2 cache is concerned, this

[3]Our current implementation generates a separate loop nest for each allocation(p,step). Further optimizations are possible to minimize the number of nests generated.

algorithm is expected to improve both intra-core locality and iter-core locality. First, at any given step m of the code in Figure 5.2.1, the set of iterations assigned to different cores have the *same* data access patterns (the *same* tag), (i.e., they access the same set of data blocks). Consequently, chances for converting inter-core data reuse to inter-core locality (L2 cache hit) are very high. In other words, this helps us reduce the reuse distance for the shared data. Second, as we move from step m to step $m+1$, we can expect that some of the data used in step m will also be used in step $m+1$ since the corresponding tags have the minimum Hamming Distance. Therefore, our scheme (the algorithm in Figure 5.2.1) exploits both intra-step and inter-step data reuses.

5.2.2 Case with Dependences

In this case, we are given a set of loop iterations that will be executed in parallel, and some of these iterations may be dependent on others. There are changes to be made to the algorithm in Figure 5.2.1 when there are data dependencies between loop iterations. Clearly, when there are data dependencies between loop iterations, we may not always be able to schedule next the iteration block whose tag has the minimum Hamming Distance from the tag of the currently-scheduled iteration block. We define an *iteration block dependence graph* $\mathcal{G}$ as follows. Each iteration block ϕ^T is represented using a node v_T in $\mathcal{G}$, and there is an edge from node v_{T_1} (which represents iteration block ϕ^{T_1}) to node v_{T_2} (which represents iteration block ϕ^{T_2}) iff $\exists \vec{I_1}, \vec{I_2}$ and $\exists \mathcal{R}_1, \mathcal{R}_2 \in \Re$: $\vec{I_1} \in \phi^{T_1}$ and $\vec{I_2} \in \phi^{T_2}$ and $\vec{I_1} \preceq \vec{I_2}$ and $\mathcal{R}_2(\vec{I_2}) = \mathcal{R}_1(\vec{I_1})$, assuming that either $\mathcal{R}_1$ or $\mathcal{R}_2$ is a write-reference and $\preceq$ means "lexicographically smaller than or equal to". Suppose that v_{T_1} represents the last iteration block that has been processed so far (i.e., its iterations have been distributed across processor cores). We define a set of schedulable iteration blocks for v_{T_1} (denoted $\mathcal{S}(v_{T_1})$) as the set of iteration blocks that can start execution when v_{T_1} finishes. The next node v_{T_2} to be processed is selected such that v_{T_2} belongs to $\mathcal{S}(v_{T_1})$ and, for any other node $v_{T_3} \in \mathcal{S}(v_{T_1})$, we have $\delta(T_1, T_2) \leq \delta(T_1, T_3)$. That is, among all schedulable nodes, we select the one whose tag has the minimum Hamming Distance from the tag of the currently-scheduled node. It needs to be noted that, once $\mathcal{G}$ is built, we can employ any scheduling algorithm (such as list scheduling [21]) and use the Hamming Distance metric as the tie-breaker when there are more than one schedulable nodes. Once a node (iteration block) is scheduled, the iterations it contains are distributed over available cores, as explained in the case of dependence-free case. Figure 9 illustrates an example application of our integrated allocation-scheduling scheme. The figure also highlights the allocation of core 2. In the rest of this section, when no confusion occurs, we use the terms "node" and "iteration block" interchangeably.

However, there are still two problems we need to address. First, for a given tag T, the iterations in ϕ^T can have data dependencies amongst themselves, preventing us from executing all of them in parallel using multiple cores. Our solution to this problem is as follows. We divide ϕ^T into s subsets: ϕ^T_1, ϕ^T_2, $\cdots$,ϕ^T_s. Each subset is small enough so that all its iterations can be scheduled together (i.e., there is no circular dependence which can be formulated using Presburger sets, between any (ϕ^T_m, ϕ^T_n), where $m \neq n$). Such a division is always possible since we can have only one iteration in each subset in the extreme case. However, in general, we still want to keep the size of each subset as large as possible to reduce the number of subsets and the associated scheduling and code generation costs. This partitioning, which can be formulated using Presburger sets, is applied to all ϕ^Ts whose iterations are dependent on each other. After this partitioning, the newly-generated nodes are added to the iteration block dependence graph ($\mathcal{G}$), and the allocation and scheduling are carried out as explained earlier.

Handling Cyclic Dependencies. The last problem to address is due to potential cyclic dependencies across the iteration blocks. Note that iterations are placed into iteration blocks based only on their data block access patterns. As a result, we can have cyclic dependencies between two iteration blocks. Unless all the cycles are eliminated, it is not possible to schedule an iteration block dependence graph. An example cyclic graph is illustrated in Figure 10(a). Observe that there are at least two ways of eliminating a given cycle from an iteration block dependence graph. First, we can *merge* the nodes involved in the dependence cycle into one node. Second, we can *split* some of the nodes involved in the cycle so that the cycle can be eliminated. Figures 10(b) and (c) illustrate node merging and node splitting, respectively, for the example iteration block dependence graph in Figure 10(a). Both these techniques have their drawbacks. The tag of the merged node is the bit-wise union of the tags of the nodes involved in the cycle. Consequently, the merged node (iteration block) may not exhibit very good data locality (in the extreme case, it can access all the data blocks manipulated by the loop nest being optimized). While node splitting does not have this problem, it increases the number of nodes in the graph which in turn may affect the size of the output code. Consequently, both these techniques should be applied with care. In particular, we need to minimize the number of merge and split operations in converting a cyclic graph to a non-cyclic one. Below, we discuss a solution to this problem. After preliminary experiments with these two techniques that can eliminate cycles, we found that node splitting generally performs better. Therefore, in the remainder of this discussion, we focus on node splitting.

The question that we need to address is: *"What is the minimum number of nodes to split to make a cyclic graph non-cyclic?"* Our approach to this problem uses the *feedback vertex set problem* [22], which is a graph-theoretical NP-complete problem. Given an undirected graph $G = (V, E)$, the feedback vertex set problem returns the *minimum* set of nodes such that removal of those nodes makes the resulting graph cycle-free. Karp was the first one to show that this problem is NP-complete on directed graphs; but it is known today that the undirected version is also NP-complete. Fortunately, there exist several heuristic algorithms proposed in the literature for the feedback vertex set problem. In this work, we use the heuristic discussed in [17]. Since the details of this linear heuristic are beyond the scope of this paper, we do not discuss them here.

Our approach for handling the cyclic graphs operates as follows. We first invoke the algorithm in [17] to determine the minimum set of nodes to be removed from the graph to make it cycle-free. We use $\mathcal{J}$ to denote this set. Then, for each of these nodes, we try node splitting to see whether the node can be split satisfactorily. What is meant by "satisfactorily" in this context is that, although in theory we can always split a node into two or more nodes, the particular split we are interested in has the properties explained below.

Assume that $\phi^T \in \mathcal{J}$ is the node to be split (i.e., it is one of the nodes returned by the algorithm in [17]). Let $\mathcal{V}$ be the set of nodes from which there are dependences to node ϕ^T. That is, for each member ϕ^{T_v} of $\mathcal{V}$, there is a dependence from ϕ^{T_v} to ϕ^T. Assume further that $\mathcal{W}$ is the set of nodes to which we have dependences from ϕ^T. In other words, we have a dependence from ϕ^T to each node ϕ^{T_w} of $\mathcal{W}$. Suppose now that ϕ^T is divided into two sub-nodes: ϕ^{T_1} and ϕ^{T_2}. We call this split "satisfactory" if the following three conditions are satisfied after the split:

- No dependence goes from any $\phi^{T_v} \in \mathcal{V}$ to ϕ^{T_2}. Put it another way, all in-coming dependences of original ϕ^T are directed to ϕ^{T_1}.

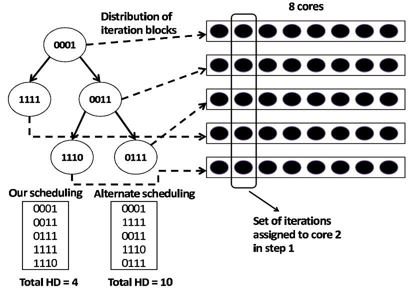

Figure 9: Allocation and scheduling for an example iteration block dependence graph (upper-left) with five iteration blocks. With our scheme, the total Hamming Distance is 4. The figure also shows an alternate legal schedule which gives a total Hamming Distance of 10.

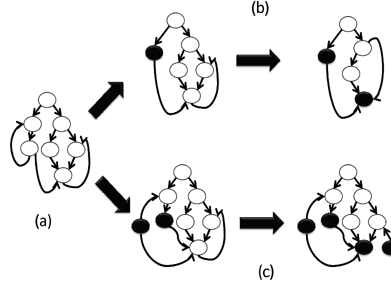

Figure 10: (a) A sample iteration block dependence graph with two cycles. (b) Node merging. (c) Node splitting. In (b) and (c) the newly-created nodes are colored black.

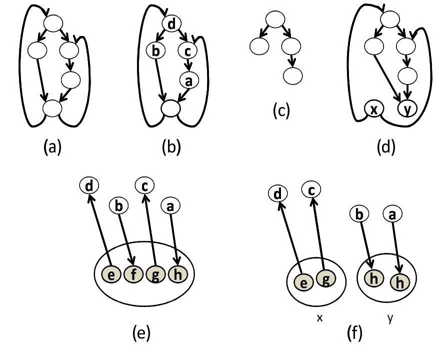

Figure 11: An example showing the details of our node splitting strategy.

1: divide data into d blocks $\lambda_1, \lambda_2, \cdots, \lambda_d$
2: compute iteration blocks $\phi^{T_1}, \phi^{T_2}, \phi^{T_3}, \cdots, \phi^{T_{2^d-1}}$
3: build iteration block dependence graph $\mathcal{G}$
4: **if** there are intra-node dependences **then**
5:   **for** each ϕ^T with intra-node dependences **do**
6:     break ϕ^T into sub-nodes with no cyclic dependences
7:     update $\mathcal{G}$
8:   **end for**
9: **end if**
10: **while** there are cyclic dependences **do**
11:   $\mathcal{H}$:= feedback-vertex-set($\mathcal{G}$)
12:   **for** each $\phi^T \in \mathcal{H}$ **do**
13:     **if** check(ϕ^T) **then**
14:       split(ϕ^T) and update $\mathcal{G}$
15:     **end if**
16:   **end for**
17: **end while**
18: step := 1
19: **while** there are nodes in $\mathcal{G}$ to be scheduled **do**
20:   ϕ^T := next-node($\mathcal{G}$)
21:   **for** (p:=1; $p \leq P$; p++) **do**
22:     allocation(p,step) := share(ϕ^T,p)
23:   **end for**
24:   step := step + 1
25:   remove ϕ^T from $\mathcal{G}$
26: **end while**
27: **for** (p:=1; $p \leq P$; p++) **do**
28:   emit-code(p,allocation(p,1),allocation(p,2),allocation(p,3),$\cdots$)
29: **end for**

Figure 12: Integrated allocation/scheduling algorithm for the case with dependences.

- No dependence goes from any ϕ^{T_1} to any ϕ^{T_w} of $\mathcal{W}$. In other words, all out-going dependences of original ϕ^T are directed from ϕ^{T_2}.
- No dependence exists between ϕ^{T_1} and ϕ^{T_2}.

Our approach tries to find two sets (iteration blocks) ϕ^{T_1} and ϕ^{T_2} such that $\phi^{T_1} \cup \phi^{T_2} = \phi^T$ and they satisfy all three conditions listed above. While we do not present the details due to space concern, this partitioning problem is also formulated using Presburger sets. Our approach tries to find such a split for all nodes in $\mathcal{J}$. If all splits are satisfactory, then we are done. If not, we update the graph with the satisfactory splits made so far, and run the algorithm in [17] again to find alternate nodes to split.

Figure 11(a) gives an example cyclic dependence graph, and Figure 11(b) shows the node selected by the algorithm in [17]. Figure 11(c) depicts the state of the graph when this node is removed, and Figure 11(d) shows the situation after split. The in-coming and out-going dependences of the node removed (and split) are highlighted in Figure 11(e). In this figure, the four iterations contained by the node are marked using e, f, g and h. Finally, Figure 11(f) shows the details of how two new nodes (marked as x and y) are created after splitting (these are the same x and y nodes in Figure 11(d)). The pseudo-code for our integrated allocation-scheduling algorithm that handles the case with data dependencies is given in Figure 5.2.2. In this code $check(\phi^T)$ returns true if ϕ^T can be satisfactorily split. It is also important to emphasize that our scheme is good from a load balance perspective as well. This is because, for each ϕ^T, we divide the iterations in it as equally as possible among the available processor cores.

5.2.3 *Example*

We illustrate our scheme using a simple example. The original loop in Figure 14 accesses two arrays U and V. Assuming that our CMP contains 4 cores, a simple parallelization of this loop would be to assign equal iterations to each core, as illustrated in Figure 15 for core p. Under this default parallelization, if the arrays are divided into 4 data blocks each, then at any given time, elements from all 8 data blocks are accessed. This can lead to conflicts in the L2 cache (note that the loop bound is very small for the purpose of illustration). Figure 13 shows the different iteration blocks that are formed according to the access tags. Each tag in this case is a 8-bit vector with the kth bit set to 1 if the iteration block accesses the kth block. The iterations can be separated into 10 iteration blocks corresponding to the tags shown. The iteration blocks corresponding to the remaining tags are not shown as they are empty.

The next step in our scheme is to schedule the iteration blocks such that the temporally contiguous blocks have the lowest Hamming Distance possible. In this particular example, there exist no data dependences between the different iteration blocks and hence we can order the iteration blocks in any manner. The schedule of iteration blocks shown in Figure 13 (which goes from top to bottom) leads to the lowest total Hamming Distance possible. This is because each iteration block differs from the preceding and succeeding iteration blocks by a Hamming Distance of exactly 1.

We now discuss the distribution of each iteration block to our 4 cores. In this example, the iteration block $\phi^{10001000}$

Blk#	Tag	Iterations	Core 1	Core 2	Core 3	Core 4
1	10001000	4→20	4→7	8→11	12→15	16→20
2	10001100	21→24	21	22	23	24
3	01001100	25→28	25	26	27	28
4	01000100	29→45	29→32	33→36	37→40	41→45
5	01000110	46→49	46	47	48	49
6	00100110	50→53	50	51	52	53
7	00100010	54→70	54→57	58→61	62→65	66→70
8	00100011	71→74	71	72	73	74
9	00010011	75→78	75	76	77	78
10	00010001	79→95	79→82	83→86	87→90	91→95

Figure 13: Tags, iteration-to-iteration block mapping, and scheduling. Columns one, two and three show the iteration block number, tag and iterations in the block, respectively. Columns four through 7 show which iterations from each iteration block are assigned for execution on each core. For each core, the schedule goes from top to bottom.

```
for( i = 4; i < 96; i++){
    U[i] = V[i-4] + V[i-3] + V[i-2]+ V[i-1] + V[i]
           + V[i+1] + V[i+2] + V[i+3] + V[i+4];
}
```

Figure 14: Original sequential loop.

consisting of 16 iterations is scheduled first. Each core is assigned 4 iterations. Next, iteration block $\phi^{10001100}$ consisting of 4 iterations is distributed such that each core is assigned one iteration. Similarly, each iteration block is distributed as shown in Figure 13. Each core therefore is assigned a specific subset of iterations to execute. The code corresponding to core 1 is shown in Figure 16. Codes for the other cores can be generated in a similar fashion. Note that, our approach reduces the reuse distance to shared data (which reside on data block boundaries).

It is important to emphasize that, since the original loop does not have data dependences, we have the flexibility to start scheduling from any iteration block we want (as long as we minimize the Hamming Distance between the tags of the successively-scheduled iteration blocks). Note also that, if in the original code in Figure 14 the left and right hand side arrays were the same, we would have data dependences, which would force a particular scheduling order for the iteration blocks (from left to right in this case). Even in this case however, it is still possible to achieve the minimum Hamming Distance as we move from one iteration block to the next.

5.3 Scheduling for Fixed Allocation

In some cases, we may not have the flexibility to assign iterations to processor cores. For example, a prior phase in compilation may have already performed this assignment based on some objective function (e.g., to maximize inter-core parallelism, improve load balance, etc). However, even under this fixed allocation scenario, we still have the flexibility of changing the execution order of loop iterations assigned to a core. Further, by tuning the execution order of iterations for each core in a symbiotic manner, we may be able to improve data locality in the shared on-chip cache.

Without loss of generality, let us assume that we have P cores, and $\phi(p)$ represents the set of iterations assigned to core p, where $1 \le p \le P$. As before, we divide the data into blocks, λ_1, λ_2, $\cdots$, λ_d. The set of loop iterations assigned to core p ($\phi(p)$) is partitioned into $2^d - 1$ subsets, and as usual, each partition (iteration block) is assigned a tag which represents the set of data blocks it accesses. We use $\phi(p)^T$ to represent an iteration block of core p with tag T. As before,

```
for( i = 4 + (p-1)*23; i < 4*p*23-1; i++){
    U[i] = V[i-4] + V[i-3] + V[i-2]+ V[i-1] + V[i]
           + V[i+1] + V[i+2] + V[i+3] + V[i+4];
}
```

Figure 15: Default parallelization.

```
for( i = 4; i < 8; i++){
    U[i] = V[i-4] + V[i-3] + V[i-2]+ V[i-1] + V[i]
           + V[i+1] + V[i+2] + V[i+3] + V[i+4];
}
U[21] = V[17] + V[18] + V[19]+ V[20] + V[21]
       + V[22] + V[23] + V[24] + V[25];
U[25] = V[21] + V[22] + V[23]+ V[24] + V[25]
       + V[26] + V[27] + V[28] + V[29];
for( i = 29; i < 33; i++){
    U[i] = V[i-4] + V[i-3] + V[i-2]+ V[i-1] + V[i]
           + V[i+1] + V[i+2] + V[i+3] + V[i+4];
}
U[46] = V[42] + V[43] + V[44]+ V[45] + V[46]
           + V[47] + V[48] + V[49] + V[50];
U[50] = V[46] + V[47] + V[48]+ V[49] + V[50]
           + V[51] + V[52] + V[53] + V[54];
for( i = 54; i < 58; i++){
    U[i] = V[i-4] + V[i-3] + V[i-2]+ V[i-1] + V[i]
           + V[i+1] + V[i+2] + V[i+3] + V[i+4];
}
U[71] = V[67] + V[68] + V[69]+ V[70] + V[71]
           + V[72] + V[73] + V[74] + V[75];
U[75] = V[71] + V[72] + V[73]+ V[74] + V[75]
           + V[76] + V[77] + V[78] + V[79];
for( i = 79; i < 83; i++){
    U[i] = V[i-4] + V[i-3] + V[i-2]+ V[i-1] + V[i]
           + V[i+1] + V[i+2] + V[i+3] + V[i+4];
}
```

Figure 16: Output code generated for core 1.

we discuss the dependence-free case and the case with data dependences separately.

5.3.1 Dependence-Free Case

In this case, for each core, we are given a set of dependence-free iterations assigned to them (we do not change this allocation). At a high level, our approach can be summarized as reordering the set of loop iterations assigned to each core such that the iteration sets that access the same data blocks are scheduled at the same time as much as possible. For example, in order to have good shared cache locality, all the following sets should be scheduled concurrently:

$$\phi(1)^T, \phi(2)^T, \phi(3)^T, \cdots \phi(P)^T,$$

for any given tag T. It is important to note that, as we move from the current iteration block (tagged T_1) to the next one (say T_2), we need to ensure that $\delta(T_1, T_2)$ is minimum. While it is possible that, for different cores, different iteration blocks may be empty, in practice the impact of this may not be very significant. This is because, for each core, our approach tries to schedule the iteration blocks assigned to it using Hamming Distance as the optimization metric.

5.3.2 Case with Dependences

As in the previous subsection, we assume that loop iterations have already been allocated to cores. However, now there may be dependences across these loop iterations. This case is challenging, because, unlike the case with the integrated allocation/scheduling problem where we schedule one node (from the iteration block dependence graph) at a time (and distribute its iterations across all cores), in this case we have P different schedules (one per core). As a result, data dependences can exhibit complex patterns (e.g., we can have dependences flowing from one core to another, as illustrated in Figure 18). Our problem formulation for this case starts with defining schedulable iteration block sets for cores. Let $\mathcal{S}_p$ denote the set of schedulable nodes for core p, where $1 \le p \le P$. Initially, we select, for core p, ϕ^{T_p} from $\mathcal{S}_p$ such that $\sum_p \sum_{q,q \ne p} \delta(T_p, T_q)$ is minimized. Informally, we select an iteration block (to schedule) from each core such that the sum of the Hamming Distances between the tags of each pair is minimized.

This will clearly help to reduce reuse distances in this first step (in the ideal case, in a given step, the tags of the iteration blocks scheduled from different cores will be the same). After these first set of iteration blocks are scheduled, we update the set of schedulable iteration blocks for each core. The next set

```
1: divide data into d blocks λ1, λ2, · λd
2: for (p:=1; p≤P; p++) do
3:    compute S_p
4: end for
5: step := 1
6: for (p:=1; p≤P; p++) do
7:    φ^{T_p} := next-node-1(S_p)
8: end for
9: while there are nodes in any S_p to be scheduled do
10:    step := step + 1
11:    for (p:=1; p≤P; p++) do
12:       φ^{T_p} := next-node-2(S_p)
13:       allocation(p,step) := φ^{T_p}
14:       remove φ^T from S_p
15:    end for
16: end while
17: for (p:=1; p≤P; p++) do
18:    emit-code(p,allocation(p,1),allocation(p,2),allocation(p,3),···)
19: end for
```

Figure 17: Scheduling algorithm for the fixed allocation.

Application Name	Graph Size	Number of Splits	Increase in Code Size
Barnes	(206,497)	71	53%
FMM	(313,779)	88	67%
LU	(194,382)	39	63%
Ocean	(428,760)	91	49%
Radiosity	(187,411)	55	72%
Raytrace	(259,823)	81	86%
Volrend	(406,918)	76	96%
Water	(112,339)	34	38%
Cholesky	(108,467)	29	51%
FFT	(395,991)	86	66%
Radix	(337,882)	47	58%
Freqmine	(211,573)	57	69%
BodyTrack	(376,886)	83	72%

Table 3: Important statistics regarding our integrated allocation/scheduling scheme.

of iteration blocks to get scheduled (ϕ^{T_p} for core p) is selected such that $\sum_p \sum_{q,q\neq p} \delta(T_p, T_q) + \sum_p \delta(T_p, T_{p'})$ is minimized, where $T_{p'}$ represents the tag of the iteration block scheduled for core p in the previous step. Note that while the first term in this expression captures sum of the Hamming Distances of the tags of the iteration blocks scheduled in the current step, the second term represents sum (across all cores) of the Hamming Distances between the tags of the currently scheduled block and the previous block. In this way, we exploit both data reuse across the iterations scheduled for different cores in the same scheduling step and data reuse for a given core as we move from the current stage to the next. While there may be alternate cost formulations, we found that this formulation is easy to implement and works very well in practice. Figure 18 also illustrates the schedule our approach chooses for the scenario shown.

The algorithm in Figure 5.3.2 gives the pseudo-code for this scheduling strategy. Note that we do not explicitly show the code that identifies cycles and eliminates them (it is very similar to that in Figure 5.2.2). Note also that select-node-1(.) and select-node-2(.) are different from each other. Specifically, as explained above, select-node-1(.) selects iteration blocks to be scheduled (for each p) considering the Hamming Distances among the tags in the same step, whereas select-node-2(.) considers the tags in both the current step and the previous step. In addition, select-node-2(.) does not select the node with the minimum Hamming Distance if doing so reduces parallelism. In other words, our scheduling scheme tries to schedule an iteration block for each core at each step if it is possible to do so. Among all alternatives that satisfy this constraint, it selects the one that minimizes the Hamming Distance, as explained above.

6. IMPLEMENTATION AND EVALUATION

We used two software infrastructures, SUIF [26] and Omega Library [42], to implement our approach. Specifically, once the input program is read by SUIF, we build our data and iteration sets and transform them using the Omega Library. The Omega Library, which is a polyhedral tool that manipulated Presburger formulas, is also used for generating the output code for each core, which is subsequently converted to the internal SUIF structures. At a high level, our approach is implemented as part of a source-to-source translator. In the first step during compilation, we analyze the code and obtain, for each loop nest, the set of iterations that will be executed in parallel (ϕ). As mentioned earlier, some of these iterations may have dependences which have to be enforced. In our integrated scheme, we determine both assignment of iterations to cores and scheduling of iterations for each core (re-writing the parallel sections of the code when necessary). In the scheduling for the fixed allocation scheme however, we kept the same iteration-to-core mapping implied by the original Pthread/OpenMP based version (to quantify the benefits coming solely from optimized data locality). Note that all the versions tested using simulator exercise the same low-level gcc compiler with the O3 optimization level.

We first present the detailed results collected through our SIMICS-based simulation platform (the memory timing model is from GEMS [40], which enables detailed cycle-accurate simulation). However, before discussing our improvements in reuse distances, cache misses and execution cycles, we want to present statistics regarding the behavior of our approach. The second column in Table 3 gives, for our integrated scheme, the size of the iteration block dependence graph for our applications in the (*nodes,*edges) format. The next column gives the number of split operations performed during compilation, and under the fourth column we present the percentage increase in code size as a result our integrated allocation/scheduling scheme. Recall that we generate a separate loop nest for each allocation of a core in each step during scheduling, and this leads to an increase in code sizes. We see that the average increase in code size (with respect to original applications) is about 63%. In our experiments below, unless explicitly specified, we used the values of the simulation parameters shown in Table 3. The increase in compilation time due to our approach (over the case without our optimization) was about 35% when averaged over all applications.

Recall that Figures 4 and 5 present the distribution of intra-core and inter-core reuse distances, respectively, for the original applications. We now present in Figure 19 the distribution of inter-core reuse distances when our integrated allocation-scheduling scheme is used. We note that, as compared to the distribution in Figure 5, this new distribution is much better as most of the data reuses have very short distances. In comparison, Figure 20 presents the distribution of reuse distances under the fixed allocation case. While, as expected, these results are not as good as those in Figure 19, they are still much better than those in Figure 5. Overall, we see that our scheduling and allocation schemes are very successful in reducing the reuse distances for the L2-resident data.

While these reductions in reuse distances are encouraging, it is also important to quantify their impact on L2 cache misses and execution cycles. The reductions in inter-core conflict misses in L2 due to our approach (over the original applications) are presented in Figure 21. We see that, on average, our integrated scheme reduces inter-core cache misses by 67%, and our scheduling under fixed allocation achieves an L2 miss reduction of 51%. However, since inter-core conflict misses in L2 are not the sole contributor of execution time, we also present overall execution time savings with our schemes (in Figure 22), over the original applications. The last bar in this figure represents the optimal savings and is reproduced from Figure 6. We observe that our integrated scheme and scheduling under fixed allocation result in average performance improvements of 29% and 24%, respec-

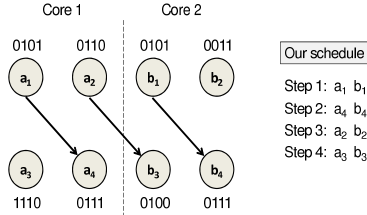

Figure 18: Left: a scenario with two cores and eight iteration blocks. Right: the result of our scheduling.

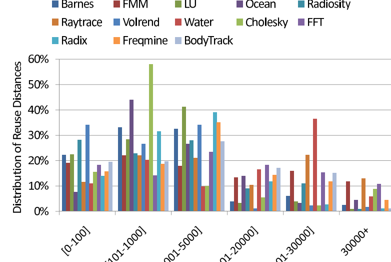

Figure 19: Distribution of inter-core reuse distances with the integrated scheme.

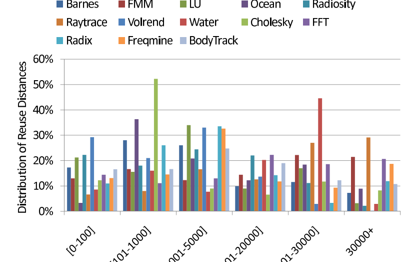

Figure 20: Distribution of inter-core reuse distances with the scheduling for fixed allocation.

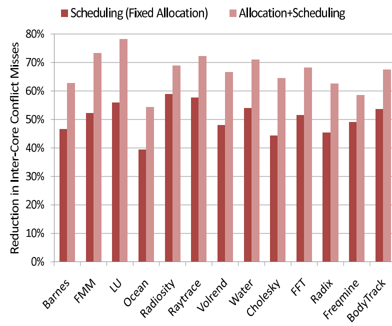

Figure 21: Reduction in inter-core conflict misses.

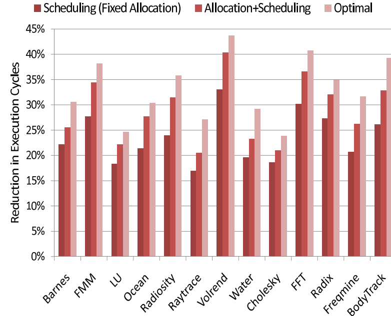

Figure 22: Execution time improvements with our scheme (simulation).

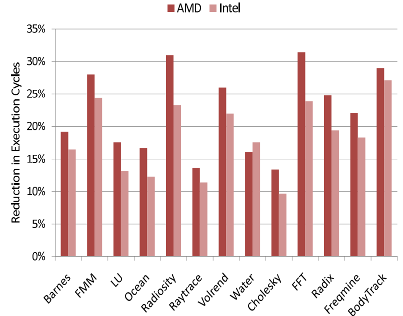

Figure 23: Execution time improvements with the integrated scheme (AMD and Intel).

tively. Considering that the average optimal saving is around 33%, these results are very good. We want to mention that the remaining performance difference between our approach and the optimal results are mostly due to data dependencies. That is, in several cases, data dependencies prevented our scheduler from eliminating some L2 misses. We also noticed that a better scheduler could cut more misses in some loop nests in our applications.

As mentioned earlier, we also conducted experiments on two commercial multi-core platforms (we gave cache details of these machines in Section 3). The bar-chart in Figure 23 gives the execution time improvement—over original applications when using the highest optimization flag in each machine—when our integrated allocation/scheduling scheme is used. The average improvements we obtain are 22% in AMD and 18% in Dunnington, when all 13 applications are considered. To sum up, the results obtained through both simulation and real-platform experiments clearly show that restructuring loop iterations for exploiting sharing in L2/L3 can be very beneficial in practice.

We also compared our approach to a state-of-the-art *intra-core* data locality optimization strategy. In this alternate scheme, after the conventional parallelization step, for each loop nest of each thread, we use a set of locality-enhancing transformations to maximize cache performance. More specifically, each loop nest of each thread is optimized independently using well-known loop restructurings such as loop permutation and tiling (tile sizes are determined after experimenting with several values). Note that, in many cases, such extensive intra-core locality optimizations generate better results than the locality optimizations supported by current commercial compilers. An example comparing this alternate approach (intra-core scheme) and our approach was given earlier in Figure 2. Figure 24 presents the percentage improvements in execution cycles, under the default values of our simulation parameters, this alternate scheme brings over the original codes. For ease of comparison, we also reproduce the results with our schemes. We see from these results that there is a huge performance difference between optimizing intra-core reuse alone and optimizing both intra-core and inter-core reuse (as in the case of our approach). In terms of average performance improvements, optimizing intra-core reuse alone (which is not effective in eliminating inter-core conflict misses) brings only 13% improvement, which is much lower than the average saving achieved by our integrated scheme (29%). This is because the original applications already have good single thread data access patterns. In fact, intra-core reuse distances are mostly on the lower side, as has already been discussed. Instead, significant benefits can be obtained by symbiotically reorganizing and scheduling data accesses by considering all threads together, which is achieved by our scheme.

In the rest of our experiments, we vary the values of some of our simulation parameters, and study their impact. Recall that the results presented and discussed so far are collected using the base simulation parameters given in Table 3. Figure 25 shows the results with different L2 associativities and the number of cores (the values of all other parameters are as given in Table 3). Each bar in this graph represents a value (percentage improvement in execution latency) when *averaged* over all 13 applications we have. When we increase the number of cores, our percentage savings increase. This is because the behavior of the original applications gets worse with the increased core count (the data accesses are spread more, creating more inter-core conflict misses). Since the results in Figure 25 are with respect to the original case, we observe

an improvement with increasing core count. Our second observation is that our approach performs better with smaller associativities. As we reduce associativity, the chances for data access conflicts in L2 increase, and code restructuring for reorganizing accesses to shared data becomes more important.

We next evaluate the impact of data block size in our results. This is an important parameter as it determines the formation of iteration blocks, which in turn determines the shape of dependencies. As indicated in Table 3, the default data block size used in our experiments so far is 32KB. The results with different block sizes are given in Figure 26. Clearly, as we reduce the data block size, we achieve better savings. This is because a smaller block size enables finer granular distribution of iterations into iteration blocks, i.e., we can group them more accurately. Consequently, we do a better job during scheduling. For example, when we reduce the block size from 32KB to 8KB, the average savings with our integrated scheme jump from 29% to 39%. While this certainly motivates for smaller block sizes, one may also want to consider the impact on code size. Small block sizes means more increase in size of the generated code. Consequently, if code size is a concern, one may not want to work with very small data blocks. As stated earlier, with our default data block size, the increase in code size was about 63% on average. We also found that with a data block size of 8KB, the increase in code size jumped to nearly 224%.

7. DISCUSSION OF RELATED WORK

Related work on locality optimizations for single core systems is mostly based on loop transformations. In [55] Wolf and Lam define reuse vectors and reuse spaces, and show how these concepts can be exploited by an iteration space optimization technique. Li [38] also uses reuse vectors to detect the dimensions of the loop nest that carry some form of reuse. Carr et al [11] employ a simple metric to re-order computation to enhance data locality. Zhang et al [57] study reference affinity, which characterizes a group of data that are always accessed together in computation. Tiling is also a well-known technique for enhancing data locality [34, 36]. Previous research [12] also discusses how code restructuring can be used for improving data locality in embedded applications. Cierniak and Li [20] were among the first to offer a scheme that unifies loop and data transformations.

Chatterjee et al [13] use Presburger formulas to express cache misses including the state of the cache in each loop nest. Andrade et al [4] propose a framework to model the cache behavior of codes with indirections using analytical models. Srikantaiah et al [51] present a shared cache management approach called set pinning, and propose a new classification system for cache misses. Sorenson and Flanagan [50] propose a cache characterization scheme using locality surfaces to predict cache miss rates. Ghosh et al [25] propose Cache Miss Equations (CME), a framework to express memory reference and cache conflict behavior in terms of sets of equations. Vera et al [54] discuss a scheme that estimates the solution of the CMEs by using sampling techniques.

A critical challenge faced by CMP hardware designers is how to efficiently manage the on-chip shared resources such as caches and register files. Previous work in this category includes [7, 18, 47]. Ballapuram et al [5] analyze the internal and external snoop behavior in a CMP system. They propose Selective Snoop Probe (SSP) and Essential Snoop Probe (ESP), where the snoopy cache coherence protocol is relaxed. Baskaran et al [6] present automatic data management for on-chip memories, where buffers are seated in on-chip (local) memories for holding portions of the data accessed. In [16], the authors focus on careful scheduling of threads to minimize destructive interactions on shared on-chip caches. Our approach is different from these previous studies as we focus on automated compiler-directed restructuring of data accesses for parallel applications targeting better L2 cache locality.

Anderson et al [2] propose a transformation technique that makes data elements accessed by the same processor contiguous in the shared address space. Chen and Sheu [15] minimize interprocessor communication by first dividing the iteration space into blocks without inter-block communication, and then assigning data and iterations to processors. Tim et al [53] propose a general data partitioning heuristic that considers both parallelism and communication cost. Several research groups address partitioning and scheduling problem for multiprocessors using tiling [3, 24]. In [33], authors present a profile-guided compiler technique for cache-aware partitioning of iteration spaces of parallel loops. Bondhugula et al [10] propose an automatic polyhedral source-to-source transformation framework to optimize a given code for both parallelism and locality. Kandemir [32] presents a data locality optimization scheme for CMPs which uses reuse-vectors (in a linear-algebraic framework) and can therefore only handle codes in which compiler can extract data reuse vectors accurately. It cannot handle any of the codes used in this submission. In comparison, our approach uses polyhedral arithmetic which is more general. Bikshandi et al [9] propose Hierarchically Tiled Arrays (HTAs) that enable direct manipulation of tiles for parallel program improvement as well as achieving locality. Liao et al [39] introduce a parallel compiler for the Brook streaming language with aggressive data and computation transformations. Chu and Mahlke [19] propose a compiler-directed approach to partition data and computation across multiple clusters in a locality aware fashion. Our work is complementary to many of these prior studies as it can be used in conjunction with them. In fact, they can be used in the same application: our approach can handle loop-nests with outer-loop parallelism and [19] can be used for the nests with inner-loop/intra-loop parallelism.

8. FUTURE WORK AND CONCLUDING REMARKS

This paper has presented and evaluated a compiler based data locality optimization scheme for shared L2 (or L3) based CMPs. The proposed scheme determines both assignment of loop iterations to processor cores and the order in which the iterations assigned to each core will be executed. The goal is to bring accesses to shared data from different cores together, thereby reducing inter-core conflict misses. We tested this scheme using all applications from the Splash-2 benchmark suite and two applications from Parsec. The results collected, through both simulation and experiments on a quad-core AMD and six-core Intel multi-core machines, show that the proposed scheme is very successful in bringing together the loop iterations that access shared data blocks. In this way, different cores operate on the data they share at around the same time. This helps to reduce reuse distances for shared data and, as a result, improves the performance of the shared on-chip cache. Our future work includes experimenting with different data block orientations (e.g., column-wise, diagonal) and with other commercial CMP systems. Work is also underway in integrating our approach with compiler techniques that try to automatically extract parallelism from sequential codes.

9. REFERENCES

[1] AMD Athlon 64 X2 Dual-Core processor for desktop. http://www.amd.com /us-en/Processors/ProductInformation/0,,30_118_9485_13041,00.html

[2] J. M. Anderson et al. Data and computation transformations for multiprocessors. In *Proc. POPL*, 1995.

[3] A. Agarwal et al. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. In *TPDS*, 1995.

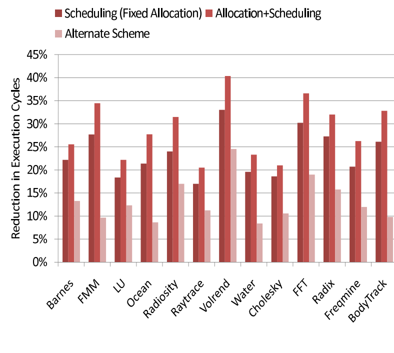

Figure 24: Execution time improvement with our schemes and an alternate scheme that improves data locality for each core in isolation.

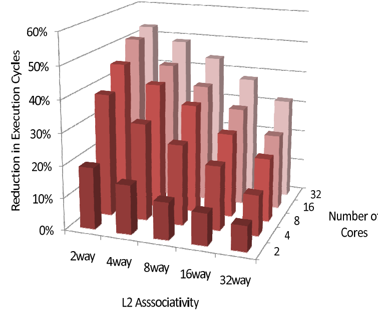

Figure 25: Impact of the number of cores and the number of cache ways (integrated scheme).

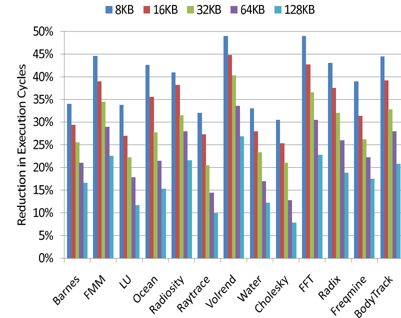

Figure 26: Impact of the data block size (integrated scheme).

[4] D. Andrade et al. Precise automatable analytical modeling of the cache behavior of codes with indirections. In *TACO*, 2007.

[5] C.S. Ballapuram et al. Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. In *Proc. ASPLOS*, 2008.

[6] M. Baskaran et al. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proc. PPoPP*, 2008.

[7] D. Beckmann, D. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proc. MICRO*, 2004.

[8] C. Bienia, S. Kumar, J. P. Singh and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. PACT,* October 2008.

[9] G. Bikshandi et al. Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. PPOPP*, 2006.

[10] U. Bondhugula et al. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proc. CC*, 2008.

[11] S. Carr et al. Compiler optimizations for improving data locality. In *Proc. ASPLOS*, 1994.

[12] F. Catthoor et al. Code transformations for data transfer and storage exploration preprocessing in multimedia processors. In *IEEE Design Test*, 2001.

[13] S. Chatterjee et al.Exact analysis of the cache behavior of nested loops. In *SIGPLAN Not.*, 2001.

[14] J. Chang, G. Sohi. Dynamic partitioning of shared cache memory. In *Proc. ICS*, 2007.

[15] T. S. Chen, J. P. Sheu. Communication-free data allocation techniques for parallelizing compilers on multicomputers. In *TPDS*, 1994.

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

[17] M. Cheng et al. A TDI system and its application to approximation algorithms. In *Proc. FOCS*, 1998.

[18] Z. Chishti et al. Optimizing replication, communication, and capacity allocation in CMPs. In *Proc. ISCA*, 2005.

[19] M. L. Chu, S. A. Mahlke. Compiler-directed data partitioning for multicluster processors. In Proc. CGO, 2006.

[20] M. Cierniak, W. Li, Unifying Data and control transformations for distributed shared memory machines. In *Tech. Rep. U. Rochester*, 1994.

[21] K. Cooper L. Torczon. Engineering a compiler. 2008.

[22] T. H. Cormen et al. Introduction to algorithms. 2001.

[23] D. Culler et al. Parallel computer architecture: a hardware/software approach. 1999.

[24] E. DŠ'Hollander. Partitioning and labeling of loops by unimodular transformations. In *TPDS*, 1992.

[25] S. Ghosh et al. Cache miss equations: An analytical representation of cache misses.In *Proc. ICS*, 1997.

[26] M. W. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. In *Computer*, 1996.

[27] http://www.intel.com/p/en_US/products/server/processor/xeon7000?iid=servproc+body_xeon7400subtitle

[28] Intel quad-core Xeon. http://www.intel.com/quad-core/?cid=cim:ggl|xeon_us_clovertown|k7449|s

[29] http://www.intel.com/idf/.

[30] J. Kahle et al. Introduction to the Cell Multiprocessor. In *IBM Journal of Research and Development*, 2005.

[31] R. Kalla et al. IBM Power5 chip: a dual-core multithreaded processor. In *IEEE Micro*, 2004.

[32] M. Kandemir. Data locality enhancement for CMPs. In *Proc. ICCAD*, 2007.

[33] A. Kejariwal et al. Cache-aware iteration space partitioning. In *Proc. PPoPP*, 2008.

[34] I. Kodukula, K. Pingali. Data-centric transformations for locality enhancement. In *IJPP*, 2001.

[35] P. Kongetira et al. Niagara: A 32-way multithreaded SPARC processor. In *IEEE Micro*, 2005.

[36] M. Lam et al. The cache performance of blocked algorithms. In *Proc. ASPLOS*, 1991.

[37] H. Q. Le, et al. IBM POWER6 microarchitecture. In *IBM Jrnl. of R&D*, 2007.

[38] W. Li. Compiling for NUMA parallel machines. In *Ph.D. Thesis, Cornell University*, 1993.

[39] S. Liao et al. Data and computation transformations for Brook streaming applications on multiprocessors. In *Proc. CGO*, 2006.

[40] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. R. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News,* September 2005.

[41] R. McGowen. Adaptive designs for power and thermal optimization. In *Proc. ICCAD*, 2005.

[42] Omega library. http://www.cs.umd.edu/projects/omega.

[43] W. Pugh. Counting solutions to Presburger formulas: how and why. *Proc. PLDI*, 1994.

[44] Quad-core AMD Opteron. http://multicore.amd.com/us-en/quadcore/

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

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

[47] A. Shayesteh et al. Dynamically configurable shared CMP helper engines for improved performance. In *SIGARCH Comput. Archit.*, 2005.

[48] A. Silberschatz et al. Operating system concepts. 2008.

[49] SIMICS. http://www.virtutech.com/simics/simics.html.

[50] E. Sorenson, J. K. Flanagan. Using locality surfaces to characterize the SPECint 2000 benchmark suite. In *Workload Characterization of Emerging Computer Applications*, 2001.

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

[52] G. E. Suh et al. Dynamic partitioning of shared cache memory. In *Journal of Supercomputing*, 2004.

[53] J. Tims et al. Dataflow analysis driven dynamic data partitioning. In *Proc. of Workshop. on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1998.

[54] X. Vera et al. A fast and accurate framework to analyze and optimize cache memory behavior. In *TOPLAS* 2004.

[55] M. Wolf, M. Lam. A data locality optimizing algorithm. In *Proc. PLDI*, 1991.

[56] S. Woo et al. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. ISCA*, 1995.

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