Incremental Hierarchical Memory Size Estimation for Steering of Loop Transformations

Q. Hu and P. G. Kjeldsberg
Norwegian University of Science and Technology
and
A. Vandecappelle, M. Palkovic and F. Catthoor
IMEC

Modern embedded multi-media and telecommunications systems need to store and access huge amounts of data. This becomes a critical factor for the overall energy consumption, area and performance of the systems. Loop transformations are essential to improve the data access locality and regularity in order to optimally design or utilize a memory hierarchy. However, due to abstract high level cost functions, current loop transformation steering techniques do not take the memory platform sufficiently into account. They usually also result in only one final transformation solution. On the other hand, the loop transformation search space for real-life applications is huge, especially if the memory platform is still not fully fixed. Use of existing loop transformation techniques will therefore typically lead to sub-optimal end-products. It is critical to find all interesting loop transformation instances. This can only be achieved by performing an evaluation of the effect of later design stages at the early loop transformation stage.

This paper presents a fast incremental hierarchical memory size requirement estimation technique. It estimates the influence of any given sequence of loop transformation instances on the mapping of application data onto a hierarchical memory platform. As the exact memory platform instantiation is often not yet defined at this high level design stage, a platform independent estimation is introduced with a Pareto curve output for each loop transformation instance. Comparison among the Pareto curves helps the designer, or a steering tool, to find all interesting loop transformation instances that might later lead to low power data mapping for any of the many possible memory hierarchy instances. Initially the source code is used as input for estimation. However, performing the estimation repeatedly from the source code is too slow for the large search space exploration. An incremental approach, based on local updating of the previous result, is therefore used to handle sequences of different loop transformations. Experiments show that the initial approach takes a few seconds, which is two orders of magnitude faster than state-of-the-art solutions but still too costly to be performed interactively many times. The incremental approach typically takes just a few milliseconds, which is another two orders of magnitude faster than the initial approach. This huge speedup allows us for the first time to handle real-life industrial size applications and get realistic feedback during loop transformation exploration.

Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Memory Design*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; *Optimization*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Optimization*

General Terms: Algorithms, Design, Performance

Author's address: (Qubo Hu, Per Gunnar Kjeldsberg), Norwegian University of Science and Technology, Trondheim, Norway; email: {qubo.hu, pgk}@iet.ntnu.no; (Martin Palkovic, Arnout Vandecappelle, Francky Catthoor), IMEC DESICS, Leuven, Belgium; email: {palkovic, vdcappel, catthoor}@imec.be

Additional Key Words and Phrases: data optimization, memory size estimation, code transformation, memory architecture exploration, high-level synthesis

1. INTRODUCTION

In modern advanced real-time multimedia and telecommunication applications, the manipulation of large array-based data sets has a major effect on both the energy consumption and performance of the system. This is due to the huge amounts of data transfers and storage to/from large, energy consuming off-chip data memories. To alleviate the impact of these memory accesses, high-level memory exploration and optimization techniques have been proposed to transform the application and also to optimally utilize the memory hierarchy [Catthoor et al. 1998]. An important step in these optimization techniques is the improvement of data accesses locality and regularity through loop transformations [Banerjee 1993] [Wolf and Lam 1991] [Kelly and Pugh 1993] [McKinley et al. 1996] [Darte and Robert 1995] [Danckaert et al. 2000] [Verdoolaege et al. 2003] [Song et al. 2004] [Girbal et al. 2006]. Figure 1(a) shows a small code example with two array accessing statements inside two loop nests. When the two loop nests are fused as shown in Figure 1(b), the accesses to the same array elements are getting closer and data access locality is improved. Loop interchange can further be performed on the two outer-most loop dimensions resulting in the code shown in Figure 1(c).

```
for (i=0; i<=109; i++)
  for (j=0; j<=69; j++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++)
S1:       ... = f(A[40i+k][20j+l]);
for (i=1; i<=109; i++)
  for (j=1; j<=69; j++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++)
S2:       ... = g(A[40i+k-40][20j+l-20]);
```

(a)

```
for (i=0; i<=109; i++)
  for (j=0; j<=69; j++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++) {
S1:       ... = f(A[40i+k][20j+l]);
        if (i>0 and j>0)
S2:       ... = g(A[40i+k-40][20j+l-20]);
      }
```

(b)

```
for (j=0; j<=69; j++)
  for (i=0; i<=109; i++)
    for (k=0; k<=59; k++)
      for (l=0; l<=29; l++) {
S1:       ... = f(A[40i+k][20j+l]);
        if (i>0 and j>0)
S2:       ... = g(A[40i+k-40][20j+l-20]);
      }
```

(c)

1: Code examples: (a) original code, (b) after loop fusion, (c) after loop interchange

Loop transformations are usually performed at an early system level design stage, where the data memory platform, e.g., the number of layers and the exact size of the memories in each layer, is often not known yet. Improved data locality and regularity usually bring repeated accesses of the same data closer together in time. The data elements will then typically be in a smaller and faster memory located near the processor. Improving locality

also shortens the data lifetime, freeing up memory for other data and typically reducing the memory requirement for an application-specific processor realization [Verbauwhede et al. 1989]. Thus, by exploiting the memory hierarchy, power and performance savings can be obtained by accessing heavily used off-chip data from smaller on-chip memories. Such optimizations either have to rely on hardware cache controllers which copy relevant data into the cache based on quite local criteria, or they rely on scratch-pad memories (SPMs), also called software-controlled memories. For the data dominated applications in our target domains, SPM is highly preferred over cache due to the stringent low-power and real-time requirements of embedded systems. Caches incur a significant penalty in area, energy, hit latency and real-time guarantees. SPM does not need any extra hardware. It instead requires source code transformations that exploit on-chip memory layers to which frequently used data will be stored or copied. Specifically, copies of data will be introduced from larger off-chip memories to smaller on-chip memories. As most of the data access patterns in our application domain are known at compile time, a global view can be taken during code optimization and data mapping [Panda et al. 1997] [Steinke et al. 2002]. Memory hierarchy layers can thus contain normal SPMs and/or caches. An application has to be mapped efficiently on this memory hierarchy and different techniques [Brockmeyer et al. 2003] [Kandemir and Choudhary 2002] [Benini et al. 2000] have been presented. Many embedded processors have already integrated SPM(s), some even leaving out the cache.

However, improving data locality and regularity are very abstract cost functions. They do not represent how the data will be mapped onto the memory platform at later design stages. On the other hand, there usually exist a huge number of (combined) loop transformation possibilities for real applications with multiple loop nests (up to dozens). [Darte 2000] has proven that even performing loop fusion is an NP-complete problem. The task is even more complex when various other loop transformation techniques are considered at the same time, e.g., loop interchange, loop reverse, loop skewing, loop shifting, etc. Different loop transformations may result in optimal utilization for different memory platform instances as will be demonstrated later in this paper. Ad-hoc loop transformation decisions without estimating their impact on the actual hierarchy utilization usually lead to final sub-optimal solutions. It is hence crucial to perform estimation of memory mapping during the early loop transformation exploration in order to find all interesting (including intermediate) loop transformation instances. The state-of-the-art loop transformation algorithms all result in a single final loop transformation solution. Their solution may be optimal for certain memory hierarchy instances, but typically not for all. As mentioned, the data memory platform is typically not defined at the early design stage. The estimation must hence be platform independent. At the same time it is not enough simply to estimate the actual size of a given array and of the given application, since the minimal size may still be too large to fit on-chip. In addition, if sufficient locality between read accesses is present, a local copy of part of the array to on-chip memory may already remove most of the off-chip accesses [Wuytack et al. 1998], making the actual size of that array less relevant. To select all the interesting loop transformation alternatives, it is therefore critical to identify the frequently accessed data and estimate their mapping on the hierarchical memory platform. Later, when the details of the memory platform is decided, the set of transformation solutions can help the designer or a steering tool to find the optimal version of code while trading off memory size and power (i.e., off-chip accesses).

This paper presents a fast Hierarchical Memory Size Estimation (HMSE) methodology. We use a multi-layer memory hierarchy template with a main memory that is large enough to store all required data and (typically on-chip) SPM layers that we can instantiate to any size we choose. This paper discusses a two-layer memory hierarchy, but the technique presented here is general and also applicable to deeper hierarchies. Based on this template, we evaluate a sequence of loop transformation instances and select the ones which will result in low power design for at least one of many possible low power memory platform instances. As the name indicates, the HMSE methodology performs estimation and can as such be used together with any state-of-the-art loop transformation algorithm. The loop transformation algorithm decides the sequence of loop transformations, which can then be evaluated by our estimation technique. It helps the designer, or a steering tool, by providing accurate and very fast feedback during the loop transformation exploration. Moreover, the complete Pareto curve of potentially good code versions are generated as explained below. HMSE estimates the data reuse analysis (DRA) and memory hierarchy layer assignment (MHLA) performed at later design stages. DRA identifies the frequently reused data at each loop dimension and MHLA decides how to map all the data to the different layers [Catthoor et al. 1998].

We have previously presented an early version of our technique [Hu et al. 2004] [Hu et al. 2006]. This was mainly limited to the so-called initial HMSE in which the instantiated transformed source code is used as input. Often, however, a series of loop transformations are performed incrementally, where the outcome of one transformation is used as starting point for the next transformation. The effect of the transformation is then typically limited to a certain code region [Cohen et al. 2004]. It is then highly inefficient to perform all transformations and estimations on the whole source code. In [Hu et al. 2006] we handled this in a simple manner for a restricted set of loop transformations (more specifically loop fusion and loop shifting). In this paper we present a more advanced technique for incremental HMSE that works for all affine loop transformations and strip mining, based on local updating of the effect of the transformed code region. After each transformation, HMSE generates a Pareto curve which shows a trade-off between SPM size and the number of main memory accesses. Finally, based on all the individual curves, a global Pareto curve is generated. It keeps track of all (sequences of) transformations that can potentially lead to low power data mapping for any specific memory platform selected at a later design stage.

An alternative to estimation, is to perform an exact data mapping onto a given memory hierarchy. This is done in [Brockmeyer et al. 2003], but their approach requires minutes of CPU-time for our experimental applications, and even more as applications grow larger. This is too slow to be used for exploration steering purposes. For comparison, our initial HMSE just needs up to a few seconds for our experimental applications, that is two orders of magnitude faster. However, even seconds per estimation is too slow to be performed repeatedly for the large number of transformation possibilities. Our incremental approach can require as little as a few milliseconds CPU-time, which is up to another two orders of magnitude faster than the initial approach. This speedup is essential to make it feasible to perform the estimation during the loop transformation exploration. Further, experiments have shown that the incremental approach is scalable for larger applications. Together this allows us for the first time to handle real-life industrial size applications and still get realistic feedback during system-level exploration.

The rest of this paper is organized as follows. We start with an introduction to previous work and the basics of performing loop transformations on the geometrical model (GM). This is followed by the presentation of HMSE methodology, which includes both the initial and incremental approaches. Towards the end, experiments on real-life test vehicles are presented. Finally, we draw our conclusions.

2. PREVIOUS WORK

Loop transformations have been extensively used to optimize application code for customizable architectures or for a given architecture. In particular, the vector machine, systolic array, and parallel compiler communities have used this to a very large extent for a long time, e.g. [Banerjee 1993] [Wolf and Lam 1991] [Bacon et al. 1994] [McKinley et al. 1996]. Their main goal is, however, to reveal and exploit code and data parallelism to improve performance, so that multiple instantiations of (parts of) a loop nest can be executed simultaneously. Loop transformations have also been used successfully for low power embedded system design [Kelly and Pugh 1993] [Danckaert et al. 2000] [Verdoolaege et al. 2003] [Darte and Robert 1995] [Fraboulet et al. 1999] [Song et al. 2004]. It is a crucial step within high level optimization methodologies, such as the data transfer and storage exploration (DTSE) methodology. It systematically improves the efficiency (e.g. in terms of power consumption and memory footprint) of applications, both for custom architectures (ASICs) [Catthoor et al. 1998] and programmable processor platforms with predefined memory organizations [Catthoor et al. 2002]. [Danckaert et al. 2000] [Verdoolaege et al. 2003] split the execution of affine loop transformations in three phases: a linear transformation phase for improving data regularity, a translation phase (with loop fusion and loop shifting) for improving data locality and an ordering phrase for deciding the execution order. This is one example of state-of-the-art work that can select the transformations to be performed. It can then use our estimation technique to evaluate the alternatives. [Kandemir and Choudhary 2002] propose to integrate loop transformation optimization together with the memory hierarchy optimization, as will further be discussed later in this section. Their approach performs a limited number of loop transformations and come up with one final transformation instance. [Girbal et al. 2006] presents a framework to facilitate an automatic search of a sequence of loop transformations. The HMSE is also useful in their context, and can even be integrated within their framework by providing accurate and very fast feedback to evaluate their loop transformation exploration.

Previous work on array-oriented storage requirement estimation [Verbauwhede et al. 1994] [Balasa et al. 1995] [Zhao and Malik 1999] [Grun et al. 1998] [Kjeldsberg et al. 2003] [Zhu et al. 2006] [Hu et al. 2006] all consider a single layer memory. More details about them can be found in [Hu et al. 2006]. In contrast, we perform a two layer memory hierarchy estimation where we not only estimate the storage requirement but also identify the frequently accessed data and estimate their mapping on a hierarchical memory platform.

Several research groups have studied how to perform data reuse exploration. [Wuytack et al. 1998] presented a formalized methodology for data reuse exploration. It is systematic and manually applicable but not directly implementable in a fully automated tool. [Beyls and D'Hollander 2001] presents an exact data reuse analysis for all array element accesses. It is however very computationally expensive. In [Van Achteren et al. 2002], a reuse analysis technique is presented that explores tradeoffs between SPM size and power within the

complete search space. However, it is limited to two nested loops with one array reference. In [Issenin et al. 2004], the data reuse is explored at each loop dimension both between different array accesses as well as for the same access. It generates a hierarchical set of scratch-pad buffers, any of which can be selected to be used later. However, for all these techniques the analysis is slow, which is not acceptable for use during loop transformation search space exploration.

For memory allocation, [Panda et al. 2001] gives an overview of early approaches. In addition, [Steinke et al. 2002] presents an ILP approach to optimally identify and assign data and instructions to the SPM. [Udayakumaran and Barua 2006] presents a technique for data allocation of both affine and non-affine code, but without full data reuse analysis exploration. In general, all of them focus on one layer memory assignment. [Benini et al. 2000] proposes to generate application-specific scratch-pad memories with an additional decode for distinguishing between a hit and a miss. [Brockmeyer et al. 2003] presents a backtracking algorithm to find an optimized data mapping on a given memory platform, as part of the Atomium tool [IMEC 2006] for the DTSE methodology. All previous work presented so far either focuses on one layer memory assignment or targets one specific memory hierarchy configuration. [Kandemir and Choudhary 2002] presents a somewhat different approach. Either they assume a given memory hierarchy and find good loop transformations for the current program, or they design an optimized memory hierarchy for the current program. As motivated previously, both of these goals can benefit from accurate estimation. [Nguyen et al. 2006] proposes to take advantage of compiler analysis of the data access pattern to make code portable across SPMs of any size at run-time. Their work does not target estimation and computation speed is thus not as critical as it is in our case.

3. LOOP TRANSFORMATIONS ON THE GEOMETRICAL MODEL

For the target class of data dominated applications, the high-level description is typically characterized by large multi-dimensional loop nests and arrays with mostly manifest and affine index expressions. Loop transformations are performed by reordering the loop iterations and are usually classified as affine and non-affine loop transformations. Typical affine loop transformations are loop interchange, loop reverse, loop fusion, loop shifting (also called bumping), and loop skewing. Loop unrolling, strip mining, loop coalescing and loop tiling are examples of non-affine transformations.

3.1 The geometrical model

Loop transformations are usually performed on a geometrical model (GM)[1]. In the GM, multi-dimensional spaces, or domains, are used to represent all information about data, operations, dependencies, and the order in which they are handled. This section gives a brief introduction to the GM with an emphasis on giving an intuitive understanding of the main concepts on which our techniques are based. More details can be found in [Wilde 1993].

The iteration domain (I) of a statement is a set of integer points where each point represents exactly one execution of this statement. Its description is derived from the constraints corresponding to the boundaries of the surrounding loops and conditions that restrict the

[1]Sometimes called a polyhedral model. However, since we approximate the domains to bounding boxes, it would not be an appropriate name in our case.

execution of the statement. Each node within the iteration domain can be identified by its loop iterator $\vec{i} = [i_1, i_2, \ldots, i_m]^T$, where i_g is the gth iterator in the loop nest, counting from the outermost to innermost dimension. We will illustrate the concepts in this section based on the example code in Figure 1. Sometimes, an extra time dimension is also inserted before each loop dimension in order to identify the exact execution ordering between multiple statements within the same loop nest and between multiple loop nests. The time dimensions are left out in the example to keep the illustration simple.

In this work, a simplified GM is used in which all iteration domains are extended to bounding boxes. A bounding box iteration domain is a rectangular approximation of the original iteration domain which can have any convex shape. The bounding box domain can easily be defined by the lower bound (L) and upper bound (U) for each dimension. This is an efficient and appropriate approximation since most convex shapes in the targeted application domain are rectangular and have regular accesses, i.e., images, blocks, etc. Using the bounding boxes we can define the iteration domain for a statement S as

$$I_S = \{\vec{i} = [i_1, \ldots, i_m]^T \mid \bigwedge_{g=1}^{m} L_g \leq i_g \leq U_g\} \tag{1}$$

where the comparison is applied componentwise and the gth elements of L_g and U_g are the lower bound and upper bound values at the gth dimension, respectively. For the statement S_1 in Figure 1(a), the bounding box iteration domain is

$$I_{S_1} = \{[i, j, k, l]^T | 0 \leq i \leq 109 \wedge 0 \leq j \leq 69 \wedge 0 \leq k \leq 59 \wedge 0 \leq l \leq 29\}$$

For this example, the bounding box represents the exact shape of the original iteration domain.

Each statement has a number of accesses to array variables. For the analysis, it is important to identify which data that is accessed by a given statement. Therefore, each access Y has an associated index function F_Y. It maps the iterators of the statement to the accessed index

$$F_Y = \{[i_1, \ldots, i_m]^T \mapsto [d_1, \ldots, d_n]^T \mid \bigwedge_{h=1}^{n} d_h = \sum_{g=1}^{m} c_g^h i_g + c_{m+1}^h\} \tag{2}$$

where d_h is the hth dimension of the array. c_g^h is the index function coefficient for the gth loop iterator of the hth array dimension, c_{m+1}^h is the constant offset of the hth array dimension.

The set of all indices encountered by an access over all iterations of the surrounding loops is called the data domain D_Y.

$$\begin{aligned} D_Y &= F_Y(I_S) \\ &= \{[d_1, \ldots, d_n]^T \mid \exists [i_1, i_2, \ldots, i_m]^T \in I_S : \bigwedge_{h=1}^{n} d_h = \sum_{g=1}^{m} c_g^h i_g + c_{m+1}^h\} \end{aligned} \tag{3}$$

As long as I_S is a bounding box iteration domain, the data domain is also a bounding box since the index function is affine. The data domain for one array access can hence be represented as

$$D_Y = \{[d_1, \ldots, d_n]^T \mid \bigwedge_{h=1}^{n} L_h \leq d_h \leq U_h\} \tag{4}$$

where the comparison is applied componentwise and the L_h and U_h are the lower bound and upper bound values of the hth array dimension. For example, the index function and data domain for array A accessed in statement S_1 in Figure 1.(a) are:

$$\begin{aligned} F_{A,S_1} &= \{[i,j,k,l]^T \mapsto [d_1,d_2]^T \mid d_1 = 40i+k \wedge d_2 = 20j+l\} \\ D_{A,S_1} &= F_{A,S_1}(I_{S_1}) \\ &= \{[d_1,d_2]^T \mid \exists [i,j,k,l]^T \in I_{S_1} : d_1 = 40i+k \wedge d_2 = 20j+l\} \\ &= \{[d_1,d_2]^T \mid 0 \le d_1 \le 4419 \wedge 0 \le d_2 \le 1409\} \end{aligned}$$

3.2 The fundamentals of loop transformations

The initial HMSE approach starts from the source code and can work for any kind of loop transformations, i.e. strip mining, fusion, shift, interchange, unrolling, tiling etc. The incremental HMSE is performed based on the initial HMSE result and works for matrix manipulation based affine loop transformations and strip mining. Strip mining creates a new dimension and divides the old dimension with the new dimension's bound. An affine loop transformation can be represented with a linear transformation A and a translation vector a as

$$\vec{i'} = A \cdot \vec{i} + \vec{a} \tag{5}$$

It can also be represented with homogeneous coordinates by adding an extra column corresponding to the translation vector and by an extra row as

$$\begin{bmatrix} \vec{i'} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{a} \\ \vec{0}^A & 1 \end{bmatrix} \begin{bmatrix} \vec{i} \\ 1 \end{bmatrix} \tag{6}$$

this is simply written as

$$\tilde{i'} = \tilde{A} \cdot \tilde{i} \tag{7}$$

$\vec{0}^A$ is a vector of as many zeros as there are columns in A. The main benefit of this notation is that the composition of a series of loop transformations amounts to multiplication of the corresponding homogeneous transformation matrices. When a sequence of affine loop transformations are performed, the transformations can be represented as

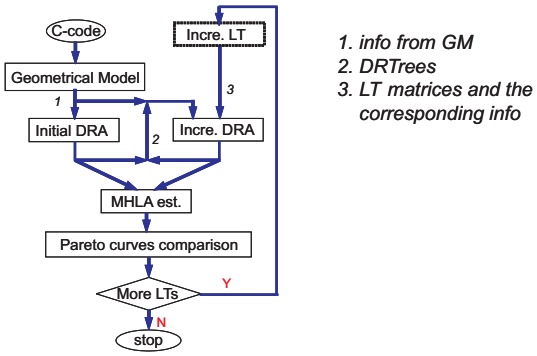
$$\tilde{i'} = \tilde{A}_N \cdot \tilde{A}_{N-1} \cdot \ldots \cdot \tilde{A}_1 \cdot \tilde{i} \tag{8}$$

in which $\tilde{A}_1$ to $\tilde{A}_N$ are the sequence of transformation matrices, corresponding to a sequence of N incremental loop transformations.

Loop transformations are performed for each statement individually. For example, when loop interchange is performed on the code in Figure 1(b) resulting in Figure 1(c), the transformation matrix in the homogeneous coordinate format for both statements (in this case they are identical) is

$$\tilde{A}_{S_1} = \tilde{A}_{S_2} = \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

In the above transformation matrix, each row (except the bottom row) corresponds to a dimension in the loop nest. The matrix identifies how each loop dimension will be transformed including both linear loop transformation and translation. More precisely, if the



2: HMSE flow graph

diagonal element of one row is different from 1 and any of the other elements on that row are non-zero, the corresponding loop dimension is transformed for that statement. The last element in each row (except the last row) corresponds to the translation part (also called the constant offset part) for that dimension. If it has a value other than zero, that dimension is shifted and the value determines the shifting distance. Loop fusion and loop shifting only have effect on the translation part.

Loop transformations improve data locality by reordering the loop iterations while keeping the functionality unchanged. When a loop transformation is performed, the iteration domain of one statement is transformed and hence also the index functions of the arrays accessed within that statement. For a given transformation matrix $\tilde{A}$, the iteration domain and index function after transformation are given by

$$\tilde{I}' = \tilde{A} \cdot \tilde{I} \tag{9}$$

$$\tilde{F}' = \tilde{F} \cdot \tilde{A}^{-1} \tag{10}$$

The transformed data domain for one array access will be

$$\tilde{D}' = \tilde{F}' \cdot \tilde{I}' = \tilde{F} \cdot \tilde{A}^{-1} \cdot \tilde{A} \cdot \tilde{I} = \tilde{F} \cdot \tilde{I} = \tilde{D} \tag{11}$$

This shows that the transformed data domain is always equivalent to the original. This is fundamental to ensure unchanged functionality during a sequence of loop transformations.

When a sequence of loop transformations is performed incrementally, the iteration domain after these transformations can be calculated based on Equation 8 and Equation 9 with the sequence of transformation matrices. The index function after the transform can be calculated based on Equation 10 by multiplying with the inverse transformation matrices. This transformed matrix information is the basis for the incremental estimation when performing sequences of loop transformations. This will be discussed in detail in the next section.

4. HIERARCHICAL MEMORY SIZE ESTIMATION

In this section we present the HMSE methodology. As shown in Figure 2, HMSE is composed of four steps: initial DRA, incremental DRA, MHLA estimation with Pareto curve output and Pareto curve comparison. We divide the overall HMSE approach in two: initial

HMSE and incremental HMSE. The initial approach consists of initial DRA and MHLA estimation, which has been previously presented in [Hu et al. 2006]. The incremental approach consists of incremental DRA and MHLA estimation. The initial approach uses the source C-code as input. It is parsed into the GM on which loop transformations and our estimations are performed. Since only one Pareto curve output is created for the initial HMSE, the output Pareto curve is simply kept during the Pareto curve comparison step. Then we check if we are going to perform additional loop transformations. If not, the estimation stops. If yes, incremental HMSE is performed. Note that HMSE is independent of any specific incremental loop transformation algorithms.

The typical use would be to have HMSE integrated in a state-of-the-art interactive or automatic loop transformation tool. As shown, the incremental DRA uses the previous DRA output as its input, i.e., the previous GM information and the loop transformation matrices and the corresponding statement information. Each incremental HMSE generates a new Pareto curve. This curve is then compared to those generated previously, and only those that have at least one Pareto point that is better than all other curves, are kept. This procedure is repeated until there are no more incremental loop transformations to perform. Let us first briefly demonstrate how the initial approach works on a small example before the incremental approach is presented.

4.1 Initial Data Reuse Analysis

Data reuse analysis identifies data that is accessed repeatedly. This can be arrays or parts of arrays which are then denoted as copy candidates (CCs). It is normally beneficial to copy these CCs from the main memory to smaller (on-chip) memories from where they are accessed multiple times. This can both save energy and improve performance since accessing on-chip memory is faster and more energy efficient.

Initially, data reuse analysis is performed for all array accesses (both write and read) of one array assuming all loops iterate over their complete set of iterator values. It results in the root for that array. Then, the analysis is performed at each loop dimension, with CC(s) as output, starting from the outermost dimension. The analysis is performed both for each individual array access and between different array accesses. The root, together with the CCs at every loop dimension, form the data reuse tree for a given array. Each array has its own data reuse tree.

At one loop dimension, we analyze the data domain accessed within one iteration of that loop by keeping all outer iterators constant and expanding the inner iterators. Since the index function is affine and the iteration domains are bounding boxes, the data domain in any given iteration is the same as in the first iteration, only shifted. We can therefore simply set all outer iterators to 0. For instance, when analyzing the j-dimension of Figure 1(a) we set i to 0 while k iterates from 0 to 59 and l iterates from 0 to 29. The data domains at two consecutive iterations (e.g., $j = 0$ and $j = 1$) of the j loop are therefore

$$D_{A,S_1,j} = F_{A,S_1}(I_{S_1}(i = 0, j = 0)) = \{[d_1, d_2]^T \mid 0 \leq d_1 \leq 59 \wedge 0 \leq d_2 \leq 29\}$$
$$D^{+}_{A,S_1,j} = F_{A,S_1}(I_{S_1}(i = 0, j = 1)) = \{[d_1, d_2]^T \mid 0 \leq d_1 \leq 59 \wedge 20 \leq d_2 \leq 49\}$$

An interesting CC exists since the data domains accessed at the two consecutive iterations are overlapping. If the CC is assigned to the on-chip SPM layer, the overlapping part is reused during the second iteration without fetching it from main memory again. It is

therefore called the *reuse_part*. In this case the *reuse_part* is

$$reuse\_part = D_{A,S_1,j} \cap D^{+}_{A,S_1,j} = \{[d_1,d_2]^T \mid 0 \leq d_1 \leq 59 \wedge 20 \leq d_2 \leq 29\}$$

The non overlapping part accessed at the second iteration value needs to be fetched from off-chip main memory before it is accessed at that iteration and is called the *update_part*. The *update_part* is calculated by taking the difference between the data domain accessed at one iteration and its *reuse_part*.

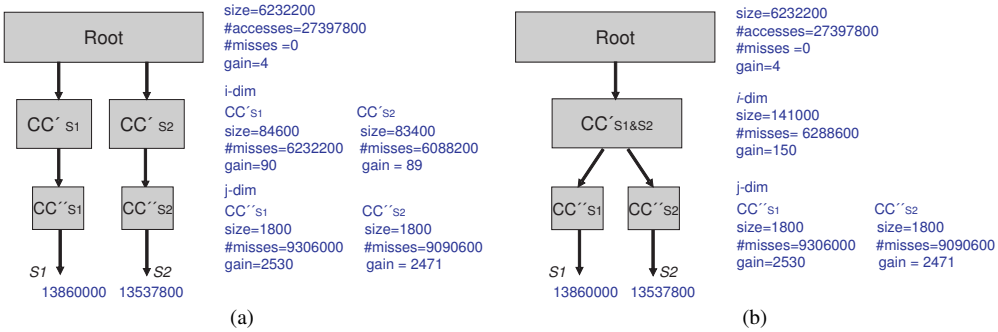
$$update\_part = D^{+}_{A,S_1,j} \setminus reuse\_part = \{[d_1,d_2]^T \mid 0 \leq d_1 \leq 59 \wedge 30 \leq d_2 \leq 49\}$$

For each interesting CC, we need to know three numbers: the size it occupies in the SPM (*size*), the total number of accesses (to SPM instead of main memory) covered by the CC (#*accesses*) and the number of accesses to main memory still needed to fetch the data (#*misses*). Details on how to calculate them can be found in [Hu et al. 2006]. For the example code in Figure 1(a), the data reuse tree is shown in Figure 3(a). The CCs at the i and j dimensions are potentially interesting, but at the inner k and l dimensions the *reuse_part* is empty so no interesting CCs are found here. The same analysis is also performed for the other array access in statement S_2.

A reuse gain (*gain*) is also calculated for each CC. It will be used by the following MHLA estimation as discussed further in Section 4.2. It is defined as the number of accesses to main memory that are avoided, per size unit, by assigning the CC to the SPM. In general, the higher reuse gain a CC has, the more beneficial it is to copy it on-chip. When the gain of a CC is smaller than that of its parent, that CC is discarded.

Above we have discussed how to perform the data reuse analysis for one array access. This analysis is also performed between multiple array accesses. We do this when, for the current and all outer loop dimensions, the iteration domains of the two accesses are overlapping and their index function coefficients are identical. The minimal lower bound and the maximal upper bound are used instead of individual bounds for these iteration dimensions. The union CC, if it exists, will replace the individual CCs in the DRTree, provided it has a larger *gain*. For the example code in Figure 1(b), the union CC between the two array accesses at the i-dimension has larger *gain* than the individual CCs and is kept in the DRTree instead of the individual ones. This is shown in Figure 3(b). At the j-dimension, a union CC between the two accesses also exists ($size = 5000$, $\#misses = 23320000$, $gain = 815.56$). It is not kept because it has lower *gain* than the individual CCs (average $gain = 2500$).

Obviously no data reuse exists for the update parts at the first and last iteration of the dimension being analyzed. We simply ignore this boundary case and assume the data not reused at the boundaries are also fetched from the main memory to the SPM. In contrast, [Issenin et al. 2004] [Van Achteren et al. 2002] [Beyls and D'Hollander 2001] take this boundary effect into account using a more complex analysis. The bounding box approach can also result in overestimates compare to previous solutions when the bounding box does not exactly represent the original iteration domain (e.g., when the domain is diagonal). Consequently, these three previous approaches are more accurate but much slower. In general, the simplified bounding box technique performs sufficiently accurate analysis for most practical cases. It is also extremely fast, which is essential when using HMSE estimation during loop transformation exploration.



3: (a) DRTree for the original code and (b) DRTree for the fused code

4.2 Memory Hierarchy Layer Assignment Estimation

The goal of MHLA is to map the CCs, together with the original arrays, onto a memory platform in order to minimize the overall energy consumption [Brockmeyer et al. 2003]. We perform an estimation of this mapping based on the DRTrees of all arrays for a given version of the application code. As usually no detailed memory platform is defined at the loop transformations stage, we propose a platform-independent MHLA estimation approach based on a two-layer memory hierarchy template: the on-chip SPM layer and the main memory. The size of the main memory is assumed to be large enough to hold all arrays while the on-chip SPM layer has an unfixed size varying from zero and up to the size required to store all arrays.

At the start, the SPM is empty and all accesses from the processor go to the main memory. Then at each pass of the estimation algorithm, the unassigned CC or array with the biggest *gain* is assigned to the SPM layer (replacing its children if they are assigned previously). The rationale behind assignment according to *gain* is that the CC with the highest *gain* replaces, per size unit increase of SPM, the largest number of main memory accesses with accesses to the SPM. Since accessing main memory is more costly than SPM, this represents the most power saving per size unit increase of SPM. Each assignment results in a Pareto point between the number of main memory accesses and that SPM size requirement. The number of main memory accesses for one SPM size is the sum of the number of misses from the assigned CCs/roots and the number of accesses to the unassigned roots. This procedure is repeated until all CCs and arrays in the DRTrees are assigned. For each Pareto point, the information regarding its size requirement and which CCs that have been assigned are kept. The Pareto points together result in the Pareto curve. For the fused code with DRTree shown in Figure 3(b), CC''_{S_1} has the largest gain and is assigned to SPM first followed by CC''_{S_2}. Then CC'_{S_1,S_2} is assigned while its children CC''_{S_1} and CC''_{S_2} are removed from the SPM. Finally the array root is assigned and CC'_{S_1,S_2} is removed resulting in no misses to the main memory. The Pareto curve output is shown in Figure 4.

Because of the greedy stepwise assignment, where each array and CC are considered for assignment only once, our algorithm is very fast. Its complexity is $O(n \log n)$ (where n is the number of CCs and arrays) because of the sorting of CCs and arrays in the DRTrees. n is limited since only CCs with *gain* larger than its parents are included in the DRTrees. For comparison, the algorithm used in [Brockmeyer et al. 2003] has a complexity of $O(2^n n^2 \log n)$ for a given two-layer memory platform instance. Note that this is only for

4: the Pareto curve output for the fused code

one instance of the platform. For m instances the complexity would be m times larger. To estimate the mapping of a K-layer memory platform, the complexity of their approach is $O(k^n n^k \log n)$. Our approach gives a quick MHLA estimate and the experiments in Section 5 show a reasonable accuracy compared to the exact MHLA results. Their work is also targeting (and requires) one specific memory platform instance at a time, while our MHLA estimation is platform instance independent. This means that our MHLA estimation only needs to be performed once for a given version of application code.

For a memory hierarchy with three layers, a four-dimensional Pareto curve would have to be built, with the sizes and misses of L1 and L2 as axes. It is possible to develop a similar heuristic as above for such a situation. However, we don't consider this very useful, as the size/misses trade-off of one layer looks more or less the same for all sizes for the other layer. Therefore in practice we limit the exploration to two-layer memory hierarchies.

For each size and #accesses combination, a rough estimate of the energy consumed by the SPM and main memory can be calculated (e.g. using the #accesses and the energy required per access for a given size). The designer can use this as part of a basis for a size and energy trade-off when selecting the actual memory hierarchy. Since the data layout is not known, it is however difficult to take into account optimization for page mode and burst access [Kim et al. 2003] when calculating power. An early estimation of these effects can be an interesting line of future work. To compare alternative loop transformations, #accesses is still currently the most relevant factor. State-of-the-art power estimation tools could also benefit from the information available in our Pareto curves.

4.3 Incremental Data Reuse Analysis

We use incremental DRA to further speed up the estimation when loop transformations are performed incrementally. The transformations then usually only have local effect, e.g., their effect is restricted to a limited number of arrays and statements, to some array accesses but not the whole array, and/or only to certain loop dimensions. This varies depending on the incremental loop transformation performed. Compared to Figure 1(b), the code in Figure 1(c) corresponds to a loop interchange of the outermost two dimensions of statements S_1 and S_2. In this case, loop transformations are performed at the outermost two dimensions. It is hence only necessary to perform DRA at these two transformed dimensions. Sometimes this has an effect on inner loop dimensions as well, and analysis must then also be performed at these inner dimensions. Otherwise, the analysis at the inner dimensions can be skipped. The results from the previous run of DRA (initial or incremental) can be reused instead. For real life applications, loop transformations usually also have effect on a limited number of arrays. The untransformed arrays are not changed and DRA hence does

not need to be redone for them. Because of this, incremental DRA can significantly save computation time when the transformation effect is local.

An evaluation is performed to decide whether to perform a complete DRA or to perform the normally much faster local updating. Identification of the dimensions that are transformed is based on evaluation of the transformation matrix. As mentioned in Section 3.2, a dimension is transformed if the diagonal element of a row is different from 1 or any of the other elements on that row are non-zero. Thus it is necessary to identify the outermost dimension that is transformed (denoted as tra_OMD) and the innermost dimension that is transformed (denoted as tra_IMD) for each of the transformed array accesses. We then assume that the array accesses are transformed for all dimensions within this range.

The strip mining transform is not simply a matrix multiplication since it creates a new dimension and divides the old dimension with the new dimension's bound difference. The tra_OMD and tra_IMD for strip mining are hence the old dimension and the new dimension. The updating of the iteration domain bounds during strip mining is trivial and we will not go into further details. If an array is transformed for all its dimensions, we choose to rebuild its DRTree from start. Otherwise, local updating is performed by recomputing DRA only at the transformed loop dimension range. Figure 5 shows the pseudo code of the incremental DRA algorithm.

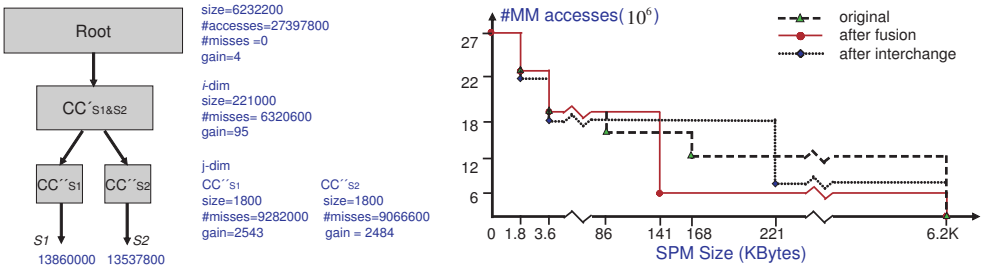
In the procedure $incremental\_DRA$, GM_update is executed for the first time at line 11. At this line the iteration domain of each transformed statement and the index function of all array accesses in the transformed statements are updated based on the Equations 9 and 10. This is required even though the data domain of the whole array is unchanged as proved by Equation 11. The reason is that the data domain calculated for one iteration value of a certain loop dimension, see Section 4.1, is usually changed by the loop transformation. Consequently, it must be calculated based on the updated iteration domain and updated index function.

Let us now illustrate the incremental DRA algorithm based on the loop interchange example in Figure 1(b) and (c). We assume that we have already run HMSE on Figure 1(b). The resulting GM model and Pareto curve for this code is consequently available. After the execution of GM_update, tra_OMD and tra_IMD for the transformed arrays are identified at line 13. Since only the outermost two loop dimensions have been interchanged for the two statements, the if-condition at line 14 is false and the if-condition at line 17 is true. The procedure $local\_update$ is called at line 18 for the parent that are one dimension above tra_OMD. In this case the parent is the root. If the if-condition at line 14 had been true, initial DRA would have been performed. If the if-condition at line 17 had been false, all parents that are one dimension above tra_OMD would have been identified at line 20 and $local\_update$ would have been called for each of the parents at line 21.

Within the $local\_update$ procedure call, the parent is the root for this example and the if-condition at line 31 is true. The CCs at the dimension below the parent is recomputed (denoted as CCs_new) based on $GM\_updated$. Since CCs_new is not equal to the old CCs at the same position within the $DRTree$, the if-condition at line 33 is true. In this case there is only one CC and the CCs_new is updated at line 38. Since both the old and new CC contain the two array accesses, the if-condition at line 39 is true and $local\_update$ is called again at line 40. This function call will continue the evaluation at the next inner dimension, in this case at the second outermost dimension. The updated CC at the outer dimension now becomes parent. The if-condition at line 31 is true, and the new CCs below the parent

```
0:  DRTrees : The data reuse trees for all arrays
    GM : the GM info before incremental loop transformation
    OMD : the outermost dimension
    IMD : the innermost dimension
    tra_OMD : the outermost dimension that has been transformed
    tra_IMD : the innermost dimension that has been transformed
    LTs_info: the performed LT info

10: procedure incremental_DRA()
11:     GM_updated = GM_update(LTs_info, GM)
12:     for (each transformed array)
13:         identify tra_OMD and tra_IMD based on LTs_info
14:         if (tra_OMD = OMD and tra_IMD = IMD )
15:             recompute DRTrees[array] based on GM_updated
16:         else
17:             if (tra_OMD = OMD )
18:                 local_update(DRTrees[array], GM_updated, LTs_info)
19:             else
20:                 locate each parent whose children's dimension is OMD_tra
22:                     local_update(parent, GM_updated, LTs_info)

30: procedure local_update(parent, GM_updated, LTs_info)
31:         if (parent.children != {} and (parent contains transformed array accesses))
32:             CCs_new = CCs_calc(parent, GM_updated, LTs_info)
33:             if (parent.children != CCs_new) /* children have been changed * /
34:                 for (i=0; i<#parent.children; i++)
35:                     if (parent.children[i] exists in CCs_new)
36:                         local_update(parent.children[i], GM_updated, LTs_info)
37:                     else
38:                         update parent.children[i] from the CC within CCs_new
39:                         if (parent.children[i] and CC contains the same array accesses)
40:                             local_update(parent.children[i], GM _updated, LTs_info)
41:                         else /* DRTree for parent.children[i] is changed after transformation*/
42:                             compute DRTree below parent.children[i]
43:             else
44:                 if (parent's dim < (tra_IMD of array accesses parent contains))
45:                     local_update(parent.children[i], GM_updated, LTs_info)
46:         else
47:             compute DRTree below parent

50: function GM_update(LTs_info, GM)
        /* update the iteration domain of each transformed statement */
        /* and the index function of the corresponding array accesses */
        return GM_updated

60: function CCs_calc(parent, GM_updated, LTs_info)
        return CCs_new /* recalculate the children CCs of parent */
```

5: Pseudo code of incremental DRA algorithm



6: (a) DRTree output for the code after loop interchange and (b) Pareto curves comparison

are calculated at the second outermost dimension. Since the loop interchange takes effect at the two outermost dimensions, the new CC is not equal to the corresponding old CC in the DRTrees. Hence it is updated with the new CC at line 38. As both the new and old CC contain the same array accesses at the second dimension, the function *local_update* is again called at line 40. The parent now becomes the updated CC at the second outer dimension. This time the analysis is performed at the third dimension even if it is inside *tra_IMD*. If the analysis detects any CC changes, *DRTree* will be updated with the new CCs at this dimension and the analysis continues at the next inner dimension. If no changes are detected, we go to line 44 and check if the parent contains any array accesses which are transformed at that or inner dimension(s). In this case, no changes are present in the third dimension, so the analysis stops for this branch. Since the if-conditions at line 33 and 44 are both false, the analysis stops without analyzing the third and fourth dimensions. The analysis also stops for this array since only this CC exists. Figure 6(a) shows the DRTree output after performing loop interchange.

For this small example, the local DRA needs to be performed on the three outermost dimensions. Consequently, we do not save much computation time with incremental DRA compared to using the initial DRA. For real life applications, e.g., the QSDPCM driver in Section 5, there can be more than ten loop dimensions. Typical loop transformations are performed at the outer loop dimensions with no effect on the remaining inner dimensions, or at inner dimensions with no effect on other statements. In that case, the incremental DRA with local updating can significantly reduce the computation time. As explained in Section 3.1, it is often necessary to insert an extra time dimension before each loop dimension. The computation time saving of incremental DRA is then even more evident. Furthermore, the number of CCs in a DRTree can potentially increase exponentially at each level in the loop nest. The number of CCs to recompute will therefore become very large for deep loop nests. As a consequence, it is very beneficial to only recompute the CCs that have actually been changed.

Compared to the incremental DRA algorithm presented here, the initial algorithm presented in [Hu et al. 2006] only works for loop fusion and loop shifting. It takes the translation part of the transformation matrix as input as loop fusion and loop shifting only have effect on that part. It is very straight-forward to identify the transformed loop dimensions if their coprresonding element has a non-zero value in the translation part. The initial algorithm only exploits very limited local update by recomputing the DRTree from the outer most transformed loop dimension *tra_OMD* till the inner most dimension. The GM is updated based on the transformed translation part without performing matrix manipulations.

In contract, the algorithm presented here works for all affine loop transformations and strip mining, based on the analysis and manipulation of the whole transformation matrix as shown in Section 3.2. It exploits a transformation's full local effect.

4.4 Pareto curves comparison

As explained in Section 4.2 we generate one Pareto curve after each incremental loop transformation. Many of these can be discarded as they do not have any global Pareto points. A global Pareto point is a point that has a smaller number of main memory accesses than any other points on other curves with an SPM size not larger than the one we have at this point. A global Pareto point can hence result in the most energy efficient use of a certain two layer memory hierarchy instance with an SPM size equal to that of this point. This is because, for a given two layer memory hierarchy instance, the SPM size at the global point always results in more accesses to the SPM and less misses to the main memory than any other point. This means that the loop transformation corresponding to the curve containing this point is more power efficient for that specific memory hierarchy instance and should to be kept among the potential solutions. For deeper memory hierarchies, a set of Pareto points are selected in the same manner and each point describes the estimated data mapping for one SPM layer. For evaluation of loop transformations, this is however not very relevant. This is because the global Pareto curve already indicates if the loop transformation is beneficial for any SPM layer size. If it is (in number of accesses) not better for some SPM size in a two-layer memory hierarchy, it cannot be better for any multiple-layer memory hierarchy either.

Figure 6(b) shows a comparison of the three Pareto curves for the three versions of the code shown in Figure 1. We can see that for small SPM sizes, the interchanged code will result in less main memory accesses than the fused code. This means that the interchanged code can result in the most energy efficient use of certain two layer memory platform instances. On the other hand, the fused code has the smallest number of main memory accesses for larger SPM sizes. Hence these two loop transformations are both interesting and should be kept until the actual memory platform instance is defined and the right version of code is selected. For this small example, the difference of the memory accesses is not that significant. It still demonstrates the fact that different loop transformations might be optimal for different memory platform instances and also shows how the HMSE technique can be used.

The Pareto curve comparison allows us to find all the interesting loop transformations. Later, when the memory hierarchy is given, we can easily find the right transformed version of code. Based on the size/#accesses combination at each global Pareto point, energy of the SPM and main memory can also be calculated as outlined in Section 4.2.

4.5 Summary

This Section has presented the incremental HMSE, which is differentiated from the initial HMSE by exploiting the loop transformation's local effect and performing an advanced incremental DRA algorithm. Multiple (intermediate) loop transformation instances which may result in a low power memory hierarchy usage are retained for further evaluation. Note, however, that loop transformations influence issues such as scheduling freedom, control flow complexity, and instruction locality. High level estimation would be beneficial here as well. This is outside the scope of our current work, but if it would exist, it would be easy to plug into our framework.

When the memory hierarchy configuration is not defined or the SPM size is flexible at the early loop transformation stage, the Pareto curves are used to retain all the interesting loop transformation instances. It can also assist in finding the most appropriate low power memory hierarchy configuration. If the memory hierarchy is already given, HMSE can be used to find the (intermediate) loop transformation instances which may result in low power usage for that specific memory hierarchy instance. In both cases HMSE can naturally be integrated in, and be beneficial for, state-of-the-art loop transformation exploration tools.

5. EXPERIMENTS

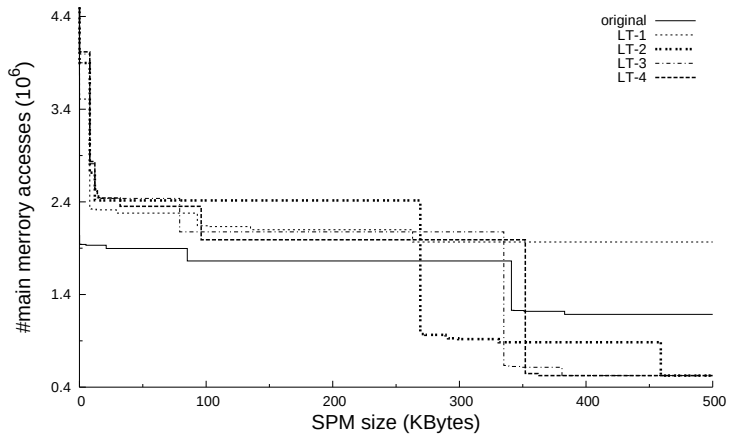
A prototype CAD tool for HMSE has been implemented in Python, incorporating the techniques and algorithms described in this paper. We have performed experiments on several real life test vehicles: 2D Wavelet transform, QSDPCM and Cavity Detection algorithms. For our experiments, the incremental sequences of loop transformations performed are generated manually, but they could alternatively have come from any of the state-of-the-art exploration tools discussed in Section 2. The Wavelet transform is an important part of modern multimedia codecs like Scalable Video Coding (SVC). The QSDPCM algorithm is an inter-frame compression technique for video images. The Cavity Detection algorithm is used for detection of cavity perimeters in medical imaging. Our experiments also include intra-array inplace estimation using our fast estimation approach. See [Hu et al. 2007] for details regarding this.

Figure 7 shows the Pareto curves output for the 2D Wavelet transform after having performed initial HMSE and four incremental HMSEs corresponding to four sets of loop transformations. Each set of loop transformations is in fact a fairly long sequence of transformations on different statements of the code. These transformations are driven by abstract locality measures [Danckaert et al. 2000], without taking into account a realistic memory hierarchy. However, the hierarchical memory size estimation shows that for SPM sizes smaller than 270K, the original code actually performs better (because some data reuse opportunities are destroyed by the transformation). On the other hand, as the SPM size increases, LT-2, LT-3 and LT-4 have the least number of main memory accesses for certain SPM sizes. LT-1 is not optimal for any SPM sizes, and can be discarded. This demonstrates the importance of performing HMSE in order to find the right versions of code during loop transformation exploration. Without doing so, we would end up with one single solution, which would be sub-optimal for many of the possible SPM sizes that could be selected at a later design space.

	GM parsing/update	Intra- + DRA	MHLA est.	incr./init. HMSE (%)
initial HMSE	208.5	911.1	6.6	100
incre. HMSE (LT-1)	142.0	249.4	6.4	27.9
incre. HMSE (LT-2)	140.1	253.3	6.4	28.4
incre. HMSE (LT-3)	140.4	252.2	6.4	28.2
incre. HMSE (LT-4)	140.4	254.7	6.4	28.5

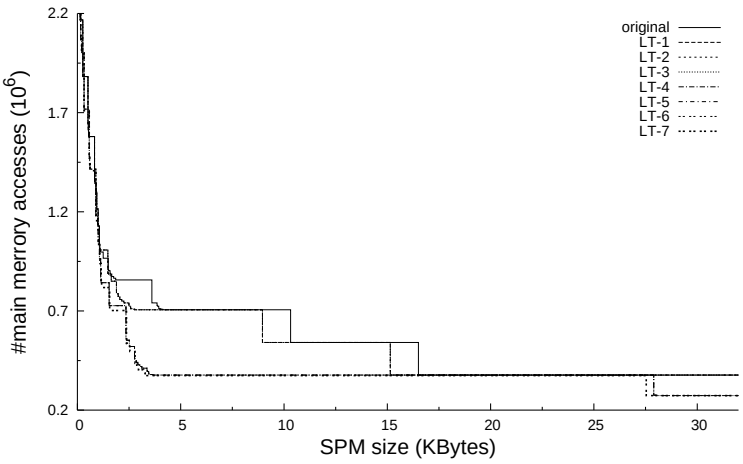
I: Execution time comparison for 2D Wavelet transform [ms]

Table I compares the execution time required for the initial HMSE and each of the incremental HMSEs. In this table, "Intra- + DRA" means the execution time required to perform intra-inplace estimation and DRA. All incremental HMSEs take less than 30% of



7: HMSE output of Pareto curves for 2D Wavelet

the time required by the initial HMSE. The execution time for performing HMSE is also compared to the time needed to read in the GM information for the initial HMSE case and to update GM for incremental HMSE. This GM read-in and updating must in any case be done once during loop transformations and is in fact not part of our iteratively applied HMSE.



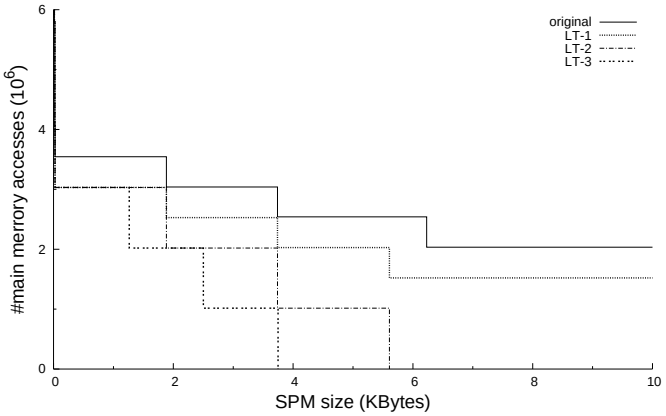
8: HMSE output of Pareto curves for QSDPCM

Figure 8 and Tab. II show the Pareto curves output and the execution time comparison for QSDPCM. For QSDPCM, the fully transformed code LT-7 is always the best. This

	GM parsing/update	Intra- + DRA	MHLA est.	incr./init. HMSE (%)
initial HMSE	542.7	1627.6	10.2	100
incre. HMSE (LT-1)	6.1	9.1	9.3	1.2
incre. HMSE (LT-2)	1.6	2.0	9.5	0.7
incre. HMSE (LT-3)	1.5	2.1	9.3	0.7
incre. HMSE (LT-4)	4.7	8.2	9.0	1.1
incre. HMSE (LT-5)	64.3	222.1	11.5	14.3
incre. HMSE (LT-6)	61.1	177.9	8.9	11.4
incre. HMSE (LT-7)	3.5	9.3	8.9	1.1

II: Execution time comparison for QSDPCM [ms]

means that this code version will result in optimal memory usage for all possible hierarchy instances. For QSDPCM, the incremental HMSEs take between 0.7% and 15% of the execution time needed for the initial HMSE. This is because incremental loop transformations are all performed at the outermost two of over 12 loop dimensions and only a limited number of arrays are transformed. Most of the loop transformations performed have no effect at the inner dimensions and the incremental DRA only needs to locally update the transformed arrays at the two outermost dimensions. This significantly reduces the computation time. LT-5 and LT-6 are big transformation steps which affect about 50% of the array accesses. Still, since not all dimensions are transformed, 85% of the time can be saved by working incrementally.

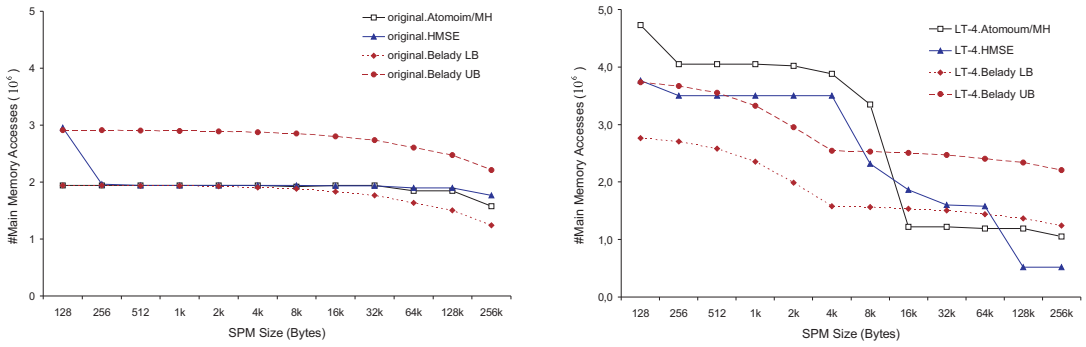


9: HMSE output of Pareto curves for Cavity Detection

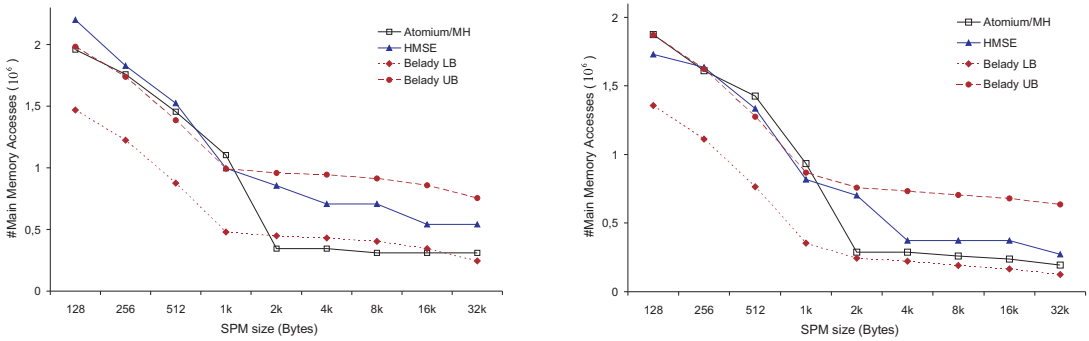
	GM parsing/update	Intra- + DRA	MHLA est.	incr./init. HMSE (%)
initial HMSE	208.5	120.0	1.2	100
incre. HMSE (LT-1)	142.0	19.0	1.2	16.7
incre. HMSE (LT-2)	140.1	30.1	1.3	25.9
incre. HMSE (LT-3)	140.4	19.2	1.2	16.8

III: Execution time comparison for Cavity Detection [ms]

For the Cavity Detection algorithm, the fully transformed version LT-3 is always the best as shown in Figure 9. Compared to the first two code versions original and LT-1, which require over 1M memory to store all arrays, the last two versions only require 5K and 3K, respectively. It is therefore possible to keep all the data of the last two versions on a small SPM layer. With the same SPM size (3K), the original code would need more than 3 million accesses to main memory. Figure 9 also shows why it is important for the memory size estimation to take into account the memory hierarchy. The total memory size requirement is 3838 Bytes for LT-3 and over 1M for the LT-1. Without taking into account the memory hierarchy exploration, the conclusion would therefore be that the code LT-1 is not interesting at all. However, when the hierarchical memory size estimation is performed, it turns out that for SPM sizes up to 1286 Bytes, the code LT-1 is viable alternative. Since the LT-1 code has lower complexity (the loop shifting of LT-3 adds `if`-clauses), it is actually preferred for small SPM sizes. Analysis of the code complexity as a third trade-off axis can be performed at a later stage when the memory platform is given. Even for this small application, incremental HMSE still reduces the execution time significantly as shown in Tab. III.



10: Accuracy comparison for 2D Wavelet between HMSE, Atomium/MH and Belady's algorithm on (a) original and (b) LT-4 codes



11: Accuracy comparison for QSDPCM between HMSE, Atomium/MH and Belady's algorithm on (a) original and (b) LT-7 codes

To evaluate the estimation speed and accuracy of our early system level HMSE, we compare our estimates with the results obtained with the Atomium/MH tool [IMEC 2006], and with Belady's algorithm which has an optimal replacement policy [Belady 1966]. The Atomium/MH tool is partially implemented based on the techniques presented in [Brockmeyer et al. 2003]. Note that their tool can only be performed for one memory hierarchy instance at a time and it hence needs to be performed for all the memory hierarchy instances. In order to make a fair comparison, the inter-array inplace optimization option in their tool is not used since this option is not implemented yet in our current work. The optimal replacement policy is found by simulating an address trace, and finding out how many different addresses are accessed before the same address is accessed again. Unfortunately, this technique does not take into account write-back accesses. To compensate for these, we define a lower and an upper bound (LB and UB in the figures): the lower bound assumes no writing to main memory at all, the upper bound assumes write-through behavior. In addition, the algorithm does not avoid compulsory misses (the first time an address is written, it is always a miss). HMSE and Atomium/MH, on the other hand, can place a complete array in SPM, which removes all main memory accesses for that array. Therefore, the HMSE and Atomium/MH results can sometimes be better than the optimal.

As shown above, our initial approach takes less than 1 second for 2D Wavelet and less than 2 seconds for QSDPCM while the incremental approach further significantly reduces the execution time to milliseconds. In contrast, the Atomium/MH takes minutes for both the 2D Wavelet and QSDPCM applications. For Cavity Detection, the Atomium/MH takes several seconds while our incremental HMSE just takes a couple of milliseconds. The simulation for the optimal replacement policy takes days. In summary, experiments show that our initial HMSE is at least two orders of magnitude faster than Atomium/MH, and our incremental HMSE can further speed up the estimation another two orders of magnitude. Our prototype tool is implemented in Python while Atomium/MH is implemented in C++. Our tool would be even faster if it was implemented in C/C++.

Now let us look at the accuracy of our estimation. Figure 10 and Figure 11 show a comparison of the output between HMSE, Atomium/MH and Belady's algorithm for 2D Wavelet and QSDPCM on several realistic two layer memory hierarchy instances. For the original version of the 2D Wavelet code, HMSE produces estimates that are very close to the Atomium/MH tool. For the transformation $LT-4$, there exists estimation difference between HMSE and Atomium/MH. This is partially due to that the different intra-array memory size estimation technquies give different results. When the two layer memory hierarchy instances have an SPM layer larger than 64K, incorrect HMSE results occur also due to that the intra-array memory technique used in HMSE gives incorrect array size requirement estimation. This, in turn, leads to incorrect HMSE output.

Compared to Belady's algorithm, both HMSE and Atomium/MH perform fairly well on the original code. On the transformed code, there is a large difference for intermediate SPM sizes, because no interesting copy candidates exist with a size between 256 bytes and 8K. Belady's algorithm, on the other hand, can fill up the remaining size with parts of copy candidates.

For QSDPCM, the results of HMSE are in general also close to those of Atomium/MH for the different versions of codes. When the two layer memory hierarchy instances have an SPM size between 2K and 8K, HMSE gives some overestimates compared to the Atomium/MH tool. This is because the Atomium/MH tool finds data dependent copies for some

arrays that our HMSE does not find. The reason is that the simplified geometrical model which is the input of our HMSE does not model data dependent terms in the index expressions accurately. Accuracy compared to Belady's algorithm is again fairly good. For the Cavity Detection algorithm, the HMSE estimates are identical to the results of Atomium/MH. The fidelity is also high for all applications.

We have also studied the scalability of our incremental approach for larger applications. To demonstrate this, we have duplicated the QSDPCM code 8 times (8*QSDPCM) giving approximately 7500 lines of C-code. Furthermore, we have built a simple incremental loop transformation generator that performs automatic loop transformation exploration. The automatic loop transformation generator explores hundreds of alternatives in seconds. For the 8*QSDPCM, the initial HMSE takes 16 seconds. The time required for incremental HMSE varies depending on the loop transformations performed and their effect on the array accesses. For the sequence of incremental loop transformations used for QSDPCM above, the incremental HMSEs for 8*QSDPCM take approximately the same execution time as for the original QSDPCM. This is not unnatural since the loop transformations generated by the generator, like those of more general techniques, have mostly local effect. This is the case even when the application is large like 8*QSDPCM and demonstrates that our incremental approach is scalable to industrial-sized applications.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an automatic hierarchical memory size requirement estimation methodology that can be used to steer loop transformation exploration. It can report all the potentially good loop transformations which may result in optimal usage for any memory hierarchy instance. When the memory hierarchy is given, the good loop transformations for that specific memory hierarchy can easily be selected. Several advanced techniques such as bounding box data reuse analysis, platform independent MHLA estimation, and locally update the data reuse analysis have been introduced to make the incremental estimation fast. Experiments also show that our incremental approach is scalable for large applications and that the accuracy is adequate for our steering purposes. For future work, we will integrate inplace mapping estimation, which is another important factor highly influenced by loop transformations.

REFERENCES

BACON, D. F., GRAHAN, S. L., AND SHARP, O. J. Dec. 1994. Compiler transformations for high-performance computing. *ACM computing surveys 26,* 4, 245–420.

BALASA, F., CATTHOOR, F., AND DE MAN, H. 1995. Background memory area estimation for multidimensional signal processing systems. *IEEE Trans. on VLSI Systems 3,* 2 (June), 157–172.

BANERJEE, U. 1993. *Loop transformation for restructuring compilers: the foundations*. Kluwer Acad. Publ., Boston, USA.

BELADY, L. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal 5,* 6, 78–101.

BENINI, L., MACII, A., AND PONCINO, M. 2000. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design and Test of Computers 17,* 2 (Apr.), 74–85.

BEYLS, K. AND D'HOLLANDER, E. 2001. Reuse distance as a metric for cache behavior. In *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems*, T. Gonzalez, Ed. IASTED, Anaheim, California, USA, 617–622.

BROCKMEYER, E., MIRANDA, M., CORPORAAL, H., AND CATTHOOR, F. 2003. Layer assignment techniques for low energy in multi-layered memory organisations. In *Proc. 6th ACM/IEEE Design and Test in Europe Conf.* Munich, Germany, 1070–1075.

CATTHOOR, F., andC. KULKARNI, K. D., BROCKMEYER, E., KJELDSBERG, P. G., VAN ACHTEREN, T., AND OMNES, T. 2002. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Acad. Publ., Boston, USA.

CATTHOOR, F., WUYTACK, S., DE GREEF, E., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology – Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Acad. Publ., Boston, USA.

COHEN, A., GIRBAL, S., AND TEMAM, O. 2004. A polyhedral approach to ease the composition of program transformations. In *Proc. EuroPar Conf.* Number 3149 in Lecture notes in computer science. Springer Verlag, Pisa, Italy, 292–303.

DANCKAERT, K., CATTHOOR, F., AND DE MAN, H. 2000. A loop transformation approach for combined parallelization and data transfer and storage optimization. In *Proc. ACM Conf. on Par. and Dist. Proc. Techniques and Applications, PDPTA'00*. Las Vegas NV, USA, 2591–2597.

DARTE, A. 2000. On the complexity of loop fusion. *Parallel Computing 26,* 9, 1175–1193.

DARTE, A. AND ROBERT, Y. 1995. Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *Journal of Parallel and Distributed Computing 29,* 1 (Aug.), 43–59.

FRABOULET, A., HUARD, G., AND MIGNOTTE, A. 1999. Loop alignment for memory access optimization. In *Proc. 12th ACM/IEEE Int. Symp. on System Synthesis*. San Jose, CA, 71–77.

GIRBAL, S., VASILACHE, N., BASTOUL, C., COHEN, A., PARELLO, D., SIGLER, M., AND TEMAM, O. 2006. Simi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming 34,* 3, 261–317.

GRUN, P., BALASA, F., AND DUTT, N. 1998. Memory size estimation for multimedia applications. In *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design(Codes)*. Seattle WA, USA, 145–149.

HU, Q., BROCKMEYER, E., PALKOVIC, M., KJELDSBERG, P. G., AND CATTHOOR, F. 2004. Memory hierarchy usage estimation for global loop transformations. In *Proc. IEEE Norchip Conference*. Oslo, Norway, 301–304.

HU, Q., VANDECAPPELLE, A., KJELDSBERG, P. G., CATTHOOR, F., AND PALKOVIC, M. 2007. Fast memory footprint estimation based on dependency distance vector calculation. In *Proc. 10th ACM/IEEE Design and Test in Europe Conf.* Nice, France.

HU, Q., VANDECAPPELLE, A., PALKOVIC, M., KJELDSBERG, P. G., BROCKMEYER, E., AND CATTHOOR, F. 2006. Hierarchical memory size estimation for loop fusion and loop shifting in data-dominated applications. In *Proc. 11st Proc. IEEE Asia and South Pacific Design Autom. Conf. (ASPDAC)*. Yokohama, Japan, 606–611.

IMEC. 2006. Atomium web site. http://www.imec.be/design/atomium/.

ISSENIN, I., BROCKMEYER, E., MIRANDA, M., AND DUTT, N. 2004. Data reuse analysis technique for software-controlled memory hierarchies. In *3rd ACM/IEEE Design and Test in Europe Conf.* Paris, France, 202–207.

KANDEMIR, M. AND CHOUDHARY, A. 2002. Compiler-directed scratch pad memory hierarchy design and management. In *Proc. 38th ACM/IEEE Design Automation Conf.* New Orleans, USA, 628–633.

KELLY, W. AND PUGH, W. 1993. A framework for unifying reordering transformations. Report UMIACS-TR-92-126.1, University of Maryland at College Park, Institute for Advanced Computer Studies, MD, USA.

KIM, H. S., VIJAYKRISHNAN, N., KANDEMIR, M., BROCKMEYER, E., CATTHOOR, F., AND J. IRWIN, M. 2003. Estimating influence of data layout optimizations on sdram energy consumption. In *Proc. Intnl. Symp. on Low Power Electronics and Design (ISLPED'03)*. 40–43.

KJELDSBERG, P. G., CATTHOOR, F., AND AAS, E. J. 2003. Data dependency size estimation for use in memory optimization. *IEEE Trans. on Comp.-aided Design 22,* 7 (July), 908–921.

MCKINLEY, K., CARR, S., AND TSEND, C. W. Jul. 1996. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems 18,* 4.

NGUYEN, N., DOMINGUEZ, A., AND BARUA, R. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embedded Comput. Syst.(TECS) 5,* 2, 472–511.

PANDA, P., CATTHOOR, F., DUTT, N. D., DANCKAERT, K., BROCKMEYER, E., KULKARNI, C., VANDERCAPPELLE, A., AND KJELDSBERG, P. G. 2001. Data and memory optimization techniques for embedded systems. *ACM Trans. Design Automation of Electronic Systems 6,* 2 (Apr.), 149–206.

PANDA, P. R., DUTT, N. D., AND NICOLAU, A. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proc. 5th ACM/IEEE Europ. Design and Test Conf.* Paris, France, 7–11.

SONG, Y., XU, R., WANG, C., AND LI, Z. 2004. Improving data locality by array contraction. *IEEE transactions on computers 53,* 9, 1073–1084.

STEINKE, S., WEHMEYER, L., LEE, B.-S., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proc. 5th ACM/IEEE Design and Test in Europe Conf.* Paris, France, 409–415.

UDAYAKUMARAN, S. AND BARUA, R. 2006. An integrated scratch-pad allocator for affine and non-affine code. In *Proc. 2006 ACM/IEEE Design and Test in Europe Conf.* Munich, Germany, 925 – 930.

VAN ACHTEREN, T., DECONINCK, G., CATTHOOR, F., AND LAUWEREINS, R. 2002. Data reuse exploration techniques for loop-dominated application. In *Proc. 5rd ACM/IEEE Design and Test in Europe Conf.* Paris, France, 428–535.

VERBAUWHEDE, I., CATTHOOR, F., VANDEWALLE, J., AND DE MAN, H. 1989. Background memory management for the synthesis of algebraic algorithms on multi-processor dsp chips. In *Proc. VLSI'89, Intnl. Conf. on VLSI*. Munich, Germany, 209–218.

VERBAUWHEDE, I., SCHEERS, C., AND RABAEY, J. 1994. Memory estimation for high-level synthesis. In *Proc. 31st ACM/IEEE Design Automation Conf.* San Diego CA, USA, 143–148.

VERDOOLAEGE, S., BRUYNOOGHE, M., JANSSENS, G., AND CATTHOOR, F. 2003. Multi-dimensional incremental loop fusion for data locality. In *Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors(ASAP'03)*. Leiden, The Netherlands, 17–27.

WILDE, D. K. 1993. A library for doing polyhedral operations. M.S. thesis, Oregon State University, Corvallis, OR. also Technical Report PI-785, IRISA, Rennes, France.

WOLF, M. E. AND LAM, M. S. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems 2,* 4 (Oct.), 452–471.

WUYTACK, S., DIGUET, J. P., CATTHOOR, F., AND DE MAN, H. 1998. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. *IEEE Trans. on VLSI Systems 6,* 4 (Dec.), 529–537.

ZHAO, Y. AND MALIK, S. 1999. Exact memory size estimation for array computation without loop unrolling. In *Proc. 36th ACM/IEEE Design Automation Conf.* New Orleans, USA, 811–816.

ZHU, H., LUICAN, I. I., AND BALASA, F. 2006. Memory size computation for multimedia processing applications. In *Proc. 11st Proc. IEEE Asia and South Pacific Design Autom. Conf. (ASPDAC)*. Yokohama, Japan, 802–807.