A Novel Technique to Use Scratch-pad Memory for Stack Management

Soyoung Park Hae-woo Park Soonhoi Ha

School of EECS, Seoul National University, Seoul, Korea
{soy, starlet, sha}@iris.snu.ac.kr

Abstract

Extensive work has been done for optimal management of scratch-pad memory (SPM) all assuming that the SPM is assigned a fixed address space. The main target objects to be placed on the SPM have been code and global memory since their sizes and locations are not changed dynamically. We propose a novel idea of dynamic address mapping of SPM with the assistance of memory management unit (MMU). It allows us to use SPM for stack management without architecture modification and complier assistance. The proposed technique is orthogonal to the previous works so can be used at the same time. Experiments results show that the proposed technique results in average performance improvement of 13% and energy savings of 12% observed compared to using only external DRAM. And it also gives noticeable speed up and energy saving against a typical cache solution for stack data.

1. Introduction

Compared with the cache, the scratch-pad memory (SPM) has potential advantage for low power embedded systems since it does not need additional hardware logic for managing the contents. Consequently, to make the SPM the alternative solution of on-chip SRAMs, extensive work has been done for optimal management of SPM. Optimal management of SPM aims to find out the best memory objects to be placed on SPM statically or dynamically to maximize the performance.

The previous works assume that the SPM is assigned a fixed address space. Thus the main target objects for SPM management have been code and/or global data since their sizes and locations are not varied at run time. On the other hand, local variables that are stored in stack vary their locations dynamically.

Most of the previous works are based on compile time analysis that examines the access pattern of memory objects to find out the most frequently used. As the application size gets large, the computational complexity of compile time analysis becomes a limiting factor for optimal decision. And, it requires compiler modification to put the explicit copy operation of memory objects onto the SPM.

In this paper, we propose a novel idea of dynamic address mapping of SPM with the assistance of memory management unit (MMU). It allows us to use SPM for stack area without architecture modification and complier assistance. We just assume the environment supporting MMU, which is getting more popular in SoC products. The proposed technique is implemented with a reset handler and a permission fault handler of MMU software. In spite of its simplicity, it takes the advantage of the stack locality quite well. The proposed technique is orthogonal and complementary to the previous works.

The rest of this paper is organized as follows. Section 2 reviews related work and section 3 gives the basic idea of our approach. In section 4, we present details on the proposed dynamic slot paging method. Section 5 describes the experimental results and discussion. Finally section 6 concludes the paper with summary of our work.

2. Related Work

Contrary to cache systems, SPM requires careful management to maximize the performance. So there have been extensive works performed recently on optimal management of SPM. They can be categorized by what objects to allocate and by when to perform allocation. To determine the candidate objects to place in the SPM several techniques such as static analysis, memory tracing, and profiling are used to obtain the size, life-times, access frequency of the objects.

Static allocation methods ([3], and [4]) fix the placement of code or data before run time. For instance, the optimal placement algorithm proposed in [3] is formulated as the knapsack problem[14] that is NP-complete. Therefore, ILP is used as a heuristic in order to reducing computation overhead.

In dynamic allocation techniques, contents in SPM are changed at runtime. Dynamic overlay of SPM[8] copies both code and data during run time paying the cost of copy overhead. It has been shown that the advantage of dynamic overlay is larger than the copy overheads. In order to assist dynamic managing procedure, [9] splits data arrays into tiles, [10] uses data compression, [11] adds DMA hardware to reduce the copy overhead, and [6] uses compiler-inserted code to exploit dynamic program behavior.

As mentioned in the previous section, all previous works, to our best knowledge, assume that the address map of the SPM is fixed. Therefore they mainly consider code or global data for candidate objects for SPM allocation. Local variables positioned in the stack area at run time are not good candidates since their locations vary. In [3], compile-time analysis on local variables is performed in much the same way as global data. Call graph analysis is applied in order to use the same SPM space for different functions.

978-3-9810801-2-4/DATE07 © 2007 EDAA

It proposed to split the stack into two separate memory units, the SPM and the DRAM, as shown in Figure 1. For this scheme, architecture modification is necessary, which is not feasible in many cases.

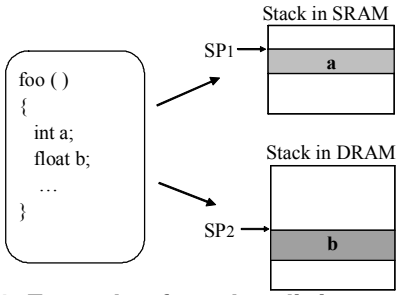

Figure 1. Example of stack split into two separate memory units.[3]

Compared with these related works, the proposed technique has the following unique characteristics:

1. The address map of SPM is changed dynamically at run time.
2. No architecture modification or compiler assistance is needed.
3. No explicit copying is needed in the application software.

As a result, the proposed technique enables us to use SPM for stack data to exploit the locality of local variables. It is the main theme of this paper.

3. Basic Idea

The proposed approach concerns about the locality characteristic that can be generally observed in stack data access as shown in Figure 2 with a simple example.

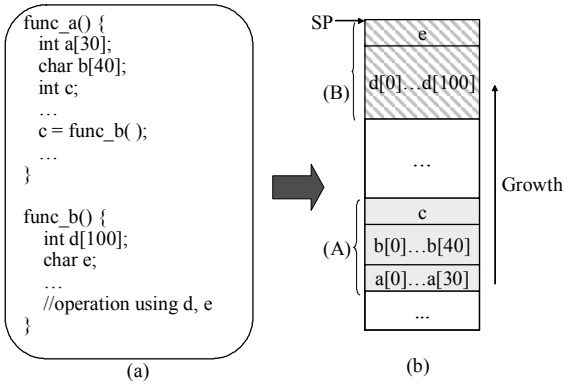

Figure 2. (a) Example program, (b) Stack view at point of executing func_b()

Suppose that func_b() is currently being executed in the program shown in Figure 2-(a) with the status of stack as displayed in Figure 2-(b). Then, the region (B) of stack is the most frequently accessed region (we call it the working set in this paper) during execution of func_b(). After func_b() ends, the stack becomes shrunken down and now the working set of stack is changed to region (A) that stores the local variables of func_a().

While the working set of the stack varies dynamically, the following fact remains consistent. The working set of the stack is located near the stack pointer. To exploit this locality characteristic, we propose to change the address map of SPM dynamically following the stack pointer. The proposed technique is similar to the virtual memory systems: SPM is to DRAM what DRAM is to hard disk. The SPM is logically partitioned into slots as the DRAM is partitioned into pages in the virtual memory systems.

Initially, the entire SPM space is mapped to the bottom of the stack area. SPM can be used for entire stack if the stack does not overgrow the SPM space. However, as the stack size gets larger than the SPM size, we find an SPM slot for replacement, expel it to DRAM area after copying its contents, and map the top portion of the stack to the slot. The resultant stack area consists of the SPM and the DRAM.

To understand the management policy in more details, we consider two cases separately.

CASE 1: It is the case when the stack grows over the SPM space as mentioned above. If there is any attempt to access the stack region that is over the SPM region, a permission fault occurs for non-allocated region. Then, the fault handler selects some slot of the SPM, copies it to a DRAM region, and allocates the slot as requested.

CASE 2: It is the case when the access below the SPM region is made inside the stack. It may occur when the current function completes its execution or the current working set is larger than the SPM area. This case also signals the fault handler. The fault handler finds the deallocated SPM slots if exist and copies the DRAM slots into the deallocated SPM slots.

In this way, our method behaves like sliding SPM up and down as the stack pointer moves. Since the data near the top always exists in the SPM, it takes advantage of the stack locality.

4. Dynamic Slot Paging

The proposed technique is based on an assumption that the target architecture includes MMU. The proposed technique modifies the address mapping of the SPM by modifying the associated page table entries at run time. When the modification is needed, we let permission faults be generated so that the fault handler modifies the page table and performs memory management. The key technique is to manipulate the access permission bits, shortly AP bits in ARM processors, of the page table entries. As explained earlier, we divide the SPM area into slots: a slot is a group of pages and its size is determined empirically, as will be discussed in the next section. Since a slot is the unit of operation, we name the proposed technique “dynamic slot paging”.

Initially, the reset handler maps the SPM at the bottom of the stack as shown in Figure 3. In the figure, a slot contains only one page of size 1K. On top of the SPM, the external DRAM is placed, also divided into slots logically. We mark the beginning address of the DRAM space above

the SPM space in the stack as *DRAM Base*. The rest handler disables the AP bits of the DRAM page table entries in the stack area to generate permission faults for any access into the DRAM space.

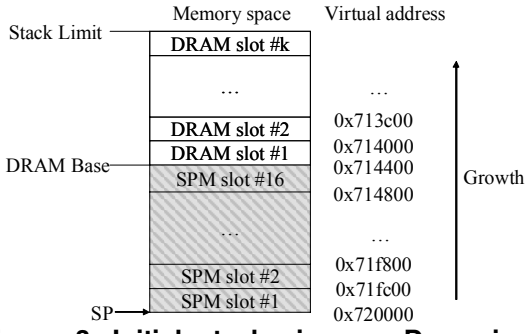

Figure 3. Initial stack view on Dynamic Slot Paging (16KB SPM, slot size : 1KB)

We classify permission faults into two cases, Case 1 and Case 2. When a permission fault occurs, the fault handler compares the aborted logical address with the DRAM Base. If the logical address is lower than the DRAM Base, the fault is classified into Case 1. Otherwise it is classified as Case 2. Next the handler performs different operations for two cases.

4.1. SPM Management for Case 1

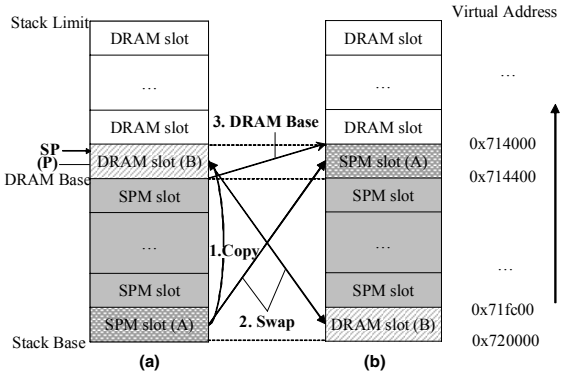

Figure 4. Stack structure: (a) at the abort condition leading to Case 1 fault, (b) after fault handler completion

The Case 1 fault usually occurs when the stack grows over the SPM size, so the stack pointer points to a DRAM slot (B) as illustrated in Figure 4-(a). As the program tries to access the data located at (P), the MMU aborts the access and signals an access permission fault. Then the operation of the fault handler consists of 6 steps as follows:

1. Pick the SPM slot that is located closest to the stack base. In the figure, SPM slot (A) is selected for replacement.
2. Enable the AP bits of the page table entries associated with the aborted DRAM slot. Then we can access the DRAM slot without write permission fault in the next step.
3. Copy the valid stack data in the SPM slot (A) into the aborted DRAM slot (B) using load-store instructions.
4. Swap the address map of the SPM slot and the aborted DRAM slot in the page table. Now we call this DRAM slot as a *backup slot*. A backup slot contains valid stack data in DRAM below the SPM space.
5. After address swapping is completed, disable again the AP bits of the page table entry associated with the DRAM slot. It allows us to detect Case 2 fault afterwards.
6. Finally, update the DRAM Base to point the next DRAM slot. The DRAM base always indicates the top of the SPM space.

After the fault handler finishes its operation, the stack becomes like Figure 4–(b). Note that the SPM space moves one slot up to cover the top of the stack. So the locality of the stack access is fully exploited in the proposed technique. Experiments reveal that the copy overhead of step 3 is relatively small, so we could get significant performance improvement for the stack data, which will be discussed in the next section.

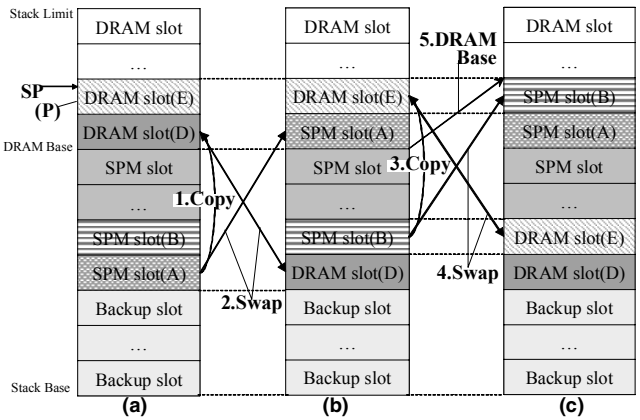

Figure 5. Stack structure: (a) an example of Case 1 permission fault when the stack pointer grows over multiple DRAM slots, (b) after the aborted slot (D) management, (c) after fault handler completion

We may need to apply this management for several DRAM slots when the stack pointer moves over multiple DRAM slots. Let us consider the situation of Figure 5-(a) where the aborted DRAM slot(E) is located one slot over the SPM space. In this case, the handler performs its operations sequentially for all DRAM slots between (P) and the DRAM base. In the figure, we swap two DRAM slots (D) and (E) with two SPM slots at the bottom (A) and (B).

4.2. SPM Management for Case 2

Due to step 5 of Case 1 fault handler operation, access to the backup DRAM slot also generates a permission fault. Note that even though the DRAM contains the valid data, it generates a fault. We can further classify Case 2 into two situations. The first situation is when the stack shrinks below the entire SPM space as presented in Figure 6-(a).

Then the operation of the fault handler consists of the following 7 steps.

1. Select the top SPM slot just below the DRAM Base. In the figure, SPM slot (F) is selected for replacement.
2. Enable the AP bits of the page table entries associated with the aborted DRAM backup slot. Then we can access the DRAM slot without read permission fault in the next step.
3. Copy the valid stack data in the aborted backup slot (A) into the SPM slot (F) using load-store instructions.
4. Swap the address map of the SPM slot and the DRAM backup slot in the page table.
5. For each backup slot above the updated stack pointer, select the top SPM slot and swap the address map without copying data. After stack pointer update, no SPM slot except the bottom slot contains valid data. Therefore data copy is not needed.
6. After address swapping is completed, disable again the AP bits of the page table entry associated with the DRAM slots.
7. Finally, update the DRAM Base to point the first DRAM slot over the SPM space.

Figure 6-(c) displays the final stack structure after fault handler execution.

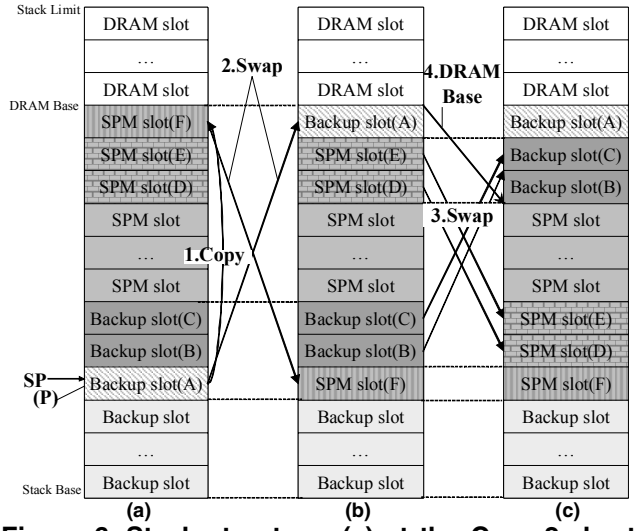

Figure 6. Stack structure: (a) at the Case 2-abort status with shrinking the stack pointer, (b) after the aborted slot (A) management, (c) after fault handler completion

The other situation corresponds to the case when an access is attempted to a backup slot without modifying the stack pointer. We call this backup slot as the target backup slot. Then we swap the target backup slot with a free SPM slot above the stack pointer if exists. If there is no free SPM slot, we just enable the access permission of the backup slot. If there is a free SPM slot as depicted in Figure 7, the fault handler operates by the following steps.

1. Select the backup slot below the SPM space. In this figure, Backup slot (B) is selected.
2. Enable the AP bits of the page table entries associated with the backup slot. Then we can access the DRAM slot without read permission fault in the next step.
3. Copy the valid stack data in the backup slot (B) into the SPM slot (D) using load-store instructions.
4. Swap the address map of the SPM slot and the backup slot in the page table. The backup slot becomes a DRAM slot above the SPM space.
5. After address swapping is completed, disable again the AP bits of the page table entry associated with the DRAM slot.
6. Select the backup slot below the SPM space again until we meet the target backup slot and perform steps 3 to 6.
7. Finally, update the DRAM Base to point the first DRAM slot over the SPM space.

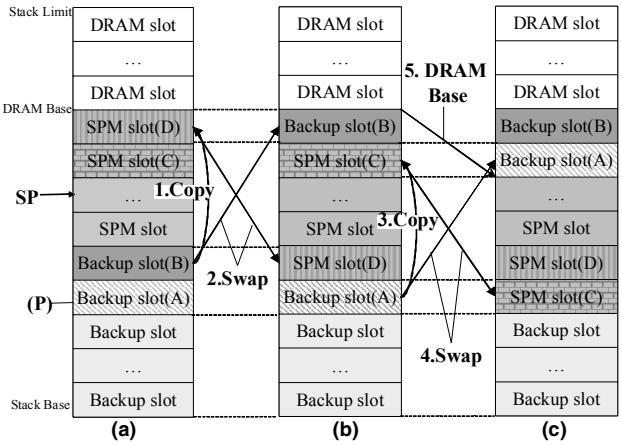

Figure 7. Stack structure: (a) at the Case2-abort status without shrinking the stack pointer, (b) after backup slot (B) management, (c) after fault handler completion

The resultant stack structure after the fault handler operation is shown in Figure 7-(c). The proposed technique places the SPM space always at the top of the stack assuming that the top region of the stack tends to be accessed more often than the other region.

5. Experiments

The target architecture consists of an ARM926EJ-S that has an MMU, 4-associative on-chip cache, and on-chip scratchpad memory, and an off-chip RAM. We implemented a reset handler and a permission fault handler in ARM assembly language for the target architecture. We used page size of 1KB for the stack, and 1MB for the other memory regions. The total execution cycle is measured by cycle accurate simulation using ARMulator[1]. In cycle measurement we assumed that the access delay of the off-chip DRAM is 20 cycles. And we estimated the energy consumption based on the cycle information. The energy consumption per access for on-chip cache, scratchpad memory, and TLBs is obtained from the CACTI cache

model[12] for 0.13μm technology. We borrowed the parameters of each memory component for the cache model from [15]. And the other figures for the energy consumption are taken from [4] and [7].

Table 1. Benchmark applications (size: bytes)

Application	Description	Global data	Stack depth	On-chip
gunzip	Uncompress a gzip file	340,584	442	1,024
minizip	Compress a gzip file	16,588	17,976	4,096
jpeg-comp	Compress a jpeg file	12,244	2,000	1,024
jpeg-trans	Transpose a jpeg image	10,324	2,608	2,048
x264	Encode h264/avc video streams[16]	3,096,828	25,584	8,192
mp3	Decode mp3 audio streams	120,776	49,784	16,384

The benchmarks used in experiments include gzip applications[17] and several multimedia applications from Mediabench[18]. The application characteristics and the on-chip memory size we used are summarized in Table 1. In the first set of experiments, we used the slot size of 1KB, which is same with the page size for stack region.

Figure 8 displays the normalized total execution cycles of benchmark applications. The reference performance is the case without any on-chip SRAM as denoted by (A) in the figure. To focus on the performance comparison only for the stack data, we disabled the cacheable flags of pages related to the global data and the code sections in this experiment. We compared the proposed technique, denoted by (C), with the cache solution, denoted by (B), for the stack data. The proposed technique outperforms the DRAM only solution by average 13% in the total execution time. It reveals that the stack operation takes significant portion of execution time. If the total execution time is reduced by using caches for code and global data, the percentage improvement will be larger. An important observation is that the proposed technique gives better performance than the cache solution by 2% to 18%. It indicates that our approach takes more advantage of stack locality than general cache policy.

We measured the runtime-overhead for managing permission fault in Figure 9 for the benchmark applications, except gunzip that has no fault generated. The figure shows that the overhead is insignificant, below 0.1% of total execution time for all applications. In fact the number of page faults is pretty low due to stack locality. It means the proposed technique does not require any hardware overhead such as DMA to reduce the overhead.

Figure 10 presents the estimated energy consumption for each case. Though TLBs give extra energy consumption, the proposed technique gives 12% power saving on average over (A) and average of 8% over (B). In case of x264, the speed up(7%) and the energy saving(6%) is relatively small compared with the other applications because the reference code defines large size of global data and reuses the global data even for the local usage inside functions. So stack access takes relatively low portion of the total execution.

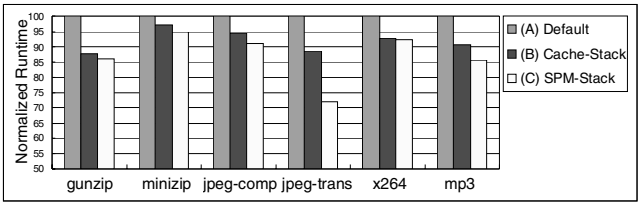

Figure 8. Execution time comparison among three: DRAM-only solution, cache solution, and SPM solution for stack data.

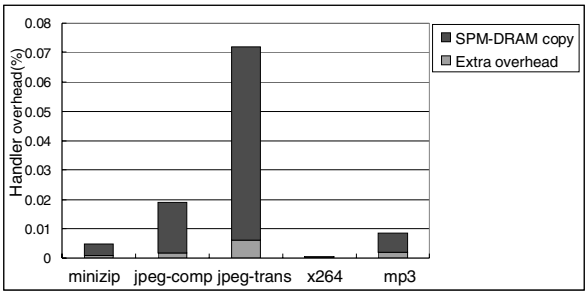

Figure 9. Runtime-overhead caused by the fault handler

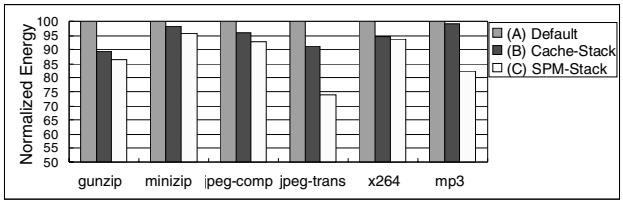

Figure 10. Normalized energy consumption

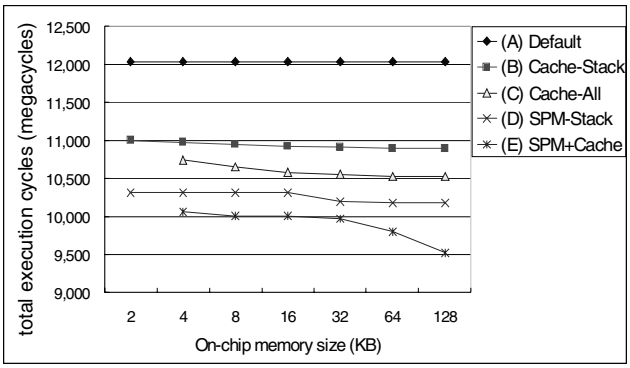

Figure 11. MP3 : Total execution cycles

For mp3 application, we investigated the effect of on-chip memory size for various configurations and plotted the total execution cycles in Figure 11. (B) is the cache solution, and (D) is the proposed SPM solution for stack data. When we use 64KB SPM, all stack variables are allocated into the SPM so that no performance improvement is achieved with larger size of memory. The figure shows that the performance variation is not significant over the entire

range. It is because the working set is usually not larger than 2KB. Additionally, we tested two more configurations. In (C), data cache is used not only for the stack data but also for the global data. The speedup is not significant, compared with (B). In (E), we use a half size- data cache and a half size-SPM. In this case, the data cache is used for the global data, and we apply our approach to the SPM for the stack data. (E) outperforms (C) consistently by 7% on average. It confirms the viability of the proposed technique: the SPM management for the stack data is superior to cache solutions.

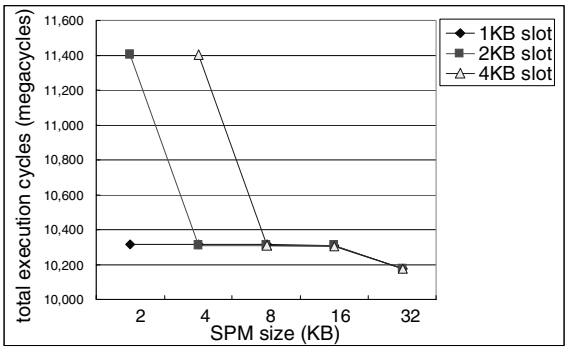

Figure 12. Total execution cycles with three different slot sizes for MP3 application

The final experiment is performed to examine the effect of slot size on the performance. As shown in Figure 12, the worst case occurs when the slot size is the same as the SPM size. It results in performance degradation of 10% in experiments. For the other cases, the slot size has different effect depending on the applications. In general, performance tends to get better as the slot size getting smaller, because the size of the working set is usually smaller than the page size. However, the performance variation is not significant in almost all cases, around 0.3% in MP3.

6. Conclusions

In this paper, we propose a novel technique to use scratch-pad memory for stack management. Unlike the previous works, we change the address map of the SPM region by managing the page table in the target architecture with MMU. The experimental results show that the proposed technique exploits the stack locality better than the cache solution and so gives better performance. It does not require any additional hardware and compiler technique so that it can be cheaply implemented. One drawback of the proposed technique is lack of predictability.

The proposed technique is complementary to the previous works. And, it can be applied separately with the global data management. As an example, we tested a configuration where an SPM is used for the stack data and a normal cache is used for the global data, and obtained the better results than the other configurations. How to obtain the optimal configuration remains as a future work.

Acknowledgement

This work was supported by IT leading R&D Support Project funded by Korean MIC, IT-SoC project, and BK21 project. The ICT and ISRC at Seoul National University and IDEC provided research facilities for this study.

References

[1] ARM. *Advanced RISC Machines Ltd.* http://www.arm.com.
[2] W Wolf, M Kandemir., Memory system optimization of embedded software, *In Proc. of the IEEE*, 2003.
[3] Oren Avissar , Rajeev Barua , Dave Stewart, An optimal memory allocation scheme for scratch-pad-based embedded systems, *ACM Transactions on Embedded Computing Systems (TECS)*, v.1 n.1, p.6-26, November 2002.
[4] S. Steinke , L. Wehmeyer , B. Lee , P. Marwedel, Assigning Program and Data Objects to Scratchpad for Energy Reduction, In *Proc. of the conference on Design, automation and test in Europe*, p.409, March 04-08, 2002.
[5] Manish Verma , Stefan Steinke , Peter Marwedel, Data partitioning for maximal scratchpad usage, *In Proc. of the 2003 conference on Asia South Pacific design automation*, Kitakyushu, Japan, January 21-24, 2003.
[6] Sumesh Udayakumaran, Angel Dominguez, Rajeev Barua, Dynamic allocation for scratch-pad memory using compile-time decisions, *ACM Transactions on Embedded Computing Systems (TECS)*, v.5 , n.2, p.472-511, May 2006.
[7] Federico Angiolini , Luca Benini , Alberto Caprara, Polynomial-time algorithm for on-chip scratchpad memory partitioning, In *Proc. of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, San Jose, California, USA, October 30-November 01, 2003.
[8] Manish Verma , Lars Wehmeyer , Peter Marwedel, Dynamic Overlay of Scratchpad Memory for Energy Minimization, In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*, p.104-109, September 08-10, 2004.
[9] M. Kandemir , J. Ramanujam , J. Irwin , N. Vijaykrishnan , I. Kadayif , A. Parikh, Dynamic management of scratch-pad memory space, In *Proc. of the 38th conference on Design automation*, p.690-695, Las Vegas, Nevada, United States June 2001.
[10] Ozcan Ozturk , Mahmut Kandemir , Mary Jane Irwin, Increasing on-chip memory space utilization for embedded chip multiprocessors through data compression, In *Proc. of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Jersey City, NJ, USA, September 19-21, 2005.
[11] Poletti Francesco , Paul Marchal , David Atienza , Luca Benini , Francky Catthoor , Jose M. Mendias, An integrated hardware/software approach for run-time scratchpad management, In *Proc. of the 41st annual conference on Design automation*, San Diego, CA, USA, June 07-11, 2004.
[12] S.J.E. Wilton and N.P.Jouppi., CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677-688, May 1996.
[13] Richard Ernest Bellman, Dynamic Programming, Dover Publications, Incorporated, 2003.
[14] Martello, S.; Toth, P., "Knapsack Problems", John Wiley & Sons, Chichester, 1990.
[15] ARM. *Advanced RISC Machines Ltd.* ARM926EJ-S Technical Reference Manual, http://www.arm.com.
[16] X264, http://developers.videolan.org/x264.html.
[17] gzip, http://www.gzip.org.
[18] Mediabench, http://cares.icsl.ucla.edu/MediaBench