Compiler-Directed Cache Polymorphism [*]

J. S. Hu, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, H. Saputra and W. Zhang
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802.

ABSTRACT

Classical compiler optimizations assume a fixed cache architecture and modify the program to take best advantage of it. In some cases, this may not be the best strategy because each loop nest might work best with a different cache configuration and transforming a nest for a given fixed cache configuration may not be possible due to data dependences. Working with a fixed cache configuration can also increase energy consumption in loops where the best required configuration is smaller than the default (fixed) one. In this paper, we take an alternate approach and modify the cache configuration for each nest depending on the access pattern exhibited by the nest. We call this technique *compiler-directed cache polymorphism* (CDCP). More specifically, in this paper, we make the following contributions. First, we present an approach for analyzing data reuse properties of loop nests. Second, we give algorithms to simulate the footprints of array references in their reuse space. Third, based on our reuse analysis, we present an optimization algorithm to compute the cache configurations for each nest. Our experimental results show that CDCP is very effective in finding the near-optimal data cache configurations for different nests in array-intensive applications.

Categories and Subject Descriptors

B.3 [**Hardware**]: Memory Structures; D.3.4 [**Programming Languages**]: Processors—*Compilers;Optimization*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Embedded software, compilers, cache polymorphism, data reuse, cache locality, energy consumption.

[*]This work was supported in part by grants from PDG, National Science Foundation grants CAREER 0093082&0093085, 0103583, 0082064 and a GSRC grant.

1. INTRODUCTION

Most of today's microprocessor systems include several special architectural features (e.g., large on-chip caches) that use a significant fraction of on-chip transistors. These complex and energy-hungry features are meant to be applicable across different application domains. However, they are effectively wasted for applications that cannot fully utilize them, as they are implemented in a rigid manner. For example, not all the loops in a given array-based embedded application can take advantage of a large on-chip cache. Also, working with a fixed cache configuration can increase energy consumption in loops where the best required configuration (from the performance angle) is smaller than the default (fixed) one. This is because a larger cache can result in a large per access energy.

The conventional approach to address the locality problem for caches (that is, the problem of maximizing the number of cache hits) is to employ compiler optimization techniques [8]. Current compiler techniques generally work under the assumption of a fixed cache memory architecture, and try to modify the program behavior such that the new behavior becomes more compatible with the underlying cache configuration. However, there are several problems with this method. First, these compiler-directed modifications sometimes are not effective when data dependences prevent necessary program transformations. Second, the available cache space sometimes cannot be utilized efficiently, because the static configuration of cache does not match different requirements of different programs and/or of different portions of the same program. Third, most of the current compiler techniques (adapted from scientific compilation domain) do not take energy issues into account in general.

An alternative approach to the locality problem is to use reconfigurable cache structures and dynamically tailor the cache configurations to meet the execution profile of the application at hand. This approach has the potential to address the locality problem in cases where optimizing the application code alone fails. However, previous research on this area [1, 9] is mainly focused on the implementation and the employment mechanisms of these designs, and lacks software-based techniques to direct dynamic cache reconfigurations. Recently, a compiler-directed scheme to adapt the cache assist was proposed in [6]. Our work focuses on the cache as opposed to the cache assist.

In this paper, we propose a strategy where an optimizing compiler decides the best cache configuration for each nest in the application code. More specifically, in this paper, we make the following contributions. First, we present techniques for analyzing the data reuse properties of a given loop nest and constructing formal expressions of these reuse patterns. Second, we develop algorithms to simulate the footprints of array references. Our simulation approach is much more efficient than classical cycle-based simulation techniques as it simulates only data reuse space. Third, we develop an optimization algorithm for computing the optimized cache configurations for each loop nest. We also provide a program level algorithm for selecting dynamic cache configurations. We focus on the behavior of array references in loop nests as loop nests are the most important part of array-intensive media and signal processing application programs. In most cases, the computation performed in loop nests dominates the execution time of these programs. Thus, the behavior of the loop nests determines both performance and energy behavior of applications. Previous research [8] shows that the performance of loop nests is directly influenced by the cache behavior of array references. Also, recently, energy consumption has become an important issue in embedded systems [9]. Consequently, determining a suitable combination of cache memory configuration and optimized software is a challenging problem in embedded design world.

The rest of this paper is organized as follows. Section 2 reviews basic concepts, notions, and representations for array-based codes. In Section 3, concepts related to cache behavior such as cache misses, interferences, data reuse, and data locality are analyzed. Section 4 introduces our compiler-directed cache polymorphism technique, and presents a complete set of algorithms to implement it. We present experimental results in Section 5 to show the effectiveness of our technique. Finally, Section 6 concludes the paper with a summary and discusses some future work on this topic.

2. ARRAY-BASED CODES

This paper is particularly targeted at the array-based codes. Since the performance of loop nests dominates the overall performance of the array-based codes, optimizing nests is particularly important for achieving best performance in many embedded signal and video processing applications. Optimizing data locality (so that the majority of data references are satisfied from the cache instead of main memory) can improve the performance and energy efficiency of loop nests in the following ways. First, it can significantly reduce the number of misses in data cache, thus avoiding frequent accesses to lower memory hierarchies. Second, by reducing the number of accesses to the lower memory hierarchies, the increased cache hit rate helps promote the energy efficiency of the entire memory system. In this section, we discuss some basic notions about array-based codes, loop nests, array references as well as some assumptions we made.

2.1 Representation for Programs

We assume that the application code to be optimized has the format which is shown in Figure 1.

Assumption 1. *Each array in the application code being optimized is declared in the global declaration section of the program. The arrays declared in the global section can be referenced by any loop in the code.*

This assumption is necessary for our algorithms that will be discussed in following sections. In the optimization stage of computing the cache configuration for the loop nests, Assumption 1 ensures an exploitable relative base address of each array involved.

```
#include < header.h >
...
Global Declaration Section of Arrays;
...
main(int argc, char *argv[ ])
{
  ...
  Loop Nest No. 0;
  ...
  Loop Nest No. 1;
      ⋮
  Loop Nest No. l;
  ...
}
```

Figure 1: Format for a Program.

```
for($i_1 = l_1; i_1 \leq u_1; i_1 += s_1$)
  for($i_2 = l_2; i_2 \leq u_2; i_2 += s_2$)
    ...
      for($i_n = l_n; i_n \leq u_n; i_n += s_n$)
        {
          $\cdots AR_1[f_{1,1}(\vec{i})][f_{1,2}(\vec{i})]\cdots[f_{1,d_1}(\vec{i}))]\cdots;$
          $\cdots AR_2[f_{2,1}(\vec{i})][f_{2,2}(\vec{i})]\cdots[f_{2,d_2}(\vec{i}))]\cdots;$
                  ⋮
          $\cdots AR_r[(f_{r,1}(\vec{i})][f_{r,2}(\vec{i})]\cdots[f_{r,d_r}(\vec{i}))]\cdots;$
        }
```

Figure 2: Format for a Loop Nest.

Since loop nests are the main structures in array-based programs, program codes between loop nests can be neglected. We also assume that each nest is independent from the others. That is, as shown in Figure 1, the application contains a number of independent nests, and no inter-loop-nest data reuse is accounted for. This assumption can be relaxed to achieve potentially more effective utilization of reconfigurable caches. This will be one of our future research. Note that several compiler optimizations such as loop fusion, fission, and code sinking can be used to bring a given application code into our format [12].

Assumption 2. *All loop nests are at the same program lexical level, the global level. There is no inter-nesting between any two different loop nests.*

Assumption 3. *All nests in the code are perfectly-nested, i.e., all array operations and array references only occur at the innermost loop.*

These assumption, while not vital for our analysis, make our implementation easier. We plan to relax these in our future work.

2.2 Representation for Loop Nests

In our work, loop nests form the boundaries at which dynamic cache reconfigurations occur. Figure 2 shows the format for a loop nest.

In this format, $\vec{i}$ stands for the loop index vector, $\vec{i} = (i_1, i_2, \cdots, i_n)^T$. Notations l_j, u_j and s_j are the corresponding lower bound, upper bound, and stride for each loop index i_j, where $j = 1, 2, \cdots, n$. AR_1, AR_2, $\cdots$, and AR_r correspond to different instances of array references in the nest. Note that these may be same or different references to the same array, or different references to different arrays. Function $f_{j,k}(\vec{i})$ is the subscript (expression) function (of $\vec{i}$) for the

k^{th} subscript of the j^{th} array reference, where $j = 1, 2, \cdots, r$, $k = 1, 2, \cdots, d_k$, and d_k is the number of dimensions for the corresponding array.

2.3 Representation for Array References

In a loop nest with the loop index vector $\vec{i}$, a reference AR_j to an array with m dimensions is expressed as:

$$AR_j[f_{j,1}(\vec{i})][f_{j,2}(\vec{i})] \cdots [f_{j,m}(\vec{i})].$$

We assume that the subscript expression functions $f_{j,k}(\vec{i})$ are affine functions of the loop indices and loop-invariant constants. A row-major storage layout is assumed for all arrays as in C language. Assuming that the loop index vector is an n depth vector; that is, $\vec{i} = (i_1, i_2, \cdots, i_n)^T$, where n is the number of loops in the nest, an array reference can be represented as:

$$\begin{pmatrix} f_{j,1} \\ f_{j,2} \\ \vdots \\ f_{j,m} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{pmatrix} + \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{pmatrix} \tag{1}$$

The vector at the left side of the above equation is called array reference subscript vector $\vec{f}$. The matrix above is defined as access matrix A. The rightmost vector is known as the constant offset vector $\vec{c}$. Thus, the above equation can be also written as [12]:

$$\vec{f} = A\vec{i} + \vec{c} \tag{2}$$

3. CACHE BEHAVIOR

In this section, we review some basic concepts about cache behavior. As noted earlier, in array-intensive applications, cache behavior is largely determined by the footprints of the data manipulated by loop nests. In this paper, we first propose an algorithm for analyzing the cache behavior for different arrays and different array references in a given loop nest. Based on the information gathered from this analysis, we then propose another algorithm to compute the cache memory demand in order to achieve a perfect cache behavior for the loop nest being analyzed, and suggest a cache configuration.

3.1 Cache Misses

There are three types of cache misses: compulsory (cold) misses, capacity misses, and conflict (interference) misses. Different types of misses influence the performance of program in different ways. Note that, most of the data caches used in current embedded systems are implemented as set-associative caches or direct-mapping caches in order to achieve high speed, low power, and low implementation cost. Thus, for these caches, interference misses can dominate the cache behavior, particularly for array-based codes. It should be stressed that since the cache interferences occur in a highly irregular manner, it is very difficult to capture them accurately [11]. Ghosh et al. proposed cache miss equations in [4] as an analytical framework to compute potential cache misses and direct code optimizations for cache behavior.

3.2 Data Reuse and Data Locality

Data reuse and data locality concepts are discussed in [12] in detail. Basically, there are two types of data reuses: temporal reuse and spatial reuse. In a given loop nest, if a reference accesses the same memory location across different loop iterations, this is termed as temporal reuse; if the reference accesses the same cache block (not necessarily the same memory location), we call this spatial reuse. We can consider temporal reuse is a special case of spatial reuse. If there are different references accessing the same memory location, we say that a group-temporal reuse exists; whereas if different references are accessing the same cache block, it is termed as group-spatial reuse. Note that group reuse only occurs among different references of the same array in a loop nest. When the reused data item is found in the cache, we say that the reference exhibits *locality*. This means that data reuse does *not* guarantee data locality. We can convert a data reuse into locality only by catching the reused item in cache. Classical loop-oriented compiler techniques try to achieve this by modifying the loop access patterns.

4. ALGORITHMS FOR CACHE POLYMORPHISM

The performance and energy behavior of loop nests are largely determined by their cache behavior. Thus, how to optimize the cache behavior of loop nests is utmost important for satisfying high-performance and energy efficiency demands of array-based codes.

There are at least two kinds of approaches to perform optimizations for cache behavior. The conventional way is compiler algorithms that transform loops using interchange, reversal, skewing, and tiling transformations, or transform the data layout to match the array access pattern. As mentioned earlier, the alternative approach is to modify the underlying cache architecture depending on the program access pattern. Recent research work [7] explores the potential benefits from the second approach. The strategy presented in [7] is based on exhaustive simulation. The main drawback of this simulation-based strategy is that it is extremely time consuming and can consider only a fixed set of configurations. Typically, simulating each nest with all possible cache configurations makes this approach unsuitable for practice. In this section, we present an alternative way for determining the suitable cache configurations for different sections (nests) of a given code.

4.1 Compiler-directed Cache Polymorphism

The existence of cache interferences is the main factor that degrades the performance of a loop nest. Cache interferences disrupt the data reuse in a loop nest by preventing data reuse from being converted into locality. Note that both self-interferences or cross-interferences can prevent a data item from being used while it is still in the cache. Our objective is then to determine the cache configurations that help reduce interferences. The basic idea behind the compiler-directed cache polymorphism (CDCP) is to analyze the source code of an array-based program and determine data reuse characteristics of its loop nests at compile time, and then to compute a suitable (near-optimal) cache configuration for each loop nest to exploit the data locality implied by its reuse. The near-optimal cache configuration determined for each nest eliminates most of the interference misses while keeping the cache size and associativity under control. In this way, it optimizes execution time and energy at the same time. In fact, increasing either cache capacity or associativity further only increases energy consumption. In this approach, the source codes are not modified (obviously, they can be optimized be-

```
for(i = 0; i ≤ N_1; i++)
  for(j = 0; j ≤ N_2; j++)
    for(k = 0; k ≤ N_3; k++)
      for(l = 0; l ≤ N_4; l++)
        {
          a[i + 2*k][2*j + 2][l] = a[i + 2*k][2*j][l];
          b[j][k + l][i] = a[2*i][k][l];
        }
```

Figure 3: Example Code – a Loop Nest.

fore our algorithms are run; what we mean here is that we do not do any further code modifications for the sake of cache morphism).

At the very high level, our approach can be described as follows. First, we use compiler to transform the source codes into an intermediate format. In the second step, each loop nest is processed as a basic element for cache configuration. In each loop nest, references of each array are assigned into different uniform reference sets. Each uniform set is then analyzed to determine the reuse they exhibit over different loop levels. Then, for each array, an algorithm is used to simulate the footprints of the reuse space within the layout space of this array. Following this, a loop nest level algorithm optimizes the cache configurations while ensuring data locality. Finally, the code is generated such that these dynamic cache configurations are activated at runtime (in appropriate points in the application code).

4.2 Array References and Uniform Reference Sets

Every array reference is expressed in Equation 2, $\vec{f} = A\vec{i} + \vec{c}$, in which $\vec{f}$ is the subscript vector, A is the access matrix, $\vec{i}$ is the loop index vector and $\vec{c}$ is the constant vector. All the information are stored in the array reference leaf, array node and its parent loop-nest node of the intermediate codes. Consider a piece of code in Figure 3, which is a loop nest:

The first reference of array a is represented by the following access matrix A_a and constant offset vector $\vec{c_a}$,

$$A_a : \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \vec{c_a} : \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}.$$

The reference to array b is also represented by its access matrix A_b and constant offset vector $\vec{c_b}$:

$$A_b : \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \vec{c_b} : \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

The definition of uniform reference set is very similar to the uniformly generated set [3]. If two references to an array have the same access matrix and only differ in constant offset vectors, these two references are said to belong to the same uniform reference set. Constructing uniform reference sets for an array provides an efficient way for analyzing the data reuse for the said array. This is because all references in an uniform reference set have same data access patterns and data reuse characteristics. Also, identifying uniform reference sets allows us to capture group reuse easily.

4.3 Algorithm for Reuse Analysis

In the following sections, we use a bottom-up approach to introduce the algorithms for implementing our compiler-directed cache polymorphism technique. First, algorithms analyzing the data reuses including self-reuses and group-reuses are provided for each uniform reference set in this subsection.

```
INPUT: access matrix $A_{m*n}$ of a uniform reference set
          array node, loop-nest node
          a given cache block size: BK_SZ
OUTPUT: self-reuse pattern vector $\overrightarrow{SRP_n}$ of this uniform set
Begin
  Initial self-reuse pattern vector: $\overrightarrow{SRP_n} = \vec{0}$
  Set current loop level CLP to be the innermost loop:
      $CLP = n$
  Do
    Set current dimension level CDN to be the highest
        dimension: $CDN = 0$
    Set index occurring flag IOF: IOF = FALSE
    Do
      If Element in access matrix $A[CDN][CLP] \neq 0$
        Set IOF = TRUE
        Break
      Go up to the next lower dimension level
    While $CDN == $ the lowest dimension
    If $IOF == FALSE$
      Set reference has temporal reuse at this level:
          $SRP[CLP]$ = TEMP-REUSE
    Else If $CDN == m$
      If $A[CDN][CLP] * s[CLP] < BK\_SZ/ELMT\_SZ$
        Set reference has spatial reuse at this level:
            $SRP[CLP]$ = SPAT-REUSE
    Go up to the next higher loop level
  While $CLP ==$ the outermost loop level
End.
```

Figure 4: Algorithm 1: Self-Reuse Analysis.

4.3.1 Self-Reuse Analysis

Before the reuse analysis, all references of an array in a loop nest are first constructed into several uniform reference sets. Self-reuses (both temporal and spatial) are analyzed at the level of uniform set. This algorithm works on access matrix. The detailed algorithm is shown in Figure 4.

This algorithm checks each loop index variable from the innermost loop to the outermost loop to see whether it occurs in the subscript expressions of the references. If the j^{th} loop index variable i_j does not occur in any subscript expression, the reflection in access matrix is that all elements in the j^{th} column are 0. This means that the iterations at the j^{th} loop do not change the memory location accessed, i.e., the array reference has self-temporal reuse in the j^{th} loop. If the index variable only occurs in the lowest (the fastest-changing) dimension (i.e., the m^{th} dimension), the distance between the contiguous loop iterations is checked. In the algorithm, $s[CLP]$ is the stride of the CLP^{th} loop, BK_SZ is a given cache block size and ELMT_SZ is the size of array elements. If the distance ($A[CDN][CLP] * s[CLP]$) between two contiguous iterations of this reference is within a cache block, it has spatial reuse in this loop level.

4.3.2 Group-Reuse Analysis

Group reuses only exist among references in the same uniform reference set. Group-temporal reuse occurs when different references access the same data location across the loop iterations, while group-spatial reuse exists when different references access the same cache block in the same or different loop iterations. Algorithm 2 in Figure 5 exploits a simplified version of group reuse which only exists in one loop level.

When a group-spatial reuse is found at a particular loop level, the algorithm in Figure 5 first checks whether this level

```
INPUT: a uniform reference set with A and $\vec{c}$s
       array node, loop-nest node
       a given cache block size: BK_SZ
OUTPUT: group-reuse pattern vector $\overrightarrow{GRP_n}$ of this uniform set
Begin
  Initial group-reuse pattern vector: $\overrightarrow{GRP_n} = \vec{0}$
  For each pair of constant vectors $\vec{c_1}$ and $\vec{c_2}$
    If $\vec{c_1}$ and $\vec{c_2}$ only differ at the $j^{th}$ element
      Set init_dist = $| c_1[j] - c_2[j] |$
      Check the $j^{th}$ row in access matrix A
      Find the first occurring loop index variable (non-zero
          element) starting from the innermost loop, say $i_k$
      If $k < 1$
        If $j == m$ and $init\_dist < BK\_SZ/ELMT\_SZ$
          Continue
      Else
        Check the $k^{th}$ column of access matrix A
        If $i_k$ only occurs in the $j^{th}$ dimension
          If $j == m$ //m is the lowest dimension of array
            If $init\_dist \% A[k][m] == 0$
              Set GRP[k] = TEMP-REUSE
            Else If GRP[k] == 0
              Set GRP[k] = SPAT-REUSE
          Else //$j < m$
            If $init\_dist \% A[k][m] == 0$
              Set GRP[k] = TEMP-REUSE
End.
```

Figure 5: Algorithm 2: Group-Reuse Analysis.

has group-temporal reuse for other pairs of references. If it does not have such reuse, this level will be set to have group-spatial reuse. Otherwise, it just omits the current reuse found. For group-temporal reuse found at some loop level, the element corresponding to that level in the group-reuse vector $\overrightarrow{GRP_n}$ will be directly set to have group-temporal reuse.

Now, for each array and each of its uniform reference sets in a particular loop nest, using Algorithm 1 and Algorithm 2, the reuse information at each loop level can be collected. As for the example code in subsection 4.3, references to array a have self-spatial reuse at loop level l, self-temporal reuse at loop level j and group reuse at loop level j. Reference of array b has self-spatial reuse at loop level i.

Note that, in contrast to the most of the previous work in reuse analysis (e.g., [12]), this approach is simple and computes reuse information without solving a system of equations.

4.4 Simulating the Footprints of Reuse Spaces

The next step in our approach is to transform those data reuses into real data localities. A straightforward idea is to make the data cache large enough to hold all the data in these reuse spaces of the arrays. Note that data which are out of reuse spaces are not necessary to be kept in cache after the first reference since there is no reuse for those data. As discussed earlier, the cache interferences can significantly affect the overall performance of a nest. Thus, the objective of our technique is to find a near-optimal cache configuration, which can reduce or eliminate the majority of the cache interferences within a nest. An informal definition of a near-optimal cache configuration is as follows:

DEFINITION 1. *A near-optimal cache configuration is the possibly smallest cache in size and associativity which achieves a near-optimal number of cache misses. And, any increase in either cache size or associativity over this configuration does not deliver further significant improvement.*

In order to figure out such a near-optimal cache configuration that would contain the entire reuse space for a loop nest, the real cache behavior in these reuse spaces must be made available for potential optimizations. In this section, we provide an algorithm that simulates the exact footprints (memory addresses) of array references in their reuse spaces.

Suppose, for a given loop index vector $\vec{i}$, an array reference with a particular value of $\vec{i} = (i_1, i_2, \cdots, i_n)^T$ can be expressed as follows:

$$f(\vec{i}) = SA + Cof_1 * i_1 + Cof_2 * i_2 + \cdots + Cof_n * i_n. \quad (3)$$

Here, SA is starting address of the array reference, which is different from the base address (the memory address of the first array element) of an array. It is the constant part of the above equation. Suppose that the data type size of the array elements is $elmt\_sz$, the depth of dimension is m, the dimensional bound vectors are $\overrightarrow{dl_m} = (dl_1, dl_2, \cdots, dl_m)^T$, $\overrightarrow{du_m} = (du_1, du_2, \cdots, du_m)^T$, and the constant offset vector $\vec{c} = (c_1, c_2, \cdots, c_m)^T$, SA is derived from the following equation:

$$SA = elmt\_sz * \sum_{j=1}^{m} \prod_{k=j+1}^{m+1} dd_k * c_j, dd_k = \begin{cases} 1, & k = m+1 \\ du_k - dl_k, & k \le m \end{cases} \quad (4)$$

$Cof_j (j = 1, 2, \cdots, n)$ are integrated coefficients of the loop index variables. Suppose the access matrix is A_{m*n}, Cof_j is derived as follows:

$$Cof_j = elmt\_sz * \sum_{l=1}^{m} \prod_{k=l+1}^{m+1} dd_k * a_{lj}, dd_k = \begin{cases} 1, & k = m+1 \\ du_k - dl_k, & k \le m \end{cases} \quad (5)$$

Note that, with Equation 3, the address of an array reference at a particular loop iteration can be calculated as the offset in the layout space of this array. The algorithm provided in this section is using these formulations to simulate the footprints of array references at each loop iteration within their reuse spaces. Following two observations give some basis as to how to simulate the reuse spaces.

OBSERVATION 1. *In order to realize the reuse carried by the innermost loop, only one cache block is needed for this array reference.*

OBSERVATION 2. *In order to realize the reuse carried by a non-innermost loop, the minimum number of cache blocks needed for this array reference is the number of cache blocks that are visited by the loops inner than it.*

Since we have assumed that all subscript functions are affine, for any array reference, the patterns of reuse space during different iterations at the loop level which has the reuse are exactly the same. Thus, we only need to simulate the first iteration of the loop having the reuse currently under exploiting. For example, loop level j in loop vector $\vec{i}$ has the reuse we are exploiting, the simulation space is defined as $SMS_j = (i_1 = l_1, i_2 = l_2, \cdots, i_j = l_j, i_{j+1}, \cdots, i_n)$, in which $i_{k>j}$ varies from its lower bound l_k to upper bound u_k.

Algorithm 3 (shown in Figure 6) first calls Algorithms 1 and 2. Then, it simulates the footprints of the most significant reuse space for an array in a particular loop nest. These footprints are marked with a array bitmap.

4.5 Computation and Optimization of Cache Configurations for Loop Nests

```
INPUT: an array node, a loop-nest node
        a given cache block size: BK_SZ
OUTPUT: an array-level bitmap for footprints
Begin
  Initial array size AR_SZ in number of cache blocks
  Allocate an array-level bitmap ABM with size AR_SZ
    and initial ABM to zeros
  Initial the highest reuse level $RS\_LEV = n$
  //n is the depth of loop nest
  For each uniform reference set
    Call Algorithm 1 for self-reuse analysis
    Call Algorithm 2 for group-reuse analysis
    Set $URS\_LEV$ = highest reuse level of this set
    If $RS\_LEV > URS\_LEV$
      Set $RS\_LEV = URS\_LEV$
  If $RS\_LEV == n$
    For all references of this array
      Set $\vec{i} = \vec{l}$//only use the lower bound
      apply equation 3 to get the reference address $f(\vec{i})$
      transfer to block id: $bk\_id = f(\vec{i})/BK\_SZ$
      set array bitmap: $ABM[bk\_id] = VISITED$
  Else
    For all loop indexes $i_j$, $j > RS\_LEV$
      varies the value of $i_j$ from lower bound to upper bound
      For all references of this array
        apply equation 3 to get the reference address $f(\vec{i})$
        transfer to block id: $bk\_id = f(\vec{i})/BK\_SZ$
        set array bitmap: $ABM[bk\_id] = VISITED$
End.
```

Figure 6: Algorithm 3: Simulation of Footprints in Reuse Spaces.

```
INPUT: loop-nest node
        global list of arrays declared
        lower bound of block size: Bk_SZ_LB
        upper bound of block size: Bk_SZ_UB
OUTPUT: optimal cache configurations at diff. BK_SZ
Begin
  Set BK_SZ = BK_SZ_LB (lower bound)
  Do
    For each array in this loop nest
      Call algorithm 3 to get the array bitmap ABM
    create and initial a loop-nest level bitmap LBM,
      with the size is the smallest $2^n$ that is $\geq$
      the size of the largest array (in block): LBM_size
    For each array bitmap ABM
      map ABM into the loop-nest bitmap LBM
        with the relative base-address of array: base_addr
        to indicate the degree of conflict at each block
      For $block\_id < array\_size$
        $LBM[(block\_id + base\_addr)\%LBM\_size] +=$
            $ABM[block\_id]$
    set assoc = the largest degree of conflict in LBM
    set $cache\_sz = assoc * LBM\_size$
    set optimal cache conf. to current cache conf.
    For assoc < assoc upper bound
      half the number of sets of current cache by
        $LBM\_size /= 2$
        For $i \leq LBM\_size$
          $LBM[i] += LBM[i + LBM\_size]$
      set assoc = highest value of $LBM[i], i \leq LBM\_size$
      set $cache\_size = assoc * LBM\_size$
      If assoc < assoc upper bound
          and cache_size < optimal cache size
        set optimal cache conf. to current cache conf.
    give out optimal cache conf. at BK_SZ
    doubling $BK\_SZ *= 2$
  while $BK\_SZ > BK\_SZ\_UB$ (upper bound)
End.
```

Figure 7: Algorithm 4: Compute and Optimize Cache Configurations for Loop Nests.

In previous subsections, the reuse spaces of each array in a particular loop nest have been determined and their footprints have also been simulated in the layout space of each array. Each array has a bitmap indicating the cache blocks which have been visited by the iterations in reuse spaces after applying Algorithm 3. As we discussed earlier, the phenomena of cache interferences can disturb these reuses and prevent the array references from realizing data localities across loop iterations. Thus, an algorithm that can reduce these cache interferences and result in better data localities within the reuse spaces is crucial.

In this subsection, we provide a loop-nest level algorithm to explicitly figure out and display the cache interferences among different arrays accessed within a loop nest. The main point of this approach is to map the reuse space of each array into the real memory space. At the same time, the degree of conflict (number of interferences among different arrays) at each cache block is stored in a loop-nest level bitmap. Since the self-interference of each array is already solved by Algorithm 3 using an array bitmap, this algorithm mainly focuses on reducing the group-interference that might occur among different arrays. As is well-known, one of the most effective way to avoid interferences is to increase the associativity of data cache, which is used in this algorithm. Based on the definition of near-optimal cache configuration, this algorithm tries to find the smallest data cache with smallest associativity that achieves significantly reduced cache interferences and nearly perfect performance of the loop nest. Figure 7 shows the detailed algorithm (Algorithm 4) that computes and optimizes the cache configuration.

For a given loop nest, Algorithm 4 starts with the cache block size (BK_SZ) from its lower bound, e.g., 16 bytes and goes up to its upper bound, e.g., 64 bytes. At each particular BK_SZ, it first applies Algorithm 3 to obtain the array bitmap ABM of each array. Then it allocates a loop-nest level bitmap LBM for all arrays within this nest, whose size is the smallest value in power of 2 that is greater or equal to the largest array size. All ABMs are remapped to this LBM with their relative array base addresses. The value of each bits in LBM indicates the conflict at a particular cache block. Following this, the optimization is carried out by halving the size of LBM and remapping LBM. The largest value of bits in LBM also shows the smallest cache associativity needed to avoid the interference in the corresponding cache block. This process is ended when the upper bound of associavitity is met. A near-optimal cache configuration at block size BK_SZ is computed as the one which has smallest cache size as well as the smallest associativity.

4.6 Global Level Cache Polymorphism

The compiler-directed cache polymorphism technique does not make changes to the source code. Instead, it uses compiler only for source code parsing and generates internal code with the intermediate format which is local to our algorithms. A global or program level algorithm, Algorithm 5 (in Figure 8) is presented in this subsection to obtain the directions (cache configurations for each nest of a program) of the cache reconfiguration mechanisms.

This algorithm first generates the intermediate format of the original code and collects the global information of arrays in source code. After that, it applies Algorithm 4 to each of its loop nests and obtains the near-optimal cache configurations for each of them. These configurations are stored in the cache-configuration list (CCL). Each loop nest has a corresponding

```
INPUT: source code(.spd)
OUTPUT: Performance data and its cache configurations
          for each loop nest
Begin
  Initial cache-configuration list: CCL
  Use one SUIF pass to generate the intermediate code format
  Construct a global list of arrays declared with its
    relative base address
  For each loop nest
    For each array in this loop nest
      Construct uniform reference sets for all its references
    Call algorithm 4 to optimize the cache configurations
      for this loop nest
    store the configurations to the CCL
  For each block size
    activate reconfiguration mechanisms with each loop nest
      using its configuration from the CCL
    Output performance data as well as the cache configuration
      of each loop nest
End.
```

Figure 8: Algorithm 5: Global Level Cache Polymorphism.

```
#define N 8
int a[N][N][N], b[N][N][N];
intN1 = 4, N2 = 4, N3 = 4, N4 = 4;
main()
{
  int i, j, k, l;

  for(i = 0; i <= N1; i++)
    for(j = 0; j <= N2; j++)
      for(k = 0; k <= N3; k++)
        for(l = 0; l <= N4; l++)
          {
            a[i+k][j+2][l] = a[i+k][j][l];
            b[j][k+l][i] = a[2*i][k][l];
          }
}
```

Figure 9: An Example: Array-based Code.

node in CCL which has its near-optimal cache configurations at different block sizes. After the nest-level optimization is done, Algorithm 5 activates the cache reconfiguration mechanisms, in which a modified version of the Shade simulator is used. During the simulation, Shade is directed to use the near-optimal cache configurations in CCL for each loop nest before its execution. The performance data of each loop nest under different cache configurations is generated as output.

Since current cache reconfiguration mechanisms can only vary cache size and cache ways with fixed cache block size, the cache optimization is done for different (fixed) cache block sizes. This means that the algorithms in this paper suggest a near-optimal cache configuration for each loop nest for a given block size. In the following section, experimental results verifying the effectiveness of this technique are presented.

4.7 An Example

In this subsection, we focus on the example code in Figure 9 to illustrate how the compiler-directed cache polymorphism technique works. For simplicity, this code only contains one nest.

Algorithm 5 starts with one SUIF pass to convert the above source code into intermediate code, in which the program node only has one loop-nest node. The loop-nest node is represented by its index vector $\vec{i} = (i, j, k, l)^T$, with an index lower bound vector of $\overrightarrow{il} = (0, 0, 0, 0)^T$, an upper bound vector of $\overrightarrow{iu} = (N_1, N_2, N_3, N_4)^T$ and a stride vector of $\overrightarrow{is} = (1, 1, 1, 1)^T$. Within the nest, arrays a and b have references AR_{a^1}, AR_{a^2}, AR_{a^3} and AR_b, which are represented in access matrices and constant vectors as follows:

$$A_{a^1} : \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \overrightarrow{c_{a^1}} : \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix},$$

$$A_{a^2} : \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \overrightarrow{c_{a^2}} : \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

$$A_{a^3} : \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \overrightarrow{c_{a^3}} : \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

$$A_b : \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \overrightarrow{c_b} : \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix},$$

Also, a global array list is generated as $< a, b >$. Then, for array a, references AR_{a^1} and AR_{a^2} are grouped into one uniform reference set, and AR_{a^3} is put to another one. Array b, on the other hand, has only one uniform reference set.

Then, Algorithm 4 is invoked and starts from the smallest cache block size, BK_SZ, say 16 bytes. It uses Algorithm 3 to obtain the array bitmap ABM_a for array a and ABM_b for array b at BK_SZ. Within Algorithm 3, we first call Algorithm 1 and Algorithm 2 to analyze the reuse characteristics of a given array. In our example, the first uniform set of array a has self-spatial reuse at level l, group-temporal reuse at level j, the second uniform set has self-spatial reuse at level l and self-temporal reuse at level j. Reference of array b has self-spatial reuse at level i. The highest level of reuse is then used for each array by Algorithm 3 to generate the ABM for its footprints in the reuse space. We assume an integer has 4 bytes in size. In this case, both ABM_a and ABM_b have 128 bits shown as follows:

ABM_a:

0-31	1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
32-63	1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
64-95	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
96-127	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

ABM_b:

0-31	1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
32-63	1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
64-95	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
96-127	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

These two ABMs are then passed by Algorithm 3 to Algorithm 4. In turn, Algorithm 4 creates a loop-nest bitmap LBM with size being equal to the largest array size, $MAX($ $ABMs)$, and re-maps ABM_a and ABM_b to LBM. Since array a has relative base address at 0 (byte), and array b at 2048, we determine LBM as follows:

0-31	2 0 2 0 2 0 2 0 1 0 1 0 1 0 1 0 2 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0
32-63	2 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0 2 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0
64-95	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
96-127	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Name	Arrays	Nests	Brief Description
adi.c	6	2	Alternate Direction Integral
aps.c	17	3	Mesoscale Hydro Model
bmcm.c	11	3	Molecular Dynamic of Water
eflux.c	5	6	Mesh Computation
tomcat.c	9	8	Mesh Generation
tsf.c	1	4	Array-based Computation
vpenta.c	9	8	Nasa Ames Fortran Kernel
wss.c	10	7	Molecular Dynamics of Water

Table 1: The Array-based Benchmarks Used in the Experiments.

The maximum value of bits in LBM indicates the number of interference among different arrays in the nest. Thus, it is the least associativity that is required to avoid this interference. In this example, Algorithm 4 starts from a cache associativity of 2 to compute the near-optimal cache configuration. Each time, the size of LBM is halved and the LBM is re-mapped until the resulting associativity reaches the upper bound, e.g., 16. Then it outputs the smallest cache size with smallest associativity as the near-optimal configuration at this block size BK_SZ. For this example, the near-optimal cache configuration is 2KB 2-way associative cache at 16 byte block size. The LBM after optimization is shown as follows:

0-31	2 0 2 0 2 0 2 0 1 0 1 0 1 0 1 0 2 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0
32-63	2 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0 2 0 1 0 2 0 1 0 1 0 1 0 1 0 1 0

Following this, Algorithm 4 continues to compute the near-optimal cache configurations for larger cache block sizes by doubling the previous block size. When the block size reaches its upper bound, e.g., 64 bytes, this algorithm stops to pass all the near-optimal configurations at different block sizes to Algorithm 5. On receiving these configurations, Algorithm 5 activates Shade to simulate the example code (executable) with these cache configurations. Then the performance data is generated as the output of Algorithm 5.

5. EXPERIMENTS

5.1 Simulation Framework

In this section, we present our simulation results to verify the effectiveness of the CDCP technique. Our technique has been implemented using SUIF [5] compiler and Shade [2]. Eight array-based benchmarks are used in this simulation work. In each benchmark, loop nests dominate the overall execution time. Our benchmarks, the number of arrays (for each benchmark) and the number of loop nests (for each benchmark) are listed in Table 1.

Our first objective here is to see the cache configurations returned by our CDCP scheme and a scheme based on exhaustive simulation (using Shade). We consider three different block (line) sizes: 16, 32 and 64 bytes. Note that our work is particularly targeted at L1 on-chip caches.

5.2 Selected Cache Configurations

In this subsection, we first apply an exhaustive simulation method using the Shade simulator. For this method, the original program codes are divided into a set of small programs, each program having a single nest. Shade simulates these loop nests individually with all possible L1 data cache configurations within the following ranges: cache sizes from 1K to 128K, set-associativity from 1 way to 16 ways, and block size at 16, 32 and 64 bytes. The number of data cache misses is used as the metric for comparing performance. The optimal cache configuration at a certain cache block size is the smallest one in terms of both cache size and set associativity that achieves a performance (the number of misses) which cannot be further improved (the number of misses cannot be reduced by 1%) by increasing cache size and/or set associativities. The left portion of Table 2 shows the optimal cache configurations (as selected by Shade) for each loop nest in different benchmarks as well as at different cache block sizes.

The compiler-directed cache polymorphism technique directly takes the original source code in the SUIF .spd format and applies Algorithm 5 to generate the near-optimal cache configurations for each loop nest in the source code. It does not do any instruction simulation for configuration optimization. Thus, it is expected to be very fast in finding the near-optimal cache configuration. The execution engine (a modified version of Shade) of CDCP directly applies these cache configurations to activate the reconfiguration mechanisms dynamically. The cache configurations determined by CDCP are shown on the right part of Table 2. To sum up, in Table 2, for each loop nest in a given benchmark, the optimal cache configurations from Shade and near-optimal cache configurations from CDCP technique at block sizes 16, 32, and 64 bytes are given. A notation such as 8k4s is used to indicate a 8K bytes 4-way set associative cache with a block size of 32 bytes. In this table, B means bytes, K denotes kilobytes and M indicates megabytes.

From Table 2, we can observe that CDCP has the ability to determine cache capacities at byte granularity. In most cases, the cache configuration determined by CDCP is less than or equal to the one determined by the exhaustive simulation.

5.3 Simulation Results

The two sets of cache configurations for each loop nests given in Table 2 are both simulated at the program level. All configurations from CDCP with cache size less than 1K are simulated at 1K cache size with other parameters unmodified. For best comparison, the performance is shown as the cache hit rate instead of the miss rate. Figure 10 gives the performance comparison between Shade (exhaustive simulation) and CDCP using a block size of 16 bytes.

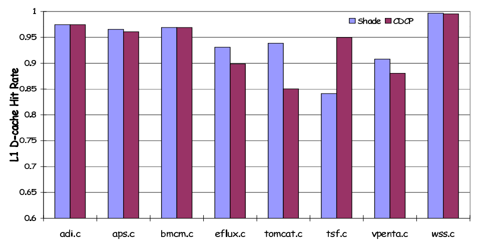

Figure 10: Performance Comparison of Cache Configurations at Block Size of 16: Shade Vs CDCP.

We see from Figure 10 that, for benchmarks *adi.c*, *aps.c*, *bmcm.c* and *wss.c*, the results obtained from Shade and CDCP are very close. On the other hand, Shade outperforms CDCP in benchmarks *eflux.c*, *tomcat.c* and *vpenta.c*, and CDCP

Codes	Shade			CDCP		
adi	16	32	64	16	32	64
1	1k4s	1k4s	1k4s	64B4s	128B4s	256B4s
2	16k16s	16k16s	16k16s	16k16s	16k16s	16k16s
aps	16	32	64	16	32	64
1	2k4s	4k8s	64k4s	2k8s	4k4s	8k8s
2	16k8s	16k16s	32k16s	16k4s	16k8s	32k8s
3	4k2s	4k8s	8k8s	2k16s	4k8s	8k8s
bmcm	16	32	64	16	32	64
1	1k8s	2k8s	4k8s	64B1s	128B1s	256B1s
2	1k8s	2k8s	4k8s	64B2s	128B4s	256B1s
3	32k4s	64k4s	128k4s	32k4s	64k4s	128k4s
eflux	16	32	64	16	32	64
1	16k4s	32k4s	64k4s	2k8s	4k4s	8k8s
2	16k8s	32k4s	64k4s	8k4s	16k2s	32k4s
3	128k16s	128k16s	128k1s	128k8s	256k2s	256k2s
4	2k8s	2k8s	4k8s	128B4s	256B2s	256B4s
5	16k16s	32k4s	64k4s	8k16s	16k8s	32k4s
6	128k16s	128k16s	128k1s	128k8s	256k2s	256k2s
tomcat	16	32	64	16	32	64
1	1k2s	1k1s	1k1s	32B2s	64B2s	128B1s
2	1k1s	1k1s	1k1s	32B1s	64B1s	128B2s
3	128k4s	128k8s	128k1s	64k1s	128k2s	256k2s
4	1k2s	1k4s	2k8s	32B2s	64B2s	128B1s
5	64k8s	128k8s	128k2s	64k1s	128k2s	256k2s
6	1k2s	1k4s	2k4s	64B4s	128B4s	256B2s
7	64k4s	128k8s	128k8s	32k4s	64k8s	128k16s
8	32k1s	128k2s	128k4s	32k1s	64k2s	128k4s
tsf	16	32	64	16	32	64
1	4k4s	8k1s	8k1s	4k1s	4k1s	4k1s
2	128k16s	128k16s	128k16s	1M1s	1M1s	1M1s
3	4k4s	4k16s	8k4s	4k1s	4k1s	4k1s
4	128k16s	128k16s	128k16s	1M1s	1M1s	1M1s
vpenta	16	32	64	16	32	64
1	64k1s	128k1s	128k16s	64k1s	128k1s	256k8s
2	1k8s	2k4s	2k8s	128B8s	256B8s	512B8s
3	1k4s	2k2s	2k8s	256B4s	512B2s	1k2s
4	128k8s	128k16s	128k16s	128k2s	256k8s	512k2s
5	1k4s	2k4s	4k2s	256B4s	512B2s	1k2s
6	1k2s	2k2s	2k8s	128B8s	256B4s	512B8s
7	1k2s	1k2s	1k16s	64B1s	128B2s	256B4s
8	64k8s	128k2s	128k1s	64k1s	128k1s	256k1s
wss	16	32	64	16	32	64
1	4k4s	8k4s	8k16s	2k2s	4k4s	8k8s
2	1k8s	2k8s	4k4s	64B4s	128B4s	256B4s
3	1k2s	1k2s	1k2s	64B2s	128B4s	256b4s
4	64k4s	64k4s	64k4s	64k2s	64k2s	64k2s
5	4k4s	8k8s	16k8s	2k4s	4k4s	8k8s
6	1k2s	1k2s	1k2s	32B2s	64B1s	128B2s
7	2k8s	4k4s	4k4s	64B4s	128B1s	256B2s

Table 2: Cache Configurations for each Loop Nest in Benchmarks: Shade Vs CDCP.

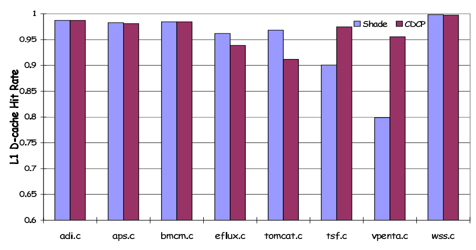

Figure 11: Performance Comparison of Cache Configurations at Block Size of 32: Shade Vs CDCP.

outperforms Shade in *tsf.c*. Figures 11 and 12 show the results with block sizes of 32 and 64 bytes, separately.

We note that, for most benchmarks, the performance difference between Shade and CDCP decreases as the block size is increased to 32 and 64 bytes. Especially for benchmarks *adi.c*, *aps.c*, *bmcm.c* and *wss.c*, the performances from the two approaches are almost the same. For other benchmarks such as *tsf.c* and *vpenta.c*, our CDCP strategy consistently outperforms Shade when block size is 32 or 64 bytes. This is because the exhaustive Shade simulation has a searching range (for cache sizes) from 1K to 128K as explained earlier, while CDCP has no such constraints (that is, it can come up with a non-standard cache size too). Obviously, we can use much larger and/or much finer granular cache size for exhaustive simulation. But, this would drastically increase the simulation time, and is not suitable for practice. In contrast, the CDCP strategy can determine any near-optimal cache configuration without much increase in search time.

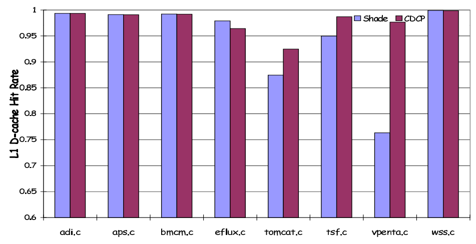

Figure 12: Performance Comparison of Cache Configurations at Block Size of 64: Shade Vs CDCP.

For more detailed study, we break down the performance comparison at loop nest level for benchmark *aps.c*. Figure 13 shows the comparison for each loop nest of this benchmark at different cache block sizes.

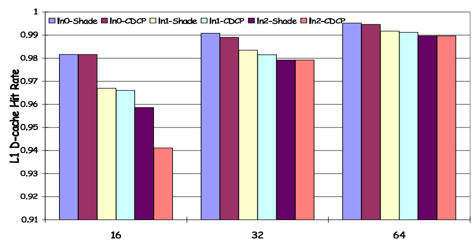

Figure 13: Loop-nest Level Performance Comparison of Cache Configurations for asp.c: Shade Vs CDCP.

The results from the loop nest level comparison show that the CDCP technique is very effective in finding the near-optimal cache configurations for loop nests in this benchmark, especially at block sizes of 32 and 64 bytes (the most common block sizes used in embedded processors). Since CDCP is analysis-based not simulation-based, we can expect that it will be even more desirable in codes with large input sizes.

From energy perspective, the Cacti power model [10] is used to compute the energy consumption in L1 data cache for each loop nest of our benchmarks at different cache configurations listed in Table 2. We use 0.18 micron technology for all the cache configurations. The detailed energy consumption figures are given in Table 3.

Codes	Shade			CDCP		
adi	16	32	64	16	32	64
1	318.6	287.4	-[1]	318.6	287.4	-
2	12154.4	13164.5	16753.6	12154.4	13164.5	16753.6
aps	16	32	64	16	32	64
1	322.3	771.7	540.1	661.2	335.4	822.0
2	125599.5	279985.9	368764.9	65461.7	122847.2	145962.2
3	7907.7	33273.5	34697.7	64275.4	33273.5	34697.7
bmcm	16	32	64	16	32	64
1	314.6	342.9	393.4	31.7	30.5	31.1
2	314.6	342.9	393.4	83.0	155.2	31.1
3	26826.7	32203.8	36989.1	26826.7	32203.8	36989.1
eflux	16	32	64	16	32	64
1	366.7	386.4	433.3	648.4	320.1	776.6
2	1068.8	610.3	700.1	534.8	301.7	598.5
3	2366.1	2435.0	329.4	1220.7	727.5	749.6
4	310.2	321.7	370.7	146.0	77.0	-
5	2326.5	636.5	731.2	2399.6	1121.7	624.5
6	2573.0	2666.1	375.0	1323.3	795.5	821.3
tomcat	16	32	64	16	32	64
1	895.0	280.4	260.0	895.0	748.4	260.0
2	28.4	27.5	28.1	28.4	27.5	74.3
3	66507.5	117366.5	40582.4	26846.9	40767.0	83199.2
4	78.1	147.5	-	78.1	77.1	29.5
5	25678.1	27508.1	14394.7	9448.6	14978.6	25989.1
6	80.8	152.7	167.6	152.8	152.7	86.5
7	9461.3	18865.2	25190.9	9647.7	21984.0	57050.0
8	2051.1	5050.0	8406.6	2051.1	4046.2	8406.6
tsf	16	32	64	16	32	64
1	160.9	38.5	41.4	34.7	34.7	35.9
2	18858.6	18245.8	19320.4	6263.6	9501.5	14293.2
3	163.5	787.9	173.9	35.2	35.2	42.5
4	18769.0	18159.7	19230.0	6234.3	9452.6	14226.7
vpenta	16	32	64	16	32	64
1	4111.6	5130.1	87386.9	4111.6	5130.1	22364.9
2	350.7	184.6	-	350.7	-	-
3	189.4	102.3	-	189.4	97.7	98.6
4	77075.1	235412.6	268609.7	27835.4	90080.5	100849.2
5	188.4	216.9	108.7	188.4	97.4	98.3
6	99.1	101.7	-	-	185.8	-
7	90.2	89.0	-	32.7	89.0	-
8	36158.0	13557.1	15249.6	8994.2	12456.5	21512.0
wss	16	32	64	16	32	64
1	268.8	279.9	1610.6	138.6	261.1	624.0
2	288.5	317.1	168.1	143.9	143.6	-
3	75.1	74.1	74.9	75.1	141.8	-
4	22641.6	23665.1	22935.2	13274.4	13051.5	14560.2
5	326.7	672.8	775.3	325.7	319.6	756.3
6	74.8	73.8	74.6	74.8	27.6	74.6
7	302.8	155.6	166.6	142.4	27.9	75.1

Table 3: Energy Consumption (microjoules) of L1 Data Cache for each Loop Nest in Benchmarks with Configurations in Table 2: Shade Vs CDCP.

From our experimental results, we can conclude that (i) our strategy generates competitive performance results with exhaustive simulation, and (ii) in general it results in a much lower power consumption than a configuration selected by exhaustive simulation. Consequently, our approach strikes a balance between performance and power consumption.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new technique, compiler-directed cache polymorphism, for optimizing data locality of array-based embedded applications while keeping the energy consumption under control. In contrast to many previous techniques that modify a given code for a fixed cache architecture, our technique is based on modifying (reconfiguring) the cache architecture dynamically between loop nests. We presented a set of algorithms that (collectively) allow us to select a near-optimal cache configuration for each nest of a given application. Our experimental results obtained using a set of array-intensive applications reveal that our approach generates competitive performance results and consumes much less energy (when compared to an exhaustive simulation based framework). We plan to extend this work in several directions. First, we would like to perform experiments with different sets of applications. Second, we intend to use cache polymorphism at granularities smaller than loop nests. And finally, we would like to combine CDCP with loop/data based compiler optimizations to optimize both hardware and software in a coordinated manner.

[1]Energy estimation is not available from Cacti due to the very small cache configuration.

7. REFERENCES

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proc. of the 32nd Micro*, 1999.

[2] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proc. of the 1994 ACM SIGMETRICES Conf. on the Measurement and Modeling of Computer Systems*, May 1994.

[3] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[4] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proc. of ICS'97*.

[5] Stanford Compiler Group. *The SUIF Library, version 1.0 edition.* 1994.

[6] X. Ji, D. Nicolaescu, A. Veidenbaum, A. Nicolau, and R. Gupta. Compiler-directed cache assist adaptivity. Technical Report ICS-TR-00-17, ICS Department, University of California-Irvine, June 2000.

[7] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and J. Ramanujam. Morphable cache architectures: potential benefits. In *ACM Workshop on LCTES'01*, June 2001.

[8] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Lanaguages and Systems*, 18(4):424–453, July 1996.

[9] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *Proc. of the 27th ISCA*, June 2000.

[10] G. Reinman and N. Jouppi. An integrated cache timing and power model. Cacti 2.0 technical report, COMPAQ Western Research Lab, 1999.

[11] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proc. of ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, 1994.

[12] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. of PLDI'91*, pages 30–44, 1991.