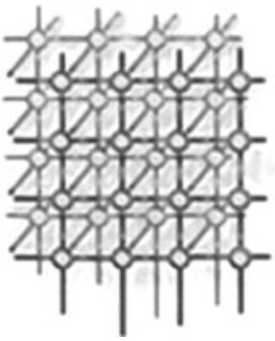
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **22**:1874–1892
Published online 7 June 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1602



Dynamic scratch-pad memory management with data pipelining for embedded systems

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

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

SUMMARY

In this paper, we propose an effective data pipelining technique, SPDP (Scratch-Pad Data Pipelining), for dynamic scratch-pad memory (SPM) management with DMA (Direct Memory Access). Our basic idea is to overlap the execution of CPU instructions and DMA operations. In SPDP, based on the iteration access patterns of arrays, we group multiple iterations into a block to improve the data locality of regular array accesses. We allocate the data of multiple iterations into different portions of the SPM. In this way, when the CPU executes 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. We perform code transformation to insert DMA instructions to achieve the data pipelining. We have implemented our SPDP technique with the IMPACT compiler, and conduct experiments using a set of loop kernels from DSPstone, Mibench, and Mediabench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique achieves performance improvement compared with the previous work. Copyright © 2010 John Wiley & Sons, Ltd.

Received 15 December 2009; Revised 17 February 2010; Accepted 10 April 2010

KEY WORDS: scratch-pad memory management; data pipelining; embedded systems

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

Contract/grant sponsor: Hong Kong Special Administrative Region, China; contract/grant number: GRF PolyU 5269/08E
Contract/grant sponsor: Hong Kong Polytechnic University; contract/grant number: HK PolyU A-ZV5S
Contract/grant sponsor: National 863 Program of China; contract/grant number: 2008AA01Z106
Contract/grant sponsor: National Natural Science Foundation of China (NSFC); contract/grant number: 60725208

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 with its advantages in energy and area [1]. A recent study [1] shows that SPM has 34% smaller area and 40% lower power consumption than the 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. Owing to these advantages, SPM has become the most common SRAM in embedded processors. However, it poses a big 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 method [2] and dynamic method [3,4]. 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 [3], and many embedded applications in image and video processing domains have significant data transfer requirements in addition to their computational requirements. To reduce off-chip memory access overheads, the dedicated cost-efficient hardware, DMA (Direct Memory Access), is used to transfer data. In this paper, we focus 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 [4–8]. In [4], DMA is applied for data transfer between SPM and off-chip memory. The same cost model using DMA for data transfer has been used in [7] to accelerate data transfer between off-chip memory and SPM. The work in [8] used DMA to pre-fetch data only from off-chip memory to SPM. However, the above work focuses on array allocation for SPM without considering the data parallelization between DMA and CPU. In our technique, we show that we can achieve data parallelization for multiple iterations of a loop.

In this paper, we propose an effective data pipelining technique, SPDP (Scratch-Pad Data Pipelining), for dynamic SPM management with DMA. Our basic idea is to overlap the execution of CPU instructions and DMA operations. In SPDP, based on the iteration access patterns of arrays, we group multiple iterations into a block to improve the data locality of regular array accesses. We allocate the data of multiple iterations into different portions of the SPM. In this way, when the CPU executes 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. We perform code transformation to insert DMA instructions to achieve the data pipelining. We implement our technique with IMPACT [9], and conduct experiments using a set of loop kernels from DSPstone [10], Mibench [11], and Mediabench [12] on the cycle-accurate VLIW simulator of Trimaran [13]. The experimental results show that the SPDP technique achieves performance improvement compared with the previous work [5,6,8].

DOI: 10.1002/cpe

The remainder of this paper is organized as follows. In Section 2, we present the system model. In Section 3, we give motivational examples. We propose the SPDP technique in Section 4. We present the experimental results in Section 5. The related work is presented in Section 6. The conclusion is given in Section 7.

2. BASIC CONCEPTS AND MODELS

2.1. System model

The system model shown in Figure 1 has a similar architecture as that in [4,14]. Basically, 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 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 compiler to insert instructions to control DMA operations for data transfer. In this paper, we assume that the SPM supports simultaneous accesses based on the dual ported SPM system in [14] that is associated with multiple software managed SRAMs with specialized address generators.

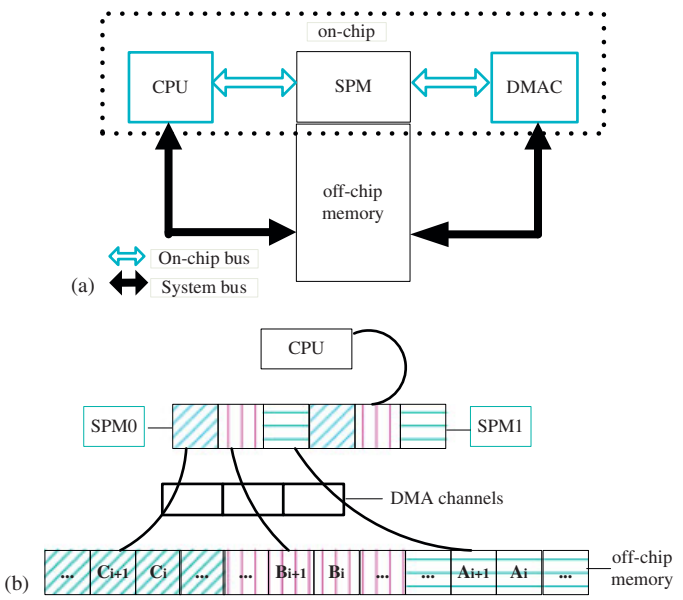


Figure 1. The system model.

DOI: 10.1002/cpe

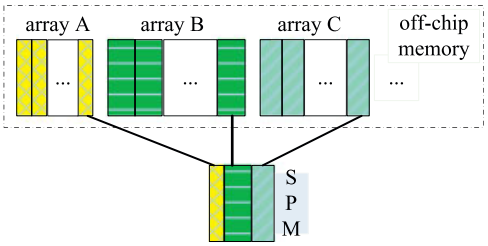


Figure 2. Data transfer between the SPM and off-chip memory.

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 that 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 the later sections, we will discuss how to determine the size of data block and insert DMA operations for block-level data transfer.

Based on the model in [4], the cost of transferring a data block between SPM and off-chip memory is approximated by (Cdi+Cdt*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*(Cdi+Cdt*n), where Nb is the number of blocks of an array. In our approach, we can reduce Nb by exploiting 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 this way that the total number of blocks for an array is reduced.

2.2. Data pipelining

We assume that DMA operations can be executed simultaneously with CPU instructions, which is supported by the dual ported SPM system in [14]. Based on this architecture, as shown in Figure 3(a), CPU accesses the first part of SPM, SPM0, whereas DMA transfers data blocks between SPM1 and off-chip memory at the same time. In this way, data pipelining is achieved. We focus on data pipelining for array elements in this paper.

3. MOTIVATIONAL EXAMPLES

In this section, we present motivational examples to show how our technique works. The C code of the example loop is shown in Figure 4(a). In this example, there are three arrays, W, X, and Y. We have load operations for array X and store operations for array Y. For array W, we have both load and store operations. The data of array references is put into SPM for execution. We assume that the SPM can hold 64 array elements, and we have two DMA channels. With this SPM size,

DOI: 10.1002/cpe

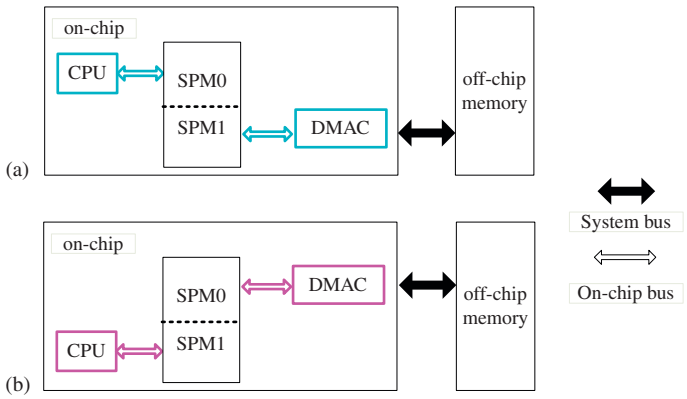


Figure 3. Data pipelining.

we group 20 iterations of the loop into a block, and put the data of this block into SPM. DMA is used to transfer the required data block between the off-chip memory and the SPM.

The SPM layout without data pipeline is shown in Figure 4(b). In this figure, based on the iteration access patterns of arrays, the array elements are allocated to SPM. These elements are required by or calculated in iterations 20–40. The execution of this case is shown in Figure 4(d). Without the data pipeline, we have to use DMA to load the required data into SPM for execution. After CPU instructions finish computation, DMA operations are performed to store data to the off-chip memory. Accordingly, the iterations of the loop are executed sequentially.

Using our technique, the SPM layout with data pipeline is shown in Figure 4(c). Our SPDP technique divides the SPM into two parts. We put the array elements of iterations 20–30 into the first half of the SPM, and put the array elements of the next 10 iterations into the second half. In this way, the data pipeline is generated. With this pipeline, the execution status is shown in Figure 4(e). Using our technique, when CPU instructions of the first 10 iterations execute, DMA operations are performed to transfer the data of the next 10 iterations at the same time. Thus, the data required by the next 10 iterations are ready when CPU finishes the computation of the current 10 iterations. Similarly, when CPU processes the data in the second half of the SPM, DMA operations are executed simultaneously to fetch or store data for the first half of the SPM. With the overlapping between the execution of CPU instructions and DMA operations, our SPDP technique can improve the performance of loops.

4. THE PROPOSED ALGORITHM

The overview of our SPDP technique is shown in Figure 5. It mainly consists of the following three steps:

- *Array classification*: Based on iteration access patterns of arrays, we classify arrays into four groups: write-only arrays, read-only arrays, write-advance-read arrays, and others. Then, we allocate the SPM space to the array elements.

DOI: 10.1002/cpe

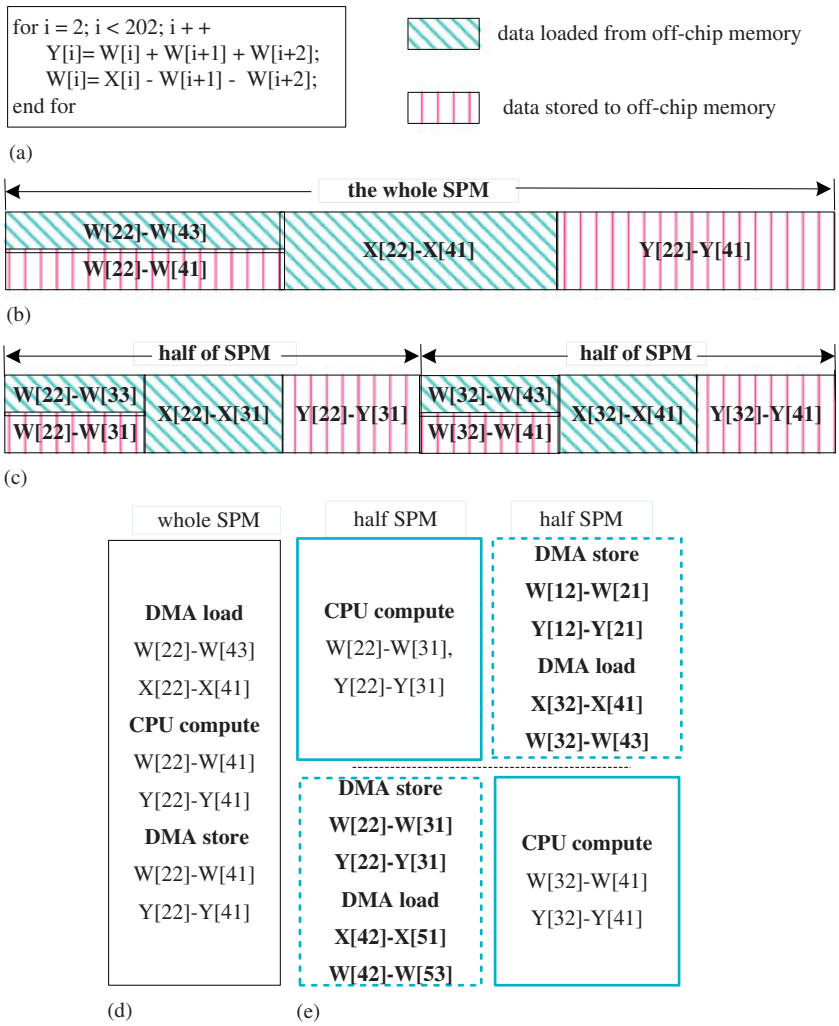


Figure 4. Data pipelining.

- *Data pipelining*: In this step, we discuss how to compute the number of iterations in a block considering the SPM size.
- *Code transformation*: We propose an algorithm to generate the transformed code by applying data pipelining. Basically, we insert instructions for DMA transfer and transform array references.

Next, we present the details of each step in the later sections.

DOI: 10.1002/cpe

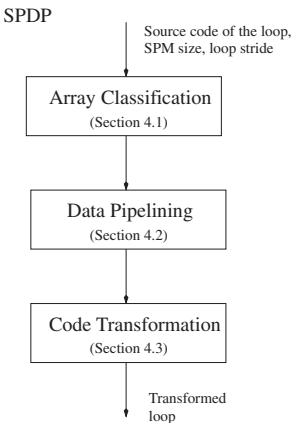


Figure 5. The overview of our SPDP technique.

4.1. Array classification

In SPDP, 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 maximum subscript among all of the array elements related to the read operations must not be larger than the maximum subscript among all of 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.

4.2. Data pipelining

Based on the iteration access patterns of arrays, we perform data pipelining considering the size of SPM. Our basic idea is to execute CPU instructions and DMA operations at the same time. In our technique, we divide the SPM into two blocks for data allocation. Considering the size of SPM, we can calculate how many iterations we can put into one block based on the memory required for all arrays. To initialize the SPM, we group a number of iterations into the first block, and put the data accessed by the arrays of next iterations into the second block. When CPU instructions execute and access data from the second block, we use DMA operations to transfer and update data between the first block and the off-chip memory. In this way, we have a data pipeline in which there is an

DOI: 10.1002/cpe

overlapping between the execution of CPU instructions and DMA operations. In the following, we present how we calculate the number of iterations to be put into one block.

In our SPDP technique, taking the SPM size into account, we can group *Nit* iterations into a block. Assume that the arrays of a loop are in the form of '$X[f(i)]=X[\text{coef}(X)*i+\text{offset}(X)]$'. Here, X is an array, the expression $f(i)$ is an affine function of loop index i, coef(X) is the coefficient and offset(X) is the offset. As we assign one space for each array in each block of SPM, we have the following equation:

$$S_{\text{spm}}=2\times\sum_{i=1}^{N_a}S_{A_i} \tag{1}$$

Here, S_{spm} denotes the size of the whole SPM. 'Na' denotes the number of arrays in the iteration. S_{A_i} denotes the SPM space occupied by array A_i for N_{it1} iteration, and A_i is any array in iteration. We express Equation (1) in detail as follows:

$$S_{\text{spm}}=2\times\sum_{i=1}^{N_a}(\max(\text{offset}(A_i))-\min(\text{offset}(A_i))+1)+(N_{\text{it1}}-1)\times\text{coef}(A_i)\times k) \tag{2}$$

According to the above equations, we can compute the number of iterations, N_{it1}, as follows:

$$N_{\text{it1}}=\left\lfloor\frac{S_{\text{spm}}\times 0.5-\sum_{i=1}^{N_a}\text{bound}(A_i)}{\sum_{i=1}^{N_a}\text{coef}(A_i)\times k}+1\right\rfloor \tag{3}$$

$$\text{bound}(A_i)=\max(\text{offset}(A_i))-\min(\text{offset}(A_i))+1 \tag{4}$$

For example, in the motivational example of Figure 4, we have the following parameters: $S_{\text{spm}}=64$, bound(W)=3, bound(X)=bound(Y)=1, coef(W)=coef(X)=coef(Y)=1, and $k=1$. According to Equation (3), we can get the value of N_{it1} as 10.

4.3. Code transformation

Based on the data analysis in Section 4.2, in this section, we perform code transformation to implement the data pipelining. The code transformation algorithm is shown in Figure 6.

Basically, based on the number of iterations for one block, the code transformation consists of the following steps: (1) group iteration into blocks and change array references and (2) insert DMA operations for data transfer between SPM and off-chip memory. In Section 4.2, we have illustrated how to compute the number of iteration N_{it} in a block. Next, we introduce the instructions we adopt for DMA transfer.

There are two kinds of DMA instructions: DMA initialization and DMA ready. The instruction of DMA initialization includes three fields: the source address, the destination address, and the size of the data block. When DMA is used to load blocks from off-chip memory to SPM, the source address is AD_address and the destination address is AS_address. The size of the block is determined by the number of iterations in a block. Here, A_i is an array and AD_address denotes the off-chip memory address of block of array A_i. AS_address denotes the SPM address of block of array A_i. When DMA is used to store blocks from SPM to off-chip memory, the source address is AS_address, and the destination address is AD_address.

DOI: 10.1002/cpe

```
Procedure: code transformation

1. initialize AS_address  for all the arrays in iteration
2. initialize AD_address for all the arrays in iteration
3. initialize sequential number of block, snb=1
   //data pipeline_1 fill stage
4. initialize DMA loading for the snb^th block
5. DMA ready
6. initialize DMA loading for the (snb+1)^th block
7. execute the iteration of the snb^th block
8. DMA ready
   //data pipeline_1 full stage
9. for (i=3, i< N_b+1, i+1)
10.    do  initialize DMA storing for the snb^th block
11.        initialize DMA loading for the (snb+2)^th  block
12.        execute the iteration of the (snb+1)^th block
14.        snb++
15.        DMA ready
   //data pipeline_1 empty stage
16. initialize DMA storing for the snb^th block
17. execute the iteration of the (snb+1)^th block
18. initialize DMA storing for the (snb+1)^th block
19. DMA ready
```

Figure 6. The code transformation algorithm.

The details of the DMA ready instruction are as follows. When DMA finishes loading or storing blocks, DMA enters the ready state and we use the DMA ready instruction to wait for DMA completing block transfer. As shown in line 5 of Figure 6, the instruction denotes that DMA has finished loading the data for the first iteration block. In lines 6 and 7 of Figure 6, as the data for computing the first block is ready, CPU initializes DMA for the next block and executes the instructions of the first block.

The block mapping between off-chip memory and SPM is shown in Figure 7(a). In our technique, the odd blocks in the off-chip memory are mapped to SPM1, and the even blocks are mapped to SPM0. The mapped SPM address can be calculated as AS'_address$=$ (AS_address$+0.5*S_{\text{spm}}$) mod S_{spm}. Here, AS_address denotes the SPM address of one block, and AS'_address denotes that of the other block. For example, the address mapping of the motivational example in Figure 4 is shown in Figure 7(b). In this example, *WB*, *YB*, and *XB* denote the blocks of array W, Y, and X, respectively. We assume the SPM addresses of arrays W, X, and Y are 32, 44, and 54, respectively. We can get the addresses of the other blocks as 0, 12, and 22, respectively.

5. EXPERIMENTS

5.1. Experimental setup

We have implemented our technique with the IMPACT compiler [9] and conducted experiments on the cycle-accurate VLIW simulator of Trimaran [13]. To make the Trimaran simulator support

DOI: 10.1002/cpe

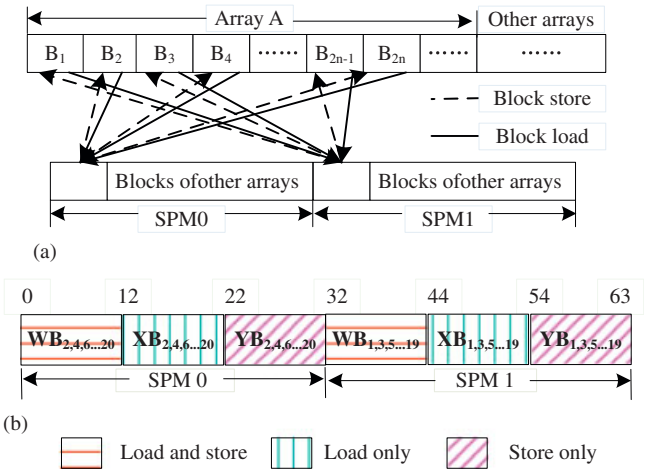


Figure 7. (a) The address mapping and (b) an example of address mapping for the example of Figure 4.

Table I. The configurations of the simulator.

Parameters	Configurations
Functional units	Two integer ALUs; two floating point ALUs; two load-store units; one branch unit; five issue slots
Instruction latency	One cycle for integer ALU; one cycle for floating point ALU; two cycles for load in cache; one cycle for store; one cycle for branch

SPM, we extend it by adding an additional data buffer as the SPM and the customized DMA instructions for data transfer between the main memory and SPM. In the experiments, for each benchmark, we first generate Lcode by IMPACT. Based on the Lcode obtained, we apply the code transformation, insert DMA instructions into loop kernels and conduct experiments with the loop kernels on the Trimaran simulator. The configurations for the Trimaran simulator are shown in Table I.

In the experiments, we extract nine loop kernels from DSPstone [10], Mediabench [12], and Mibench [11] as shown in Table II based on the Lcode generated by IMPACT. In Table II, the required memory size for each loop kernel is given in column 'Data Size'.

To evaluate the run-time performance, we use five parameters, Cdi, Cdt, Ccs, Cct, and the SPM size. Here, Cdi is the number of clock cycles for the DMA initialization. Cdt is the number of clock cycles for the DMA to transfer one byte between the SPM and off-chip memory. Ccs is the number of clock cycles for the CPU to transfer one byte between the CPU and SPM, and Cct is the number of cycles for CPU to transfer one byte between the CPU and off-chip memory. Based on the parameters in [4], we set Ccs as 1. In the experiments, we adopt five ratios of ⟨Cdi:Cdt:Cct⟩, and they are ⟨5:5:10⟩, ⟨9:1:20⟩, ⟨10:2.5:10⟩, ⟨20:1:20⟩, and ⟨100:1:100⟩.

DOI: 10.1002/cpe

Table II. The benchmarks.

Benchmark	Description	Data size (kB)
DSPstone		
Dbiq	*biquad_section*	90.01
Dlms	lms filter	155.98
Dfir	fir2dim filter	80.03
Dupd	*n_real_updates*	62.92
Mediabench		
Mrea	Read source pictures	48.04
Rfft	Fourier analysis	75.95
Rpos	Auditory processing	78
Mibench		
Tfou	FFT operations	128.04
Hreo	Loop-reorder test	89.96

5.2. Results and discussion

In the experiments, we compare our SPDP technique with the following three techniques:

- *NDMA*: DMA is not used for data transfer [5].
- *ADMA*: DMA is used to accelerate data transfer, and the data transfer between off-chip memory and SPM relies on DMA completely [6].
- *PDMA*: DMA is used to pre-fetch data for the next data block [8].

In the following, we first present the results of the performance improvement obtained by our technique with a fixed SPM size. Then, we give the results to show the performance improvements with different parameters.

(1) Performance improvement with fixed SPM size. Figure 8 shows the normalized run-time performance when we use a fixed SPM size. The run-time performance is normalized to that of NDMA. In the NDMA technique, DMA is not used in data transfer and there exists no overlapping between CPU execution and DMA data transfer. Figures 8(a)–(e) demonstrate the normalized run-time performance with various parameters $\langle 5:5:1:10\rangle$, $\langle 9:1:1:20\rangle$, $\langle 20:1:1:20\rangle$, $\langle 100:1:1:100\rangle$, and $\langle 10:2.5:1:10\rangle$, respectively. Figure 8(f) shows the average normalized run-time performance. On average, compared with the performance obtained by NDMA, ADMA, and PDMA, our SPDP achieves a performance improvement of 46.74, 33.27, and 58.68%, respectively, when the parameter is set as $\langle 5, 5, 1, 10\rangle$. When we set the parameter as $\langle 10, 2.5, 1, 10\rangle$, the average gain is 71.09, 46.53 and 79.07%, respectively. When the parameter is $<9, 1, 1, 20>$, the average gain is 91.79, 55.37 and 94.38%, respectively. When the parameter $\langle 20, 1, 1, 20\rangle$ is considered, the gain is 91.73, 55.25 and 94.27%, respectively. With the parameter of $\langle 100, 1, 1, 100\rangle$, the respective gain is 98.2, 57.92, and 98.65%.

From the results, we have two observations. The first observation is that, for all the applications and experimental parameters, our SPDP technique achieves the best run-time performance. This is because compared with the other techniques, SPDP achieves the overlapping of CPU execution and DMA data transfer. The second observation is that the parameter, $\langle$Cdi, Cdt, Ccs, Cct$\rangle$, affects the run-time performance. For example, in Figure 8(f) when the parameter is set as $<5, 5, 1, 10>$,

DOI: 10.1002/cpe

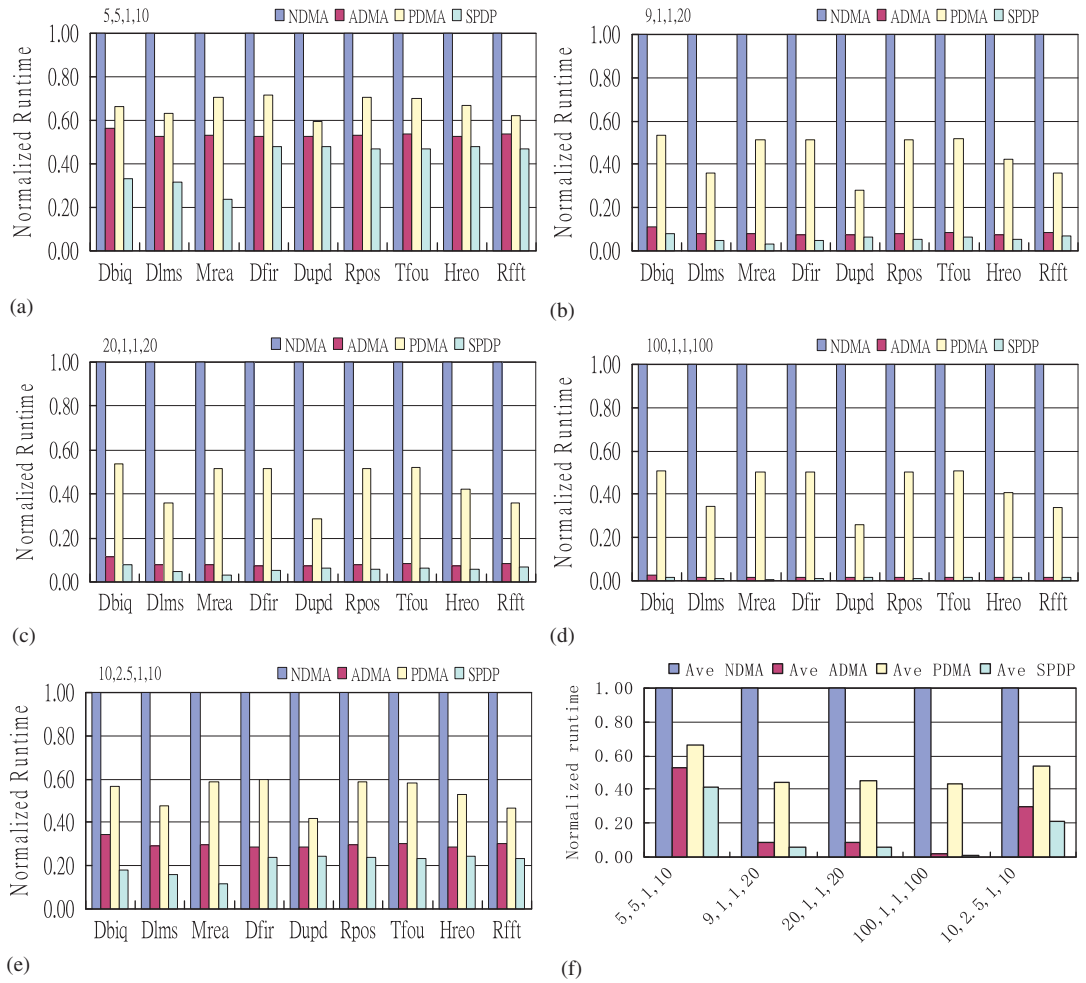


Figure 8. Normalized performance for different techniques (SPM_Size = 2k).

the average gain 46.74, 33.27, and 58.68% is obtained for ADMA, PDMA, and SPDP, respectively. Figure 9 shows the normalized run-time performance when the SPM size is 5% of each application data size. We can see that the same trend is obtained as that of Figure 8.

(2) Performance improvement with different parameters. Table III shows a quantitative comparison between SPDP and the other three techniques when the SPM size is 2 k and 5% of each data size. When the parameters change, the ratios of SPDP: NDMA and SPDP: PDMA are changed accordingly. Compared with NDMA and PDMA, the less the ratio of ⟨Cdt:Cct⟩ is applied, the more gain SPDP can obtain. For example, when the ratio ⟨Cdt:Cct⟩ is ⟨1 : 100⟩, it is the least ratio among these parameters, and the most gain is obtain by SPDP compared with NDMA and PDMA. The reasons are as follows.

DOI: 10.1002/cpe

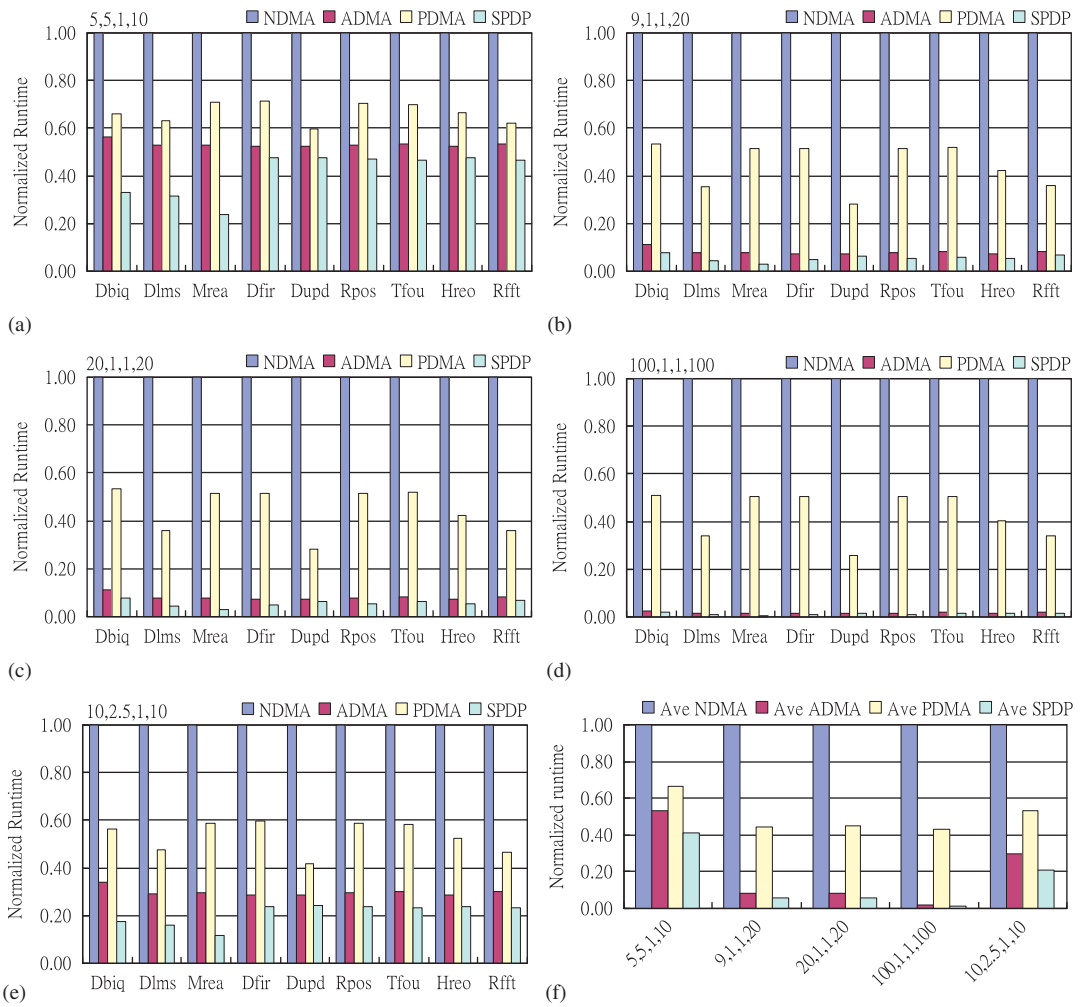


Figure 9. Normalized runtime for different techniques (SPM_Size = 5% of the data size).

Our SPDP technique applies DMA to transfer data completely, whereas NDMA transfer data by software approach and PDMA stores data to off-chip memory also relies on software approach. Compared with ADMA, SPDP gives a relative steady influence. Data transfer of both techniques is by DMA. SPDP divides SPM into two parts, hence the total *Cdi* cost is twice as ADMA. Table III shows that SPDP reduces the cost over that of ADMA. The experimental results demonstrate that the additional Cdi cost can be compensated by the gain of overlapping.

Table IV shows the results of the performance influenced with different values of parameters. From the results, we can see that the same parameter ratio with different values gives less influence on the performance.

DOI: 10.1002/cpe

Table III. The performance improvement of SPDP.

	Average improvement (%) over		
Parameters	NDMA	ADMA	PDMA
SPM_Size = 2k			
⟨5, 5, 1, 10⟩	0.4132	0.7772	0.6223
⟨9, 1, 1, 20⟩	0.0562	0.6846	0.1331
⟨20, 1, 1, 20⟩	0.0573	0.6929	0.1354
⟨100, 1, 1, 100⟩	0.0135	0.7494	0.0337
⟨10, 2, 5, 1, 10⟩	0.2093	0.7035	0.3990
SPM_Size = 5% of the data size of the benchmark			
⟨5, 5, 1, 10⟩	0.4126	0.7766	0.6221
⟨9, 1, 1, 20⟩	0.0557	0.6809	0.1321
⟨20, 1, 1, 20⟩	0.0562	0.6849	0.1332
⟨100, 1, 1, 100⟩	0.0124	0.7131	0.0309
⟨10, 2, 5, 1, 10⟩	0.2082	0.7014	0.3980

Table IV. Normalized runtime for different techniques with various parameters.

Parameters	NDMA	ADMA	PDMA	SPDP
SPM_Size = 2k				
⟨9, 1, 1, 20⟩	1.0000	0.0821	0.4463	0.0562
⟨18, 2, 1, 40⟩	1.0000	0.0666	0.4471	0.0454
⟨27, 3, 1, 60⟩	1.0000	0.0613	0.4487	0.0444
⟨36, 4, 1, 80⟩	1.0000	0.0586	0.4500	0.0446
⟨45, 5, 1, 100⟩	1.0000	0.0570	0.4508	0.0446
SPM_Size = 5% of the data size of the benchmark				
⟨5, 5, 1, 10⟩	1.0000	0.5346	0.6718	0.4170
⟨18, 2, 1, 40⟩	1.0000	0.5193	0.6939	0.4304
⟨27, 3, 1, 60⟩	1.0000	0.5112	0.7055	0.4374
⟨36, 4, 1, 80⟩	1.0000	0.5084	0.7095	0.4399
⟨45, 5, 1, 100⟩	1.0000	0.5070	0.7115	0.4411
⟨45, 5, 1, 100⟩	1.0000	0.5062	0.7127	0.4418

(3) Impact of the cost for DMA initialization. Based on the above analysis, we can find that both SPDP and ADMA are independent of the parameter of *Cct*. With different settings of Cdi, in Figure 10, we compare the corresponding performance improvements obtained by our SPDP technique and that of the ADMA technique. The results for the comparison are reported using a ratio of ⟨SPDP: ADMA⟩.

According to Figure 10, we can see that when parameter Cdt is unchanged, the bigger the parameter Cdi, the less the gain obtained by our SPDP technique. Even with the worst case ratio of 100:1, our SPDP still outperforms the ADMA technique. For all of the benchmarks, comparing our SPDP technique with ADMA, we achieve performance improvements ranging from 7.76 to 59.94%, with an average of 28.69%. The improvements are achieved due to the overlapping between data transfer and CPU execution generated by our technique.

(4) Impact of SPM size. Table V shows the normalized run-time performance with different SPM size. From the results, we can observe that the performance of SPDP, ADMA, and PDMA is slightly

DOI: 10.1002/cpe

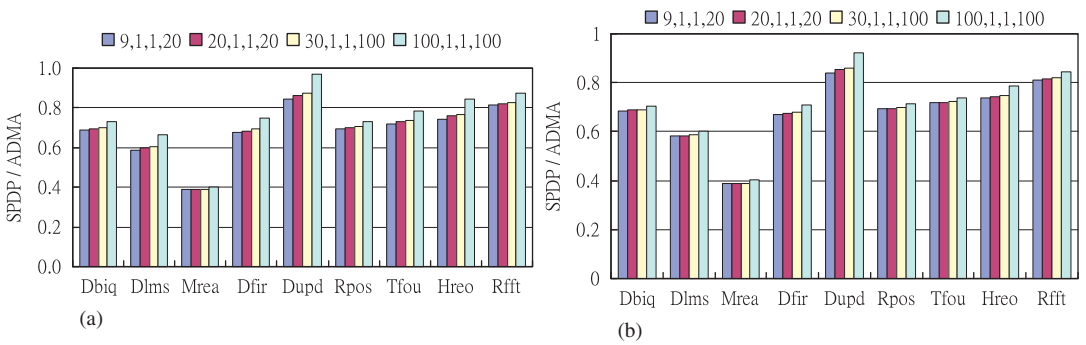


Figure 10. The impact of Cdi on the performance improvements obtained by our SPDP technique over that of the ADMA technique, when (a) SPM_Size = 2k and (b) SPM_Size = 5% of the data size of the benchmark.

Table V. Normalized performance improvements for different techniques with different SPM sizes.

SPM Size	NDMA	ADMA	PDMA	SPDP
The parameter is set as ⟨5:5:1:10⟩				
2k	1	0.53256	0.66734	0.41315
4k	1	0.53228	0.66673	0.41265
8k	1	0.53214	0.66642	0.41240
16k	1	0.53207	0.66627	0.41227
32k	1	0.53204	0.66619	0.41221
64k	1	0.53202	0.66615	0.41218
128k	1	0.53201	0.66614	0.41216
256k	1	0.53200	0.66613	0.41215
The parameter is set as ⟨10:2.5:1:10⟩				
0.5%	1	0.30322	0.54332	0.21680
1%	1	0.30061	0.53786	0.21205
5%	1	0.29852	0.53349	0.20824
10%	1	0.29826	0.53294	0.20777
20%	1	0.29813	0.53267	0.20753
40%	1	0.29807	0.53253	0.20741
50%	1	0.29806	0.53251	0.20739
60%	1	0.29805	0.53249	0.20737
80%	1	0.29804	0.53247	0.20735
100%	1	0.29803	0.53245	0.20734

affected by the various SPM sizes. And, the increasing SPM size contributes to small gains. This is because the data of small data size is frequently used in the loops. When the SPM size portion is varied from 0.5 to 5%, the performance improvements obtained from our SPDP method varies from 78.32 to 79.18%. Starting from the portion of 10%, the performance gain increases slowly. This is because the larger the SPM size, the less the block number. As a result, the total cost of DMA initialization is decreased. When the portion is larger than 10%, the total DMA initial cost becomes a smaller part compared with that of the total data transfer and execution. The results show that our SPDP can achieve high performance improvement with a small SPM size.

DOI: 10.1002/cpe

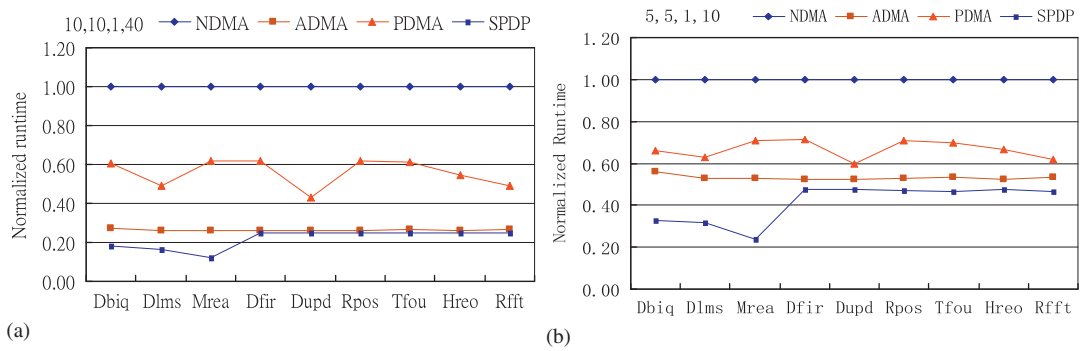


Figure 11. The normalized performance for benchmarks with different data access patterns, when (a) SPM_Size = 2k and (b) SPM_Size = 5% of the data size of the benchmark.

(5) Impact of iteration access pattern. Figure 11 illustrates the normalized performance for the benchmarks with different iteration access patterns. From the results, we can observe that SPDP has more gain in the first three applications as compared with the other six applications. The discussions are as follows. Basically, our SPDP improves the performance from two aspects. One is to reduce the total size of data transfer, and the other is to achieve the overlapping between the CPU execution and data transfer. However, for the latter six applications, our technique can only generate the overlapping. These results show that our SPDP is able to take the iteration access pattern into account.

6. RELATED WORK

In this section, we present the related work in terms of SPM allocation methods and DMA-based SPM management approaches, in Sections 6.1 and 6.2, respectively.

6.1. SPM allocation methods

The objects of SPM allocation can be classified into three categories: stack variables, global variables, and heap data. The study in [15] was the first compiler-managed method for allocating heaps data to SPM. In [7], the authors presented a general-purpose compiler approach, called memory coloring, to automatically allocate the arrays in a program to an SPM. In studies [4,5,16,17], both data and code can be allocated into SPM, such as [4] allocate global variable, stack variables, and program code to SPM. Except for studies [17,18], the size of the SPM must be known at compile time to determine the optimal SPM allocation. The scope of the SPM allocation object is either in a loop [19–22] or in whole program [4,5,23].

There are two methods of SPM allocation: static and dynamic. When the size of object of SPM allocation is greater than the size of SPM, the dynamic method is superior to the static method. This is because the content of SPM can be changed during run-time using dynamic method. The technique

DOI: 10.1002/cpe

proposed in [23] can place all global and stack variables in SPM dynamically, even in the presence of unrestricted pointers. In [4], the authors analyzed the program to identify locations called program pints where it may be beneficial to insert code to copy a variable from off-chip memory into SPM. However, the static SPM allocation method can not demonstrate the run-time SPM behavior since the content of SPM is fixed before execution. In both static method and dynamic methods, there are two common approaches to allocate SPM. On one side, [1] is a knapsack-based allocation. On another, the studies in [2,5,16,24–26] model the allocation problem as an integer linear programming model (ILP). The authors in [2] proposed an algorithm for assigning data elements onto the SPM, and the proposed algorithm was based upon profiling the application and solving a system of binary linear equations. In [16], the authors proposed a compiler extension which partitions program code and data into smaller segments whenever it is beneficial. In their scheme, the best set of program and data values are identified using integer linear programming, and the selected objects are placed onto SPM. In [27,28], SPM allocation methods for multi-processor and multiple applications have been proposed.

6.2. DMA-based SPM management approaches

Many embedded applications have significant data transfer requirements. In particular, many codes from video processing and signal-processing domains manipulate large arrays. Important issues of compiler-managed SPM are to maintain good data locality and transfer data between SPM and off-chip memory. To transfer data transfer with compiler control, software approaches have been used, in which the compiler not only inserts instructions of loading data from off-chip memory to SPM for computing, but also inserts instructions of storing data from SPM to off-chip memory for saving the result data and making space for incoming data. In the software approaches, CPU is fully responsible for both data transfer and data computing. Previous studies [29,30] show that high off-chip memory latencies are likely to be the limiting factor for future embedded systems. To reduce the serious off-chip memory access overheads, many studies have adopted the cost efficient hardware, DMA [4,8,15,23,31,32] or the same transfer cost model as DMA [6,7,19,26], to accelerate data transfer. Most of the above DMA-based SPM management techniques only focus on the data access frequency. For example, in [23], a profile-driven cost model was presented to estimate the benefit and cost. In this scheme, the complier identifies the variables repeatedly accessed and inserts instructions of DMA to allocate the variable into SPM. To reduce the run-time overhead and code size, the work in [4] used DMA to implement the data copy between off-chip memory and SPM.

7. CONCLUSION

In this paper, we proposed a compiler-assisted iteration-access-pattern-based data pipelining technique for dynamic SPM management with DMA. In the proposed technique, we exploited the chance to overlap the execution of CPU instructions and DMA operations so as to fully utilize the limited SPM space and to improve the performance of applications. We implemented our technique with the IMPACT compiler, and conducted experiments using a set of loop kernels from DSPstone, Mibench and Mediabench on the cycle-accurate VLIW simulator of Trimaran. The experimental results show that our technique achieves performance improvements compared with the previous work.

DOI: 10.1002/cpe

ACKNOWLEDGEMENTS

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF PolyU 5269/08E), the Hong Kong Polytechnic University (HK PolyU A-ZV5S), the National 863 Program of China (2008AA01Z106), and National Natural Science Foundation of China (60725208).

REFERENCES

1. Banakar R, Steinke S, Lee B, Balakrishnan M, Marwedel P. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. *Proceedings of the 10th International Symposium on Hardware/Software Codesign* (*CODES*). ACM: New York, 2002; 73–78.
2. Avissar O, Barua R, Stewart D. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems* (*TECS*) 2002; **1**(1):6–26.
3. 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*. ACM: New York, 2007; 65–74.
4. 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.
5. Verma M, Wehmeyer L, Marwedel P. Dynamic overlay of scratchpad memory for energy minimization. *Proceedings of the Second IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM: New York, 2004; 104–109.
6. Li L, Nguyen Q, Xue J. Scratchpad allocation for data aggregates in superperfect graphs. *Proceedings of the 2007 ACM Conference on Languages*, *Compilers and Tools for Embedded Systems* (*LCTES*). ACM: New York, 2007; 207–216.
7. Li L, Gao L, Xue J. Memory coloring: A compiler approach for scratchpad memory management. *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques* (*PACT*). IEEE: New York, 2005; 329–338.
8. Dasygenis M, Brockmeyer E, Durinck B, Catthoor F, Soudris D, Thanailakis A. An optimal memory allocation scheme for scratch-pad-based embedded systems. *IEEE Transactions on Very Large Scale Integration Systems* (*TVLSI*) 2006; **14**(3):279–291.
9. Chang PP, Mahlke SA, Chen WY, Warter NJ, Hwu WW. Impact: An architectural framework for multiple-instruction-issue processors. *Proceedings of the 18th International Symposium on Computer Architecture*. ACM: New York, 1991; 266–275.
10. Zivojnovic V, Martinez J, Schlager C, Meyr H. DSPstone: A DSP-oriented benchmarking methodology. *Proceedings of the 1994 International Conference on Signal Processing Applications and Technology*. IEEE: New York, 1994.
11. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB. Mibench: A free, commercially representative embedded benchmark suite. *Proceedings of the IEEE International Workshop on Workload Characterization*. IEEE: New York, 2001; 3–14.
12. Lee C, Potkonjak M, Mangione-Smith W. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*). ACM: New York, 1997; 330–335.
13. *The Trimaran Compiler Research Infrastructure*. Available at: http://www.trimaran.org/ [2010].
14. Mathew B, Davis A. A loop accelerator for low power embedded VLIW processors. *Proceedings of the Second IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (*CODES+ISSS '04*). ACM: New York, 2004; 6–11.
15. Dominguez A, Udayakumaran S, Barua R. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing* 2005; **1**(4):521–540.
16. Verma M, Steinke S, Marwedel P. Data partitioning for maximal scratchpad usage. *Proceedings of the 2003 Conference on Asia South Pacific Design Automation*. ACM: New York, 2003; 77–83.
17. Nguyen N, Dominguez A, Barua R. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *Proceedings of the ACM International Conference on Compilers*, *Architecture*, *and Synthesis for Embedded Systems*. ACM: New York, 2005; 115–125.
18. Egger B, Lee J, Shin H. Scratchpad memory management for portable systems with a memory management unit. *Proceedings of the International Conference on Embedded Software*. ACM: New York, 2006; 321–330.
19. Kandemir M, Ramanujam J, Irwin MJ, Vijaykrishnan N, Kadayi I, Parikh A. Dynamic management of scratch-pad memory space. *Proceedings of the 38th Annual Design Automation Conference*. ACM: New York, 2001; 690–695.

20. 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*. ACM: New York, 2007; 195–206.
21. 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*. ACM: New York, 2002; 213–218.
22. Anantharaman S, Pande S. Compiler optimizations for real time execution of loops on limited memory embedded systems. *Proceedings of the IEEE Real-Time Systems Symposium*. IEEE: New York, 1998; 154–164.
23. Udayakumaran S, Barua R. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM: New York, 2003; 276–286.
24. 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*. ACM: New York, 2001; 34–43.
25. Jan S, Carl P. Storage allocation for embedded processors. *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM: New York, 2001; 15–23.
26. Li L, Wu H, Feng H, Xue J. Towards data tiling for whole programs in scratchpad memory allocation. *Proceedings of the 12th Asia-Pacific Computer Systems Architecture Conference*. IEEE: New York, 2007; 63–74.
27. Kandemir M, Ramanujam J, Choudhary A. Exploiting shared scratch pad memory space in embedded multiprocessor systems. *Proceedings of the 39th Annual Design Automation Conference*. ACM: New York, 2002; 219–224.
28. Ozturk O, Kandemir M, Kolcu I. Shared scratch-pad memory space management. *Proceedings of the Seventh International Symposium on Quality Electronic Design*. IEEE: New York, 2006; 576–584.
29. Catthoor F, de Greef E, Suytack S. *Custom Memory Management Methodology*: *Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers: Dordrecht, 1998.
30. Vijaykrishnan N, Kandemir M, Irwin MJ, Kim HS, Ye W. Energy-driven integrated hardware-software optimizations using simplepower. *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM: New York, 2000; 95–106.
31. Chen G, Ozturk O, Kandemir M, Karakoy M. Dynamic scratch-pad memory management for irregular array access patterns. *Proceedings of the Conference on Design, Automation and Test in Europe*, 2006; 931–936.
32. Francesco P, Marchal P, Atienza D, Benini L, Catthoor F, Mendias JM. An integrated hardware/software approach for run-time scratchpad management. *Proceedings of the 41st Annual Design Automation Conference*. ACM: New York, 2004; 238–243.

DOI: 10.1002/cpe