SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **41**:737–752
Published online 28 October 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1020

Compiler-assisted dynamic scratch-pad memory management with space overlapping for embedded systems

Yanqin Yang[1,2], Haijin Yan[3], Zili Shao[4,*,†] and Minyi Guo[1]

[1]*Department of Computer Science and Engineering, Shanghai Jiao-Tong University, Shanghai 200240, People's Republic of China*
[2]*Department of Computer Science and Technology, East China Normal University, Shanghai 200241, People's Republic of China*
[3]*Motorola Inc., Chicago, IL, U.S.A.*
[4]*Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong*

SUMMARY

Scratch-pad memory (SPM), a small, fast, software-managed on-chip SRAM (Static Random Access Memory) is widely used in embedded systems. With the ever-widening performance gap between processors and main memory, it is very important to reduce the serious off-chip memory access overheads caused by transferring data between SPM and off-chip memory. In this paper, we propose a novel compiler-assisted technique, ISOS (**I**teration-access-pattern-based **S**pace **O**verlapping **S**PM management), for dynamic SPM management with DMA (Direct Memory Access). In ISOS, we combine both SPM and DMA for performance optimization by exploiting the chance to overlap SPM space so as to further utilize the limited SPM space and reduce the number of DMA operations. We implement our technique based on IMPACT and conduct experiments using a set of benchmarks from DSPstone and Mediabench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique achieves run-time performance improvement compared with the previous work. The average improvements are 13.15, 19.05, and 25.52% when the SPM sizes are 1KB, 512 bytes, and 256 bytes, respectively. Copyright © 2010 John Wiley & Sons, Ltd.

Received 28 April 2009; Revised 19 August 2010; Accepted 20 August 2010

KEY WORDS: compiler; scratch-pad memory; embedded system

1. INTRODUCTION

The ever-widening performance gap between CPU and off-chip memory requires effective techniques to reduce memory accesses. To alleviate the gap, scratch-pad memory (SPM), a small, fast, software-managed on-chip SRAM (Static Random Access Memory) is widely used in embedded systems [1–5] with its advantages in energy and area [6–9]. A recent study [10] shows that SPM has 34% smaller area and 40% lower power consumption than a cache of the same capacity. As the cache typically consumes 25–50% of the total energy and area of a processor, SPM can help to significantly reduce the energy consumption for embedded processors. Embedded software is usually optimized for specific applications, hence we can utilize SPM to improve the performance and predictability by avoiding cache misses. With these advantages, SPM has been widely used

*Correspondence to: Zili Shao, Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong.
†E-mail: cszlshao@comp.polyu.edu.hk

in embedded systems. However, it poses a huge challenge for the compiler to fully explore SPM since it is completely controlled by software.

To effectively manage SPM, two kinds of compiler-managed methods have been proposed: static methods [6, 8, 10–17] and dynamic methods [1, 18–30]. Basically, based on the static SPM management, the content in SPM is fixed and is not changed during the running time of applications. With the dynamic SPM management, the content of SPM is changed during the running time based on the behavior of applications. For dynamic SPM management, it is important to select an effective approach to transfer data between off-chip memory and SPM. This is because the latency of off-chip memory access is about 10–100 times of that of SPM [1, 6, 18, 30], and many embedded applications in image and video processing domains have significant data transfer requirements in addition to their computational requirements [9, 31, 32]. To reduce off-chip memory access overheads, the dedicated cost-efficient hardware, DMA (Direct Memory Access) [33], is used to transfer data. The focus of this paper is on how to combine SPM and DMA in dynamic SPM management for optimizing loops that are usually the most critical sections in some embedded applications, such as DSP and image processing.

Our work is closely related to the work in [20, 29, 34–37]. In [20], Kandemir *et al*. proposed a dynamic SPM technique for loops that can determine memory layouts and best loop access patterns, partition the available SPM space, and restructure the code for explicit data transfer. In [29], DMA is applied for data transfer between SPM and off-chip memory by applying graph coloring for SPM management. In [34, 35], a two-level loop tiling technique with partitioning and pre-fetching is proposed for optimizing loop nests. The technique focuses on overlapping memory latency with data pre-fetching. In [36, 37], array folding was first proposed as a means to reduce the size of temporary arrays. Basically, the technique is to linearize a temporary array in some canonical way and fold it with a modulo operation; in such a way, the size of the array can be reduced and a memory cell can be reused when it contains a dead value, i.e. a value no longer used. The above work, however, does not consider optimizing DMA transfer. Because SPM is a small on-chip memory, we cannot put all the necessary data at one time taking the power and size of embedded systems into account. Therefore, multiple DMA transfers are needed for arrays in loops in dynamic SPM management and the pre-fetching. Considering the overhead caused by DMA operations, it becomes an important research issue to reduce the number of DMA operations.

In this paper, we propose a novel technique called ISOS (**I**teration-access-pattern-based **S**pace **O**verlapping **S**PM management) for SPM management by transferring blocks of data based on iteration access patterns. The basic idea is that the SPM space of some array elements can be overlapped if the array elements are not used in later iterations. In this way, if there are spaces that can be overlapped, we can reduce the number of DMA operations so as to change the ratio between array elements allocated to the SPM and DMA operations performed. We implement our technique based on IMPACT [38] and conduct experiments using the benchmarks from DSPstone and Mediabench on the cycle-accurate VLIW simulator of Trimaran [39]. The experimental results show that ISOS achieves significant run-time performance improvement compared with the previous work [20].

The remainder of this paper is organized as follows. In Section 2, we present the system model and the basic concepts. Our ISOS technique is proposed in Section 3. We present the experiments and conclusion in Sections 4 and 5, respectively.

2. MODEL AND BASIC CONCEPTS

2.1. System model

The system model is shown in Figure 1 which has the similar architecture to [18, 20]. The system consists of CPU, SPM, DMAC (Direct Memory Access Controller), and off-chip memory. On-chip SPM can be accessed by CPU through on-chip bus, and DMA is used to transfer data between SPM and off-chip memory. CPU can directly access data in off-chip memory through the system bus, and the access time is much bigger than that between CPU and SPM. A DMAC may have more

DOI: 10.1002/spe

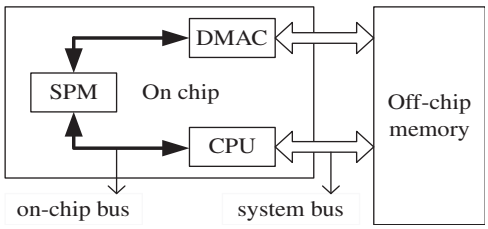

Figure 1. The system model.

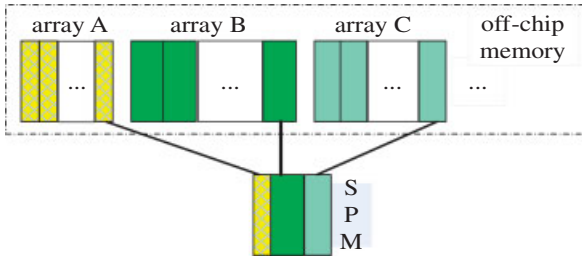

Figure 2. The data transfer between the SPM and off-chip memory.

than one DMA channels, and every channel can be used to transfer a block of data. The block-level data transfer between CPU and SPM is controlled by setting DMA control registers with source and destination accesses and data size. Thus, we can use the compiler to insert instructions to control DMA operations for data transfer. In this paper, we assign DMA channels based on a simple as-early-as-possible scheduling scheme, i.e. an available DMA channel will be assigned to the earliest DMA request in the request queue.

In this paper, we focus on array allocation in SPM. Each array is divided into blocks, and block-level data are transferred between SPM and off-chip memory through DMA. Only necessary data blocks that are accessed by a set of iterations are put into SPM. Therefore, we can more effectively utilize the space of SPM which is usually small.

An example is given in Figure 2. Based on iteration access patterns, we divide each array into blocks and allocate space for each array in the SPM. At one time, for one array, only one block of data is put into the SPM. In Sections 3.2 and 3.3, we will discuss how to determine the size of data blocks and insert DMA operations for block-level data transfer.

Based on the model in [18, 20], the cost of transferring a data block between SPM and off-chip memory is approximated by $(Cdi + Cdt \times n)$ in cycles, where Cdi is the initialization cost of DMAC for one block, including all the latencies of arbitration and synchronization, Cdt is the cost per byte transfer, and n is the number of bytes in a block. The total cost of transferring an array is approximated by $Nb \times (Cdi + Cdt \times n)$, where Nb is the number of blocks of an array. In our approach, we can reduce Nb by exploiting the space overlapping among arrays in iterations. The basic idea is that if we can allocate more space for arrays, then we can put more array elements into one block in such a way that the total number of blocks for an array is reduced.

2.2. *Space overlapping*

In this section, we use an example to illustrate our basic idea for space overlapping. Suppose we need to allocate space for four arrays, A, B, C, D, for a given loop in an application. We have six DMA channels. Based on the iteration-level data access pattern in the loop, we find that array A is read-only and array B is write-only. In Figure 3(a), we show the ordinary space allocation and DMA settings for the arrays A, B, C, and D. We allocate space for all the arrays and call DMA operations for data transfer. Since array A is read only, after one data have been used and will not be used again, we can use that space to hold the data generated by array B. In this way, by a

DOI: 10.1002/spe

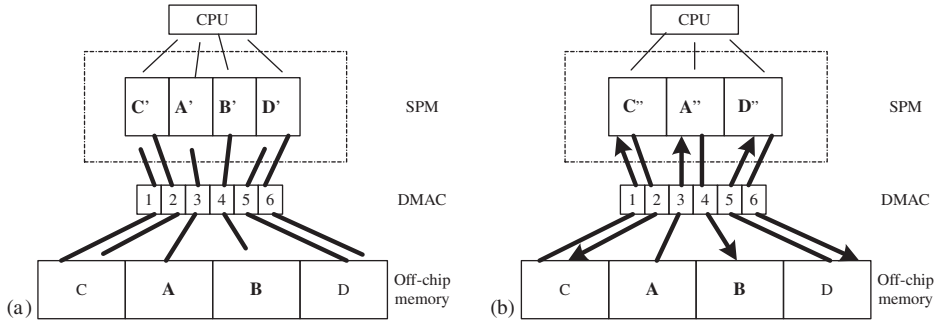

Figure 3. Two SPM management strategies: (a) the SPM space allocation without space overlapping and (b) the SPM space allocation with space overlapping by ISOS.

complete space overlapping, we do not need to assign space for array B. In Figure 3(b), we show the case for completely overlapping arrays A and B. With such an overlapping, we can assign more space to other arrays so as to utilize the space of the SPM more effectively.

3. THE ISOS TECHNIQUE

In this section, we propose our ISOS technique. We first give an overview of our technique in Figure 4 and then provide the details for each important step of ISOS from Sections 3.1 to 3.3, respectively.

The ISOS technique mainly consists of the following three steps as shown in Figure 4:

- *Step* 1, array classification: Based on iteration access patterns of arrays, we first classify arrays into four groups: write-only arrays, read-only arrays, write-advance-read arrays, and others. The SPM spaces of read-only arrays may be utilized for space overlapping for write-only arrays or write-advance-read arrays.
- *Step* 2, space overlapping exploration: In this step, we discuss how to conduct space overlapping. We first identify the space overlapping between the space of a read-only array for the prior iterations and the space of a write-only or write-advance-read array for the successive iterations and then compute the number of iterations in a block.
- *Step* 3, code transformation: We propose an algorithm to generate the transformed code by applying space overlapping. We insert instructions for DMA transfer and transform array references to map the space overlapping.

3.1. Step 1: Array classification

In ISOS, we first classify arrays into four different groups based on memory access patterns. To achieve this, for each array in a loop, its corresponding memory operations are collected. Then we classify arrays into the following groups:

- *read-only arrays*: For an array, if all of its memory operations in the loop are load operations, then this array is a read-only array;
- *write-only arrays*: For an array, if all of its memory operations in the loop are write operations, then this array is a write-only array.
- *write-advance-read arrays*: For an array, if the following two conditions are satisfied for all of its memory operations in the loop, this array is a write-advance-read array: (1) if there are both read and write (load/store) operations for the same memory location, the first operation must be the write operation and (2) the reference range of all the array elements related to the read operations must not be larger than the reference range of all the array elements related to the write operations in the loop.
- *others*: For an array, if it is not in any one of the above three groups, then it is in others.

DOI: 10.1002/spe

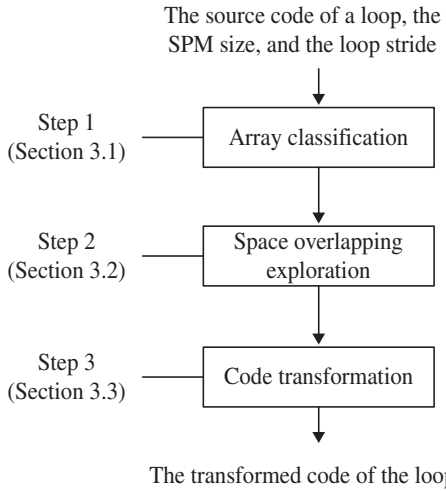

Figure 4. The overview of the ISOS technique.

```
for (i=2; i<242; i++)
{
    W[i]=X[i]-W[i-1]-W[i-2];
    Y[i]=W[i]+W[i-1]+W[i-2];
}
```

Figure 5. A loop kernel.

An example is given in Figure 5 in which a loop kernel is shown which is extracted from the IIR biquad filter in the DSP benchmark by neglecting the constant coefficient of the array elements. There are three arrays, W, X, and Y, in the loop. Based on the above array classification, X is a read-only array because it has only read operations; Y is a write-only array because it has only write operations; W is a write-advance-read array because the two conditions are satisfied, which means that the values read by all read operations are generated by the write operations in the current/previous iterations or outside the loop.

3.2. Step 2: Space overlapping exploration

Based on the array classification above, we can then perform space overlapping. The space allocated for read-only arrays may be utilized for space overlapping for write-only or write-advance-read arrays. In this section, we first propose how to select candidate arrays and discuss how to determine the size of data block.

We first determine how to select candidate arrays from read-only arrays for possible space overlapping. The basic idea of space overlapping is that the SPM space of some array elements can be overlapped after the array elements are not used in later iterations. Therefore, a candidate array must have the following features: part or all of the memory space allocated for the array in one iteration will not be used later, and the unused spaces from consecutive iterations form a consecutive address space. For each qualified array, we record the memory size we can reuse and put the array with the size into a set called *Reused_Array_Set*.

To select candidate arrays from write-only or write-advance-read arrays, we need to consider the block-level data transfer. With the block-level data transfer, we need to write a block of data with consecutive memory addresses back to the off-chip memory from the SPM by DMA after we finish the execution of a set of iterations for a loop. Therefore, for a candidate array selected from write-only or write-advance-read arrays, the related data generated in iterations must be put

DOI: 10.1002/spe

in consecutive memory accesses. Based on this, for each qualified array, we record the memory size we need and put the array with the size into a set called *Overlapping_Array_Set*.

After we obtain *Reused_Array_Set* and *Overlapping_Array_Set*, we try to find all possible array pairs for space overlapping. In order to maximally overlap space, in ISOS, we sort the arrays in *Overlapping_Array_Set* in descending order based on the memory sizes they need. Then based on the order, for each array from *Overlapping_Array_Set*, we find the best array from *Reused_Array_Set* whose memory size we can reuse is not less than, and is the closest to the size of the array from *Overlapping_Array_Set*. If we can find such a pair of arrays, we remove them from *Reused_Array_Set* and *Overlapping_Array_Set*, respectively, and then put them into a set called *Array_Pair_Set*. The above steps are repeated until all arrays in *Overlapping_Array_Set* have been checked.

After this step, we can calculate how many iterations we can put into one block based on the memory required for all arrays and the SPM size. Assume that arrays in a loop are in the form, $X[f(i)] = X[coef(X) \times i + offset(X)]$, where X is an array, the subscripts expression $f(i)$ is an affine function of loop index i, and $coef(X)$ and $offset(X)$ are the coefficient and offset, respectively. Taking the above overlapping space into account, we have

$$S_{spm} = \sum_{i=1}^{Npair} S_{ABi} + \sum_{j=1}^{Na-2*Npair} S_{C_j} \tag{1}$$

Here, S_{spm} is the total size of SPM, Na is the number of arrays in a loop, N_{pair} is the number of array pairs that can overlap SPM space with each other, S_{ABi} is the SPM size for arrays Ai and Bi (with space overlapping), and S_{Cj} is the SPM space for array Cj in a block (the SPM space of array Cj cannot be overlapped with other arrays).

Let $S_{AiBioverlap}$ be the overlapped space between array Ai and array Bi. We have

$$S_{ABi} = S_{Ai} + S_{Bi} - S_{AiBioverlap} \tag{2}$$

According to Equations (1) and (2), we have

$$S_{spm} = \sum_{j=1}^{Na} S_{Cj'} - \sum_{i=1}^{Npair} S_{AiBioverlap} \tag{3}$$

Without overlapping space, we have

$$S_{spm} = \sum_{j=1}^{Na} S_{Cj''} \tag{4}$$

Considering Equations (3) and (4), we find that $S_{Cj'}$ can be bigger than $S_{Cj''}$, so that we can allocate more elements of each array into SPM when space overlapping is applied. We assume that all the array elements occupy the same SPM size, and memory size is expressed by the number of array elements we can place into. Then in Equations (1) and (2), S_{Cj} and $S_{AiBioverlap}$ can be computed as

$$S_{Cj} = upper\_bound(Cj) - lower\_bound(Cj) + 1 + (N_{it} - 1) \times coef(Cj) \times k \tag{5}$$

$$S_{AiBioverlap} = (N_{it} - 1) \times \min(coef(Ai), coef(Bi)) \times k \tag{6}$$

Here, *upper_bound*(Cj) is the maximum offset of *Cj*, *lower_bound*(*Cj*) is the minimum offset of *Cj*, and k is the loop stride. Correspondingly, we can compute the number of iterations in a block, N_{it}, as

$$N_{it} = \frac{S_{spm} - \sum_{j=1}^{Na} bound(Cj')}{\sum_{j=1}^{Na} coef(Cj') \times k - \sum_{i=1}^{Npair} \min(coef(Ai), coef(Bi)) \times k} + 1 \tag{7}$$

Here, $bound(Cj') = upper\_bound(Cj') - lower\_bound(Cj') + 1$.

DOI: 10.1002/spe

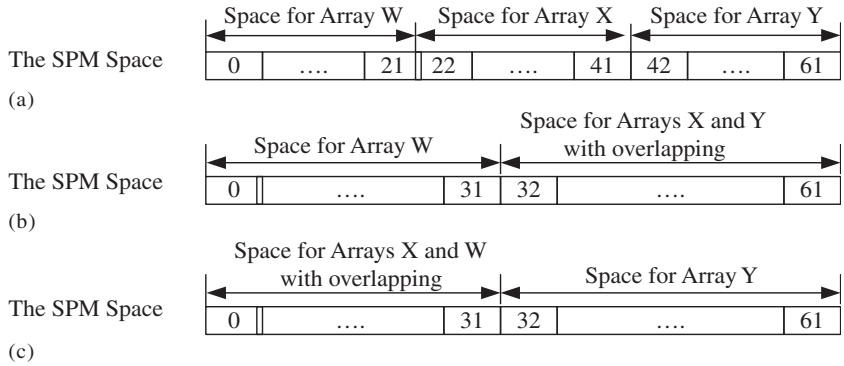

Figure 6. The memory layouts of three cases: (a) no space overlapping; (b) space overlapping between array X and array Y; and (c) space overlapping between array X and array W.

If space overlapping is not applied, we have the number of iterations in a block, N'_{it}, as

$$N'_{it} = \frac{S_{spm} - \sum_{j=1}^{Na} bound(Cj'')}{\sum_{j=1}^{Na} coef(Cj'') \times k} + 1. \tag{8}$$

Equations (7) and (8) can be used to determine how to compute the number of iterations in a block, which means how many iterations we can group together by transferring data together by DMA.

In Figure 6, we give an example for the space overlapping of the loop shown in Figure 5. Here, we assume that the size of the SPM is 62, which means that the SPM can contain at most 62 array elements. Figure 6(a) shows the case without space overlapping among arrays W, X, and Y. In this case, each array needs the SPM space. Therefore, we allocate 22 elements to array W, and 20 elements to X and Y, respectively, and accordingly the block size is 20 iterations.

Figure 6(b) shows the case that the unused space of array X is utilized for space overlapping for the space of array Y. In Figure 6(b), we do not allocate the space for Y initially because we can utilize the unused space of array X during the execution. Therefore, we can allocate more space (31 elements) to each of arrays X and W, and accordingly put more iterations into one block. In this way, we can reduce the number of DMA operations to improve the performance. During the execution, in each iteration, we can put the element of array Y generated in each iteration into the unused space allocated for X. Similarly, we can implement space overlapping between arrays X and W as shown in Figure 6(c).

3.3. Step 3: Code transformation

Based on the space overlapping exploration in Section 3.2, we can perform the code transformation to implement space overlapping. The algorithm is shown in Figure 7.

Based on the number of iterations for one block and *Array_Pair_Set*, the code transformation consists of the following steps: (1) change array references based on each array pair in *Array_Pair_Set*; (2) transfer the loop into the two-level loop; and (3) insert DMA operations for data transfer between SPM and off-chip memory. As mentioned in Section 3.2, when we apply space overlapping, we pick up the candidate arrays from read-only arrays and write-only/write-advance-read arrays and put them into *Array_Pair_Set*.

For an array pair in *Array_Pair_Set*, the SPM space of the read-only array is not less than the space needed by the write-only/write-advance-read array based on our selection criteria; therefore, in our code transfer, we first replace the write-only/write-advance-read array by their corresponding read-only array for each array pair in *Array_Pair_Set*. In this way, we can utilize the space of read-only arrays. Correspondingly, we need to transfer initial values between two arrays in an array pair. Next, we allocate SPM space for each array in the loop. If an array can utilize the space of an other array, it will be replaced in the first step. Therefore, we do not need to allocate space

DOI: 10.1002/spe

Input: *Array_Pair_Set, the original code for a loop, and the iteration number N in one block.*
Output: The transformed code.
Algorithm:

1. **for** each array pair <*Ai, Bi*> in *Array_Pair_Set*
2. **do** Change *Bi* to *Ai in the loop;*
3. Add the prologue for initial value assignment;
4. **for** each array *Ai* in the *loop*
5. **do** The minimum off-chip memory address of array element of array *Ai* is mapped to the initial SPM address to store array *Ai;*
6. **for** each N iterations
7. **do** Insert the DMA instructions to transfer a block of data;
8. Transfer the given loop into a two-level loop by executing *N* iteration in the inner loop;
9. Insert the DMA instructions to store data back to off-chip memory;

Figure 7. The code transformation algorithm.

```
Ws[0] = 1; Ws[1] = 2;
for( ie = 2; ie < 242; ie = ie+20)
{
 ie' = ie mod 20 ;
 DMA( AD_X[ie], AS_Xs[ie' -2], 20) ;
 DMA ready
 for ( ib = 2 ; ib < 22 ; ib ++)
 {
  Ws[ib] = Xs[ib-2]-Ws[ib-1]-Ws[ib-2];
  Ys[ib-2] = Ws[ib]+Ws[ib-1]+Ws[ib-2];
 }
 DMA(AS_Ws[ie' ], AD_W[ie], 20);
 DMA(AS_Ys[ie' -2], AD_Y[ie], 20);
 DAM ready
 Ws[0] = Ws[ib-1]; Ws[1] = Ws[ib];
}
```

(a)

```
Ws[0] = 1; Ws[1] = 2;
for( ie = 2; ie < 242; ie = ie+30)
{
 ie' = ie mod 30 ;
 DMA( AD_X[ie], AS_Xs[ie' -2], 30) ;
 DMA ready
 for ( ib = 2 ; ib < 32 ; ib ++)
 {
  Ws[ib] = Xs[ib-2]-Ws[ib-1]-Ws[ib-2];
  Xs[ib-3] = Ws[ib]+Ws[ib-1]+Ws[ib-2];
 }
 DMA(AS_Ws[ie' ], AD_W[ie], 30);
 DMA(AS_Xs[ie' -3], AD_Y[ie], 30);
 DAM ready
 Ws[0] = Ws[ib-1]; Ws[1] = Ws[ib];
}
```

(b)

```
X[-3] = W[0] = 1; X[-2] = W[1] = 2;
for( ie = 2; ie < 242; ie = ie+30)
{
 ie' = ie mod 30 ;
 DMA( AD_X[ie], AS_Xs[ie' -2], 30) ;
 DMA ready
 for ( ib = 2 ; ib < 32 ; ib ++)
 {
  Xs[ib-3] = Xs[ib-2]-Xs[ib-4]-Xs[ib-5];
  Ys[ib-2] = Xs[ib-3]+Xs[ib-4]+Xs[ib-5];
 }
 DMA(AS_Xs[ie' -3], AD_W[ie], 30);
 DMA(AS_Ys[ie' -2], AD_Y[ie], 30);
 DAM ready
 Ws[0] = Ws[ib-1]; Ws[1] = Ws[ib];
}
```

(c)

Figure 8. The code transformation for the loop shown in Figure 5: (a) the code without space overlapping; (b) the code with space overlapping between arrays X and Y; and (c) the code with space overlapping between arrays X and W.

for it. Then we transform the loop into two levels by grouping N iterations into an inner loop, where N is the number of iterations we can put into one block and is obtained by Equation (7). Finally, we insert DMA operations for implementing block-level data transfer. When inserting DMA operations, for arrays with space overlapping, we need to transfer the data back to original arrays. An example is given in Figure 8 to show the code transformation for the loop shown in Figure 5.

3.4. Discussion

In this paper, we focus on optimizing arrays with the properties as described in Section 3.1. All the loops in the benchmarks we tested shown in Section 4 have such properties. To deal with general arrays, the dynamic SPM technique with data pipelining proposed in [40] can be applied. In this technique, memory accesses of multiple iterations are grouped and put into different portions of the SPM in order to improve data locality of regular array accesses. When the CPU executes

DOI: 10.1002/spe

instructions and accesses data from one portion of the SPM, DMA operations can be performed to transfer data between the off-chip memory and another portion of SPM simultaneously.

In our technique, when two arrays are selected for space overlapping, they must be in distinct memory areas. The pointer aliasing problem may appear when arrays are inputted as the parameters of function calls, i.e. the arrays may exist in the same space due to runtime conditions. In this case, our technique can only be applied when we can guarantee that the arrays are in distinct memory areas—for example, in C programs, their pointers are labeled with the C99 keyword 'restrict'.

In this work, we focus on optimizing loop kernels as they are the most time and power consuming parts of the whole applications. For references to arrays outside loops, we can combine our technique with static SPM techniques in the previous work [6, 8, 10–17] by partitioning the SPM into two sections—one for the references to arrays outside loops using static SPM allocation techniques and the other for the references inside loops using our dynamic allocation technique. How to effectively combine the two techniques will be studied in the future work.

4. EXPERIMENTS

4.1. Experimental setup

We implement our approach based on the IMPACT compiler [38] and conducted experiments on the cycle-accurate VLIW simulator of Trimaran [39]. The experimental configuration for Trimaran simulator is shown in Table I.

In our experiments, we extract 12 loop kernels from Mediabench and DSPstone benchmarks, and the loop kernels are shown in Table II. In Table II, the required memory size for each loop kernel is given in Column 'Data Size'. In our technique, each array is assigned to a DMA channel. To test the run-time performance, we use the following DMA parameters, $Cdi = 100$ and $Cdt = 1$, that have been used in [29]. *Cdi* is the number of cycles for CPU initializing a DMA block transfer, and *Cdt* is the number of cycles for DMA transfer a byte between off-chip memory and SPM.

The benchmark characteristics are shown in Table III. For each benchmark the number of arrays and the array classification are shown in the second and third columns, respectively. The SPM allocation results for the two schemes, the technique in [20], and our proposed ISOS technique, are shown in the last column. For the sake of convenience, we name the technique in [20] as DSML (Dynamic Scratch-pad-memory Management for Loops).

The data in Table III show that ISOS can effectively achieve space overlapping, compared with DSML. For example, for benchmark *mmpen*, DSML allocates the SPM space for 4 arrays, while our ISOS technique allocates the space for 3 arrays. The reason is that space overlapping can be achieved by writing the data of the write-only array into the SPM space used by the read-only arrays. In the experiments, the block size is determined by the given SPM size and the number of

Table I. The configuration for the Trimaran simulator.

Parameters	Configuration
Function units	2 integer ALU, 2 floating point ALU, 2 load-store units, 1 branch unit
Instruction latency	1 cycle for integer ALU, 1 cycle for floating point ALU, 2 cycles for load in SPM, 1 cycle for store, 1 cycle for branch
Register file	64 integer registers, 64 floating point registers

DOI: 10.1002/spe

Table II. The loop kernels from MediaBench and DSPstone.

Source	Application	Abbreviation	Data size	Descriptions
		Mediabench		
mpeg2	enctransfrm	mmpen	10.6 KB	Forward/inverse transformation
mpeg2	decrecon	mmpde	7.8 KB	Compute the linear address based on cartesian/raster coordinates provided
mesa	drawpix	mmedr	15.6 KB	Compute shift value to scale 32-bit units down to depth values
gsm	lpc	mgslp	128 KB	Fast_Autocorrelation
epic	collapse_ortho_pyr	mepco	31.8 KB	A QMF-style pyramid using an arbitrary filter
rasta	fft	mraff	224 KB	Calling routine for complex fft of a real sequence
rasta	lpccep	mfalp	5.9 KB	Computes autoregressive cepstrum from the auditory spectrum
rasta	post_audspec	mrapo	19.6 KB	Apply equal-loudness curve
		DSPstone		
fix point	n_real_updates	dfinr	18.8 KB	N complex updates—filter benchmarking
fix point	fft_bit_reduct	dfiff	63.9 KB	Benchmarking of an integer stage scaling FFT
fix point	lms	dfilm	19.8 KB	Lms—filter benchmarking
fix point	biquad_section	dfibi	26.5 KB	Benchmarking of an one iir biquad

Table III. The benchmark characteristics.

Abbreviation	Number of arrays	Array classification	SPM space allocation
mmpen	4	1 write-only 3 read-only	DSML:4 array space ISOS:3 array space
mmpde	2	1 write-only 1 read-only	DSML:2 array space ISOS:1 array space
mmedr	2	1 write-only 1 read-only	DSML:2 array space ISOS:1 array space
mgslp	4	2 write-only 2 read-only	DSML:4 array space ISOS:2 array space
mepco	3	2 write-only 1 read-only	DSML:3 array space ISOS:2 array space
mraff	2	1 write-only 1 read-only	DSML:2 array space ISOS:1 array space
mfalp	2	1 write-only 1 read-only	DSML:2 array space ISOS:1 array space
mrapo	3	1 write-only 2 read-only	DSML:3 array space ISOS:2 array space
dfinr	4	1 write-only 3 read-only	DSML:4 array space ISOS:3 array space
dfiff	2	1 write-only 1 read-only	DSML:2 array space ISOS:1 array space
dfilm	4	2 write-advance-read 2 read-only	DSML:4 array space ISOS:2 array space
dfibi	3	1 write-only 1 read-only 1 write-advance-read	DSML:3 array space ISOS: 2 array space

arrays for each benchmark. The number of blocks for processing each loop is shown in Tables IV and V.

4.2. Results and discussion

We compare our ISOS technique with DSML. In all experiments, the time performance is normalized based on that of DSML.

DOI: 10.1002/spe

Table IV. The number of blocks for processing each loop (the SPM size is fixed).

	DSML	ISOS	DSML	ISOS	DSML	ISOS
Abbreviation	SPM: 256B		SPM: 512B		SPM: 1KB	
mmpen	42	32	21	16	11	8
mmpde	31	16	16	8	8	4
mmedr	62	31	31	16	16	8
mgslp	512	256	256	128	128	64
mepco	127	85	64	42	32	21
mraff	896	597	448	299	224	149
mfalp	24	12	12	6	6	3
mrapo	78	52	39	26	20	13
dfinr	75	56	38	28	19	14
dfiff	256	170	128	85	64	43
dfilm	79	40	40	20	20	10
dfibi	106	71	53	35	27	18

Table V. The number of blocks for processing each loop (the SPM size is the percentage of the data size).

	DSML	ISOS	DSML	ISOS	DSML	ISOS
Abbreviation	SPM: 1% of data size		SPM: 2% of data size		SPM: 3% of data size	
mmpen	100	75	50	38	34	25
mmpde	100	50	50	25	34	17
mmedr	100	50	50	25	34	17
mgslp	100	50	50	25	34	17
mepco	100	67	50	33	34	22
mraff	100	67	50	33	34	22
mfalp	100	50	50	25	34	17
mrapo	100	67	50	33	34	22
dfinr	100	75	50	38	34	25
dfiff	100	67	50	33	34	22
dfilm	100	50	50	25	34	17
dfibi	100	67	50	33	34	22

From the loop partitioning point of view, DSML is a special loop tiling technique to determine the optimal tile size with the SPM space allocation and corresponding data transfer. Similar to DSML, our ISOS technique is a loop tiling technique as well, and it can determine the tile size with the SPM allocation considering space overlapping. Different from DSML, the tile generation approach in ISOS is relatively simple by directly grouping consecutive iterations in innermost loops based on space overlapping. Therefore, by comparing DSML (a technique with a relatively complicated tile-generation optimization scheme but without space overlapping) with ISOS (a technique with a relatively simple tile-generation approach and space overlapping), we can see the impact of the space overlapping exploration.

We have carried out comprehensive experiments by showing the performance of our algorithm under different situations. First, we consider the case with the fixed SPM size, which is used to demonstrate the various run-time performances reached by applications with different sizes on a given SPM size. Second, we consider the case of taking the SPM size as the percentage of data size, which can be used to show the performance of the algorithms independent from data sizes of applications.

The experimental results with fixed SPM sizes are shown in Figure 9. Figure 9 shows the experimental results for different benchmarks when the SPM size varies from 128 bytes to 20 KB, respectively. From the figure we can see that ISOS can achieve better time performance compared with DSML with various SPM sizes in all loop kernels. The average improvements are 13.15, 19.05, and 25.52% when the SPM sizes are 1KB, 512 bytes, and 256 bytes, respectively.

DOI: 10.1002/spe

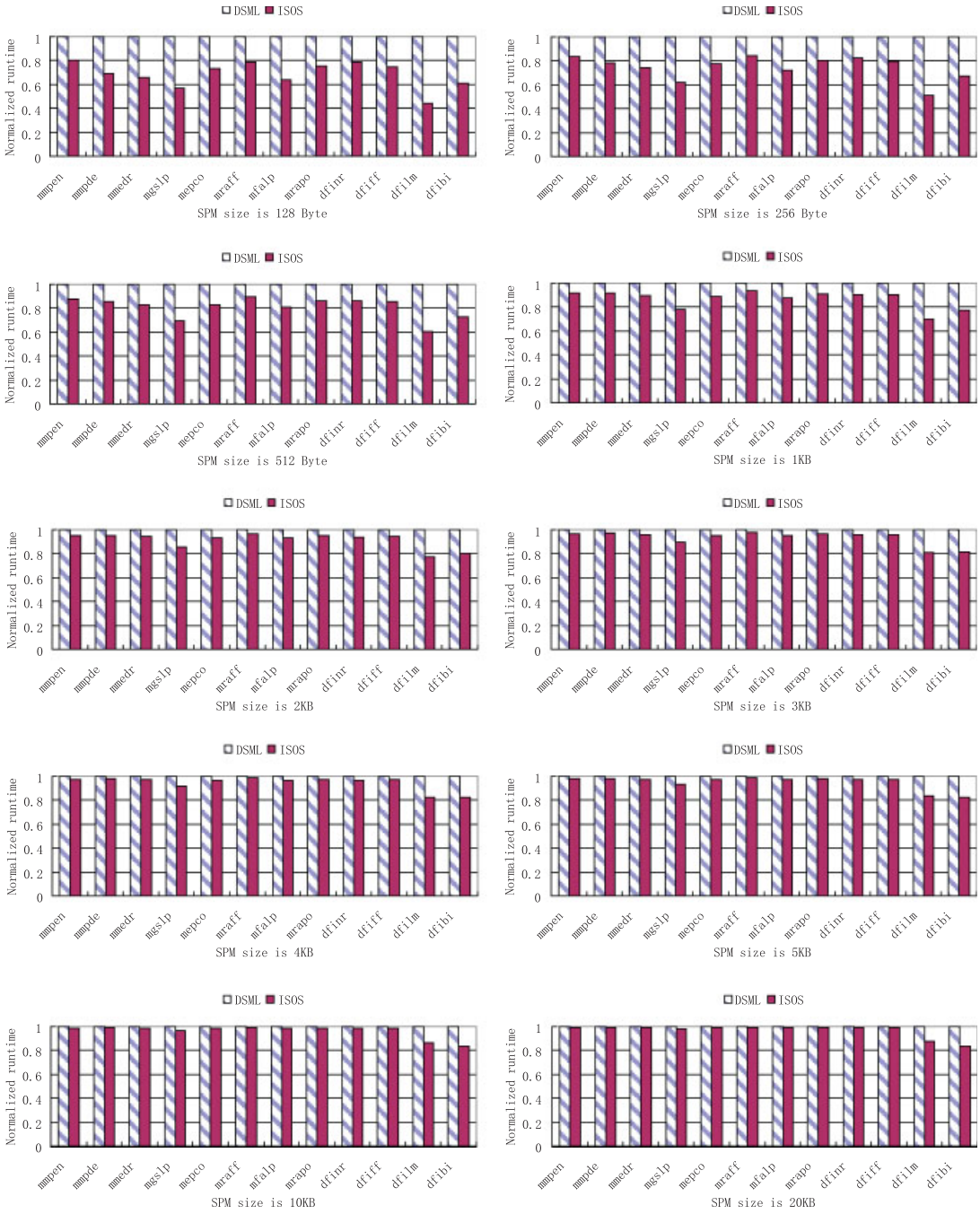

Figure 9. The time performance comparisons of ISOS and DSML with different SPM sizes (normalized to DSML).

We then conduct experiments by setting the SPM size as the percentage of the memory space each loop kernel needs (shown in Column 'Data Size' in Table II). As the memory spaces needed are different for different loop kernels, we want to compare the performances for different benchmarks with the same SPM/memory percentage. Figure 10 shows the results when the percentage of SPM/Data Size varies from 0.5 to 60%, respectively. We can see that the improvement is decreasing as the ratio varies from 0.5 to 60%. The reason is that the gap between ISOS and DSML becomes

DOI: 10.1002/spe

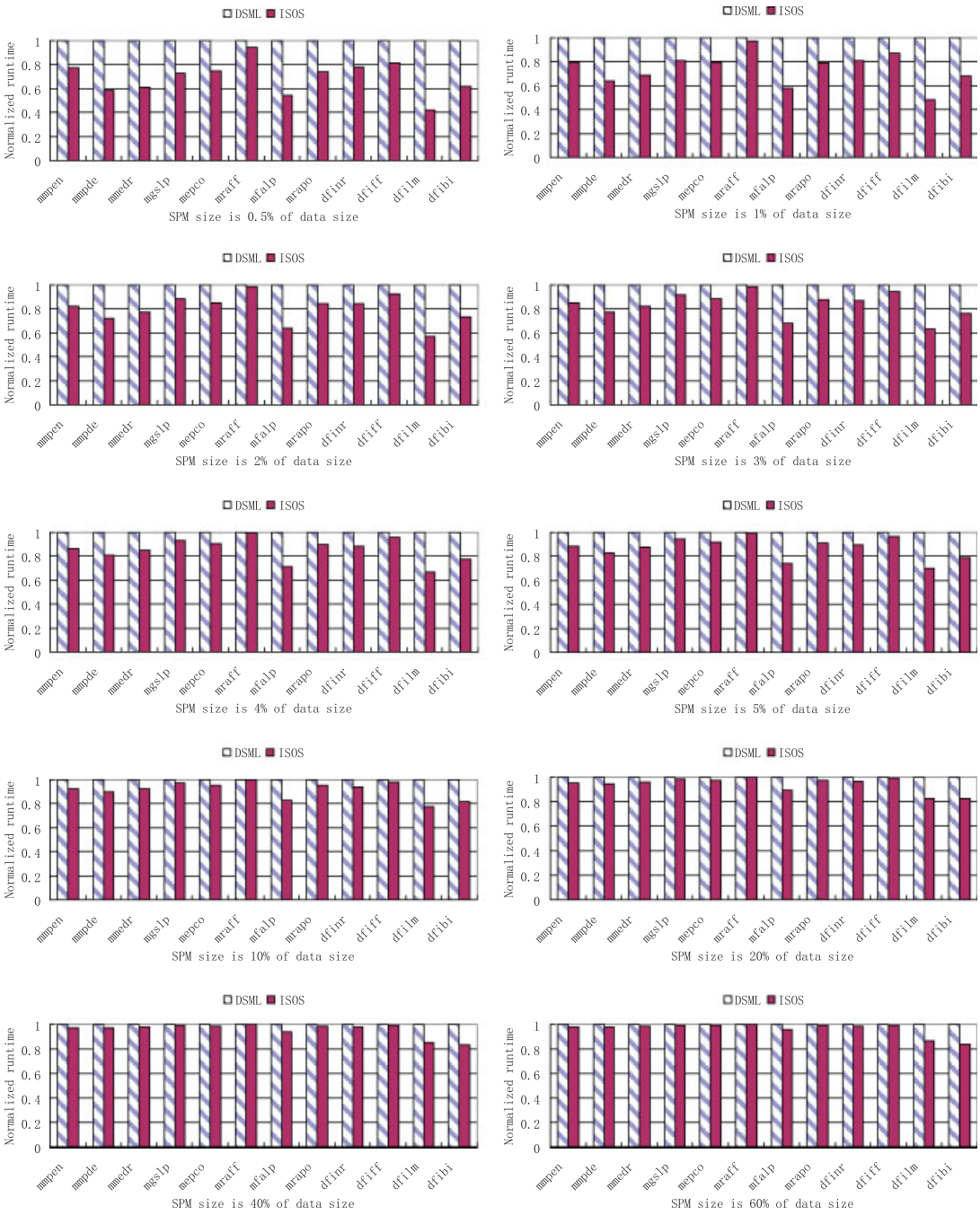

Figure 10. The time performance comparisons of ISOS and DSML with different percentages of SPM/Data Size (normalized to DSML).

smaller. From Figures 9 and 10, we can see that our ISOS can achieve better results compared with DSML in all cases.

As shown in Figures 9 and 10, the run-time performance of ISOS increases with the increase of the SPM size. We find that some applications such as '*mraff*' do not show the significant run-time performance improvement with the increase of the SPM sizes. Figure 11 shows the detailed performance comparisons among the four applications, '*mmpen*', '*mgslp*', '*mraff*', and '*dfilm*', when the SPM sizes vary from 128 bytes to 40 KB. Here, for each benchmark, the run-time performance of ISOS is normalized to that achieved when the SPM size is 128 Byte. We can see

DOI: 10.1002/spe

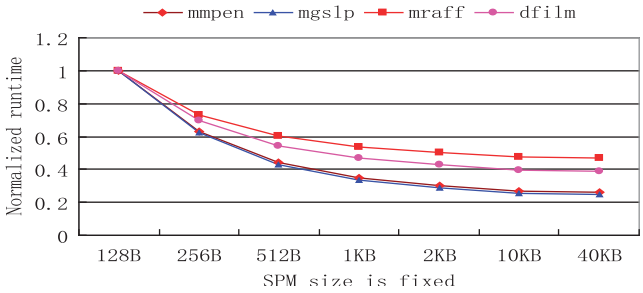

Figure 11. The normalized run-time performance on varying applications when the SPM sizes change from 128 bytes to 40 KB.

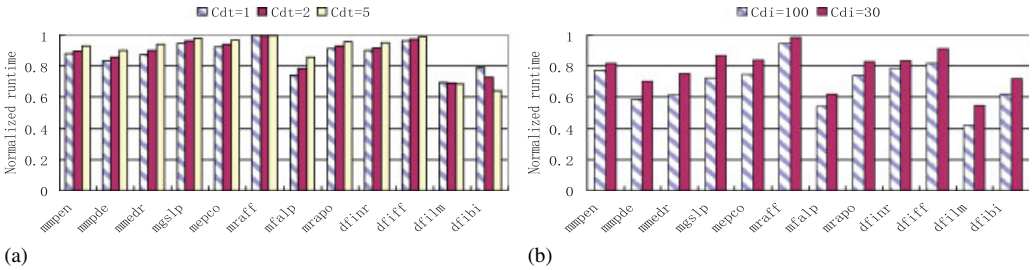

Figure 12. The normalized run-time performance: (a) various parameters for DMA transfer and (b) various parameters for DMA initialization.

that the factor of the SPM size has different influences with different applications. For example, it gives little influence on '*mraff*' but big influence on '*mmpen*'. This is because the ISOS strategy reduces the cost of data transfer with the increase of the SPM sizes; if data transfer costs dominate within an application, the run-time performance increases significantly.

Figure 12(a) shows the normalized run-time performance of all the applications when we keep the same *Cdi* but apply different *Cdt*. Most of the applications achieve less speedup when the cost of *Cdt* increases. For example, the improvements of '*mralp*' are 25.72%, 21.53%, 14.47%, when *Cdt* is equal to 1, 2, 5, respectively. This is because the costs of data transfer increase when *Cdt* is changed from 1, to 2 and 5, while the gain from space overlapping is fixed. On the other hand, when *Cdt* increases, the effect of eliminating redundancy is remarkable. Thus for some applications, it may improve the performance. For example, in '*dfilm*' and '*dfibi*', we can see that the more *Cdt* is set, the more speedup we can achieve. For application '*dlibi*', it achieves the improvements of 21.16, 27.22, and 36.03% when *Cdt* is equal to 1, 2 and 5, respectively. Figure 12(b) shows the impact on *Cdi*, the cost of DMA initialization for one channel. We can find that the bigger *Cdi* is set, the more speedup we can obtain. The speedup is proportional to the cost of DMA initialization, since the space overlapping can decrease the number of blocks by grouping more iterations into one block.

5. CONCLUSION

In this paper, we proposed a compiler-assisted iteration-access-pattern-based space overlapping technique for dynamic SPM management (ISOS) with DMA. In ISOS, we exploited the chance to overlap SPM space so as to further utilize the limited SPM space and reduce the number of DMA operations. We implemented our technique based on IMPACT and conducted experiments using a set of benchmarks from DSPstone and Mediabench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique achieves significant run-time performance improvement compared with the previous work.

DOI: 10.1002/spe

ACKNOWLEDGEMENTS

This work is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 5269/08E), HK PolyU (1-ZV5S), National 863 Program of China (Grant No. 2006AA01Z172 and 2006AA01Z199), National Natural Science Foundation of China (Grant No. 60533040 and 60773089), National Science Fund for Distinguished Young Scholars (Grant No. 60725208), and Shanghai Pujiang Program (No. 07pj14049).

REFERENCES

1. Dominguez A, Nguyen N, Barua RK. Recursive function data allocation to scratch-pad memory. *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Salzburg, Austria, 2007; 65–74.
2. The Elan™SC520 Microcontroller Technical Reference Manual, 2001. Available at: http://www.amd.com/files/connectivitysolutions/e86embedded/elansc520/22003b.pdf [August 2001].
3. The S3C2500 User's Manual, 2003. Available at: http://www.samsung.com [May 2003].
4. DaVinci™ Digital Media Processors-user Guides, 2008. Available at: http://www.ti.com [October 2008].
5. The ARM1136JF-S and ARM1136J-S Technical Reference Manual, 2009. Available at: http://www.arm.com [July 2009].
6. Panda PR, Dutt N, Nicolau A. Efficient utilization of scratch-pad memory in embedded processor applications. *Proceedings of the European Design and Test Conference*, Paris, France, 1997; 7–11.
7. Panda PR, Dutt N, Nicolau A. *Memory Issues in Embedded Systems-on-chip*. Kluwer Academic: Boston, 1999.
8. Avissar O, Barua R, Stewart D. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Systems* (*TECS*) 2002; **1**(1):6–26.
9. Kandemir M, Kadayif I, Sezer U. Exploiting scratch-pad memory using presburger formulas. *Proceedings of the 14th International Symposium on System Synthesis* (*ISSS*), Montreal, Canada, 2001; 7–12.
10. Banakar R, Steinke S, Lee B-S, Balakrishnan M, Marwedel P. Scratchpad memory: Design alternative for Cache On-chip memory in embedded systems. *Proceedings of the 10th International Symposium on Hardware/Software Codesign* (*CODES*), Colorado, U.S.A., 2002; 73–78.
11. Steinke S, Wehmeyer L, Lee B, Marwedel P. Assigning program and data objects to scratchpad for energy reduction. *Proceedings of the Design, Automation and Test Conference in Europe*, Paris, France, 2002; 409–415.
12. Wehmeyer L, Marwedel P. Influence of onchip scratchpad memories on WCET prediction. *Proceedings of the Fourth International Workshop on Worst-Case Execution Time* (*WCET*) *Analysis*, Sicily, Italy, 2004; 120–130.
13. Wehmeyer L, Helmig U, Marwedel P. Compiler-optimized usage of partitioned memories. *Proceedings of the Third Workshop on Memory Performance Issues* (*WMPI2004*), Munich, Germany, 2004; 114–120.
14. Avissar O, Barua R, Stewart D. Heterogeneous memory management for embedded systems. *Proceedings of the ACM Second International Conference on Compilers, Architectures, and Synthesis for Embedded Systems* (*CASES*), Atlanta, Georgia, U.S.A., 2001; 34–43.
15. Hiser JD, Davidson JW. EMBARC: An efficient memory bank assignment algorithm for retargetable compilers. *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, Washington, DC, U.S.A., 2004; 182–191.
16. Panda PR, Dutt N, Nicolau A. On-chip vs Off-chip memory: The data partitioning problem in embedded-processor-based systems. *ACM Transactions on Design Automation of Electronic Systems* 2000; **5**(3):682–704.
17. Sjodin J, Froderberg B, Lindgren T. Allocation of global data objects in On-chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, Washington, DC, U.S.A., 1998; 205–220.
18. Udayakumaran S, Dominguez A, Barua R. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems* (*TECS*) 2006; **5**(2):472–511.
19. Steinke S, Grunwald N, Wehmeyer L, Banakar R, Balakrishnan M, Marwedel P. Reducing energy consumption by dynamic copying of instructions onto Onchip Memory. *Proceedings of the 15th International Symposium on System Synthesis* (*ISSS*), Kyoto, Japan, 2002; 213–218.
20. Kandemir M, Ramanujam J, Irwin J, Vijaykrishnan N. Dynamic management of scratch-pad memory space. *Proceedings of the Design Automation Conference*, Las Vegas, NV, U.S.A., 2001; 690–695.
21. Li L, Nguyen QH, Xue J. Scratchpad allocation for data aggregates in superperfect graphs. *Proceedings of the ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES'07*), San Diego, U.S.A., 2007; 207–216.
22. Verma M, Wehmeyer L, Marwedel P. Dynamic overlay of scratchpad memory for energy minimization. *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*(*CODES+ISIS*), Stockholm, Sweden, 2004; 104–109.
23. Verma M, Steinke S, Marwedel P. Data partitioning for maximal scratchpad usage. *Proceedings of the 2003 Conference on Asia South Pacific Design Automation*, Kitakyushu, Japan, 2003; 77–83.
24. Angiolini F, Benini L, Caprara A. An efficient profile-based algorithm for scratchpad memory partitioning. *IEEE Transactions on Computer-Aided Design* 2005; **24**(11):1660–1676.
25. Udayakumaran S, Barua R. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (*CASES*), Philadelphia, U.S.A., 2003; 276–286.

DOI: 10.1002/spe

26. Cho H, Egger B, Lee J, Shin H. Dynamic data scratchpad memory management for a memory subsystem with an MMU. *Proceedings of the International Conference on Languages*, *Compilers*, *and Tools for Embedded Systems* (*LCTES'07*), San Diego, U.S.A., 2007; 195–206.
27. Schreiber R, Darren C. Near-optimal allocation of local memory arrays. *HP Technical Reports HPL-2004-24*, 2004; 11–13.
28. Anantharaman S, Pande S. Compiler optimizations for real time execution of loops on limited memory embedded systems. *Proceedings of the 19th IEEE Real-Time Systems Symposium* (*RTSS*), Madrid, Spain, 1998; 154–164.
29. Li L, Gao L, Xue J. Memory coloring: A compiler approach for scratchpad memory management. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, U.S.A., 2005; 329–338.
30. Dominguez A, Udayakumaran S, Barua R. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing* (*JEC*) 2005; **1**(4):521–540.
31. Kandemir M, Ramanujam J, Irwin MJ, Vijaykrishnan N, Kadayif I, Parikh A. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* 2004; **23**(2):243–260.
32. Vijaykrishnan N, Kandemir M, Irwin MJ, Kim HS, Ye W. Energy-driven integrated hardware-software optimizations using simple power. *Proceedings of the International Symposium on Computer Architecture*, Vancouver, BC, Canada, 2000; 95–106.
33. Using the STM32f101xx and STM32F103xx DMA controller, 2009. Available at: http://www.st.com [June 2009].
34. Wang Z, O'NEIL TW, Sha EH-M. Optimal loop scheduling for hiding memory latency based on two-level partitioning and prefetching. *IEEE Transactions on Signal Processing* 2001; **49**(11):2853–2864.
35. Wang Z, Sha EH-M, Hu XS. Combined partitioning and data padding for scheduling multiple loop nests. *Proceedings of the 2001 International Conference on Compilers*, *Architecture*, *and Synthesis for Embedded Systems*, Atlanta, Georgia, U.S.A., 2001; 67–75.
36. Balasa F, Catthoor F, De Man H. Exact evaluation of memory size for multi-dimensional signal processing systems. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, New York, U.S.A., 1993; 669–672.
37. De Greef E, Catthoor F, De Man H. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing* 1997; **23**:1811–1837.
38. Chang PP, Mahike SA, Chen WY, Warier NJ, Hwu WW. IMPACT: An architectural framework for multiple-instruction-issue processors. *Proceedings of the 18th International Symposium on Computer Architecture* (*ISCA*), Toronto, Canada, 1991; 266–275.
39. The Trimaran Compiler Research Infrastructure, 2010. Available at: http://www.trimaran.org/ [January 2010].
40. Yang Y, Wang M, Yan H, Shao Z, Guo M. Dynamic scratch-pad memory management with data pipelining for embedded systems. *Concurrency and Computation*: *Practice and Experience* 2010; **22**:1874–1892.

DOI: 10.1002/spe