J. Parallel Distrib. Comput. 71 (2011) 280–287

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Data locality and parallelism optimization using a constraint-based approach

Ozcan Ozturk

*Computer Engineering Department, Bilkent University, 06800, Bilkent, Ankara, Turkey*

ARTICLE INFO

*Article history:*
Received 5 January 2010
Received in revised form
4 July 2010
Accepted 4 August 2010
Available online 19 August 2010

*Keywords:*
Constraint networks
Compilers
Loop transformation
Data transformation
Parallelism
Locality

ABSTRACT

Embedded applications are becoming increasingly complex and processing ever-increasing datasets. In the context of data-intensive embedded applications, there have been two complementary approaches to enhancing application behavior, namely, data locality optimizations and improving loop-level parallelism. Data locality needs to be enhanced to maximize the number of data accesses satisfied from the higher levels of the memory hierarchy. On the other hand, compiler-based code parallelization schemes require a fresh look for chip multiprocessors as interprocessor communication is much cheaper than off-chip memory accesses. Therefore, a compiler needs to minimize the number of off-chip memory accesses. This can be achieved by considering multiple loop nests simultaneously. Although compilers address these two problems, there is an inherent difficulty in optimizing both data locality and parallelism simultaneously. Therefore, an integrated approach that combines these two can generate much better results than each individual approach. Based on these observations, this paper proposes a constraint network (CN)-based formulation for data locality optimization and code parallelization. The paper also presents experimental evidence, demonstrating the success of the proposed approach, and compares our results with those obtained through previously proposed approaches. The experiments from our implementation indicate that the proposed approach is very effective in enhancing data locality and parallelization.

## 1. Introduction

Memory is a key parameter in complex embedded systems since both the code complexity of embedded applications and amount of data they process are increasing. This is especially true for data-intensive embedded applications. In the context of data-intensive embedded applications, there have been two complementary approaches, namely, enhancing the data locality and improving the loop-level parallelism.

The data locality needs to be enhanced to maximize the number of data accesses satisfied from the higher levels of the memory hierarchy. Data locality is improved either by code restructuring or data transformations. Code structuring aims to modify the data access pattern of the loop by reorganizing the loop iterations, thereby aligning suitably with the underlying memory layout of the data structures. Although loop transformations are well studied and robust compiler techniques have been developed, the effectiveness is limited because of the intrinsic data dependencies. On the other hand, data transformations change the memory layout of data to make it suitable for a given data access pattern. Data transformations can be very useful when loop transformations are not possible for data dependency reasons.

Code parallelization in embedded chip multiprocessor systems needs to be revisited since off-chip data accesses are costlier when compared to uniprocessor architectures. Therefore, a compiler-based code parallelization needs to focus on reducing the number of off-chip requests. Moreover, previously proposed parallelization approaches mostly handle one loop nest at a time, which results in poor memory usage. This is mainly due to the fact that data sharing patterns across different loop nests will not be captured. This can be achieved by considering multiple loop nests simultaneously.

Although compilers address parallelization and data locality problems, there is an inherent difficulty in optimizing both problems simultaneously. This difficulty occurs due to the fact that effective parallelization tries to distribute the computation and necessary data across different processors, whereas locality targets placing data on the same processor [35]. Therefore, locality and parallelization may demand different loop transformations. Specifically, for a multi-dimensional array, a loop nest may demand a certain loop transformation to improve the data locality, whereas the same loop nest may require a different transformation for efficient parallelization for the said array. This results in a coupling problem, where the behaviors of two goals are coupled to each other. Note that it is also possible for a loop nest to have data dependencies that prevent the parallel execution of its iterations.

Motivated by these observations, this paper takes a different look at the data locality and code parallelization problems. Since, as mentioned above, a promising solution to both these problems should consider multiple loop nests simultaneously and each loop nest imposes additional constraints on the problem, we propose a *constraint network* (CN)-based solution to the locality

*E-mail address:* ozturk@cs.bilkent.edu.tr.

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.08.005

and parallelism problem. Constraint networks have been used for modeling and solving computationally intensive tasks in artificial intelligence [21,52]. A problem is expressed with a set of variables, along with variable domains and associated constraints. A search function is implemented to satisfy all the constraints by assigning values to variables from their specified domains. More specifically, the CN is expressed as a triplet ($\text{CN} = \langle \mathcal{P}, \mathcal{Q}, \mathcal{R} \rangle$), in which the goal is to satisfy the set of constraints ($\mathcal{R}$), for a given set of variables ($\mathcal{P}$) operating on a specific domain ($\mathcal{Q}$). The main contribution of this paper is to demonstrate that a CN is a very promising solution for addressing a combined locality and parallelism problem.

We believe that a CN is a viable and effective solution for our combined locality–parallelization optimization problem since a CN captures problems that involve multiple constraints that operate on the same domain, which is exactly what we are trying to achieve. More specifically, our optimization problem lends itself to resolving conflicts by reconciling inconsistent values in the CN, thereby providing an efficient solution.

It needs to be emphasized that, while we present CN-based solutions in this paper for the data unified locality and loop level parallelism optimization problem, it is certainly possible to use other search-based optimization techniques from the artificial intelligence (AI) domain [49,41] as well. Our point here is not to claim that a CN-based solution is superior to other possible search-based approaches. Rather, we simply demonstrate that a constraint-based approach is a promising alternative to the state-of-the-art heuristic-based schemes which currently dominate the practice in optimizing compiler implementations. Hopefully, this paper will provide part of the motivation for investigating other search-based techniques in the context of parallelism and locality related problems.

The main contributions of this paper can be summarized as follows.

- We propose a unified approach for the solution of data locality and parallelization problems, which in real-life applications must be solved simultaneously. These two problems often are solved separately and by using different techniques, and this makes this approach an important step forward.
- The CN formulation that we present can easily be modified to exclude locality or parallelization, if desired.
- We give experimental evidence showing the success of the proposed approach, and compare our results with those obtained through previously proposed approaches. The experiments from our implementation (with five embedded array-based applications) demonstrate the superiority of our approach.

The rest of this paper is organized as follows. A discussion of the related work on data locality optimization and code parallelization is given in Section 2. Our CN-based integrated locality and parallelization optimization is presented in Section 3. Section 4 presents experimental data to demonstrate the effectiveness of our strategy. Section 5 concludes the paper by summarizing our major results and briefly discusses the ongoing work.

## 2. Discussion of related work

In this section, we discuss the prior work on data locality optimization and code parallelization.

A majority of the related compiler work on cache locality optimization is based on loop transformations. Wolf and Lam [56] define reuse vectors and reuse spaces, and show how these concepts can be exploited by an iteration space optimization technique. Li [39] also uses reuse vectors to detect the dimensions of the loop nest that carry some form of reuse. Carr et al. [10] employ a simple locality criterion to reorder the computation to enhance the data locality. The loop-based locality enhancing techniques also include tiling [18,32,36]. In [20,42,53], the authors demonstrate how code restructuring can be used for improving the data locality in embedded applications. On the data transformation side, O'Boyle and Knijnenburg [45] explain how to generate code given a data transformation matrix. Leung and Zahorjan [38] focus more on minimizing memory consumption after a layout transformation. Kandemir et al. [30] propose a loop optimization technique based on an explicit layout representation. Applications of data layout optimizations to embedded codes include [34,33].

Cierniak and Li [15] were among the first to offer a scheme that unifies loop and data transformations. Anderson et al. [1] propose a transformation technique that makes data elements accessed by the same processor contiguous in the shared address space. They use permutations (of array dimensions) and strip-mining for possible data transformations. O'Boyle and Knijnenburg [46] discuss a technique that propagates memory layouts across nested loops. This technique can potentially be used for interprocedural optimization as well. Kandemir et al. [29] present an integrated compiler framework for improving the cache performance of scientific applications. In this approach, authors use loop transformations to improve the temporal locality whereas data layout optimizations are used to improve the spatial locality. Our approach is different from all these previous studies, as we employ a CN-based solution.

There has also been extensive work on reducing the memory space requirements of array-based applications [25,37,50,51,54, 59,11,12]. Note that these techniques are fundamentally different from the strategy presented in this paper, as their objective is to reduce the memory space demand rather than optimize for data locality and parallelization. While a CN-based framework can be used for reducing the memory space requirements as well, exploring that direction is beyond the scope of this paper.

Different schemes have been proposed for automatic code parallelization within different domains. In the context of high-end computing, the relevant studies include [2,26,57]. In [28], the authors implement a mechanism to use a different number of processor cores for each loop nest. Idle processors are switched to a low-power mode to reduce energy. Through the use of a pre-activation strategy, performance penalties are minimized. In the digital signal processing domain, Bondalapati [7] try to parallelize nested loops. In [24], the authors try to automatically parallelize the tiled loop nests using a message-passing interface (MPI [22]). Loop-level parallelism for coarse-grained reconfigurable architectures is introduced in [43], while Hogstedt et al. [27] investigate the parallel execution time of tiled loop nests. Ricci [48] reduces the complexity of the analysis needed for loop parallelization through an abstract interpretation. Lim et al. [40] find the optimal affine partition that minimizes the communication and synchronization costs in a parallel application. In [44], the communication and load imbalance is minimized using a locality graph and mixed integer nonlinear programming. Beletskyy et al. [5] propose a parallelization strategy for nonuniform dependences. A three-dimensional (3D) iteration space visualizer tool [58] has been designed to show the data dependencies and to indicate the maximal parallelism of nested loops. In [3], a gated single assignment (GSA)-based approach to exploit coarse-grain parallelism is proposed. The authors implement the parallelism with use-def chains between the statements.

There are also recent efforts on feedback-directed, iterative and adaptive compilation schemes [6,16,17,19,23,31]. While such techniques also search for the best solution over a search space (of possible transformations and their parameters such as tile sizes and unrolling factors), they typically execute the application to be optimized multiple times and, based on the results returned (from these executions), tune the search strategy further to obtain a better version. In comparison, our approach finds the solution in one step. We also believe that the iterative and adaptive solution techniques proposed in the literature can benefit from our approach in reducing the search space, and we plan to investigate this issue in our future research.

In our previous CN-based implementation, we exclusively address the data locality optimization [14,13]. Similarly, our DAC'06 [47] paper discusses the CN-based parallelization technique. While our previous implementations attack these two problems independently, in contrast, this work aims to simultaneously optimize the data locality and parallelization. Therefore, we address the combined data/loop optimization and parallelization problem, and thus, our framework is more general and subsumes the previous work. Moreover, a combined framework of data locality and parallelism generates much better results. The only other combined data locality and parallelization work we are aware of is that reported in [8,9]. These studies, however, focus on automatically generating OpenMP parallel code from C using polyhedral representation of programs.

## 3. Combined optimization for locality and parallelization

### 3.1. Problem description

In order to achieve an integrated locality and parallelization optimization, one needs to select the suitable loop transformation, data transformation, and correct parallelization. However, a coupling problem occurs because the same array can be accessed by multiple loop nests, and in determining its transformation–parallelization, we need to consider multiple nests. To our knowledge, only a few studies have focused on this problem previously.

From a data locality perspective, loop and data transformations should be carried out such that most of the data reuses take place in the innermost loop position. In the context of parallelization, the major goal in the past work has been maximizing the parallelism and minimizing communication as much as possible. In contrast, our target is a chip multiprocessor, and hence communication is not very expensive (it is on-chip). Our main goal instead is to minimize the number of off-chip accesses, and our CN-based approach tries to achieve that as much as possible. Note also that our approach can be integrated with any loop level optimization technique as it can take the loop level information as an input (as different ways of parallelizing a loop nest).

Our main goal in this section is to explore the possibility of employing *constraint network theory* for solving the integrated locality–parallelization optimization problem. A CN-based solution can be a promising alternative for our optimization problem since a CN is very powerful in capturing problems that involve multiple constraints that operate on the same domain. Note that we are focused on loop-based affine programs, where array subscript functions and loop bounds are affine functions of enclosing loop indices. Many programs in the embedded image/video processing domain fall into this category [12].

Locality is achieved by exploiting the temporal and spatial reuse exhibited by the innermost loop. While temporal reuse enables a data element accessed by a reference to be kept in a register, spatial reuse enables unit-stride accesses to consecutive data. Loop and data transformations aim to restructure the code such that most of the data reuses take place in the innermost loop position. The iterations of a loop nest can be represented using an iteration vector, each element of which corresponds to the value of a loop index, starting from the top loop position. For example, in a loop nest with two loops, $i_1$ (outer) and $i_2$ (inner), where $1 \leq i_1 \leq N_1$ and $1 \leq i_2 \leq N_2$, the iteration vector has two entries. Note that each value that can be taken by an iteration vector $\vec{I} = (\ i_1 \quad i_2\ )^T$, i.e., the set of potential values determined by the loop bounds, corresponds to an execution of the nest body, and all the values that can be taken by $\vec{I}$ collectively define the iteration space of the corresponding loop nest.

```
do $i = 1, N$
  do $j = 1, N$
    $Y(i,j) = X(j,i) - 3$
end do
do $i = 1, N$
  do $j = 1, N$
    $Z(i,j) = 1 - X(j,i)$
end do
do $i = 1, N$
  do $j = 1, N$
    $Y(j,i) = Y(j,i)/4$
end do
```

**Fig. 1.** An example code fragment.

One can apply a loop transformation represented by a linear transformation matrix $\mathcal{T}$. If $\mathcal{K}$ is the original access matrix, applying a loop transformation represented by the non-singular square matrix $\mathcal{T}$ generates a new access matrix $\mathcal{K}\mathcal{T}^{-1}$. On the other hand, a linear data transformation [45] can be implemented as a mapping of the index space of the array. Specifically, for an $m$-dimensional array, the data transformation represented by an $m \times m$ matrix $\mathcal{M}$ transforms the original array reference $\mathcal{K}\vec{I} + \vec{k}$ to $\mathcal{M}\mathcal{K}\vec{I} + \mathcal{M}\vec{k}$. As opposed to loop transformations [56], the iteration vector $(\vec{I})$ is not affected by data transformations, and the constant part of the reference (offset vector), $\vec{k}$, is transformed to $\mathcal{M}\vec{k}$ (the loop transformations do not affect the offset vector). Our goal is to select the best combination of loop and data transformations to achieve better data locality.

While loop/data transformations focus on improving the locality within a loop nest, it is also important to achieve locality among these different loop nests. Most of the prior techniques focus on efficiently parallelizing the individual loop nests. While this may be optimal for a certain loop, it fails to capture the data sharings among the different loop nests. Consider the example code fragment given in Fig. 1. One would prefer to either (1) parallelize the $i$ loops from the first and second nests and the $j$ loop from the third nest, or (2) parallelize the $j$ loops from the first and second nests and the $i$ loop from the third nest. This will enable that a given processor accesses the same set of array segments (of arrays $X$ and $Y$) during the execution of the different nests of the program.

We express the parallelization of a given loop nest using a vector. Consider the access pattern for array $X$ in the first loop nest given in Fig. 1. When the $i$ loop is parallelized, each processor accesses a set of consecutive columns of this array,and we can express this access pattern using $[*, \text{block}(P)]$, where $P$ indicates the processor number. More specifically, the second dimension of array $X$ is distributed over $P$ processors, whereas the first dimension is not distributed. Our goal is to select the parallelizations such that processors access the same portions of arrays as much as possible.

### 3.2. Problem formulation

We define our constraint network

$$\text{CN} = \{\langle \mathcal{P}_v, \mathcal{P}_w, \mathcal{P}_z \rangle, \langle \mathcal{Q}_v, \mathcal{Q}_w, \mathcal{Q}_z \rangle, \mathcal{R}\},$$

which captures the data locality and parallelization constraints derived by the compiler. In our CN, set $\mathcal{P}_v$ contains the transformations for the loop nests and set $\mathcal{P}_w$ contains data (layout) transformations for all the arrays that are manipulated by the code fragment being optimized, and set $\mathcal{P}_z$ contains the loop parallelizations for all loop nests. That is, we have $\mathcal{P}_v = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3, \ldots, \mathcal{V}_v\}$, $\mathcal{P}_w = \{\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3, \ldots, \mathcal{W}_w\}$, and $\mathcal{P}_z = \{\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3, \ldots, \mathcal{Z}_v\}$, where $v$ and $w$ correspond to the number of nests and the number of arrays, respectively. Note that these are unknowns we determine once the constraint network is solved; i.e., at the end of our optimization process, we want to determine a loop transformation for each loop nest in the program, a data transformation for each array in the program, and a loop parallelization for each loop nest in the program.

Sets $\mathcal{Q}_v$, $\mathcal{Q}_w$, and $\mathcal{Q}_z$ capture the domains for all variables in sets $\mathcal{P}_v$, $\mathcal{P}_w$, and $\mathcal{P}_z$, respectively.

To illustrate how this formulation is carried out in practice, we consider the example code fragment in Fig. 1. Note that, for clarity, we restrict ourselves to only loop interchange as the only possible loop transformation and dimension reindexing as the only possible data transformation. We have

$$\mathrm{CN} = \{\langle \mathcal{P}_v, \mathcal{P}_w, \mathcal{P}_z \rangle, \langle \mathcal{Q}_v, \mathcal{Q}_w, \mathcal{Q}_z \rangle, \mathcal{R}\},$$

where $\mathcal{P}_v = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\}$, $\mathcal{P}_w = \{\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3\}$, $\mathcal{P}_z = \{\mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3\}$, $\mathcal{Q}_v = \{\mathcal{D}_{V_1}, \mathcal{D}_{V_2}, \mathcal{D}_{V_3}\}$, $\mathcal{Q}_w = \{\mathcal{D}_{W_1}, \mathcal{D}_{W_2}, \mathcal{D}_{W_3}\}$, and $\mathcal{Q}_z = \{\mathcal{D}_{Z_1}, \mathcal{D}_{Z_2}, \mathcal{D}_{Z_3}\}$. Specifically, $\mathcal{V}_1$, $\mathcal{V}_2$, and $\mathcal{V}_3$ are the loop transformations for the loop nests, whereas $\mathcal{W}_1$, $\mathcal{W}_2$, and $\mathcal{W}_3$ are the data transformations for arrays. Similarly, $\mathcal{Z}_1$, $\mathcal{Z}_2$, and $\mathcal{Z}_3$ are the loop parallelizations we want to determine for the first, second, and third loop nest, respectively. In the above formulation, $\mathcal{Q}_v$, $\mathcal{Q}_w$, and $\mathcal{Q}_z$ capture our loop transformation, data transformation, and parallelization domains, respectively, and they can be expressed as follows:

$$\mathcal{Q}_v = \left\{ \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\}; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\}; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} \right\},$$

$$\mathcal{Q}_w = \left\{ \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\}; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\}; \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right\} \right\},$$

and

$$\mathcal{Q}_z = \left\{ \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}; \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}; \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} \right\}.$$

In the above expression, there are two parallelization strategies. The first one, captured by the $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ matrix in $\mathcal{Q}_z$, corresponds to the case where the first loop is parallelized and the second one is run sequentially. The second strategy, captured by $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$,on the other hand, represents an opposite approach, in which the second loop is parallelized and the first one is executed in a sequential manner. For the sake of illustration, we assume that we can parallelize only a single loop from each nest. More specifically, we have two alternatives; that is, we can parallelize the outer loop or the inner loop. Note that, in certain cases, it may not be possible to parallelize a loop which will reduce the number of entries in the CN search space.

Loop transformation is another domain we need to explore. Specifically, the $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ matrix in $\mathcal{Q}_v$ corresponds to no loop transformation case, whereas the $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$matrix represents loop interchange. Recall that we only employ a loop interchange as the loop transformation. For each of the three loop nests given in the example, we have the option to keep the loop the same or interchange it. $\mathcal{Q}_{v_1}$, $\mathcal{Q}_{v_2}$, and $\mathcal{Q}_{v_3}$ indicate the possible loop transformations for loops 1, 2, and 3, respectively. Similarly, matrices in $\mathcal{Q}_w$ correspond to no memory layout optimization and dimension reindexing, respectively.

$\mathcal{R}$, on the other hand, captures the necessary constraints for data locality and parallelism. More specifically, $\mathcal{R} = \{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$, where $\mathcal{R}_i$ captures the possible locality and parallelism constraints for the $i$th loop nest.

One feasible solution from the locality perspective is to keep the first two loops as they are and apply a loop interchange to the last loop. In order to improve the locality we also will need to dimension the reindexing on array $X$ for the first two loop nests. Note that, so far, we have not yet considered parallelism. The corresponding loop and data transformation matrices will be as the following:

$$\mathcal{R}_1 = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \epsilon \right\}$$

$$\mathcal{R}_2 = \left\{ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \epsilon, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right\}$$

$$\mathcal{R}_3 = \left\{ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \epsilon, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \epsilon \right\}.$$

In each $\mathcal{R}$, the first matrix shows the loop interchange and the remaining three matrices show the memory layout optimization for arrays $X$, $Y$, and $Z$, respectively. Our combined approach also exploits the data reuse across different (parallelized) loop nests. For the above example, our CN-based solution parallelizes the $i$ loop for the first two loop nests. On the other hand, we parallelize the $j$ loop since this will improve the locality of array $Y$ (between loop nests 1 and 3). Note that we have already performed a loop interchange on the last loop nest, and as a result $j$ is the outermost loop in the new loop nest.

### 3.3. Discussion

We now discuss how our CN-based formulation can be modified if we are to restrict our approach to only determining the locality or to only determining the parallelization. These simplified problem formulations may be necessary in some cases. For example, when we know that the loop/data structures cannot be modified in the application code being handled (e.g., as a result of explicit loop parallelization which could be distorted badly by a subsequent loop transformation), we might want to determine only parallelizations. Similarly, in some cases it may not be possible to work with any parallelization. Then, locality transformations are the only option to improve memory behavior. Actually, both these cases are easy to handle within our CN-based data locality and parallelism optimization framework. Basically, what we do is to drop the corresponding locality/parallelization constrains from consideration. For example, if we consider applying data locality optimizations only, we can redefine our network as $\mathrm{CN} = \{\langle \mathcal{P}_v, \mathcal{P}_w \rangle, \langle \mathcal{Q}_v, \mathcal{Q}_w \rangle, \mathcal{R}\}$, instead of $\mathrm{CN} = \{\langle \mathcal{P}_v, \mathcal{P}_w, \mathcal{P}_z \rangle, \langle \mathcal{Q}_v, \mathcal{Q}_w, \mathcal{Q}_z \rangle, \mathcal{R}\}$ as we did earlier. Also, in this case, $\mathcal{R}$ holds only the data locality constraints. Since this process is straightforward, we do not discuss it any further in this paper.

Note that, although a combined framework of data locality and parallelism generates much better results, it may not always be possible to achieve the best locality and best parallelization at the same time. In our experiments, we observe that most of the combined locality/parallelization frameworks generate a result

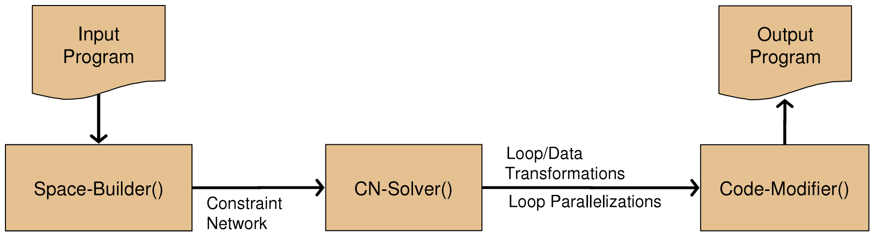

**Fig. 2.** Building blocks of our implementation.

**Table 1**
Benchmark codes.

| Benchmark name | Benchmark description | Data size (kB) | Domain size | Sol space size | Compilation time (ms) | Execution time (ms) |
|---|---|---|---|---|---|---|
| encr | Digital signature for security | 1137.66 | 324 | 7 212 | 34.41 | 434.63 |
| wood | Color-based surface inspection | 2793.81 | 851 | 17 638 | 104.33 | 961.58 |
| SP | All nodes shortest path | 2407.38 | 688 | 14 096 | 96.76 | 984.30 |
| srec | Speech recognition | 974.19 | 752 | 16 223 | 26.05 | 448.11 |
| jpegview | JPEG image display | 1628.54 | 990 | 20 873 | 82.34 | 798.76 |

with one dimension (locality or parallelization) being optimal and the other one being sub-optimal. However, in cases where both the parallelization and the locality can be optimal simultaneously, our approach optimizes both of them.

## 4. Experimental evaluation

There are two goals of our experimental analysis. First, we would like to see how much improvement our approach brings over the unoptimized (original) codes. Second, we want to compare our approach to previously proposed optimization schemes.

### 4.1. Set-up

We made experiments with six different versions of each code in our benchmark suite, which can be summarized as follows.

- Base: This is the base version against which we compare all the optimized schemes. This version does not apply any locality or parallelization optimization to the program code.
- Locality: This represents an approach that combines loop and data transformations under a unified optimizer. It is based on the algorithm described in [15]. It formulates the locality problem as one of finding a solution to a nonlinear algebraic system and solves it by considering each loop nest one by one, starting with the most expensive loop nest (in terms of execution cycles).
- Nest-Parallel: This represents a parallelization scheme, which parallelizes each loop nest in isolation and is referred to as the nest-based parallelization.
- Locality-CN: This is the locality-only version of our CN-based approach, as explained in Section 3.3.
- Parallelization-CN: This is the parallelization-only version of our CN-based approach, as explained in Section 3.3.
- Integ-CN: This is the integrated optimization approach proposed in this paper. It combines locality and parallelization optimizations under the CN-based optimization framework.

Fig. 2 depicts the implementation of our CN-based approach. Note that this implementation includes all three versions, namely, Locality-CN, Parallelization-CN, and Integ-CN. The input code is first analyzed by the compiler and possible data/loop transformations and parallelizations that go well with them are identified on a loop nest basis (note that these form the entries of our constraint set $\mathcal{R}$). The module that implements this functionality is called Space-Builder() since it builds the solution space that is explored by our search algorithms. This solution space information is subsequently fed to the CN-Solver(), which determines the desired parallelizations and loop/data transformations (if it is possible). This information is then fed to the compiler, which implements the necessary code modifications and data remappings. The compiler parts of this picture (i.e., Space-Builder() and Code-Modifier()) are implemented using the SUIF compiler [55]. The CN solver we wrote consists of approximately 1700 lines of C++ code. In our experiments, we restricted the entries of all loop/data transformation matrices to 1, −1 and 0.

Our experiments have been performed using the SimpleScalar infrastructure [4]. Specifically, we modeled an embedded processor that can issue and execute four instructions in parallel. The machine configuration we use includes separate L1 instruction and data caches; each is 16 kB, 2-way set-associative with a line size of 32 bytes. The L1 cache and main memory latencies are 2 and 100 cycles, respectively. We spawned a CPU simulation process for simulating the execution of each processor in our chip multiprocessors, and a separate communication simulation process captured the data sharing and coherence activity among the processors. Unless stated otherwise, all the experiments discussed in this section used this machine configuration.

Table 1 lists the benchmark codes used in our experimental evaluation. The second column of this table gives a description of the benchmark, and the next column gives the total data size (in kBs) manipulated by each benchmark. The fourth column gives the size of the domain for each benchmark, and the fifth column shows the size of the solution space (i.e., the total number of constraints in the $\mathcal{R}$ set). The sixth column gives the compilation times for the base version described above, and the seventh column gives the execution times, again under the base version.

### 4.2. Results

The solution times of the different optimized schemes, normalized with respect to those taken by the base version, are given in Table 2. These times are collected on a 500 MHz Sun Sparc architecture. Note that the solution times given for our CN-based approaches include the total time spent in the three components shown in Fig. 2, and are normalized with respect to the sixth column of Table 1. At least two observations can be made from these columns. First, as expected, our CN-based schemes take longer times to reach a solution compared to the previously proposed approaches. Second, among our schemes, Integ-CN takes the longest time, since it determines both loop/data transformations and parallelizations. It should be noted, however, that since compilation

**Table 2**
Normalized solution times with different schemes. The values represent the times as multiples of the base version time.

| Benchmark name | Solution times | | | | |
|---|---|---|---|---|---|
| | Locality | Nest-Parallel | Locality-CN | Parallelization-CN | Integ-CN |
| encr | 1.21 | 1.34 | 3.83 | 4.17 | 5.12 |
| wood | 1.36 | 1.37 | 3.46 | 3.77 | 4.98 |
| SP | 1.10 | 1.28 | 4.08 | 4.90 | 6.22 |
| srec | 1.47 | 1.89 | 3.61 | 4.15 | 5.46 |
| jpegview | 1.28 | 1.42 | 2.38 | 2.92 | 3.87 |

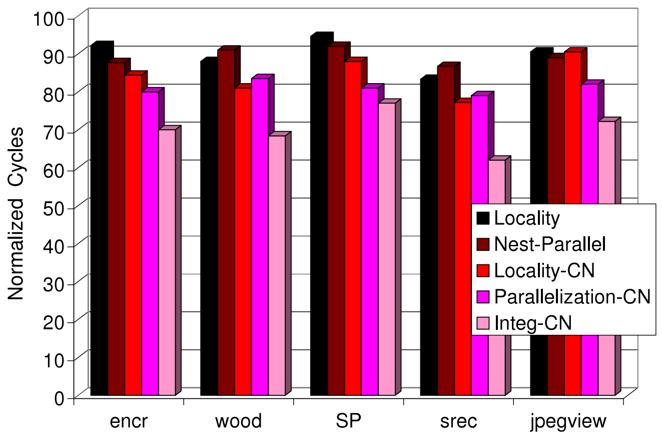

**Fig. 3.** Normalized execution cycles with different schemes.

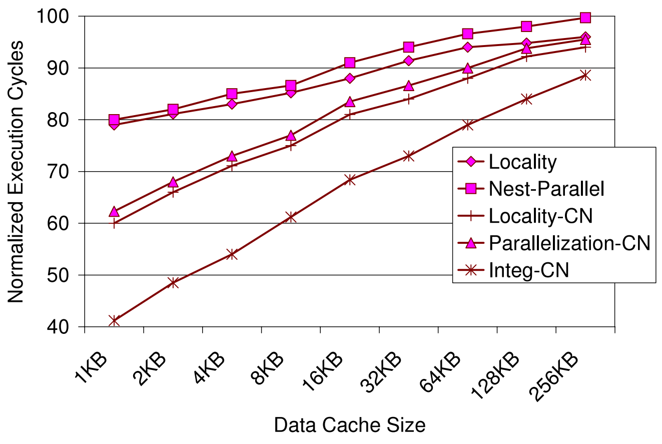

**Fig. 4.** Normalized execution cycles with different schemes and cache sizes (wood).

is essentially an offline process and code quality is a strong requirement in embedded computing, our belief is that these solution times are within tolerable limits.

The bar chart in Fig. 3 gives the execution times. Each bar is normalized with respect to the corresponding value of the base scheme (see the last column of Table 1). The first observation one can make from this graph is that the Integ-CN scheme generates the best savings across all five benchmark codes tested. The average savings it achieves are around 30.1%. In comparison, the average savings with the Locality-CN and Parallelization-CN are 15.8% and 18.9%, respectively. These results clearly emphasize the importance of considering both loop/data transformations and parallelization. When we look at the other schemes tested, the average savings are 10.3% and 10.8% for Locality and Nest-Parallel, respectively. Considering the results for Integ-CN, one can conclude that the CN-based approach is very successful in practice.

In our next set of experiments, we change the data cache size (keeping the remaining cache parameters fixed) and see how this effects the behavior of the schemes tested. The results are presented in Fig. 4. Recall that the default cache size used in our

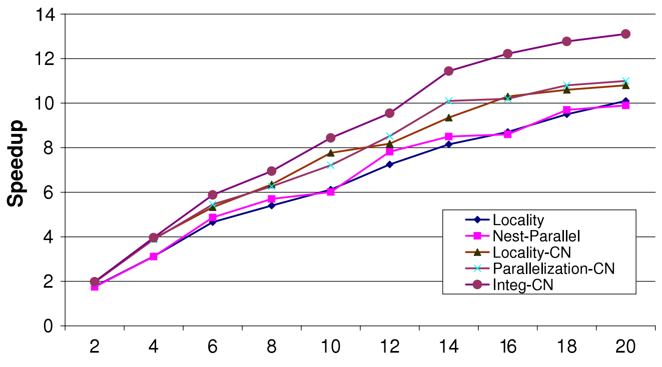

**Fig. 5.** Speedups with different number of processors (SP).

experiments was 16 kB. It is easy to see from this graph that, as we increase the size of the data cache, all the optimized versions tend to converge to each other. This is understandable when one remembers that the *y*-axis of this figure is normalized with respect to the execution cycles taken by the base version. The base version takes great advantage of increasing the cache size, and since all the versions are normalized with respect to it, we observe relative reductions in savings with increasing size. The second observation from Fig. 4 is that, when the data cache size is reduced, we witness the highest savings with the Integ-CN version. These results are very encouraging when one considers the fact that the increase in data set sizes of embedded applications is far exceeding the increase in on-chip storage capacities. Therefore, one can expect the Integ-CN scheme to be even more successful in the future.

We now evaluate the behavior of the different versions when the number of processors is changed. While we present results in Fig. 5 only for the benchmark SP, the observations we make extend to the remaining benchmarks as well. When these results are considered, one can see that scalability exhibited by the Integ-CN version is the best as we increase the number of processors. This is due to the fact that both nest-based locality optimizations and parallelization optimizations are considered simultaneously. Hence, the on-chip cache is utilized by maximizing the data reuse in the application.

## 5. Conclusions and ongoing work

The increasing gap between the speeds of processors and memory components in embedded systems makes the design and optimization of memory systems one of the most challenging issues in embedded system research. This is even more critical for chip multiprocessors, where effective utilization of limited on-chip memory space is of utmost importance due to high cost of off-chip memory accesses. The goal behind the work described in this paper is to make best use of on-chip memory components. Along this direction, this paper proposes a unified approach that integrates locality and parallelization optimizations for chip multiprocessors. We formulate the problem in a constraint network (CN), and solve it using search algorithms. The paper also presents an experimental evaluation of our CN-based approach and compares it quantitatively to alternative schemes. The results obtained from our implementation show that not only is a CN-based approach a viable option (since its solution times are reasonable) but it is also a desirable one (since it outperforms all the other schemes tested).

Our ongoing work includes incorporating other program optimizations in our CN-based infrastructure and developing customized search algorithms for different optimizations. Work is also underway in developing a CN-based solution to the problem of memory space minimization, and in using other search-based optimization schemes from the AI domain.

## Acknowledgments

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme, by a grant from IBM Corporation, and by TUBITAK grant #108E233.

## References

[1] J.M. Anderson, S.P. Amarasinghe, M.S. Lam, Data and computation transformations for multiprocessors, in: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995, pp. 166–178.

[2] J.M. Anderson, M.S. Lam, Global optimizations for parallelism and locality on scalable parallel machines, in: PLDI'93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, 1993, pp. 112–125.

[3] M. Arenaz, J. Tourino, R. Doallo, A GSA-based compiler infrastructure to extract parallelism from complex loops, in: Proc. of the 17th Annual International Conference on Supercomputing, 2003, pp. 193–204.

[4] T. Austin, E. Larson, D. Ernst, Simplescalar: an infrastructure for computer system modeling, IEEE Computer 35 (2) (2002) 59–67.

[5] V. Beletskyy, R. Drazkowski, M. Liersz, An approach to parallelizing non-uniform loops with the Omega calculator, in: Proc. of the International Conference on Parallel Computing in Electrical Engineering, 2002.

[6] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, E. Rohou, Iterative compilation in a non-linear optimisation space, in: Proc. Workshop on Profile and Feedback Directed Compilation, 1998.

[7] K. Bondalapati, Parallelizing DSP nested loops on reconfigurable architectures using data context switching, in: Proc. of the 38th Design Automation Conference, 2001, pp. 273–276.

[8] U. Bondhugula, Effective automatic parallelization and locality optimization using the polyhedral model, Ph.D. Thesis, Columbus, OH, USA, 2008.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, in: PLDI, 2008, pp. 101–113.

[10] S. Carr, K.S. McKinley, C.-W. Tseng, Compiler optimizations for improving data locality, SIGPLAN Notices 29 (11) (1994) 252–262.

[11] F. Catthoor, K. Danckaert, C. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T.V. Achteren, T. Omnes, Data Access and Storage Management for Embedded Programmable Processors, Kluwer Academic Publishers, Boston, MA, USA, 2002.

[12] F. Catthoor, E. de Greef, S. Suytack, Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[13] G. Chen, M. Kandemir, M. Karakoy, A constraint network based approach to memory layout optimization, in: Proc. of the Conference on Design, Automation and Test in Europe, 2005, pp. 1156–1161.

[14] G. Chen, O. Ozturk, M. Kandemir, I. Kolcu, Integrating loop and data optimizations for locality within a constraint network based framework, in: Proc. of International Conference on Computer-Aided Design, 2005.

[15] M. Cierniak, W. Li, Unifying data and control transformations for distributed shared-memory machines, in: PLDI'95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, 1995, pp. 205–217.

[16] A. Cohen, S. Girbal, O. Temam, A polyhedral approach to ease the composition of program transformations, in: Proc. of Euro-Par, 2004.

[17] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, N. Vasilache, Facilitating the search for compositions of program transformations, in: ICS'05: Proceedings of the 19th Annual International Conference on Supercomputing, 2005, pp. 151–160.

[18] S. Coleman, K.S. McKinley, Tile size selection using cache organization and data layout, in: PLDI'95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, 1995, pp. 279–290.

[19] K.D. Cooper, A. Grosul, T.J. Harvey, S. Reeves, D. Subramanian, L. Torczon, T. Waterman, ACME: adaptive compilation made efficient, in: LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2005, pp. 69–77.

[20] M. Dasygenis, E. Brockmeyer, F. Catthoor, D. Soudris, A. Thanailakis, Improving the memory bandwidth utilization using loop transformations, in: 15th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation, 2005, pp. 117–126.

[21] R. Dechter, J. Pearl, Network-based heuristics for constraint-satisfaction problems, Artificial Intelligence 34 (1) (1987) 1–38.

[22] M.P.I. Forum, MPI-2; extensions to the message-passing interface, 1997. URL: http://www.mpi-forum.org/docs/docs/html.

[23] G.G. Fursin, M.F.P. O'Boyle, P.M.W. Knijnenburg, Evaluating iterative compilation, in: Proc. Workshop on Languages and Compilers for Parallel Computing, 2002.

[24] G. Goumas, N. Drosinos, M. Athanasaki, N. Koziris, Automatic parallel code generation for tiled nested loops, in: Proc. of the ACM Symposium on Applied Computing, 2004, pp. 1412–1419.

[25] P. Grun, A. Nicolau, N. Dutt, Memory Architecture Exploration for Programmable Embedded Systems, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[26] M.H. Hall, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, M.S. Lam, Detecting coarse-grain parallelism using an interprocedural parallelizing compiler, in: Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, 1995, p. 49 (CDROM).

[27] K. Hogstedt, L. Carter, J. Ferrante, On the parallel execution time of tiled loops, IEEE Transactions on Parallel and Distributed Systems 14 (3) (2003) 307–321.

[28] I. Kadayif, M. Kandemir, M. Karakoy, An energy saving strategy based on adaptive loop parallelization, in: Proc. of the 39th Design Automation Conference, 2002, pp. 195–200.

[29] M. Kandemir, A. Choudhary, J. Ramanujam, P. Banerjee, Improving locality using loop and data transformations in an integrated framework, in: MICRO 31: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998, pp. 285–297.

[30] M. Kandemir, J. Ramanujam, A. Choudhary, A compiler algorithm for optimizing locality in loop nests, in: ICS'97: Proceedings of the 11th International Conference on Supercomputing, 1997, pp. 269–276.

[31] P.M.W. Knijnenburg, T. Kisuki, M.F.P. O'Boyle, Combined selection of tile sizes and unroll factors using iterative compilation, Journal of Supercomputing 24 (1) (2003) 43–67.

[32] I. Kodukula, N. Ahmed, K. Pingali, Data-centric multi-level blocking, in: PLDI'97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, 1997, pp. 346–357.

[33] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, H.D. Man, Cache conscious data layout organization for embedded multimedia applications, in: DATE, 2001, pp. 686–693.

[34] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, H.D. Man, Cache conscious data layout organization for conflict miss reduction in embedded multimedia applications, IEEE Transactions on Computers 54 (1) (2005) 76–81.

[35] M.S. Lam, Locality optimizations for parallel machines, in: CONPAR 94—VAPP VI: Proceedings of the Third Joint International Conference on Vector and Parallel Processing, 1994, pp. 17–28.

[36] M.D. Lam, E.E. Rothberg, M.E. Wolf, The cache performance and optimizations of blocked algorithms, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63–74.

[37] V. Lefebvre, P. Feautrier, Automatic storage management for parallel programs, Parallel Computing 24 (3–4) (1998) 649–671.

[38] S. Leung, J. Zahorjan, Optimizing data locality by array restructuring, Tech. Rep. TR-95-09-01, 1995. URL: citeseer.ist.psu.edu/leung95optimizing.html.

[39] W. Li, Compiling for numa parallel machines, Ph.D. Thesis, Cornell University, Ithaca, NY, USA, 1993.

[40] A.W. Lim, G.I. Cheong, M.S. Lam, An affine partitioning algorithm to maximize parallelism and minimize communication, in: Proc. of the 13th International Conference on Supercomputing, 1999, pp. 228–237.

[41] G. Luger, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, Addison-Wesley, 2004.

[42] P. Marchal, J.I. Gomez, F. Catthoor, Optimizing the memory bandwidth with loop fusion, in: Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2004, pp. 188–193.

[43] B. Mei, S. Vernalde, D. Verkest, H.D. Man, R. Lauwereins, Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling, in: Proc. of the Conference on Design, Automation and Test in Europe, 2003, pp. 10296–10301.

[44] A. Navarro, E. Zapata, D. Padua, Compiler techniques for the distribution of data and computation, IEEE Transactions on Parallel and Distributed Systems 14 (6) (2003) 545–562.

[45] M.F.P. O'Boyle, P.M.W. Knijnenburg, Nonsingular data transformations: definition, validity, and applications, International Journal of Parallel Programming 27 (3) (1999) 131–159.

[46] M.F.P. O'Boyle, P.M.W. Knijnenburg, Integrating loop and data transformations for global optimization, Journal of Parallel and Distributed Computing 62 (4) (2002) 563–590.

[47] O. Ozturk, G. Chen, M. Kandemir, A constraint network based solution to code parallelization, in: Proc. Design Automation Conference, DAC, 2006.

[48] L. Ricci, Automatic loop parallelization: an abstract interpretation approach, in: Proc. of the International Conference on Parallel Computing in Electrical Engineering, 2002, pp. 112–118.

[49] S.J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education, 2003.
[50] M.M. Strout, L. Carter, J. Ferrante, B. Simon, Schedule-independent storage mapping for loops, SIGPLAN Notices 33 (11) (1998) 24–33.
[51] W. Thies, F. Vivien, J. Sheldon, S. Amarasinghe, A unified framework for schedule and storage optimization, in: PLDI'01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, 2001, pp. 232–242.
[52] E. Tsang, A glimpse of constraint satisfaction, Artificial Intelligence Review 13 (3) (1999) 215–227.
[53] S. Verdoolaege, M. Bruynooghe, G. Janssens, F. Catthoor, Multi-dimensional incremental loops fusion for data locality, in: 14th IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2003, pp. 17–27.
[54] D. Wilde, S.V. Rajopadhye, Memory reuse analysis in the polyhedral model, in: Euro-Par'96: Proceedings of the Second International Euro-Par Conference on Parallel Processing, 1996, pp. 389–397.
[55] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, J.L. Hennessy, SUIF: an infrastructure for research on parallelizing and optimizing compilers, SIGPLAN Notices 29 (12) (1994) 31–37.
[56] M.E. Wolf, M.S. Lam, A data locality optimizing algorithm, in: Proceedings of the Conference on Programming Language Design and Implementation, 1991, pp. 30–44.
[57] M.E. Wolf, M.S. Lam, A loop transformation theory and an algorithm to maximize parallelism, IEEE Transactions on Parallel and Distributed Systems 2 (4) (1991) 452–471.
[58] Y. Yu, E.H. D'Hollander, Loop parallelization using the 3D iteration space visualizer, Journal of Visual Languages and Computing 12 (2) (2001) 163–181. URL: citeseer.ist.psu.edu/yu01loop.html.
[59] Y. Zhao, S. Malik, Exact memory size estimation for array computations without loop unrolling, in: Proceedings of the 36th ACM/IEEE Conference on Design Automation, 1999, pp. 811–816.

**Ozcan Ozturk** is an Assistant Professor in the Department of Computer Engineering at Bilkent University. Prior to joining Bilkent, he worked as a software optimization engineer in Cellular and Handheld Group at Intel (Marvell). He received his Ph.D. from Pennsylvania State University, his M.S. degree from the University of Florida, and his B.Sc. degree from Bogazici University, all in Computer Engineering. His research interests are in the areas of chip multiprocessing, computer architecture, many-core architectures, and parallel processing. He is a recipient of 2009 IBM Faculty Award and 2009 Marie Curie Fellowship from the European Commission.