# Hierarchical Memory Size Estimation for Loop Fusion and Loop Shifting in Data-Dominated Applications

Qubo Hu* Arnout Vandecappelle† Martin Palkovic†
Per Gunnar Kjeldsberg* Erik Brockmeyer† Francky Catthoor†‡

* Norwegian University of Science and Technology, Trondheim, Norway {qubo.hu,pgk}@iet.ntnu.no
† IMEC vzw, Leuven, Belgium {vdcappel,palkovic,brockmey,catthoor}@imec.be
‡ also professor at Katholieke Universiteit Leuven, Belgium

**Abstract — Loop fusion and loop shifting are important transformations for improving data locality to reduce the number of costly accesses to off-chip memories. Since exploring the exact platform mapping for all the loop transformation alternatives is a time consuming process, heuristics steered by improved data locality are generally used. However, pure locality estimates do not sufficiently take into account the hierarchy of the memory platform. This paper presents a fast, incremental technique for hierarchical memory size requirement estimation for loop fusion and loop shifting at the early loop transformations design stage. As the exact memory platform is often not yet defined at this stage, we propose a platform-independent approach which reports the Pareto-optimal trade-off points for scratch-pad memory size and off-chip memory accesses. The estimation comes very close to the actual platform mapping. Experiments on realistic test-vehicles confirm that. It helps the designer or a tool to find the interesting loop transformations that should then be investigated in more depth afterward.**

## I. Introduction

In most advanced embedded real-time communication and multimedia processing applications, the manipulation of large data sets has a major effect on both energy consumption and performance of the system. This is due to the huge amount of data transfers to/from large, energy consuming off-chip data memories. Globally optimizing the memory accesses of data-dominated applications is therefore critical for system performance and energy consumption. Loop transformations are important techniques for improving parallelism, performance and to reduce memory energy consumption [2, 19, 5]. They are usually performed on a Geometrical Model (GM) [18]. Loop fusion, or combining with loop shifting to satisfy dependency, is the basic transformation for improving data locality [8, 17]. It was shown [6] that the search for optimal loop fusion for global array contraction in general is an NP-complete problem. Heuristics based on data locality are hence used in the existing work. Still, data locality is a very abstract measure, so several techniques have been developed to estimate the real size requirements for the large data structures [1, 20, 9, 14, 10].

However, the size as such does not directly represent how the accesses to the costly off-chip memories can be reduced. Indeed, the minimal size for some array may still be larger than the local memory. In addition, if sufficient locality between read accesses is present, a local copy of part of the array may already remove most of the off-chip accesses [7], making the actual size of that array less relevant.

To select the interesting loop transformation candidates, it is therefore necessary to estimate not just the size of each array, but also their mapping on the hierarchical memory architecture. The number of costly off-chip memory accesses depends on which arrays and which copies can fit in the local memory. In addition, the platform and hence the exact size of the memories are often not yet known at this early design stage, so the estimation must take this unknown parameters into account.

In this paper we present a technique and tool support for hierarchical memory size requirement estimation for loop fusion and shifting. The basic idea has previously been introduced in [11] but it is significantly extended here. Our focus is on the Scratch-Pad Memory (SPM) based memory hierarchy, which is a more energy-efficient alternative to caches. The global view taken during data mapping replaces the area and energy consuming hardware used in caches. The SPM is filled with not only arrays [15], but also copies of the currently used part of the arrays [4].

Our hierarchical memory size estimation is performed in two main phases: a data reuse analysis phase and a Memory Hierarchy Layer Assignment (MHLA) estimation phase. Data reuse analysis is performed on the geometrical model by determining, for every loop nest, the set of data which has reuse. The different ways to copy data across the memory hierarchy is represented in data reuse trees. Earlier work using exact data reuse analysis techniques [16, 3, 12, 13] is either too limited, or too slow to be used for the exploration of a huge number of loop transformations. Instead of doing the actual geometrical computations required for this analysis, we use a bounding box approximation of the domains. This can be an over-estimate, but in practical cases it turns out to be as good as an exact analysis. To further save computation time when loop shifting and fusion are applied, we propose to incrementally compute the data reuse trees based on the previous ones.

The MHLA estimation phase selects which arrays and copies are stored in the SPM, such that the number of off-chip memory accesses is minimized. The existing technique for MHLA [4] finds the optimal selection for a given memory hierarchy (SPM size) using backtracking. Their approach

```
for (y=0; y<=399; ++y)
  for (x=0; x<=639; ++x)
    image[x][y] = ...;              // S1
for (y=0; y<=399; ++y)
  for (x=1; x<=638; ++x)
    for (z=-1; z<=1; ++z)
      ... = g(image[x+z][y]);       // S2
```

**Fig. 1.:** Code example before loop transformation

is not feasible for our estimation purpose as the memory platform instance is usually not defined at the loop transformation stage: it is not realistic to perform an estimate for each possible memory hierarchy instance. Their heuristic has high complexity and is hence too slow for large applications, making it unfeasible to be used during the exploration of loop transformations. Instead, a platform-independent heuristic is used, which is fast but usually comes very close to their result. It outputs Pareto curves (SPM size vs. off-chip accesses) for different loop transformations and helps to find the possibly good loop transformation alternatives. The Pareto curve furthermore allows an early energy estimate of any two-layer memory hierarchy instance.

The rest of this paper is organized as follows. Section II reviews the basic concepts in the geometrical model on which the loop transformations are performed. Section III presents our algorithm for doing fast hierarchical memory size requirement estimation. Experiments on real-life applications are demonstrated in Section IV. Conclusions and future work are drawn in Section V.

## II. Overview of the geometrical model

The targeted data-dominated applications are at the system level characterized by deep loop nests and multidimensional arrays as shown in Fig. 1. Loop transformations are usually performed on a geometrical model which uses multidimensional iteration domains and access mappings to represent all necessary information. The concepts needed to understand how our techniques use the geometrical model are presented below. Further details can be found in [18].

The iteration domain 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 execution of the statement. For example, the iteration domain of statement `S2` in Fig. 1 is described as:

$$ID_{S2} = \{[y,x,z] | 0 \le y \le 399 \wedge 1 \le x \le 638 \wedge -1 \le z \le 1\}$$

Note that we leave out the constraint of integer points, $[y,x,z] \in \mathbb{Z}^3$, to simplify the formulas.

Each statement has a number of accesses to variables. For our purpose, only the array accesses are important, as scalars are assumed to be mapped to local memory anyway. Each array reference (read or write) in the statement has an access mapping: a function mapping the iterators to the array indices. The access mapping for array `image` referenced in statement `S2` is described as:

$$AM_{\texttt{image},S2} = \{[y,x,z] \mapsto [a_1,a_2] | a_1 = x+z \wedge a_2 = y \wedge [y,x,z] \in ID_{S2}\}$$

The data domain of an array in a certain statement represents which elements are accessed in that statement. It is found by projecting the iteration dimensions from the access mapping:

$$\begin{aligned} DD_{\texttt{image},S2} &= \{[a_1,a_2] | \exists y,x,z : a_1 = x+z \wedge a_2 = y \\ &\qquad \wedge [y,x,z] \in ID_{S2}\} \\ &= \{[a_1,a_2] | 0 \le a_1 \le 639 \wedge 0 \le a_2 \le 399\} \end{aligned}$$

Usually, geometrical models use polytopes to represent the domains. A large set of operations can be applied on polytopes, and they are sufficient to represent many practical applications. However, polytope operations are still rather computationally expensive, especially counting the number of integer points. Therefore, we use a simplified geometrical model which uses only bounding boxes; computations on it are extremely fast. The idea was first introduced in [11] (where it is called a hyperplane). A bounding box is specified by the lower and upper bounds of the corresponding domain in each dimension. In the examples given earlier, the bounding boxes (denoted by $\overleftrightarrow{D}$) are exact: $\overleftrightarrow{ID}_{S2} = \{[0,1,-1] \rightarrow [399,638,1]\}$ and $\overleftrightarrow{DD}_{\texttt{image},S2} = \{[0,0] \rightarrow [639,399]\}$.

## III. Hierarchical memory size estimation

This section explains how to do the platform-independent hierarchical memory size estimation at the early loop transformations design stage. Our approach consists of four steps. Each of them is explained in the following subsections.

### A. Initial data reuse analysis

Our data reuse analysis is performed on the geometrical model and identifies the data (arrays or parts of arrays) that are most frequently accessed at each loop nest. It can potentially save energy and improve performance when the heavily accessed data is copied from the main memory to the smaller on-chip SPM from where it is accessed multiple times. The frequently accessed data to be copied are called copy candidates. The data reuse analysis is done for each array individually. Initially, all the array references for one array are considered together resulting in the declared array (root). This is represented geometrically with the union of the data domains of all array references. Then, the analysis is proceeded at each loop dimension, starting from the outermost dimension. The analysis is performed both for individual array references and between different references. The recursive analysis at all loop dimensions results a tree set of copy candidates, as shown in Fig. 3 (it was called copy candidate graph in [7]).

At a certain loop dimension, the data domain at that level is calculated by assuming that the current and all outermost dimensions remain constant. The descendant inner loop dimensions projected as for the total data domain. This leads to the following formulation for the data domain of array `image` in statement `S2` at the level of the $x$-loop:

$$\begin{aligned} DD_{\texttt{image},S2|y=0,x=1} &= \{[a_1,a_2] | \exists z : a_1 = 1+z \wedge a_2 = 0 \\ &\qquad \wedge -1 \le z \le 1\} \\ \overleftrightarrow{DD}_{\texttt{image},S2|y=0,x=1} &= \{[0,0] \rightarrow [2,0]\} \end{aligned}$$

```
for (y=0; y<=399; ++y)
  for (x=0; x<=639; ++x) {
    image[x][y] = ...;                  // S1
    if (x>=2)
      for (z=-1; z<=1; ++z)
        ... = g(image[x+z][y]);         // S2
  }
```

**Fig. 2.:** Code example after loop fusion and shifting

The number of points in the data domain determines the size of the copy candidate: 3 in the example. The bounding box approximation allows the use of a constant instead of a symbolic for the data domain: the size is 3 independent of $x$. In addition, it allows a very simple formula for the size.

$$\#size_{CC} = \prod_{i=1}^{n} (UB_{DD_i} - LB_{DD_i} + 1)$$

where $n$ is the number of array dimensions, $UB$ and $LB$ are the bounding box boundaries per dimension.

Data reuse is present at a certain loop dimension when the data domain at that level overlaps with the data domain at the same level in the next iteration. For instance, $\overleftrightarrow{DD}_{\texttt{image},S2|y=0,x=2} = \{[1,0] \rightarrow [3,0]\}$ is overlapping with $\overleftrightarrow{DD}_{\texttt{image},S2|y=0,x=1}$. In that case, a copy candidate is created at that dimension. The overlapping part is called the *reuse part*. The data in the *reuse part* does not need to be read from off-chip memory but can be accessed in the SPM. To keep the SPM up-to-date, in every iteration of the loop some data is copied from the off-chip memory to the SPM. This is called the *update part*. It is the difference between the data domain and the *reuse part*. In the example, the *reuse part* has 2 elements and the *update part* has 1 element. Note we only need to know the size, which makes the computations very fast again (computing set difference is rather complex, even with the bounding box approximation). The copy candidate also has to be initialized with the *reuse part* in the first iteration of the loop.

To evaluate how useful a copy candidate is, we need to know two figures: the number of accesses and the number of misses.

$$\#accesses_{CC} = \#iter_{all}$$

where $\#iter_{all}$ means the total number of the iterations for all loop dimensions surrounding the statement at where the array is referenced. In the example, *#accesses* is 768000.

$$\#misses_{CC} = \#iter_{outer} \cdot (\#size_{reuse\ part} + \#size_{update\ part} \cdot \#iter_{cur})$$

where $\#iter_{cur}$ and $\#iter_{outer}$ are respectively the number of iterations at the analyzing dimension and at all ancestor loop dimensions. *#misses* at the $x$-dimension is hence 256000, calculated as $400 \cdot (2 + 1 \cdot 640)$.

Fig. 3.a shows the data reuse tree for the example code in Fig. 1. At the root, all array references are considered together. The analysis then continues at dimensions $y$, $x$ and $z$ individually. Copy candidates with reuse are detected at the $y$ and $x$-dimensions. For the loop fused code shown in Fig. 2, the same procedure is repeated. This time, interesting copy candidates

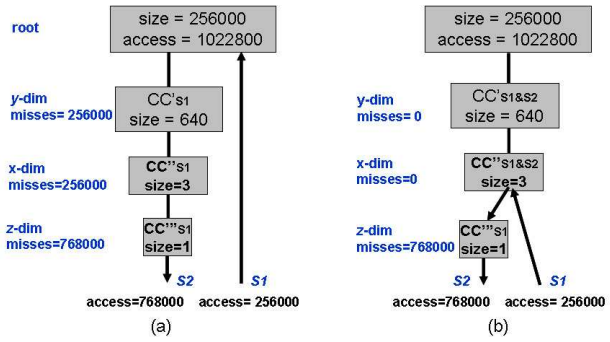



**Fig. 3.:** Data reuse trees for the example codes (a) before any transformation and (b) after loop shifting and fusion

are detected at $y$-dimension and at the $x$-dimension when the two array references are analyzed together. The analysis between multiple references is performed only if they are in the same loop nest till the current dimension and they have identical index coefficients till the current dimension. Note that the loop shifting and fusion has resulted in copy candidates with no misses to the original array. The array does not even have to be written to the main memory and can be kept completely in 3 SPM locations. This results in a significant energy reduction.

The data reuse trees show which copies are potentially interesting to put in the SPM. However, they do not yet show their impact on a hierarchical memory organization. That is analyzed in the next step.

### *B. Platform-independent MHLA estimation*

The MHLA estimation is performed based on the data reuse trees. It maps the copy candidates together with the original arrays (root) onto a memory hierarchy in order to minimize the energy consumption. As there is usually no memory platform defined at the loop transformations stage, we propose a platform-independent MHLA estimation based on a two-layer memory hierarchy template. The size of the main memory is assumed to be unlimited, while the on-chip SPM layer has a varying size. The reason behind is that this memory hierarchy template enables us to simulate any two layer memory platform instances with an early power estimate as explained afterward.

As a starting point, the SPM is empty and all accesses from the processor go to the main memory. Then at each iteration, the candidate giving the biggest potential benefit, as explained below, is assigned to the SPM (replacing its children if they were present). This procedure is repeated until all copy candidates and arrays are assigned. At each iteration, the SPM size increases and the accesses to the main memory decreases.

The potential benefit of a copy candidate is quantified by its $gain\_factor$. The one having the highest $gain\_factor$ among all the unconsidered candidates is selected for assignment.

$$gain\_factor_{CC} = \frac{\#accesses_{CC} - \#misses_{CC}}{\#size_{CC}}$$

The rationale behind this selection criterion is that the candidate with the highest $gain\_factor$ replaces, per size unit increase of SPM, the largest number of off-chip accesses with

accesses to the SPM. For example, the $gain\_factor$ for copy candidate $CC''_{S1\&S2}$ shown in Fig. 3.b is $\frac{1022800-0}{3} = 340933$.

Each iteration results in a Pareto trade-off point between the size of the SPM (denoted by $\#SPM\_size$) and the number of accesses to the main memory (denoted by $\#MM\_acc$).

$$\boxed{\#SPM\_size = \sum \#size_{CC}}$$

$$\boxed{\#MM\_acc = \#accesses_{total} + \sum (\#misses_{CC} - \#accesses_{CC})}$$

in which $\#accesses_{total}$ is the total number of accesses from the processor core, and the sum is over the currently selected copy candidates. In the end, all arrays are assigned to the SPM and there are no accesses to main memory. Different loop transformation alternatives will result in their own Pareto curves.

Because of the incremental assignment, where each array and copy candidate are only considered for assignment once, our algorithm is very fast. It has a complexity of $O(n \log n)$ where $n$ is the total number of copy candidates and arrays considered. For comparison, The algorithm used in[4] has a complexity of $O(2^n n^2 \log n)$ for a predefined two layer memory hierarchy instance. Our platform-independent algorithm, on the other hand, can be used for a whole range of possible two-layer platform instantiations (e.g. with different SPM sizes or different SPM bank activations on a configurable organization). It gives a quick MHLA estimate with reasonable result, which is acceptable for the estimation purposes. This is substantiated on real-life applications in Section IV.

### C. Data reuse analysis for incremental loop transformations

Previous techniques have no direct coupling between loop transformations and data reuse analysis when incremental transformations are performed. That is, the changed geometrical model after a loop transformation must be dumped to C-code which is then parsed back to the geometrical model for repeated data reuse analysis. This dumping and parsing procedure is time consuming and is redundant as we can simply update the geometrical model with the loop transformation performed and then rebuild data reuse trees directly based on the updated geometrical model.

Additionally, the data reuse tree rebuilding procedure can be sped up by just rebuilding the trees for the transformed arrays if not all arrays are affected at a time, as is typically the case when loop transformations are performed incrementally. If an array has only been transformed starting from a certain inner loop dimension, it is for sure no changes happened on its data reuse tree at the outer loop dimensions and those dimensions do not need to be updated. Fig. 4 shows our algorithm with three alternatives for incremental data reuse analysis. The choice of alternative is based on an evaluation of the loop transformation effects. As data reuse analysis is the most time consuming step in our estimation framework, this incremental data reuse analysis can significantly reduce the execution time especially when the incremental loop transformations only affect inner loop dimensions. Local data reuse tree update at the transformed loop dimensions only is also possible, but this cannot be proceeded without analysis of which transformations is exactly performed and it is considered for future work.

```
OMD = the outermost dimension
find all the transformed arrays in GM
update GM based on transformations
if ( all arrays are transformed at OMD ) then
    compute the trees for all arrays based on updated GM
else
    for each transformed array
        OMD_tra = its transformed outermost dimension
        if (OMD_tra == OMD)
            recompute tree for this array based on updated GM
        else
            update tree starting from OMD_tra down
```

**Fig. 4.:** Incremental data reuse analysis algorithm

### D. Comparison between different Pareto curves

Based on the Pareto curves generated for different loop transformations, we determine the potentially good loop transformation alternatives. All Pareto curves are combined into a global Pareto curve, and any alternative contributing to the global curve is good for a certain platform instance. A point belongs to the global Pareto curve if it is not dominated in both $\#SPM\_size$ and $\#MM\_acc$ by any other point. This means less accesses to the main memory then any others. Since accessing on-chip SPM memory is more energy efficient and faster than accessing main memory, the global Pareto point will result in the most energy efficient solution at least for this memory platform instance. Hence, that loop transformation alternative results in minimal energy for certain memory platform instances.

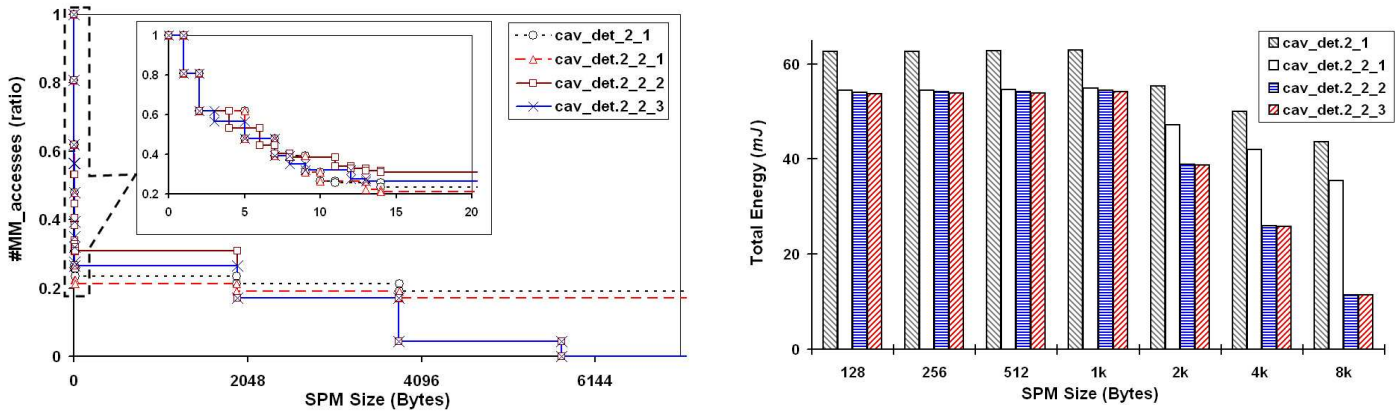
This also demonstrates how we can simulate any two-layer memory platform instances based on the Pareto curve for specific loop transformation. The Pareto point selected for simulating the on-chip SPM layer should be the one having a size as close as possible to, but not larger than, the SPM size of the selected platform. The chosen Pareto point defines which data that should be mapped on the on-chip SPM layer. The off-chip memory of the selected platform should be large enough to store the remaining data. The energy can hence be estimated based on the number of accesses to each layer together with an abstract energy-per-access model, which depends on the SPM size. Energy estimation for a number of realistic two layer memory platform instances are demonstrated on the real life applications in the next section.

## IV. Experimental results

Two realistic demonstrators are selected to present our automated estimation method. The first one is the cavity detection algorithm used for detection of cavity perimeters in medical imaging. The second one is the video compression algorithm Quadtree Structured Difference Pulse Code Modulation (QSDPCM).

### A. Cavity Detection

In the original cavity detection code, different intermediate arrays are produced and consumed in different loop nests. Data locality can be improved with loop fusion (combined with shifting to satisfy dependencies). Fig. 5(a) shows our estimation results with Pareto curves for four selected incremental loop fusion and shifting alternatives. The horizontal axis shows the SPM size required and the vertical axis shows




(a) (b)

**Fig. 5.:** (a) Pareto curves and (b) energy estimate comparison for Cavity Detection

*#MM_acc* normalized over the total number of accesses from the processor core (29 million). When the SPM size increases, the main memory accesses decrease. As shown, the four Pareto curves all contribute to global Pareto points when the SPM size is 128 or smaller. This indicates that each loop transformation instance may result in low power memory hierarchy exploration with very small SPM size. cav_det.2_2_2 and cav_det.2_2_3 both contribute to the global Pareto points when the SPM size is larger than 1927. When SPM size is larger than 5748, there is no off-chip memory accesses for these two transformation alternatives, indicating that all data can be accessed on-chip.

As mentioned, any transformation alternatives that contribute to the global Pareto points can potentially result in minimal energy consumption. This is substantiated with the estimated energy for a number of two layer memory hierarchy instances shown in Fig. 5(b). The energy is calculated based on the number of accesses to each memory layer and an abstract energy-per-access model. For this experiment our energy estimate always has over-estimate with maximal 5% margin, compared to the detailed data reuse analysis and MHLA [4]. As shown in the energy estimate, significant energy reduction can be achieved when a suitable memory hierarchy instance, together with the right version of codes, is chosen. For example, the version of code cav_det.2_2_2 or cav_det.2_2_3 is selected for two layer memory hierarchy having 4K or 8K SPM size.

Fig. 5(a) also shows why it is important for the memory size estimation to take into account the memory hierarchy. The total memory size requirement is 5745 for cav_det.2_2_3 and cav_det.2_2_3, and 2536880 for cav_det.2_1 and cav_det.2_2_1. Without taking into account the memory hierarchy exploration, the conclusion would therefore be that cav_det.2_1 and cav_det.2_2_1 are not interesting at all. However, when the hierarchical memory size estimation is performed, it turns out that for SPM sizes up to 1927, cav_det.2_2_1 is a viable alternative. Since the original code cav_det.2_1 has lower complexity (the loop shifting adds `if`-clauses), this alternative is actually preferred for small SPM sizes. Analysis of the code complexity as a third trade-off axis is hence required for future work.

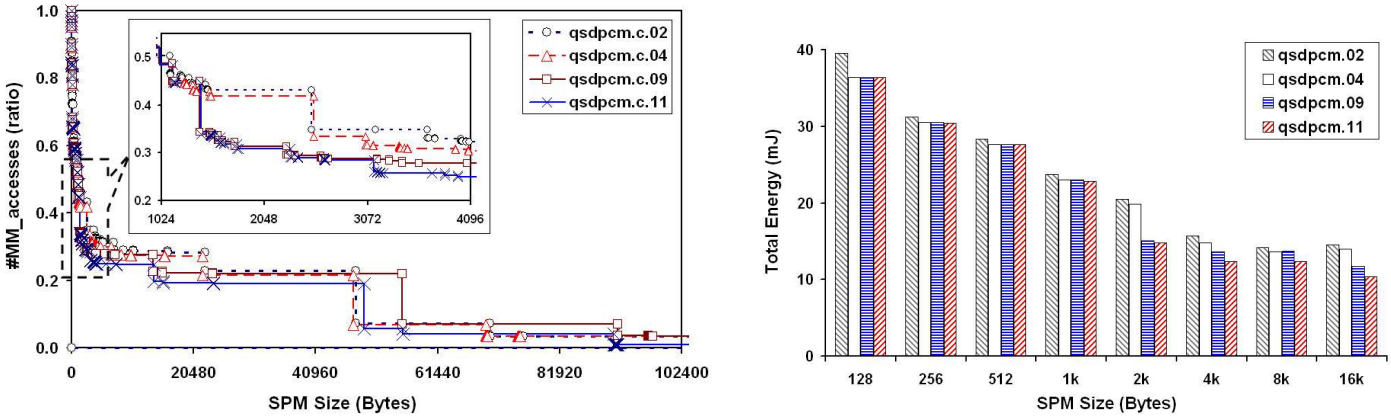
### B. QSDPCM

The QSDPCM algorithm is an inter-frame compression technique for video images, which involves hierarchical motion estimation and a quadtree-based encoding of the motion compensated frame-to-frame differences. Fig. 6(a) shows the estimation outputs with four Pareto curves corresponding to the four selected incremental loop transformations alternatives. All these four Pareto curves contribute to global Pareto points. As shown, there are significant differences in the off-chip memory accesses between the first two and the last two transformation instances, especially when SPM size is between 1312 and 2496. Fig. 6(b) shows that choosing the right loop transformations among the 4 choices can give 25% reduction in total energy consumption for the memory platform having 2k SPM size.

As mentioned, speed is critical for the estimation among the larger number of loop transformation possibilities. It is also our motivation to do a fast estimation in order to help the designer find the right loop transformation alternatives while trading off a suitable memory hierarchy instance. Experiments show the usefulness of our techniques. For the cavity detection algorithm, the approach in [4] takes 1.54 seconds of CPU time for a single SPM size, compared to 0.30 seconds for all sizes in our approach. For the QSDPCM algorithm, theirs takes between 2 to 5 minutes for a single SPM size, compared to 3.0 seconds for ours. In particular, our approach further reduces the execution time during estimation for incremental loop transformations. The time varies between close to 0 and the time required for the first round estimate, depending on the incremental loop transformations' effects on data reuse trees and the choices chosen to rebuild the new data reuse trees. In contrast, the time is constant for each round analysis of the approach in [4]. The time difference will be very significant considering the exploration among a large number of loop transformation possibilities. Note that [4] approach can only estimate for one specific memory hierarchy instance at one time and our implementation is in python which can be a factor 10 slower.

## V. Conclusion and Future Work

This paper presents a technique for hierarchical memory size requirement estimation for loop fusion and loop shifting




(a) (b)

**Fig. 6.:** (a) Pareto curves and (b) energy estimate comparison for QSDPCM

at the early loop transformations stage. As a large number of loop transformation possibilities exist and usually no memory platform is defined at this stage, we have proposed a fast platform-independent hierarchical memory size estimation algorithm. It outputs Pareto curves enabling to select the possibly good loop transformations. The Pareto curve also permits energy estimate for any two-layer memory hierarchy instances. This helps the designer to select a memory hierarchy instance, together with the right loop transformation alternatives. Experiments show the satisfactory estimation result with fast speed, which is critical for the exploration of the large number of loop transformation possibilities.

Loop transformations improving data locality also have a direct impact on array data lifetimes, which can potentially affect the memory size requirement both for individual arrays and between different arrays. This array lifetime analysis is not yet considered in our method. As taking it into account can potentially lead to better memory exploration, integrating it in our method is considered for future work. As our estimation method are in principle also applicable to any affine loop transformations, the automation of incremental estimation for general affine loop transformations is also left for future work.

## REFERENCES


[1] F. Balasa, F. Catthoor, and H. De Man. Background memory area estimation for multi-dimensional signal processing systems. 3(2):157–172, June 1995.

[2] U. K. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publ., Norwell, MA, 1993.

[3] K. Beyls et al. Reuse distance-based cache hint selection. In *International Euro-Par Conference, 2002*.

[4] E. Brockmeyer, M. Miranda, H. Corporaal, and F. Catthoor. Layer assignment techniques for low energy in multi-layered memory organisations. In *Proc. 6th ACM/IEEE Design and Test in Europe Conf.*, pages 1070–1075, Munich, Germany, Mar. 2003.

[5] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology, Exploration of memory organization for embedded multimedia system design*. Boston, MA, 1998.

[6] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[7] J.-P. Diguet, S. Wuytack, F. Catthoor, and H. De Man. Formalized methodology for data reuse exploration in hierarchical memory mappings. In *Proc. IEEE Int. Symp. on Low Power Design*, pages 30–35, Monterey CA, Aug. 1997. IEEE.

[8] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory access optimization. In *Proc. 12th ACM/IEEE Int. Symp. on System Synthesis*.

[9] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Proc. ACM/IEEE Wsh. on Hardware/Software Co-Design (CODES)*, pages 145–149, Seattle, WA, Mar. 1998.

[10] Q. Hu, M. Palkovic, and P. Kjeldsberg. Memory requirement optimization with loop fusion and loop shifting. In *Euromicro Symp. on Digital System Design (DSD'04)*, pages 272–278, Aug. 2004.

[11] Q. Hu, M. Palkovic, and P. Kjeldsberg. Memory requirement optimization with loop fusion and loop shifting. In *Euromicro Symp. on Digital System Design (DSD'04)*, pages 272–278, Aug. 2004.

[12] I. Issenin, E. Brockmeyer, M. Miranda, and N. Dutt. Data reuse analysis technique for software-controlled memory hierarchies. In *3rd ACM/IEEE Design and Test in Europe Conf.*, pages 202–207, Paris, France, Feb. 2004.

[13] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proc. 39th ACM/IEEE Design and Test in Europe Conf.*, pages 690–695, Las Vegas, NV, June 2002.

[14] P. Kjeldsberg, F. Catthoor, and E. J. Aas. Data dependency size estimation for use in memory optimization. 22(7):908–921, July 2003.

[15] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proc. 5th ACM/IEEE Europ. Design and Test Conf.*, pages 7–11, Paris, France, Mar. 1997.

[16] T. Van Achteren, F. Catthoor, R. Lauwereins, and H. De Man. Search space definition and exploration for nonuniform data reuse opportunities in data-dominant applications. 8(1):125–139, 2003.

[17] S. Verdoolaege, M. Bruynooghe, G. Janssens, and F. Catthoor. Multi-dimensional incremental loop fusion for data locality. In *Proc. Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, pages 17–27, Leiden, The Netherlands, June 2003.

[18] D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, OR, Dec. 1993. also Technical Report PI-785, IRISA, Rennes, France.

[19] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN '91 Conf.*, pages 30–44, Toronto, Canada, June 26–28 1991.

[20] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *Proc. 36th ACM/IEEE Design Automation Conf.*, pages 811–816, New Orleans, LA, June 1999.