J Sign Process Syst (2013) 72:151–164
DOI 10.1007/s11265-013-0774-y

Optimizing Data Placement of Loops for Energy Minimization with Multiple Types of Memories

Jun Zhang · Tan Deng · Qiuyan Gao · Qingfeng Zhuge · Edwin H.-M. Sha

Received: 1 November 2012 / Revised: 29 March 2013 / Accepted: 10 May 2013 / Published online: 28 June 2013

Abstract Strict real-time processing and energy efficiency are required by high-performance Digital Signal Processing (DSP) applications. Scratch-Pad Memory (SPM), a software-controlled on-chip memory with small area and low energy consumption, has been widely used in many DSP systems. Various data placement algorithms are proposed to effectively manage data on SPMs. However, none of them can provide optimal solution of data placement problem for array data in loops. In this paper, we study the problem of how to optimally place array data in loops to multiple types of memory units such that the energy and time costs of memory accesses can be minimized. We design a dynamic programming algorithm, Iterational Optimal Data Placement (IODP), to solve data placement problem for loops for processor architectures with multiple types of memory units. According to the experimental results, the IODP algorithm reduced the energy consumption by 20.04 % and 8.98 % compared with a random memory placement method and a greedy algorithm, respectively. It also reduced the memory access time by 19.01 % and 8.62 % compared with a random memory placement method and a greedy approach.

Keywords Data placement · Loops · Scratch-pad memory · Optimization

J. Zhang · Q. Zhuge (✉) · E. H.-M. Sha
College of Computer Science, Chongqing University, Chongqing, 400040, China
e-mail: qfzhuge@gmail.com

J. Zhang
e-mail: jeffjunzhang@gmail.com

E. H.-M. Sha
e-mail: edwinsha@gmail.com

T. Deng · Q. Gao
College of Information Science and Engineering, Hunan University, Changsha, 410082, China

T. Deng
e-mail: dengtan0510@gmail.com

Q. Gao
e-mail: angela.gao8908@gmail.com

E. H.-M. Sha
Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080, USA

1 Introduction

Designing energy-efficient and cost-effective system is one of the most outstanding problems for digital signal processors along with an increasing demand of high-performance applications [15]. High performance systems usually present strict requirement in energy consumption because of limited battery lifetime and concerns of sustainable computing. In order to exploit advantages of various types of memory units, many processor architectures employ multiple memory types in their design. Scratch-Pad Memory (SPM), a software-managed on-chip memory, is used in many DSP systems, including ARM10E, Analog Devices ADSPTS201S, and TI's TMX320C6XXX [4, 6, 8–10, 17], because of its advantages in power, cost, and performance. An efficient on-chip memory access design plays a critical role for reducing energy consumption as well as achieving time performance. On the other hand, the total cost of memory accesses for most computation-intensive applications is dominated by energy or time costs of accessing array data in loops. Therefore, the problem of how to optimally place array data to on-chip SPMs and second level memory

becomes a key problem for achieving high performance and low energy consumption.

Research interests have been focused on SPM to reduce the power and improve performance for a long time [4]. The process of data placement is to determine the location of every data on different memory units. There are two categories of SPM data placement methods: one is fixed data placement and the other is regional data placement. In a fixed data placement method, data placement is determined for the whole program and will not change during the program execution. Hence, it is also called static method in many papers. Avissar et al. [3] proposed a static data placement, which determines data placement on SPMs for global variables. In a regional data placement method, a program is divided into several regions by compiler. Data placement is determined for each program region. Regional data placement is also referred as dynamic method in previous works. Dynamic data placement method is superior to static method because data placement is adjusted for minimal access cost in each program region.

Our study in this paper employs dynamic method to optimize fine-grain data placement for array data in loops on processor architectures with multiple types of memory units. A dynamic programming algorithm proposed by Zhuge et al. [20] can achieve optimal data placement with minimal access cost for scalar variables in a program region. The application of the algorithm in [20], however, is very limited because it lacks a mechanism to utilize the pattern of array data accesses. Especially, it cannot effectively deal with data placement problem for array data in loops. In this paper, we consider each iteration of a loop as a program region. Data placement is then determined for each iteration of a loop. We improved the dynamic programming algorithm proposed in [20] by taking advantages of data dependencies among array data. As a result, our algorithm generates optimal data placement solution for both scalar and array variables using dynamic programming approach on a processor architecture equipped with multiple SPMs and a second-level memory.

The contributions of this work include:

- We design a greedy algorithm that can allocate array data in loop on memory architecture with different types of memory units.
- We study the data placement problem for array data in loops to reduce energy cost and time cost of memory accesses by exploiting data dependencies among loop iterations.
- We design a technique for optimally placing array data in each loop iteration with minimal energy and time cost of memory accesses. Then, we introduce an improved dynamic programming algorithm, Iterational Optimal Data Placement (IODP), based on a previous algorithm.

Experiments are conducted on DFG benchmarks for IODP algorithm and comparisons have been made with random algorithm and greedy strategy. For all the benchmarks in the experiments, IODP algorithm always performs better than the greedy strategy in terms of time cost and energy consumption. The result of our experiments shows that the average improvement of IODP algorithm over "random" technique is 20.04 % on energy consumption and 8.98 % compared with Udayakumaran's greedy algorithm. The percentage of improvement for IODP over the "random" technique in time cost is 19.01 %, which is 8.62 % when compared with Udayakumaran's algorithm. The IODP technique achieves the best improvements among all data placement techniques used in our experiments.

The rest of this paper is organized as follows: In Section 2, we provide background and related works. In Section 3, we introduce the architecture of multiple types of memory units in single-core system. In Section 4, we present a motivational example, which is solved by both greedy algorithm and IODP algorithm, to show the effectiveness of the proposed algorithm. In Section 5, we present the problem definition. In Section 6, we present details of the improved dynamic approach "Iterational Optimal Data Placement" (IODP). In Section 7, we present the experimental results. And Section 8 concludes this paper.

2 Background and Related Works

Compared with hardware-controlled cache, SPM, a software-controlled on-chip memory, has been considered an attractive solution and widely adopted in many DSP systems because of its small die area and low energy consumption. When using SPM, compilers or programmers need to decide an appropriate data placement before program execution.

Recent research has focused on developing compiler-guided dynamic data placement strategies for SPM systems. Banakar et al. [4] proposed a simple SPM data management algorithm. Their algorithm can not guarantee to achieve optimal result and is only applicable to scalar data. Some research has proposed static data placement methods, in which the data placement is determined for the whole program and will not change during the execution of the program. Avissar et al. [3] proposed an ILP formulation to obtain a data placement for the whole program, which is static data placement. Panda et al. [13, 14] also proposed static data placement techniques for array variables. Sjödin and von Platen also proposed a static data placement method [16]. The drawback of static method is that it cannot explore the benefit of varying data locality in a running environment.

The other techniques are dynamic data placement methods. The program will be divided into different regions, and data movement instructions are inserted before each region to generate data placement that is best for the current region. During the execution of a particular region, the data placement remains the same. Dynamic data placement method has better performance than static data placement since it can take better advantage of the data locality of each program region. Several works have been done with dynamic data placement methods, such as greedy strategy is used in [18, 19] to find a data placement for global data in each region by Udayakumaran. Dominguez et al. [7] proposed a regional data placement technique for heap data. In Udayakumaran's work, most frequently accessed data items are assigned to SPM for each program region using greedy strategy which cannot achieve optimal data placement with minimal memory access cost.

However, little research has been done from the aspects of the initial data location and the array data in loop. Ozturk et al. [11, 12] proposed algorithms to manage data for array-intensive nested loops with regular data access patterns. Absar et al. [2] and Chen et al. [5] proposed regional data management for irregular arrays. Their methods greatly rely on the loop's characteristics and are different from what we are aiming for. In this paper, we propose the IODP algorithm to minimize the cost of loop data accesses for memory architectures with multiple types of memory units. It can solve Iterational Data Item Placement (IDIP) problem and support both scalar and array data by taking both the dependence of array and the updating cost into consideration. In addition, our method is able to allocate parts of an array, for instance when the whole array does not fit into the SPM.

3 Hardware and Software Model

In this section, we will introduce the targeting hardware architecture and the execution model of program used by our algorithm.

3.1 Hardware Architecture

In this paper, we are targeting on a single-core system with multiple memory units architecture. A three levels memory architecture is shown in Fig. 1 for demonstration purposes. In this architecture, the single-core system has an on-chip SPM and an on-chip second level memory. The third level memory is off-chip memory. The processor is responsible for moving data around. The SPM has both tightest memory size constraint and the fastest access speed (e.g. SRAM SPM). The second level memory has medium memory size and access speed (e.g. SRAM SPM). The third level memory has the largest memory size and slowest access speed

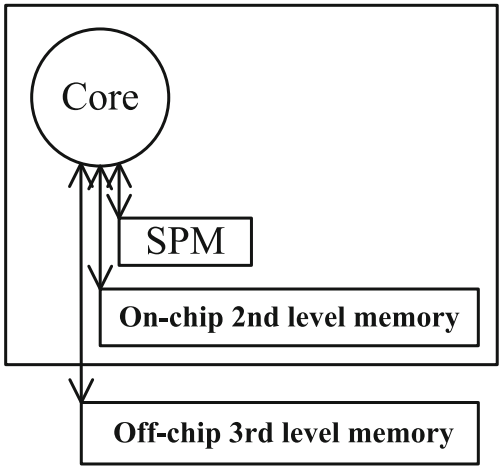


Figure 1 A single-core architecture.

(e.g. DRAM main memory). This architecture is similar to a real system with level 1 (L1) cache, level 2 (L2) cache, and main memory. In general, there may be more on-chip and off-chip memory units with different size as well.

3.2 Execution Model

The execution model is shown in Fig. 2. All the data placement instructions are inserted into the loop by compiler. Before the beginning of the loop, data items will be allocated into L1 and L2 memory from main memory as initial data placement. Then, we will have another data placement at the end of each iteration. The data placement in the loop body may not be the same for each iteration. The goal is to minimize the total cost of memory accesses by achieving the best data placement in each iteration for L1, L2 memory, and main memory. Our proposed technique is optimal for each single iteration, but does not guarantee a global optimal placement across all iterations.

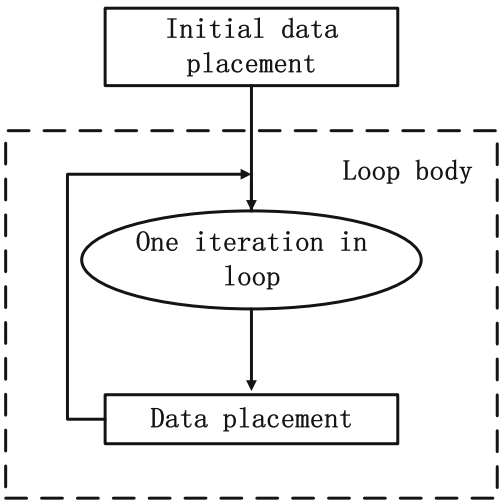


Figure 2 Loop execution model.

4 Motivational Example

A motivational example to demonstrate the main idea of our improved algorithm is presented in this section. We will compare the result of the IODP algorithm with Udayakumaran's greedy algorithm. We focus on the architecture as shown in Fig. 1. We assume that main memory is big enough to hold the whole array and each array has a data copy in main memory. The data items of array need to be updated into main memory after used to guarantee the correct array access. This example assumes that the SPM and the L2 memory can separately hold two and three data items. The main memory is large enough to store all data items.

Before presenting the examples, some notations and their definitions are shown in Table 1. The values of these notations used in this example are also shown in the table.

The loop used in this example is shown in Fig. 3b. Assuming that seven data items are accessed in this loop, which are d_1, d_2, $A[i]$, $A[i-1]$, $A[i-2]$, $B[i]$, and $B[i-1]$. The loop can be represented by a data flow graph (DFG). Each node in the DFG represents a computation. For example, in Fig. 3a, node d_2 represents the computation of d_2 in the loop body of the code shown in Fig. 3b. An edge in the DFG with some bars represents the data delay between two nodes. And the number of bars is equal to the number of delays. The problem is how to find a data placement for these data items when the total cost of memory accesses is minimized.

Suppose that in the initial placement, all data items are in the main memory. The number of accesses in one iteration

Table 1 Notations used in examples.

Notation	Time (μs)	Definition
CR_S	1	the cost of reading from SPM.
CW_S	1	the cost of writing to SPM.
CR_{L2}	15	the cost of reading from L2 memory.
CW_{L2}	15	the cost of writing to L2 memory.
CR_M	50	the cost of reading from main memory.
CW_M	50	the cost of writing to main memory.
$CM_{S\to L2}$	16	the cost of moving data from SPM to L2 memory.
$CM_{L2\to S}$	16	the cost of moving data from L2 memory to SPM.
$CM_{S\to M}$	51	the cost of moving data from SPM to main memory.
$CM_{M\to S}$	51	the cost of moving data from main memory to SPM.
$CM_{L2\to M}$	65	the cost of moving data from L2 memory to main memory.
$CM_{M\to L2}$	65	the cost of moving data from main memory to L2 memory.

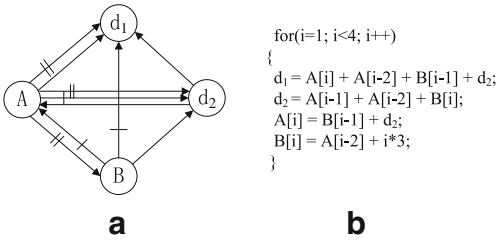


Figure 3 **a** The corresponding DFG **b** A loop for the example.

for each data item is shown in Table 2. Take data d_1 for example, it is accessed once every iteration. For data $A[i-2]$, we can see that it is accessed three times in one iteration.

We decide the data allocation for this loop with both the greedy strategy derived from Udayakumaran's algorithm and dynamic programming strategy presented in Section 6. In each loop iteration, the derived Udayakumaran's algorithm allocates the first $Size_{SPM}$ data items in SPM and the next $Size_{L2}$ data items in L2 memory. The data that do not fit in SPM and L2 are allocated to the main memory. This algorithm can guarantee that data with larger number of accesses is allocated in the faster level of memory, and the total execution time seems the least because the execution time is only related with the data's number of accesses. Section 5 discusses the greedy algorithm in detail.

The comparison of the results generated by the greedy algorithm and the proposed optimal algorithm are shown in Table 3. According to Table 3, the final data placement for the greedy algorithm results in a total time cost of memory accesses of 2841 μs as shown in Table 3. The data items $A[i-2]$ and $B[i-1]$ need to be updated to main memory. On the other hand, by using the dynamic programming algorithm as shown in Table 3, the total time cost can be reduced by 17.35 % to 2348 μs. In fact, this is the optimal data placement in each iteration of the example. For initial data placement, our goal is to get a good result with the principle of filling up all the on-chip memories.

The derived greedy algorithm does not take the initial data placement and array updating cost into consideration. In some cases it is not necessary to move data. Keeping the data in the original place might achieve better results

Table 2 Numbers of data accesses in one iteration.

Data	Accesses
d_1	1
d_2	3
$A[i]$	2
$A[i-1]$	1
$A[i-2]$	3
$B[i]$	2
$B[i-1]$	2

Table 3 Costs and Placement of each iteration with two different methods.

Uday	Initial Data Placement	Iteration 1	Iteration 2	Iteration 3	Iteration 4
SPM	$A[i-2], d_2$	$A[i-2], d_2$	$A[i-2], d_2$	$A[i-2], d_2$	$A[i-2], d_2$
L2 Mem.	$A[i], B[i], B[i-1]$	$A[i], B[i], B[i-1]$	$A[i], B[i], B[i-1]$	$A[i], B[i], B[i-1]$	$A[i], B[i], B[i-1]$
Main Mem.	$d_1, A[i-1]$	$d_1, A[i-1]$	$d_1, A[i-1]$	$d_1, A[i-1]$	$d_1, A[i-1]$
Update	$A[i-2], B[i-1]$	$A[i-2], B[i-1]$	$A[i-2], B[i-1]$	$A[i-2], B[i-1]$	$A[i-2], B[i-1]$
Cost(μs)	609	558	558	558	558
Total Cost(μs)	2841				
Optimal					
SPM	$d_2, A[i-2]$	$d_2, B[i]$	$d_2, B[i-1]$	$d_2, B[i]$	$d_2, B[i-1]$
L2 Mem.	$A[i], A[i-1], B[i]$	$A[i-1], A[i-2], B[i-1]$	$A[i], A[i-2], B[i]$	$A[i], A[i-1], B[i-1]$	$A[i-1], A[i-2], B[i]$
Main Mem.	$d_1, B[i-1]$	$d_1, A[i]$	$d_1, A[i-1]$	$d_1, A[i-2]$	$d_1, A[i]$
Update	$A[i-2]$	$A[i-2], B[i-1]$	$A[i-2], B[i-1]$	$B[i-1]$	$A[i-2], B[i-1]$
Cost(μs)	579	426	456	461	426
Total Cost(μs)	2348				

since the moving cost of data may sometimes be higher than expected. Therefore, it cannot achieve optimal results. The proposed IODP algorithm in this paper considers all these factors in order to achieve optimal solution. We will formally define the Iterational Data Item Placement (IDIP) Problem in Section 5. And the details of IODP will be presented in Section 6. In the experiment section, we will compare the derived Udayakumaran's algorithm with our improved IODP algorithm.

5 Problem Definition and A Greedy Algorithm for Loop Data Placement

In this section, we define the problem of loop data placement with multiple memory types. Then, a new greedy algorithm derived from Udayakumaran's algorithm is presented to achieve effective loop data placement with different memory types.

5.1 Notations and Definitions

In Table 4, descriptions of notations used for all algorithms in this paper are presented. Then, we formally define some terms used in this problem.

Definition 1 Iterational Data Placement Function The iterational data placement is defined as a function from D to M in one loop iteration P, $alloc^P$: $D \rightarrow M$, where D is a set of different types of data items and M is a set of different types of memory units. A function $alloc^P(d_j) = m_i$ indicates one data item d_j is allocated to a memory unit of type m_i in iteration P. Therefore, $alloc^{P-1}(d_j)$ represents the location of the data item d_j before the execution of iteration P in loop.

Definition 2 Dependent Array An array is dependent when the array data item accessed in the previous iteration is still accessed in the current iteration.

If data items $A[p]$, ..., $A[p-j]$, ..., and $A[p-k]$ ($k \geq j \geq 1$) from array A are accessed in the loop at the same iteration P. We say that array A is dependent, and the dependence of data items in array is defined as: $dependence_{A[p]} = 0$, ..., $dependence_{A[p-j]} = 0$, ..., $dependence_{A[p-k+1]} = 0$, and $dependence_{A[p-k]} = k$. The dependence of array A is defined as $dependence_A = k$.

Definition 3 Independent Array An array is independent when the array data item accessed in the previous iteration is not accessed in the current iteration.

The dependence of an independent array A is defined as $dependence_A$, which equals to 0. All data items $A[p]$

Table 4 Notations used in algorithms.

Notation	Definition
D	a set of different data items for the whole loop $d_1, d_2, ..., d_N$, where the data can be either a scalar or an array data.
M	a set of different types of memory units $m_0, m_1, ..., m_i, ..., m_j, ..., m_{T-1}$, where T is a constant, and m_0 is main memory.
$Size_{m_i}$	the size of memory type m_i.
$Read_{m_i}$	the cost of a read operation on memory type m_i.
$Write_{m_i}$	the cost of a write operation on memory type m_i.
$Size_{d_j}$	the size of data item d_j.
RN_{d_j}	the number of read operations for data item d_j in one iteration.
WN_{d_j}	the number of write operations for data item d_j in one iteration.

in independent array are also independent. We define: $dependence_{A[p]}$ = -1, $A[p]$ is a data item from array A in iteration P.

Definition 4 Updating Cost The updating cost refers to the cost of reading the array data item from its allocated location (except main memory), and writing it back to main memory.

$$Update(d_j) = \begin{cases} Read_{m_i} + Write_{m_0} & \text{if } m_i \neq m_0 \text{ and} \\ & denpendence_{d_j} \neq 0 \\ 0 & \text{else} \end{cases} \quad (1)$$

In the iterational data item placement problem (Definition 4.7), given an array data item d_j not be allocated to the main memory, if and only if $dependence_{d_j} \neq 0$, then $update(d_j) \neq 0$ (as is shown in Eq. 1).

It can ensure that there is a copy of the array with the newest data value in the main memory. Data items in array may come from dependent arrays or independent arrays. When a data item is allocated to main memory, the updating cost is always zero.

For Eq. 1, m_i is the allocated location of this data item. In the example of this paper, in iteration P, data $A[p]$, $A[p-1]$, $A[p-2]$ is used, $dependence_{A[p-2]}$ = 2. After each iteration, we need update $A[p-2]$ because $A[p-2]$ will not be allocated to main memory, $A[p]$ and $A[p-1]$ are not need to be updated. Hence, the updating cost is defined as the sum of one read operation from the memory type m_i and one write operation to the main memory, as shown in Eq. 1.

Definition 5 Moving Cost Moving cost is the cost of retrieving one data item from its original location and writing it to another memory unit with different type. Hence, the moving cost is defined as the sum of one read operation from the memory type m_i and one write operation to the memory type m_j, as shown in Eq. 2.

$$Move(m_i, m_j) = Read_{m_i} + Write_{m_j} \quad (2)$$

Definition 6 Cost of Memory Accesses for One Data Item in One Loop Iteration Let m_i be the location of data item d_j decided by a data placement function $alloc^P(d_j)$ in loop iteration P. The total cost of memory accesses for data item d_j in iteration P, represented by a cost function $Cost_{m_i}(d_j)$, is the sum of accessing data d_j during program execution, moving data d_j to a memory unit of different memory type before execution and the updating cost of that data item. The cost of memory accesses for data d_j is computed by Eq. 3.

$$\begin{aligned} Cost_{m_i}(d_j) = {} & RN_{d_j} \times Read_{m_i} + WN_{d_j} \times Write_{m_i} \\ & + Move(alloc^{P-1}(d_j), alloc^P(d_j)) \\ & + Update(d_j) \end{aligned} \quad (3)$$

Definition 7 Iterational Data Placement Problem Given a set of loop data D in a single loop iteration P, a set of various memory types M, $Size_{m_i}$ representing capacity of memory type $m_i \in M$, the iterational data placement problem is to find a many-to-one mapping between data $d_j \in D$ and memory type $m_i \in M$ in a single loop iteration P, such that the total cost of memory accesses $Cost_{alloc^P(d_j)}(d_j)$ is minimized, where $\sum_{\forall d_j \in D, alloc^P(d_j)=m_i} Size_{d_j} \leq Size_{m_i}$.

In this paper, we assume all the data items have the same size. Thus, the size of a memory unit is denoted by the number of data items that can be stored in a memory unit.

5.2 A Greedy Algorithm for Loop Data Placement in Different Types of Memory Units

In this subsection, a greedy algorithm derived from Udayakumaran's algorithm is presented in Algorithm 5.1. This adapted greedy algorithm will be used as a baseline data placement method to be compared with the proposed optimal algorithms.

Algorithm 5.1 Greedy Algorithm for Loop Data Placement

Input: A set of loop data items D, a set of memory types M, m_0 is main memory, $Size_{m_i}$ $\forall m_i \in M$, read/write cost of data accesses $Read_{m_i}$ and $Write_{m_i}$ $\forall m_i \in M$, numbers of read/write operations for each data item RN_{d_j} and WN_{d_j} $\forall d_j \in D$, and suppose the initial placement for all d_j in D are in the main memory (m_0).

Output: A data placement under which the total cost for memory access of the execution for one iteration is minimized.

1: Compute cost function $Cost_{m_i}(d_j)$, $\forall d_j \in D$ and $\forall m_i \in M$. The value of $Cost_{m_i}(d_j)$ represents memory access cost when data item d_j is assigned to memory type m_i.
2: Sort data items $D = (d_1, d_2, \ldots, d_N)$ according to their accesses in the iteration, using insertion sort to make them in a descending order.
3: initial $totalcost \leftarrow 0$.
4: **for** $j \leftarrow 1$ to $Size_{m_1}$ **do**
5: Allocate d_j to memory type m_1.
6: $totalcost \leftarrow totalcost + Cost_{m_1}(d_j)$
7: **end for**
8: **for** $j \leftarrow 1+Size_{m_1}$ to $Size_{m_1}+Size_{m_2}+1$ **do**
9: Allocate d_j to memory type m_2.
10: $totalcost \leftarrow totalcost + Cost_{m_2}(d_j)$
11: **end for**
12: **for** $j \leftarrow Size_{m_1} + Size_{m_2} + 2$ to $Size_{m_1} + Size_{m_2} + Size_{m_3} + 2$ **do**
13: Allocate d_j to memory type m_3.
14: $totalcost \leftarrow totalcost + Cost_{m_3}(d_j)$
15: **end for**
16: ...
17: **for** $j \leftarrow Size_{m_1}+Size_{m_2}+\ldots+Size_{m_{T-3}}+Size_{m_{T-2}}+T-2$ to $Size_{m_1}+\ldots+Size_{m_{T-1}}+T-2$ **do**
18: Allocate d_j to memory type m_{T-1}.
19: $totalcost \leftarrow totalcost + Cost_{m_{T-1}}(d_j)$
20: **end for**
21: **for** $j \leftarrow Size_{m_1}+Size_{m_2}+\ldots+Size_{m_{T-1}}+T-1$ to N **do**
22: Allocate the rest d_j to main memory m_0.
23: $totalcost \leftarrow totalcost + Cost_{m_0}(d_j)$
24: **end for**

The original algorithm is to sort the data by the number of accesses in decreasing order, and then allocates the first SPM size of the data into the SPM. When the SPM is full, it allocates the rest of data in the main memory. Our extension for the derived Udayakumaran's algorithm allocates the first $Size_{m_1}$ data items in m_1, the next $Size_{m_2}$ data items in m_2, the next $Size_{m_3}$ data items in m_3, and so on. The data that does not fit in m_1, m_2, ..., m_T is allocated to the main memory. This algorithm can guarantee that data with larger number of accesses is allocated in the faster level of memory, and the total execution time seems the least because the execution time is only related with the data's number of accesses.

This algorithm cannot generate the optimal solution for the iterational data placement problem because it does not consider the initial states on the on-chip and off-chip memory units and the cost of updating array data to main memory. In some cases it is not necessary to move data. Rather than moving, keep data in the original place might achieve better results since the moving cost of data may offset the benefits gained. In the experiment section, we will compare this extension of the derived Udayakumaran's algorithm with our improved algorithm.

6 Iterational Optimal Data Placement

In this section, we present details of the improved algorithm. Based on the problem we defined in last section, we present the Iterational Optimal Data Placement (IODP) algorithm, which uses dynamic programming to achieve the optimal loop data placement in each iteration. In [20], Zhuge et al. proposed a dynamic programming approach named RODA (Regional Optimal Data Allocation) to solve data placement problems. The RODA algorithm has two limitations: one is that it can only be used for scalar data items. When there are array items, the algorithm cannot be used. The other is that it cannot be used in loops. Therefore, we propose the IODP algorithm, which is improved from RODA to overcome these two limitations.

6.1 Five Key Innovations of Iterational Optimal Data Placement Algorithm

In the proposed IODP algorithm, each iteration includes four steps. First, it computes the cost of each memory access. Second, it uses dynamic programming to decide optimal data placement. Third, it reallocates the data items. Fourth, it determines when to stop the algorithm.

The overall approach of loop data placement is that we decide an initial data placement before the beginning of first iteration. After each iteration in loop, we use our algorithm to get the optimal data placement. If the result of one data placement is the same with the result in the previous iteration, we could use the same data placement for this iterations.

Compared with the RODA algorithm, our improvement which do not mention in RODA can be seen in five innovations as follows:

6.1.1 (Innovation 1) Add Updating Cost in Cost Table of Memory Accesses

The cost of memory accesses $Cost_{m_i}(d_j)$ is computed for every data item d_j on all the memory types m_i by Definition 6, where d_j is in the set of data items $D = (d_1, d_2, \ldots, d_N)$, and m_i is in the list of memory types $M = (m_0, m_1, m_2, \ldots, m_{T-1})$. In M, the element m_0 is the off-chip main memory with the largest capacity while the elements $m_1, \ldots, m_{T-1}$ are on-chip memory types. The element m_1 indicates a memory type with the smallest capacity and lowest access cost when compared with the others, e.g. SPM. For array items we add an updating cost (Definition 4) to ensure that in the main memory the copy of the newest data can be accessed. The copy is treated as a backup since each data item in the array may be scattered to different memory types. With these arrays updated into main memory, we do not need to seek scattered data items on different memories. It is convenient to access them when they are stored continuously in the main memory.

Back to the example of iterational data placement in Section 4, the result of optimal data placement is shown in "Optimal" in "Iteration 1" of Table 3. Costs of memory accesses are computed in Table 5 based on the number of accesses provided in Table 2. In Table 3, all data items' initial location is based on the placement of "Initial Data Placement". For example, d_1 is a scalar data in main memory, its updating cost is 0. According to Eq. 3, $Cost_{m_0}(d_1) = 1 \times 50 + 0 + 0 = 50$, $Cost_{m_1}(d_1) = 1 \times 1 + 51 + 0 = 52$ and $Cost_{m_2}(d_1) = 1 \times 15 + 65 + 0 = 80$. For array data like d_5 in L2, it needs to be updated to main memory, according to Eq. 3, $Cost_{m_0}(d_5) = 3 \times 50 + 65 + 0 = 215$, $Cost_{m_1}(d_5) = 3 \times 1 + 16 + 51 = 70$ and $Cost_{m_2}(d_5) = 3 \times 15 + 0 + 65 = 110$.

Table 5 Cost table for the example in Iteration 1.

Data	$Cost_{m_0}(d_j)$	$Cost_{m_1}(d_j)$	$Cost_{m_2}(d_j)$
$d_1(d_1)$	50	52	80
$d_2(d_2)$	201	3	61
$d_3(A[i])$	100	53	95
$d_4(A[i-1])$	115	17	15
$d_5(A[i-2])$	215	70	110
$d_6(B[i])$	100	53	95
$d_7(B[i-1])$	165	69	95

6.1.2 (Innovation 2) Take the Dependence of Array Items to Reduce Cost of Memory Access

We have known that for array items, they usually should be updated to main memory after each iteration. To eliminate extra cost, it is better that an array item is only updated when it has to be. So we use the dependence of array items to reduce the updating cost. The specific method is that in each dependent array, only part of data items need to be updated to main memory. Take array A for example, in iteration p, data $A[p - dependence_A]$ is updated to main memory, and the other data items in array A don't need to do that.

6.1.3 (Innovation 3) Handle Array Items in Loop

Since we keep a copy of array in main memory in our technique, and in each iteration the number of data elements in an array is computable, we can treat these data items as scalar data now. For example, in one program iteration p, array $A[p]$, $A[p-1]$ and $A[p-2]$ are used, then we treat $A[p]$, $A[p-1]$ and $A[p-2]$ as three scalar data items. In each iteration, $A[p-2]$ is also additionally written into main memory to make sure that the copy of the array stored in main memory is updated correspondingly.

6.1.4 (Innovation 4) Reallocate the Array Items

Reallocating the array items guarantees that these array items' locations are still right in the next iteration of the loop. Take array B as an example, in iteration p, data $B[p]$ and data $B[p-1]$ are accessed; In iteration t, data $B[t]$ and data $B[t-1]$ will be accessed. Obviously, if $t = p+1$, then $B[p]$ becomes $B[t-1]$, $B[p-1]$ will never be accessed, and $B[t]$ is a new to be accessed. Certainly, the memory location of $B[t-1]$ should be the location of $B[p]$ instead of $B[p-1]$. Therefore, we have to reallocate the data items to the right location before the next iteration. In the example of iterational data placement in Section 4, from Table 3 we know that after the data placement of initial data placement, the optimal allocation is: $d2$ and $A[i-2]$ in SPM, $A[i]$, $A[i-1]$ and $B[i]$ in L2 memory, $d1$ and $B[i-1]$ in main memory. Then by reallocating the array items, the initial location of each data in iteration 1 changes to be: $d2$ in SPM, $A[i-1]$, $A[i-2]$ and $B[i-1]$ in L2 memory, and $d1$, $A[i]$, $B[i]$ in the main memory.

6.1.5 (Innovation 5) Determine When to Stop

Because the data access pattern is the same in each loop iteration, the algorithm will terminate after converging between iterations. At the end of the IODP algorithm in one iteration, we will compare the placement result with the previous iteration. If they are the same, then stop doing data placement algorithm in the loop forever, we have obtained the optimal data placement. At the later iterations of the loop, we will allocate data items according to the optimal result. In the experiment, we find that the algorithm always runs no more than 5 iterations to terminate in all the benchmark programs.

6.2 Iterational Optimal Data Placement Algorithm Using Dynamic Programming

Algorithm 6.1 Iterational Optimal Data Placement (IODP) Algorithm

Input: A set of loop data items D, a set of memory types M, and the initial placement for all d_j in D.
Output: A data placement under which the total cost of the execution for each iteration is optimally minimized.

```
1:  Compute cost function Cost_{m_i}(d_j), ∀d_j ∈ D and ∀m_i ∈ M
2:  for j ← 1 to |D| do
3:     TotalCost[j,Size_{m_1},Size_{m_2},...,Size_{m_{T-1}}] ← Σ Cost_{m_0}(d_j)
4:  end for
5:  for j ← 1 to |D| do
6:     for i_1 ← Size_{m_1} to 0 do
7:        ...
8:        for i_{T-1} ← Size_{m_{T-1}} to 0 do
9:           Compute TotalCost[j,i_1,i_2, ...,i_{T-1}] according to
             Figure 4 to get the minimum TotalCost.
10:       end for
11:       ...
12:    end for
13: end for
14: for j ← |D| to 1 do
15:    if TotalCost[j,i_1,i_2,...,i_{T-1}] = TotalCost[j-
       1,i_1,i_2,...,i_{T-1}] then
16:       location_{d_j} ← 0
17:       BackPath[j,i_1,i_2,...,i_{T-1}] = (j-1,i_1,i_2,...,i_{T-1})
18:       Continue
19: else if TotalCost[j,i_1,i_2,...,i_{T-1}] = TotalCost[j-
       1,i_1+1,i_2,...,i_{T-1}]-(Cost_{m_0}(d_j)- Cost_{m_1}(d_j)) then
20:       location_{d_j} ← 1
21:       BackPath[j,i_1,i_2,...,i_{T-1}] = (j-1,i_1+1,i_2,...,i_{T-1}) i_1
22:       ← i_1 + 1
23:       Continue
24:       ...
25: else if TotalCost[j,i_1,i_2,...,i_{T-1}] = TotalCost[j-
       1,i_1,i_2,...,i_{T-1}+1]-(Cost_{m_0}(d_j)-Cost_{m_{T-1}}(d_j)) then
26:       location_{d_j} ← T − 1
27:       BackPath[j,i_1,i_2,...,i_{T-1}] = (j-1,i_1,i_2,...,i_{T-1}+1)
28:       i_{T-1} ← i_{T-1} + 1
29:       Continue
30:    end if
31: end for
32: Reallocate the data items' location for the next iteration.
33: if Current placement = Previous placement then
34:    Stop IODP algorithm forever.
35: end if
```

The IODP algorithm consists of four major steps. (1) Lines 1-4 build a cost table of memory accesses $Cost_{m_i}(d_j)$ for $\forall d_j \in D$ on different memory types. (2) Lines 5-31

determine the optimal loop data placement with dynamic programming so that the cost of memory accesses during the execution for one iteration is minimized. (3) Line 32 reallocates data items for the current iteration to make sure that at the beginning of the next iteration, the placement of every data item still remains accurate. (4) Lines 33-35 compare the result with previous iteration to determine when to stop the algorithm. Since steps 1, 3 and 4 have been discussed in the five innovations, we will concentrate on step 2 in this subsection.

Since a cost table is built in step 1, the optimal data placement could be determined by using a multi-dimensional dynamic programming table. The structure of the table is as follows: the first dimension of the table is represented by a data item d_j, each one of the other dimensions is represented by the available space on a certain memory type $m_i \in M$ except m_0, which is assumed to be large enough to hold all the data items of the program.

Let $TotalCost[j, i_1, i_2, ..., i_{T-1}]$ represent each cell of dynamic programming space. It indicates the data item d_j is considered to be allocated to a certain memory type when i_k units of available space in memory type m_k where $m_k \in M$.

Suppose the placement of d_{j-1} has been optimally determined. The value of $TotalCost[j, i_1, i_2, ..., i_{T-1}]$ is computed as the minimal total cost of memory accesses for an iteration when data item d_j is allocated to a certain memory type m_i, and the remaining data items $(d_{j+1}, d_{j+2}, \ldots, d_N)$ reside in the memory type m_0. The recursive function of dynamic programming is shown in Fig. 4.

The Iterational Optimal Data Placement (IODP) algorithm is presented in Algorithm 6.1. Before the first iteration start, all data items are stored in the main memory m_0, cost table is computed when data move to other memory types. During the computation, $location_{d_j}$ and $BackPath[j, i_1, i_2, ..., i_{T-1}]$ are used to keep the intermediate data and trace back an optimal solution. The variable $location_{d_j}$ keeps d_j's location. The array $BackPath[j, i_1, i_2, ..., i_{T-1}]$ keeps a list of the state of data placement in one iteration before the data item d_j is allocated.

To illustrate the IODP algorithm, we construct a dynamic programming table for the example's iteration 1 in Section 4. Costs of memory accesses are provided in Table 5. Since there are three types of memory units, the dynamic programming table is constructed as a 3-D table of $TotalCost[j, i_1, i_2]$. The 3-D table consists of four 2-dimensional tables as shown in Table 6. Specially it is for the first iteration of the motivation example. The 2-D table of "$i_2 = 3$" computes the costs array for all the data items when the available space on memory type m_2 equals 3. The value of i_1 indicates the available space on memory type m_1. For example, we compute the cost cell $TotalCost[1, 2, 3]$ for data item d_1 when there are two available space in memory type m_1 and three in m_2. The value of the cost is shown in row "$i_1 = 2$" and column "$j = 1$" in the first 2-D table, which is 946. Then, we compute the cost cell $TotalCost[1, 1, 3]$ when d_1 is assigned to m_1, and the other data items have not be allocated. The value of the cost is shown in row "$i_1 = 1$" and column "$j = 1$" in the 2-D table of "$i_2 = 3$", which is 948. The cost shown in row "$i_1 = 1$" and column "$j = 1$" in the 2-D table "$i_2 = 2$" indicates the minimal cost that can be achieved when there is one available space in memory type m_1 and two available space in memory type m_2. We compute costs for data item

Table 6 The dynamic programming table that computes the cost array $C[j, i_1, i_2]$ for the example in Iteration 1 of Section 4.

i_1,j	1	2	3	4	5	6	7
$i_2 = 3$							
2	**946**	946	946	946	946	946	946
1	948	**748**	**748**	748	748	748	748
0	∞	750	701	650	603	603	603
$i_2 = 2$							
2	976	806	806	806	806	806	806
1	∞	778	743	**648**	643	643	643
0	∞	∞	731	601	503	503	503
$i_2 = 1$							
2	∞	836	801	706	701	701	701
1	∞	∞	773	643	**543**	543	543
0	∞	∞	∞	631	496	**496**	433
$i_2 = 0$							
2	∞	∞	831	701	601	601	601
1	∞	∞	∞	673	538	538	473
0	∞	∞	∞	∞	526	491	**426**

Figure 4 Recursive function.

$$
TotalCost[j, i_1, i_2, ..., i_{T-1}] = \begin{cases} \sum_j Cost_{m_0}(d_j), & \text{if } j = 0, \forall k = 1, 2, ..., T-1,\ i_k = Size_{m_k}, \\ \infty & \text{if } \sum_{k=1}^{C} i_k < \sum_{k=1}^{C} Size_{m_k} - j \text{ or } \\ & \exists k \in \{1, 2, 3, \ldots, T-1\}\ i_k > Size_{m_k}, \\ \min(TotalCost[j-1, i_1, i_2, ..., i_{T-1}], & \\ TotalCost[j-1, i_1+1, i_2, ..., i_{T-1}] - (Cost_{m_0}(d_j) - Cost_{m_1}(d_j)), & \\ TotalCost[j-1, i_1, i_2+1, ..., i_{T-1}] - (Cost_{m_0}(d_j) - Cost_{m_2}(d_j)), & \\ ..., & \\ TotalCost[j-1, i_1, i_2, ..., i_{T-1}+1] - (Cost_{m_0}(d_j) - Cost_{m_{T-1}}(d_j)). & \text{if } \sum_{k=1}^{T-1} i_k \geq \sum_{k=1}^{T-1} Size_{m_k} - j. \end{cases}
$$

d_1 in columns "$j = 1$" in all the 2-D tables in a similar way. To decide the placement of data item d_2, the algorithm finds the minimal cost when both d_1 and d_2 are allocated in either memory type m_1 or m_2. The minimal cost is found when d_1 is assigned to m_0 and d_2 is assigned to m_1, which is shown in row "$i_1 = 1$" and column "$j = 2$" in the 2-D table "$i_2 = 3$". The value is 748 as shown in Table 6. The final total cost of memory accesses with the optimal data placement is 426. The backtracking path indicated by bold-faced cell in Table 6 shows one of the optimal placement as follows: data items d_2 and $B[i]$ are assigned in memory units of type m_1. Data items $A[i-1]$, $A[i-2]$ and $B[i-1]$ are assigned in memory units of type m_2. Data items d_1 and $A[i]$ are assigned in memory units of type m_0.

The IODP algorithm's time complexity is $O(N \times Size_{m_1} \times Size_{m_2} \times \ldots \times Size_{m_T-1})$. N is the number of data items in D, and T is the constant of the number of various kinds of memory.

7 Experiment

In this section, the effectiveness of IODP algorithm is evaluated by the results of the experiments. We conduct the experiments by comparing costs of memory accesses for the following data placement techniques: 1) Pick data items accessed in the loops in the program randomly and allocate them to the available memory units. 2) Generate data placement for loops in the program by the greedy algorithm derived from Udayakumaran's. 3) Use the improved IODP algorithm to generate optimal loop data placement. All the techniques are evaluated by time cost and energy consumption for memory accesses with their generated placement.

7.1 Experimental Setup

We develop a custom simulator based on SimpleScalar to simulate the process of data placement and obtain costs of memory accesses for a program. It has a single-core processor with the architecture of three types of memory units: an on-chip SPM made with SRAM, an on-chip second-level SPM made with SRAM and a block of main memory made with DRAM. A set of parameters collected from CACTI tools provided by HP for the three memory types is shown in Table 7. With the use of CACTI tools [1], we can obtain the time latency and energy consumption for these three types of memory units.

We use the following benchmarks in our experiments: diff-ct1, N-2IIR, allpole, diff2, 4-lattice, 8-lattice, ellfilter, in2, ellENC, and voltera. These benchmarks are from the DSPstone benchmark [21]. Table 8 shows the separate number of scalar data and array data in each benchmark.

Table 7 System specification for spm, l2 memory and main memory.

Component	Description
CPU Core	Number of cores: 1, frequency: 1.0 GHz
SRAM SPM	Size: 8 KB, access latency: 0.477 ns, access energy: 0.0202 nJ
SRAM SPM	Size: 16 KB, access latency: 1.914 ns, access energy: 0.0310 nJ
Main memory	Size: 256 MB, access latency: 2.781 ns, access energy: 0.7849 nJ

We integrate our data placement techniques into a compiler. Then, we run the compiled benchmark programs on the simulator with the parameters obtained form CACTI tools. The simulator runs different data placement algorithms to compare the time cost and energy consumption in run time.

7.2 Experimental Results

Table 9 shows time costs of data placement with the configuration of three memory types. Column "random" displays time costs of memory accesses when data items are randomly picked and allocated to an on-chip SPM and on-chip 2nd level memory. Because of the randomness of technique, we conduct the experiment 10 times, and get the average value. Column "Uday" displays time costs of memory accesses with placement generated by the Udayakumaran's algorithm [19]. The percentage of improvement on time cost for the Udayakumaran's algorithm over the "random" technique is shown in column "Imprv (U/r)". The average improvement of Udayakumaran's algorithm for all the benchmark programs is 11.40 %. Column "IODP" displays time costs with placement generated by our IODP algorithm. The percentage of improvement for IODP over the "random" technique is shown in column "Imprv (I/r)". The average improvement of the IODP algorithm is 19.01 %.

Table 8 Benchmarks information.

Bench.	Data	Array
$diff-ct1$	15	40
$N-2IIR$	42	6
$allpole$	21	6
$diff2$	32	12
$4-lattice$	40	2
$8-lattice$	80	4
$ellfilter$	84	18
$ellENC$	54	14
$in2$	27	18
$voltera$	156	6

Table 9 Comparison of time costs for various data placement on memory architecture with three types of memory units.

Bench.	Uday(μs)	IODP(μs)	random(μs)	Imprv(I/r)	Imprv(U/r)	Imprv(I/U)	T
diff-ct1	379.716	332.881	418.095	20.38 %	9.18 %	12.33 %	4
N-2IIR	172.907	161.692	211.757	23.64 %	18.35 %	6.49 %	4
allpole	247.134	212.447	315.513	32.67 %	21.67 %	14.04 %	2
diff2	195.796	184.061	202.688	9.19 %	3.40 %	5.99 %	2
4-lattice	183.401	155.718	211.688	26.44 %	13.36 %	15.09 %	3
8-lattice	414.222	408.417	483.328	15.50 %	14.30 %	1.40 %	4
ellfilter	380.193	369.574	421.826	12.39 %	9.87 %	2.79 %	5
ellENC	350.628	329.875	385.481	14.43 %	9.04 %	5.92 %	2
in2	130.688	112.593	135.021	16.61 %	3.21 %	13.85 %	5
voltera	525.114	481.737	594.024	18.90 %	11.60 %	8.26 %	3
Average				19.01 %	11.40 %	8.62 %	

Column “Imprv (I/U)” displays the average improvement for “IODP” over “Uday” is 8.62 %. As shown in the experimental results, our IODP algorithm achieves the best improvement of time costs on average among all other techniques. In the best case, e.g. 4-lattice, the percentage of time improvement reaches 15.09 % over Udayakumaran’s algorithm. Column “T” shows the iteration in which IODP terminates. Among all the benchmarks, the IODP algorithm always runs no more than 5 iterations.

Data access time is reduced because of the effective solution of data placement, so does the energy consumption. Table 10 compares the energy consumption of various data placement solutions generated by various techniques. Accordingly, column “random” displays energy consumption when data items are randomly picked and allocated. Column “Uday” displays the energy consumption with placement generated by the Udayakumaran’s algorithm. The average improvement of Udayakumaran’s algorithm over random technique is 12.06 %, which can be seen in column “Imprv(U/r)”. Column “IODP” indicates the energy consumption when we use IODP algorithm to allocate data items. The average improvement of IODP algorithm over random technique is 20.04 % in column “Imprv(I/r)”, and the average improvement of IODP algorithm over Udayakumaran’s algorithm is 8.98 % in column “Imprv(I/U)”. According to the table, our technique of IODP algorithm for data placement also achieves the best improvement of energy consumption. In the best case, e.g. 4-lattice, the percentage of energy improvement is 17.56 % over Udayakumaran’s algorithm.

The comparison bar-graph is presented in the Fig. 5, we can see the difference of time costs and energy consumption among three algorithms clearly.

While considering the additional cost to compute the data placement, we need to note that the cost of computing the placement is incurred during compile time, instead of run time. And it is a one-time cost. All the improvements by the IODP algorithm refer to run-time improvement. The

Table 10 Comparison of energy consumption for various data placement on memory architecture with three types of memory units.

Bench.	Uday(μJ)	IODP(μJ)	random(μJ)	Imprv(I/r)	Imprv(U/r)	Imprv(I/U)	T
diff-ct1	126.986	116.222	132.057	11.99 %	3.84 %	8.48 %	4
N-2IIR	58.219	55.229	63.649	13.23 %	8.53 %	5.14 %	3
allpole	79.301	68.692	95.091	27.76 %	16.61 %	13.38 %	2
diff2	55.465	52.909	68.477	22.73 %	19.00 %	4.61 %	2
4-lattice	58.753	48.437	64.221	24.58 %	8.51 %	17.56 %	3
8-lattice	116.486	105.747	125.418	15.68 %	7.12 %	9.22 %	3
ellfilter	127.006	116.238	137.255	15.31 %	7.47 %	8.48 %	5
in2	42.649	39.643	58.366	32.08 %	26.93 %	7.05 %	5
ellENC	95.725	89.534	121.495	26.31 %	21.21 %	6.47 %	3
voltera	198.311	179.547	201.172	10.75 %	1.42 %	9.46 %	2
Average				20.04%	12.06 %	8.98 %	

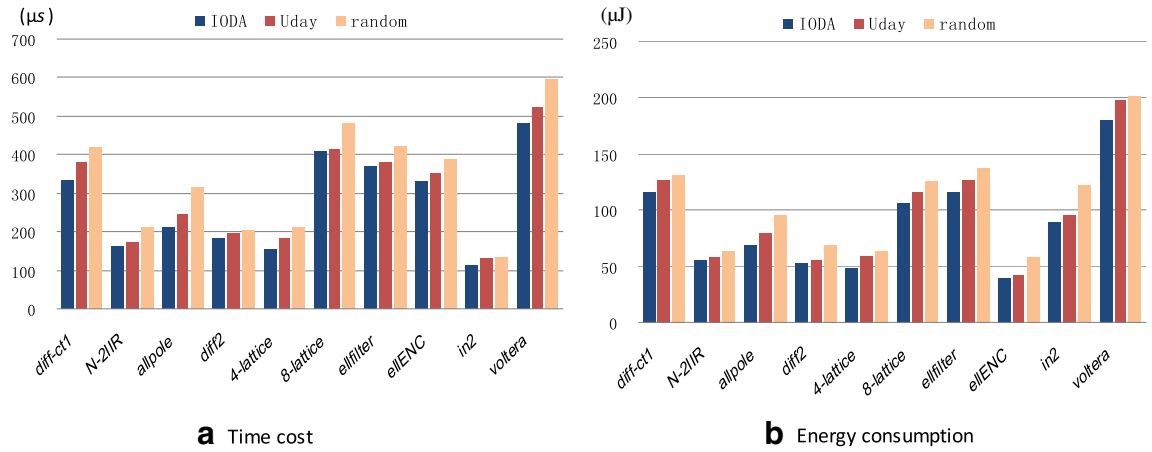


Figure 5 Time costs and energy consumption of memory access among three algorithms.

comparison of compile time cost for each benchmark is presented in the Table 11. We can see the difference of compile time cost of three algorithms clearly. In all the benchmarks, the proposed IODP takes the most time cost to get the data placement. However, it is a one-time cost. This cost will not affect the run time cost.

In all the experiments, IODP algorithm always obtains better improvements compared with the Udayakumaran's algorithm which employs a greedy strategy. There are several advantages in our techniques compared with the Udayakumaran's algorithm. First, the Udayakumaran's algorithm does not consider the initial data placement of a loop. Therefore, it does not generate the optimal solution for iterational data placement, while the IODP technique considers the effect of moving for the initial data placement. Second, the Udayakumaran's algorithm does not consider the effect of updating the array data into the main memory, while our technique carefully considers the cost of updating for minimizing the total cost.

Table 11 Comparison of compile time cost for each benchmark.

Bench.	Uday(ms)	IODP(ms)	Random(ms)
diff-ct1	337	786	245
N-2IIR	234	304	181
allpole	210	406	179
diff2	196	294	135
4-lattice	127	227	86
8-lattice	458	639	335
ellfilter	243	466	142
in2	318	359	216
ellENC	524	572	335
voltera	636	863	424

8 Conclusion

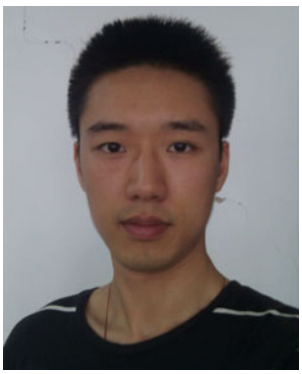
In this paper, we present an optimal data placement algorithm to efficiently minimize the cost of memory accesses for array data in loops considering memory architectures with multiple types of memory units. A dynamic programming algroithm, the Iterational Optimal Data Placement algorithm is proposed to solve data placement problem for both scalar and array data in loops by taking advantages of data dependencies among loop iterations. It achieves optimal data placement for each loop iterations. The experimental results show that IODP algorithm performs better than the greedy strategy in terms of reducing both energy and time costs. The average improvement of IODP algorithm over "random" technique is 20.04 % on energy consumption. The time cost is reduced by 8.98 % on average compared with greedy approach. In terms of time cost, the IODP algorithm achieves an average reduction of 19.01 % compared with the "random" technique. It also reduces total time cost of data accesses by 8.62 % on average compared with greedy approach.

Acknowledgments This work is partially supported by NSF CNS-1015802, Texas NHARP 009741-0020-2009, NSFC 61173014, National 863 Program 2013AA013202, Chongqing cstc2012ggC40005.

References

1. Cacti model. http://www.hpl.hp.com/research/cacti/.
2. Absar, M., & Catthoor, F. (2005). Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *Design, automation and test in Europe, 2005. Proceedings* (pp. 1162–1167). IEEE.
3. Avissar, O., Barua, R., Stewart, D. (2002). An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM*

Transactions on Embedded Computing Systems (TECS), *1*(1), 6–26.
4. Banakar, R., Steinke, S., Lee, B., Balakrishnan, M., Marwedel, P. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on hardware/software codesign* (pp. 73–78). ACM.
5. Chen, G., Ozturk, O., Kandemir, M., Karakoy, M. (2006). Dynamic scratch-pad memory management for irregular array access patterns. In *Proceedings of the conference on design, automation and test in Europe: proceedings* (pp. 931–936). European Design and Automation Association.
6. Chen, Z., Qiu, M., Niu, J., Lu, Z., Zhu, Y. (2012). Data allocation using genetic algorithm for MPSoc systems with hybrid scratch-pad memory. In *Proceedings of the 18th IEEE real time and embedded technology and applications symposium(RTAS)* (pp. 61–64).
7. Dominguez, A., Udayakumaran, S., Barua, R. (2005). Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, *1*(4), 521–540.
8. Hu, J., Xue, C., Tseng, W., He, Y., Qiu, M., Sha, E. (2010). Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation. In *Design automation conference (DAC), 2010 47th ACM/IEEE* (pp. 350–355). IEEE.
9. Hu, J., Xue, C., Tseng, W., Zhuge, Q., Sha, E. (2010). Minimizing write activities to non-volatile memory via scheduling and recomputation. In *Application specific processors (SASP), 2010 IEEE 8th symposium on* (pp. 101–106). IEEE.
10. Hu, J., Xue, C., Zhuge, Q., Tseng, W., Sha, E. (2011). Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *Design, automation & test in europe conference & exhibition (DATE)* (pp. 1–6). IEEE.
11. Ozturk, O., Kandemir, M., Kolcu, I. (2006). Shared scratch-pad memory space management. In *Quality electronic design, 2006. ISQED'06. 7th international symposium on* (pp. 6–12). IEEE.
12. Ozturk, O., Kandemir, M., Narayanan, S. (2008). A scratch-pad memory aware dynamic loop scheduling algorithm. In *Quality electronic design, 2008. ISQED 2008. 9th international symposium on* (pp. 738–743). IEEE.
13. Panda, P., Dutt, N., Nicolau, A. (1997). Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on design and test* (p. 7). IEEE Computer Society.
14. Panda, P., Dutt, N., Nicolau, A. (2000). On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *5*(3), 682–704.
15. Qiu, M., & Sha, E.H.M. (2009). Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *14*(2), 1–30.
16. Sjödin, J., & Von Platen, C. (2001). Storage allocation for embedded processors. In *Proceedings of the 2001 international conference on compilers, architecture, and synthesis for embedded systems* (pp. 15–23). ACM.
17. Tseng, W., Xue, C., Zhuge, Q., Hu, J., Sha, E. (2010). Optimal scheduling to minimize non-volatile memory access time with hardware cache. In *VLSI system on chip conference (VLSI-SoC), 2010 18th IEEE/IFIP* (pp. 131–136). IEEE.
18. Udayakumaran, S., & Barua, R. (2003). Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the 2003 international conference on compilers, architecture and synthesis for embedded systems* (pp. 276–286). ACM.
19. Udayakumaran, S., Dominguez, A., Barua, R. (2006). Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, *5*(2), 472–511.
20. Zhuge, Q., Guo, Y., Hu, J., Tseng, W., Xue, S., Sha, E. (2012). Minimizing access cost for multiple types of memory units in embedded systems through data allocation and scheduling. *Signal Processing, IEEE Transactions on*, *60*(6), 3253–3263.
21. Zivojnovic, V., Velarde, J., Schlager, C., Meyr, H. (1994). Dspstone: a dsp-oriented benchmarking methodology. In *Proceedings of the international conference on signal processing and technology*.



Jun Zhang is currently a PHD candidate at Chongqing University. He received the bachelor's and master's degrees in computer science in 2011 and 2013, respectively, from Hunan University, Changsha, China. His research interests include embedded systems, real-time systems, parallel architectures, compiler optimization, information security, and fault tolerance.

Tan Deng is currently a PHD candidate at Hunan University. Tan Deng received the bachelor's and master's degrees in software engineering in 2009 and 2011, respectively, from Xiamen University, Xiamen, China. His research interests include parallel and distributed system, real-time computing, and embedded system.

Qiuyan Gao received her BSc in information security from Hunan University, China, in 2011. She is currently pursuing her Master degree at Hunan University, Changsha, China. Her research interests include parallel architectures, embedded multi-core systems and optimization algorithms.

Dr. Qingfeng Zhuge recieved her Ph.D. from the Department of Computer Science at the University of Texas at Dallas in 2003. She obtained her BS and MS degrees in Electronics Engineering from Fudan University, Shanghai, China. She is currently a professor at Chongqing University, China. She received Best Ph.D. Dissertation Award in 2003. She has published more than 70 research articles in premier journals and conferences. Her research interests include parallel architectures, embedded systems, supply-chain management, real-time systems, optimization algorithms, compilers, and scheduling.

Dr. Edwin H.-M. Sha received Ph.D. degree from the Department of Computer Science, Princeton University, USA in 1992. From August 1992 to August 2000, he was with the Department of Computer Science and Engineering at University of Notre Dame, USA. Since 2000, he has been a tenured full professor in the Department of Computer Science at the University of Texas at Dallas. Since 2012, he served as the Dean of College of Computer Science at Chongqing University, China.

He has published more than 300 research papers in refereed conferences and journals. He has served as an editor for many journals, and as program committee and Chairs for numerous international conferences. He received Teaching Award, Microsoft Trustworthy Computing Curriculum Award, NSF CAREER Award, and NSFC Overseas Distinguished Young Scholar Award, Chang Jiang Honorary Chair Professorship and China Thousand-Talent Program.