ISOS: Space Overlapping Based on Iteration Access Patterns for Dynamic Scratch-pad Memory Management in Embedded Systems

Yanqin Yang^{1,3}, Zili Shao^{2}, Linfeng Pan^{1}, Minyi Guo^{1}
1. Dept. of Computer Science and Engineering, Shanghai Jiao-Tong Univ. Shanghai, China
2.Dept. of Computing, Hong Kong Polytechnic Univ., Hung Hom, Kowloon, Hong Kong
3. Dept. of Computer Science and technology, East China Normal Univ., Shanghai, China
Email: yang-yq@sjtu.edu.cn, cszlshao@comp.polyu.edu.hk, guo-my@cs.sjtu.edu.cn

Abstract

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 iteration-access-pattern-based space overlapping technique for dynamic SPM management (ISOS) 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 form 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.

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] with its advantages in energy and area [2-5]. A recent study [6] 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 significantly reduce the energy consumption for embedded processors. Embedded software is usually optimized for specific applications, so we can utilize SPM to improve performance and predictability by avoiding cache misses. With these advantages, SPM has become the most common SRAM in embedded processors. However, it posts a big challenge for 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 [2-5] and dynamic method [1, 7-9]. 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 very 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, 2, 7], and many embedded applications in image and video processing domains have significant data transfer requirements in addition to their computational requirements [8]. To reduce off-chip memory access overheads, the dedicated cost-efficient hardware, DMA (Direct Memory Access), 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 [7, 9, 12-13]. In [7], 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 [9, 12-13] to accelerate data transfer between off-chip memory and SPM. However, most of the above work focuses on array allocation optimization in SPM without considering optimizing DMA transfer. Because SPM is a small on-chip memory, we cannot put all the necessary data at one time taking the power, size of embedded system into account. Therefore, multiple times of DMA transfer are needed for arrays

978-0-7695-3398-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ICYCS.2008.538

in loops in dynamic SPM management. 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 iteration-access pattern-based space overlapping technique for SPM management (ISOS) 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 after the array elements will not be used in later iterations. In such a way, if there are spaces that can be overlapped, we can allocate more array elements into SPM in each DMA transfer so as to reduce the number of DMA operations. We implement our technique based on IMPACT [10] and conduct experimental using the benchmarks from DSPstone and Mediabench on the cycle-accurate VLIW simulator of Trimaran [11]. The experimental results show that ISOS achieves significant the run-time performance improvement compared with the previous work.

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

2. Model and basic concepts

2.1. System Model

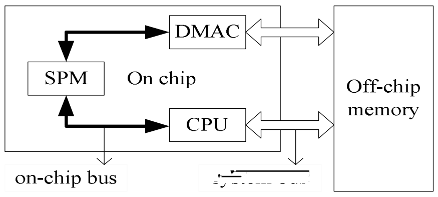


Figure 1. The system model

The system model is shown in Figure 1 which has the similar architecture as that in [7, 9]. 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 DAM 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 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.

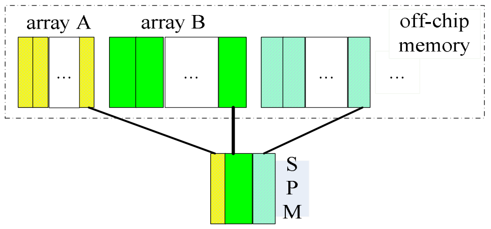


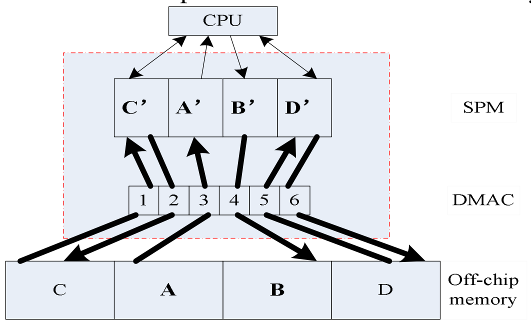
Figure 2. The data transfer between the SPM and off-chip memory

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 block and insert DMA operations for block-level data transfer.

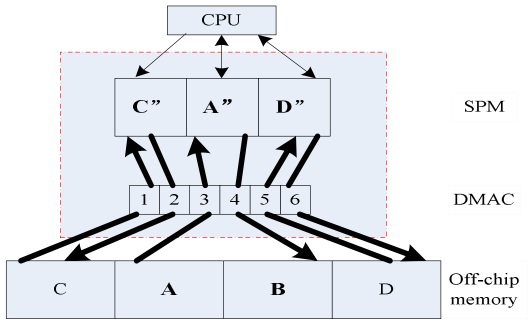
Based on the model in [7, 9], the cost of transferring a data block between SPM and off-chip memory is approximated by $(Cdi + Cdt *n)$ in cycles, where Cdi is the initialization cost of DMAC for one block, including all the latencies of arbitration and synchronization, Cdt is the cost per byte transfer, and n is the number of bytes in a block. The total cost of transferring an array is approximated by $N_b*(Cdi + Cdt *n)$, where N_b is the number of blocks of an array. In our approach, we can reduce N_b 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 such a way 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 6 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. Basically, we allocate space for all the arrays and assign DMA channels for data transfer. However, since array A is read-only, after one data has been used and will not be used again, we can use that space to hold data generated by array B. In this way, by a 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 a overlapping, we can assign more space to other arrays so as to utilize the space of the SPM more effectively.



(a) The SPM space allocation without space overlapping.



(b) The SPM space allocation with space overlapping by ISOS.

Figure 3. Two SPM management strategies.

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.

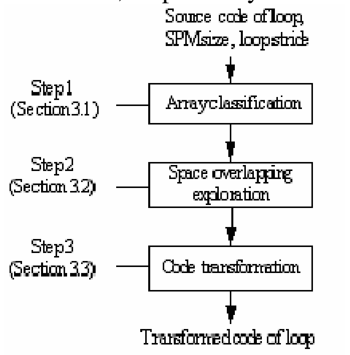


Figure 4. The overview of the ISOS technique

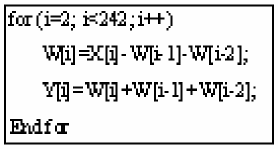
The ISOS technique mainly consists of the following three steps as shown in Figure 4:

- *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. Basically, 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. Basically, we insert instructions for DMA transfer and transform array references to map the space overlapping.

3.1 Step1: 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 operation 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 same memory location, the first operation must be the write operation; (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.



```
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];
Endfor
```

Figure 5. An exemplary loop.

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 read-only array because there is only read operation; Y is write-only array because there is only write operation; W is write-advance-read array because the two conditions are satisfied, which means the values read by all read operations are generated by the write operations in the current or previous iterations.

3.2 Step 2: Space Overlapping Exploration

Based on the array classification above, we can then perform space overlapping. Basically, 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 will not be used in later iterations. Therefore, a candidate array must have the following feature: 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 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 ISOR, 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 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 Resued_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. The DMA channels then can be allocated accordingly.

In Figure 6, we give an example for the space overlapping of the loop shown in Figure 5. Here we

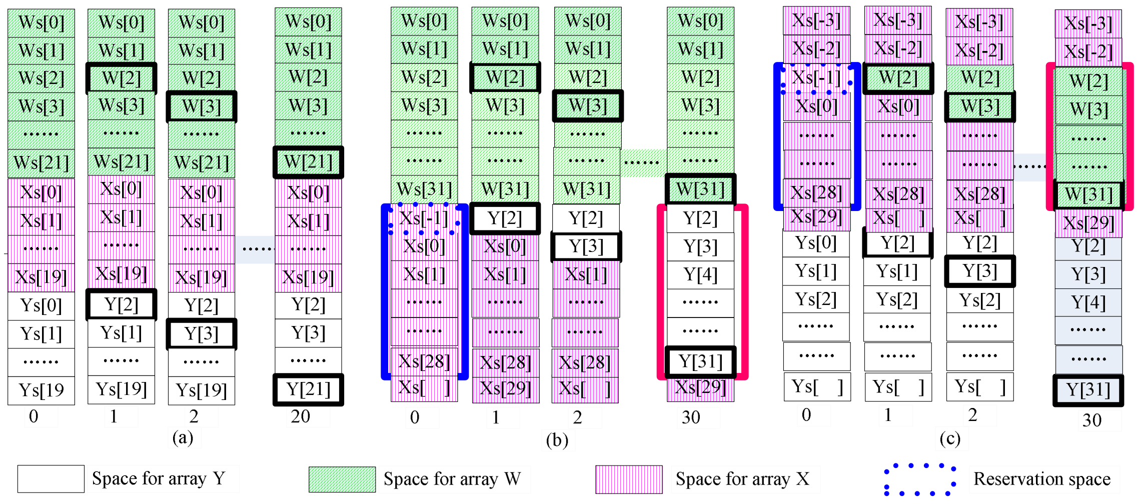


Figure 6. The memory layouts of the initial state and from iteration 1 to 20 in the case that (a) no space overlapping; (b) space overlapping between array *X* and array *Y*; (c) space overlapping between array *X* and array *W*.

```
    Ws[0] = 1;  Ws[1] = 2;
for (ie = 2; ie < 242; 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];
    end for
    DMA (AS_Ws[ie’ ], AD_W[ie], 20);
    DMA (AS_Ys[ie’ -2], AD_Y[ie], 20);
    DMA ready
    Ws[0]=Ws[ib-1]; Ws[1]=Ws[ib];
end for
```

(a)

```
    Ws[0] = 1;  Ws[1] = 2;
for (ie = 2; ie< 242; 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];
     end for
    DMA(AS_Ws[ie’ ],AD_W[ie],  30);
    DMA(AS_Xs[ie’ -3], AD_Y[ie], 30);
    DMA ready
    Ws[0]=Ws[ib-1];Ws[1]=Ws[ib];
end for
```

(b)

```
    X[-3] = W[0] = 1;  X[-2] = W[1] = 2;
for (ie = 2; i < 242; i +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];
     end for
    DMA (AS_Xs[ie’ -3], AD_W[ie], 30);
    DMA (AS_Ys[ie’ -2], AD_Y[ie], 30);
    DMA ready
    Ws[0]=Ws[ib-1];  Ws[1]=Ws[ib];
end for
```

(c)

Figure 7. 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*; (c) the code with space overlapping between arrays *X* and *W*.

assume the size of the SPM is 63, which means the SPM can contain at most 63 array elements. *Ws [ ], Ys [ ], Xs [ ]* denote the space for elements of array *W, Y, X* in the SPM, and *W [ ], Y [ ]* denote the array elements stored from the SPM to the off-chip memory.

Figure 6 (a) shows the case without space overlapping among array *W, X* and *Y*. In this case, each array needs the SPM space; therefore, we allocate 20 elements to each array in the SPM, and accordingly the block size is 20 iterations. In Figure 6(a), the memory layouts at the initial state and from iterations 1 to 20 are given. We can see that after each iteration, the space for one element from *Xs* is unused which can be utilized for space overlapping.

Figure 6 (b) shows the case that the unused space of array X is utilized for space overlapping for the space of arrays 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 (30 elements) to 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. Basically, based on the number of iterations for one block and Array_Pair_Set, the code transformation consists of the following steps: (1) Insert DMA operations for data transfer between SPM and off-chip memory; (2) Change array references based on each array pair in Array_Pair_Set; (3) Transfer the loop into the two-level loop. In Figure 7, we give an example to show the code transformation for the loop shown in Figure 5.

4. Experiments

We implement our approach based on the *IMPACT* compiler [10] and conducted experiments on the cycle-accurate *VLIW* simulator of *Trimaran* [11]. The experimental configuration for *Trimaran* simulator is shown in Table1.

Table 1. The configuration for the *Trimaran* simulator

Parameter	Configuration
Function Units	2 integer ALU, 2 floating point ALU, 2 load-store units,1 branch unit
Instruction Latency	1cycle for integer ALU, 1 cycle for floating point ALU,2 cycles for load in cache, 1 cycle for store,1 cycle for branch
Register file	64 integer registers, 64 floating point registers

In our experiments, we extract 12 loop kernels from *Mediabench* and *DSPstone* benchmarks, and the loop kernels are shown in Table 2. In Table 2, the required memory size for each loop kernel is given in Column “Data Size”. To test the run-time performance, we use the following parameters, *Cdi*=100 and *Cdt*=1, which have been used in [7, 9], where *Cdi* is the number of cycles for CPU initializing a DMA block transfer, and

Table 2: The loop kernels from MediaBench and DSPstone.

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

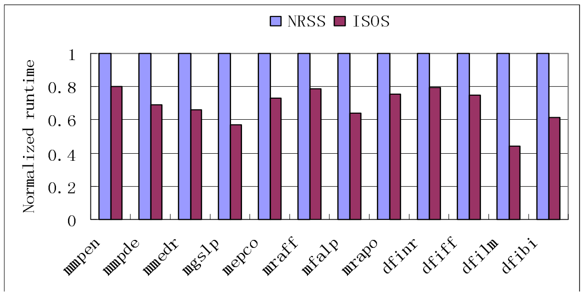
Cdt is the number of cycles for DMA transfer a byte between off-chip memory and SPM.

We compare our *ISOS* technique with the SPM management technique in [9] that is named as *NRSS*. *NRSS* is a dynamic SPM management technique to manage SPM using a combination of loop access pattern and off-chip memory layout but without space overlapping. In all experiments, the time performance is normalized based on that of *NRSS*.

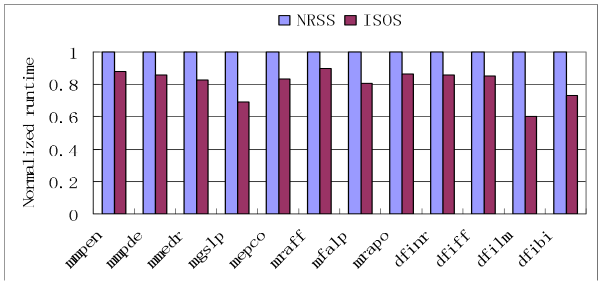
We first conduct experiments with fixed SPM sizes. The experimental results are shown in Figure 8 when the SPM sizes vary from 256 bytes, 512 bytes to 1 K bytes. From Figure 8, we can see that ISOS can achieve better time performance compared with NRSS with various SPM sizes in all loop kernels. The average improvements are 13.15%, 19.05%, and 25.52% when the SPM sizes are 1K bytes, 512 bytes and 256 bytes, respectively.

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 2). As the memory spaces needed are different from different loop kernels, we want to compare performances for different benchmarks with the same SPM/memory percentage. The experimental results are shown in Figure 9 when the percentage of SPM/Data Size vary from 1%, 2% to 3%. The average improvements are 25.73%, 20.05% and 16.73% when the percentages of SPM/Data Sizes are 1%, 2% and 3%, respectively.

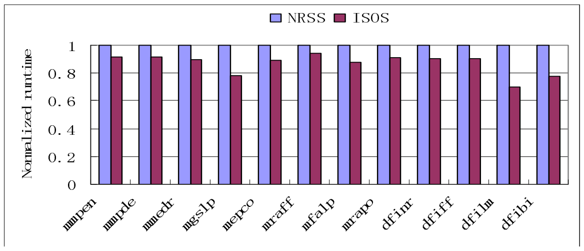
From Figures 8 and 9, we can see that our ISOS can achieve better results compared with NRSS [9] in all cases.



(a) The SPM Size is 256 bytes.

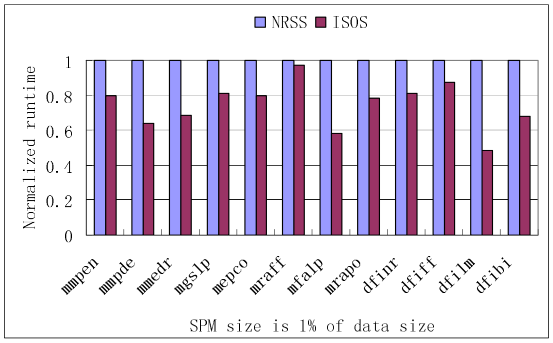


(b) The SPM size is 512 bytes.

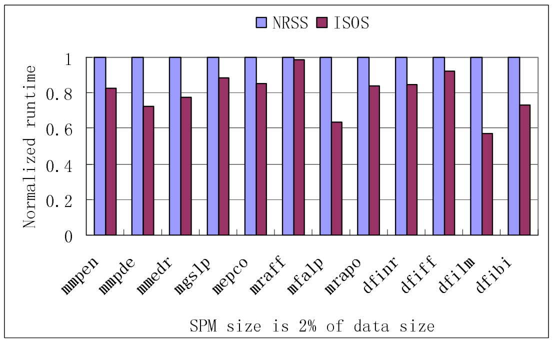


(c) The SPM size is 1K bytes.

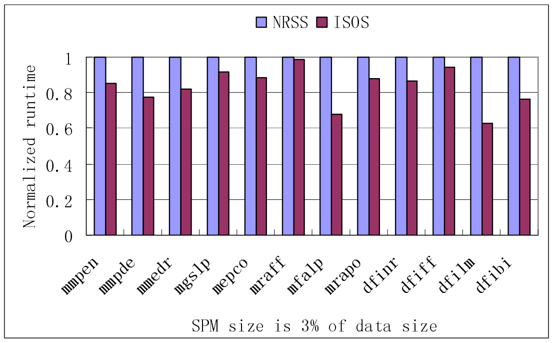
Figure 8. The time performance comparisons of ISOS and NRSS with different SPM sizes (normalized to NRSS).



(a)



(b)



(c)

Figure 9. The time performance comparisons of ISOS and NRSS with different percentages of SPM/Data Size (normalized to NRSS).

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 form 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.

Acknowledgment

This work is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (CERG 526007 (PolyU B-Q06B), PolyU A-PH41), National 863 Program of China (Grant No. 2006AA01Z172), National Natural Science Foundation of China (Grant No. 60533040), and National Science Fund for Distinguished Young Scholars (Grant No. 60725208).

References

[1] Dominguez, A., N. Nguyen, and R.K. Barua. Recursive function data allocation to scratch-pad memory. In Proceedings of CASES 2007.

[2] Panda, P.R., N.D. Dutt, and A. Nicolau, Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In Proc. of DATE 1997, pp. 7-11, 1997.

[3] Panda, P.R., Memory issues in embedded systems-on-chip. 1999: Kluwer Academic Boston.

[4] Avissar, O., R. Barua, and D. Stewart, An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Transactions on Embedded Systems (TECS), 2002. 1(1): p. 6-26.

[5] Kandemir, M., I. Kadayif, and U. Sezer, Exploiting scratch-pad memory using Presburger formulas. In Proc. of ISSS 2001, pp. 7-12, 2001.

[6] Banakar, R., et al., Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In Proc. of CODES 2002, pp. 73-78, 2002.

[7] Udayakumaran, S., A. Dominguez, and R. Barua, Dynamic allocation for scratch-pad memory using compile-time decisions. ACM Transactions on Embedded Computing Systems (TECS), 2006. 5(2): p. 472-511.

[8] Steinke, S., et al. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In Proc. of ISSS 2002.

[9] Kandemir, M., et al., Dynamic management of scratch-pad memory space, in Proc. of the 2001 Design Automation Conference, pp. 690-695, 2001.

[10] Chang, P.P., et al. IMPACT: an architectural framework for multiple-instruction-issue processors. In Proc. of ISCA, 1991.

[11] The Trimaran Compiler Research Infrastructure, http://www.trimaran.org/.

[12] Li, L., L. Gao, and J. Xue, Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In PACT 05, pp. 329-338, 2005.

[13] Dominguez, A., S. Udayakumaran, and R. Barua, Heap data allocation to scratch-pad memory in embedded systems. Journal of Embedded Computing(JEC) 2005. 1(4): p. 521-540.