IEICE TRANS. INF. & SYST., VOL.E94–D, NO.2 FEBRUARY 2011

PAPER

Core Working Set Based Scratchpad Memory Management

Ning DENG[†], ***Student Member***, **Weixing JI**[†a)], **Jiaxin LI**[†], **Qi ZUO**[†], ***and*** **Feng SHI**[†], ***Nonmembers***

SUMMARY Many state-of-the-art embedded systems adopt scratchpad memory (SPM) as the main on-chip memory due to its advantages in terms of energy consumption and on-chip area. The cache is automatically managed by the hardware, while SPM is generally manipulated by the software. Traditional compiler-based SPM allocation methods commonly use static analysis and profiling knowledge to identify the frequently used data during runtime. The data transfer is determined at the compiling stage. However, these methods are fragile when the access pattern is unpredictable at compile time. Also, as embedded devices diversify, we expect a novel SPM management that can support embedded application portability over platforms. This paper proposes a novel runtime SPM management method based on the *core working set* (*CWS*) theory. A counting-based CWS identification algorithm is adopted to heuristically determine those data blocks in the program's working set with high reference frequency, and then these promising blocks are allocated to SPM. The novelty of this SPM management method lies in its dependence on the program's dynamic access pattern as the main cue to conduct SPM allocation at runtime, thus offloading SPM management from the compiler. Furthermore, the proposed method needs the assistance of MMU to complete address redirection after data transfers. We evaluate the new approach by comparing it with the cache system and a classical profiling-driven method, and the results indicate that the *CWS*-based SPM management method can achieve a considerable energy reduction compared with the two reference systems without notable degradation on performance.

key words: *embedded processor, scratchpad memory management, core working set*

1. Introduction

Memory hierarchy is one of the most ubiquitous notions in computer system design. The main purpose of this concept is to narrow the gap between a high-speed CPU core and the memory by caching the most useful data items in a small, fast memory, with a larger but slower memory serving as a back-up store. The cache is the most popular on-chip memory in general-purpose processors due to its commonality. As for embedded systems, power consumption and the SRAM on-chip area are highly regarded, and the scratchpad memory (SPM) is adopted as a perfect on-chip memory for many embedded microprocessors.

SPM differs from the cache in several aspects: (i) SPM is explicitly manipulated by software, thus lacking the complex tag logic overhead for mapping off-chip data into the on-chip memory; (ii) SPM commonly does not contain a copy of data that is also stored in the DRAM, so there is no coherence problem in a single-level SPM architecture. In addition, SPM accesses can be completed within certain cycles, which makes it a promising choice in the hard real-time environment. Some examples of processors with SPM are Intel IXP network processor, ARMv6, IBM 440 and 405, Motorola's MCORE and 6812, and TI TMS-370. With the emergence of embedded DRAM (eDRAM) [1], the integration of larger on-chip memories with less cost and lower latency becomes possible in next-generation processors. Recent trends indicate that the dominance of SPM in embedded systems is likely to consolidate further in the future.

Traditional SPM allocation methods can be roughly classified into two classes according to whether the SPM is managed like a cache or is managed by the compiler. The first is a software-caching technique, which emulates the behavior of a hardware cache by the software. The most representative example of such methods is the local memory in the CELL BE processor [2]. However, there is no highly successful scheme to eliminate the high address translation overhead at runtime, because a single memory reference instruction is replaced by a couple of instructions for software-emulated cache lookups. The inevitable overhead greatly diminishes the merits of cache-like automatic management. The second sort of SPM allocation scheme is compiler-directed SPM management, and it is more widely adopted in embedded processors because its codes are more stable than desktop applications. Compiler-based approaches commonly utilize static analysis or profiling information as the main cue to predict memory access pattern at runtime. These methods can be used in many embedded systems whose program is tied at manufacturing and remain constant.

However, with the development of the Internet and mobile technology, more and more embedded applications are tied with the hardware platform not only by their manufacturer but by users in many cases. For example, we can download various applications for our cell phones. The programs, however, are commonly distributed in the form of binary executables and can not be tailored to fit the local SPM. Therefore, the well-known SPM's advantages are abandoned. Furthermore, for many multimedia applications and real-time applications, memory access patterns are highly affected by outside input [3]. Traditional compiler-based SPM allocation schemes may lack accurate memory reference knowledge at the compiling stage, thus decreasing SPM utilization. With the diversity of an embedded application's deployment, we consider that a compiler-independent SPM allocation method is a meaningful compensation to tra-

Manuscript received September 14, 2010.

[†]The authors are with the School of Computer, Beijing Institute of Technology, China.

a) E-mail: pass@bit.edu.cn

DOI: 10.1587/transinf.E94.D.274

ditional compiler-based approaches.

In our view, an ideal runtime SPM management scheme is expected to adjust the SPM contents based on the dynamic access pattern of the application itself. A natural proposal is that the most frequently used data items should first be considered as the candidates for SPM allocation. Denning was the first to formalized the notion of a working set [4] to depict such data items that are accessed within a certain number of instructions. The *core working set* (*CWS*) [5] extends Denning's working set concept and illustrates that a dramatic disparity exists between the usage patterns of frequently used data and those of lightly used data in the working set. It is similar to a scenario in SPM management, in which the most popular data items are expected to be maintained in SPM for future references. Therefore, we are motivated to associate SPM allocation with the *CWS* theory. Moreover, a counting-based CWS identification algorithm is adopted to determine heuristically those data blocks in the program's working set with high reference frequency, and then these promising blocks are considered as the good candidates for SPM allocation. The novelty of this SPM management method lies in its dependence on the program's dynamic access pattern as the main cue to conduct SPM allocation at runtime, thus offloading SPM management from the compiler.

The main contributions of this study include: (i) proving the existence of the *CWS* theory in embedded applications by analyzing the traces of some typical embedded applications; (ii) development of a novel runtime SPM management scheme without compiler support based on the *CWS* theory; and (iii) a comprehensive experimental evaluation to prove the rationality of the proposed method by comparing the execution time and energy consumption with a cache reference system and a classical method.

The rest of this paper is organized as follows. Section 2 introduces the main idea of *CWS* and how it can impact SPM runtime allocation. Section 3 describes the *CWS*-based runtime SPM management in detail. Section 4 describes the evaluation methodology. In Sect. 5, we prove the rationality of the proposed method through experimental results. Section 6 reviews previous research on SPM management, and Sect. 7 presents the summary.

2. Core Working Set Phenomenon

The notion of the working set was proposed by [4] to describe the set of distinct addresses referenced within a certain window of time. This definition puts all memory blocks in a working set on an equal footing. However, in real computer workloads, memory accesses are not evenly distributed in the working set space. In other words, a dramatic difference exists between the usage patterns of frequently used data and those of lightly used data. Based on this phenomenon, [6] proposed the concept of the *core working set* (*CWS*) to depict the more important core elements in the working set, which are expected to give preferential treatment when doing caching. The *CWS* theory states that at any given time, only a small fraction of all addresses is used, and this used part changes relatively slowly [7]. The *CWS* theory is an extension of the classical working set concept in a real workload. The notion of a core leads to the realization that not all data items in a working set are equally important. This core partitions the working set into two subsets: those data items that are very popular and those that are only accessed intermittently, which is often the case in practice.

To make an intuitive understanding of the embedded application's memory access pattern, we extract and plot the memory address distribution of 4 typical embedded applications from MiBench [8]. The traces are collected for a pre-defined duration of 100,000 instructions by using a simulator [9]. We plot the first 18700 memory accesses of the 4 benchmarks in a highly referenced memory region, ranging from 0X00000000(0) to 0X0000249F0(150000). In Fig. 1, the memory traces of *basicmath* and *dijkstra* are plotted, while the traces of *stringsearch* and *matrix* are shown in Fig. 2.

For applications like *dijkstra* and *matrix*, their memory references are observed to be linearly distributed in the whole address space, while those of *basicmath* and *stringsearch* are more concentrated in several core regions.

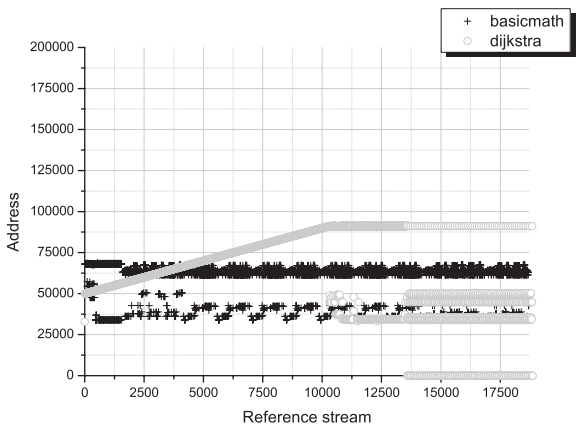

Fig. 1 Memory traces of *basicmath* and *dijkstra* from 0X00000000 to 0X000030D40.

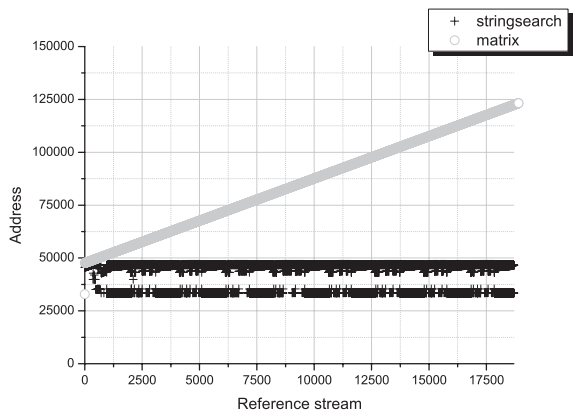

Fig. 2 Memory traces of *stringsearch* and *matrix* from 0X00000000 to 0X0000249F0.

Despite the sizes of the memory regions, they are accessed very frequently, and in fact they service most of the memory references. These provide the opportunity of predicting the frequently accessed memory regions at runtime.

There are two kinds of locality manifested by the traces: temporal locality, which means that a referenced address will probably be referenced again in the future, and spatial locality, which indicates that once an address is referenced, the addresses nearby will have a greater chance to be referenced. By analyzing the traces, we draw the conclusion that the *CWS* phenomenon indeed exists in some typical embedded benchmarks with regular memory access pattern, and the use of *CWS* knowledge to manage scratchpad memory at runtime is possible.

3. *CWS* Prediction in SPM Management

In order to identify the most frequently used data at runtime, an efficient *CWS* prediction algorithm should be adopted. First, we introduce a counting-based *CWS* prediction algorithm. Then we study the determination of predicate *nB* as the basis of the following discussion. Lastly, the implementation of our proposed method is depicted.

3.1 The Counting-Based *CWS* Prediction

An ideal *CWS* identification method should meet several conditions: first, it can obtain a *CWS*, which only occupies small portions of the whole memory but can capture the majority of references; second, a *CWS* block can be identified before too many accesses, which maximizes the access profit for future SPM references. Particularly, an efficient *CWS* prediction method with little overhead is of great significance to a runtime implementation.

According to the definition in [6], *CWS* is a set of those blocks that appear in the working set and are referenced for a multitude of times. A predicate is employed to reflect if a block is a *CWS* member or not. Indeed, evaluating the best predicate among so many alternatives is challenging. One of the most natural method of defining such predicate is based on counting the number of references to a given block. Let B represent a block of k words. Let W_i, $i = 1, \ldots, k$ be the words in block B. Let $r(w)$ be the number of references to word w within a period of time. This way, we can define a predicate that is evaluated to be true if the block is referenced n times or more:

$$nB \equiv \sum_{i=1}^{k} r(w_i) > n \tag{1}$$

For example, the predicate $3\,B$ identifies those blocks that were referenced 3 times or more. The selection of the *CWS* predicate is noted to be of great flexibility, and the *CWS* identified by a predicate is a relative concept that roughly defines only a subset of the working set.

3.2 Determination of n

In a rich set of given predicates *nB*, the selection of a suitable one for embedded applications, namely, the determination of n, is important. We evaluate a selected *nB* using the following standards: (i) memory addresses in *CWS* can satisfy as many references as possible; (ii) SPM management based on this predicate should have a small runtime overhead, which is reflected by the execution time collected by the simulator in our evaluation. Accordingly, we define the following metric

$$E(n) = \frac{T_{execution}(n)}{Ref_{CWS}(n)} \tag{2}$$

to evaluate the selection of *nB* among a number of optional predicates. On the right side of Eq. (2), $T_{execution}(n)$ represents the execution time of a program when the predicate is *nB*; $Ref_{CWS}(n)$ refers to the total memory references of the defined *CWS*. Generally, the execution time of the same program varies when using a different predicate *nB*. For a certain memory block, a smaller *nB* may have a greater chance identifying this block as a *CWS* member; thus, the size of *CWS* can become larger when using a smaller *nB*. This means there are more SPM allocations and SPM accesses during runtime, which possibly incur a different execution time due to the varied SPM allocation blocks and access numbers. Therefore, we evaluate *nB* by $E(n)$. A better performance is achieved when the $E(n)$ value is smaller, which represents a smaller execution time and a suitable *CWS* with more SPM accesses. All variables are counted using the predicate *nB*, with the assumption that a *CWS* block is referenced n times or more.

We select the *basicmath, dijkstra, matrix*, and *stringsearch* benchmarks from the benchmark suit and run them on a simulator with a varied n ($n = 2, 4, 8$ *and* 16). The detailed experiment setup is depicted in Sect. 4.1 with an SPM-only on-chip memory configuration. The simulator is employed to calculate the $T_{execution}(n)$, while the memory traces are analyzed by a simple trace analyzer to collect the statistics of $Ref_{CWS}(n)$. The counted $E(n)$ in different *nB* are plotted in Fig. 3. We observe that $n = 8$ and 16 gain a better *CWS* than $n = 2, 4$ except for *matrix*. Combined with the previous memory reference distribution in Fig. 1 and Fig. 2, we therefore consider that a smaller n in predicate *nB* is more suitable for benchmarks with a lower locality degree, while a greater *nB* achieves better performance in applications with higher locality. On the average, the *CWS* predicate with $n = 16$ can be more efficient than other *nB* choices, even though the advantage is very limited.

This standard can evaluate the selected *nB*. However, the determination of n is closely related to the system architecture and the program's access patterns. Our selection of $16\,B$ is notably achieved on the ARM926EJ-S platform and with consideration of the selected benchmarks. An even more ideal implementation of *nB* selection can be an adaptive *nB* selection, which adjusts the predicate by the dy-

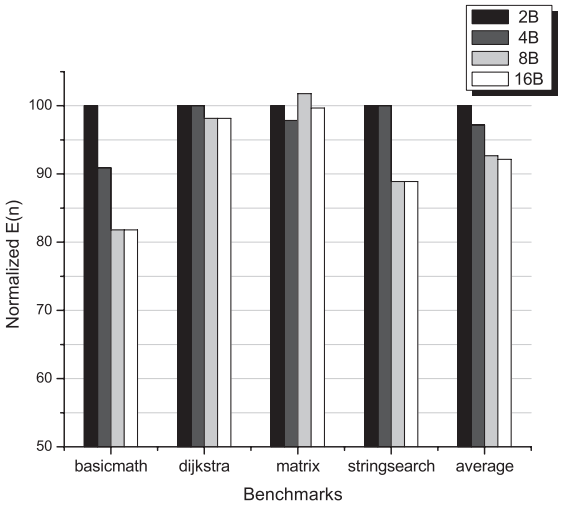

Fig. 3 Normalized E(n) for selected benchmarks with predicate 2 *B*, 4 *B*, 8 *B* and 16 *B*.

namic program behavior at runtime. Even though a dynamic *nB* selection is more flexible for different access patterns, we utilize a static *nB* in this paper for simplicity. Unless stated otherwise, we select $nB = 16\,B$ as the *CWS* predicate in the following experiment. From our evaluation, on the average, the *CWS* identified by 16 *B* captures over 90% memory references with only fewer than 1% memory addresses.

3.3 *CWS*-Based Scratchpad Memory Management Implementation

The *CWS*-based SPM management strategy adopts a software and hardware co-design in order to achieve an efficient runtime management.

3.3.1 Hardware Structure

As illustrated in Fig. 4, the data-side memory hierarchy discussed in this paper is composed of an SPM, a small cache, a TLB, and an adder. Once the CPU core issues a memory access virtual address (VA), the address is first translated into a physical address (PA) by the TLB, which is a fully-associated cache containing 8 page table entries for quick matching. The translated PA is then compared with a predetermined address in an address comparator. This determines if the memory reference is hit on the SPM. If so, the requested data item is directly returned to the requester; otherwise, a cache reference is invoked to the bypassed cache memory. If the memory reference is unfortunately not hit on either SPM or cache on-chip memory, the on-chip memory miss is then delivered to the next-level memory hierarchy.

For the convenience of managing the SPM as a whole, we evenly divided the SPM into blocks of equal sizes. The core idea of the proposed method is reference counting; a natural way is to associate a counter with each memory block. The block is identified as a *CWS* member once its counter reaches the pre-defined threshold. However, this

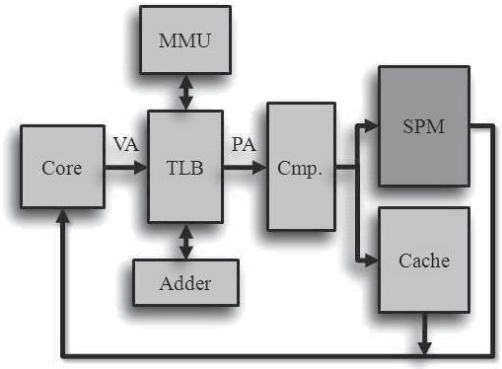

Fig. 4 On-chip memory architecture with SPM and cache.

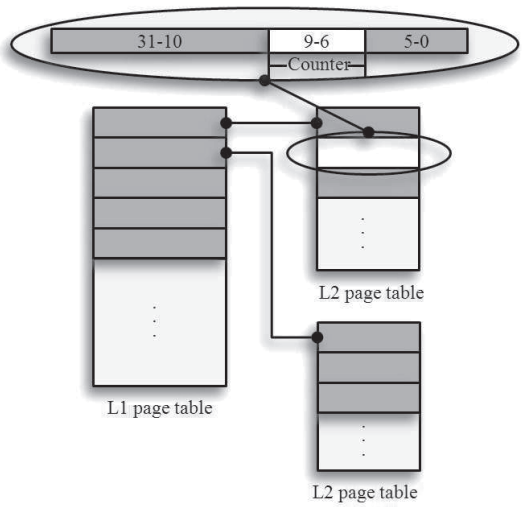

Fig. 5 Page table entries on ARM9 processors.

method incurs a huge hardware overhead to record the reference information, and the counting procedure self is time consuming. To address the problem, we use 4 unused page table entry bits in our implementation on a 32-bit ARM9EJ-S processor as a counter to record the access number of a block. As shown in Fig. 5, the ARM architecture adopts a two-level page table technique. We utilize the unused 6–9 bits in the second-level page table entry for block reference counting. A hardware adder is responsible for updating the reference number of each block in the TLB. Differing from the ordinary adder, the simple logic only updates the reference counter in the page table entry by adding one. The counter of each block is incremented only if the associated block is found to be accessed. Many types of architecture maintain some unused bits in their page table implementations [37], and these bits can be utilized to record the reference information for each block. Thus, the extra hardware overhead is effectively controlled by exploiting the potential of the existing hardware. In an extreme case wherein there is no sparse bit for counting, a separate counter should be maintained for every page table entry. In this case, the page table entry and its counter should be scanned at the

same time, which incurs slightly more overhead. With hypothetical hardware modification to allow this, we could effectively control the hardware overhead by adding only an adder and an address comparator register, which is easy to implement by hardware. Unless stated otherwise, a "block" refers to a tiny page sized $1\,KB$ in our implementation according to the ARM926EJ-S manual [10]. Even though a 4-bit counter incurs counter overflow frequently, it can still reflect the fundamentals of our method. The experimental results prove that a $16\,B$ predicate can achieve a considerable performance according to the evaluation.

Apparently, the counting-based method results in great overhead by modifying the page table entries frequently. This problem is tackled by counting only the DRAM blocks. Namely, when a block is transferred to SPM, its counter is stopped. For example, in the *basicmath* benchmark whose *CWS* is composed of 16 data blocks (see Table 6) with a pre-defined predicate $16\,B$, we count only $16 \times 16 = 256$ accesses in total, which is far fewer than the total memory accesses of the application. If an SPM block is evicted after the SPM replacement, its counter is reset. This design reduces the pressure of updating page table entries too frequently. Moreover, the execution time evaluation in the next section well supports the reasoning that *CWS* can be predicted within small references, thus avoiding an overwhelming amount of overhead by counting every reference.

3.3.2 Software Management

Listing 1 Pseudo code of CWS based SPM allocation algorithm.

```
1 CWS_SPM_Alloc() {
2     if(block[i].reference < CWS_THRESHOLD)
3       block[i].reference++;
4     //Reference time reaches the threshold
5     else{
6       if(SPM free block exists)
7           Move_to_SPM(i);
8       else
9       //No free SPM blocks
10          SPM_Replacement(i);
11
12      //Address redirection
13      Pagetable_Update(i);
14    }
15 }
```

In software management, a bitmap structure is used to keep information such as the available positions in SPM. Once a bit is set, the associated SPM block is used; otherwise, the block is available for a future allocation. Listing 1 depicts this software management algorithm by pseudo code. There are two situations in the data movement procedure: if there is available space in SPM, the selected block is copied directly into the proper location through *Load*/*Store* instructions; otherwise, a cache-like replacement algorithm is adopted to select a block from SPM for eviction. The steps followed are similar to those of the former situation. Once a memory access is hit in SPM, the requested data item is directly returned to the CPU core. If the requested data is not located in the SPM range, namely, the SPM miss, the CPU core looks up the data cache sequentially. Differing from a cache miss, the SPM replacement does not immediately happen when an SPM miss occur. It is only invoked when the memory reference counter reaches the pre-defined threshold and there is no available space in the SPM.

For simplicity, we implement the data transfer between off-chip DRAM and on-chip SPM through *Load*/*Store* instructions using the software. The function Move_to_SPM(i) transfers a data block from the off-chip DRAM to the on-chip SPM using memory access operations. The overhead involves of the cost of reading the data items from DRAM and that of writing them into SPM. SPM_Replacement(i) first write a data block back into its original address in the DRAM, and then the available space is allocated to a new block. In this procedure, the overhead of writing a block back is considered. We simply adopt a random SPM block replacement policy in our evaluation. Some optimized cache-like eviction policy, such as the LRU and MRU, may still be effective for the SPM replacement. However, for the selected benchmarks, the replacement policies make little sense for the evaluation result since that most of the applications do not evict a block from SPM during execution because the SPM is larger than the size of the *CWS*. Even though a block is evicted, it has little chance to be reused by observing the traces of the applications. Pagetable_Update(i) operation is implemented by first looking up the TLB. If the expected entry is not found in the TLB, the MMU and even the page table in DRAM is inquired. The VA is then redirected to the newly allocated SPM space by updating its mapped PA. In the ARM926 processor, updating the reference counter involves two page table walks in such a two-level page table architecture. A detailed overhead calculation is shown in Sect. 5.4.

In our experiment, once the reference counter reaches the threshold, a software exception will interrupt the execution of the running program to make an SPM allocation. However, the data transfer can be accelerated by hardware in real implementation through DMA, which can handle the data movements between off-chip and on-chip memories without interrupting the CPU core. When the data transfer is complete, an address redirection is needed to ensure that the follow-up memory references can still reach the transferred data blocks in SPM. In previous compiler-based methods, the compiler is responsible for modifying the reference addresses once the data item is allocated to SPM. In the runtime method, we turn to the virtual memory technique to reconstruct the virtual to physical address mappings after the SPM allocation is complete with the assistance of MMU. The runtime method is more flexible compared with compiler-based methods at the costs of some extra hardware.

4. Experiment Methodology

In this section, we present the evaluation of the proposed SPM runtime management method. We first describe the experimental setup which includes the selections of the sim-

ulator and benchmarks. Then, we explain the measurement metrics for energy and show the area and size configurations of the memory components.

4.1 Experimental Setup

The experimental setup includes two main parts: the simulator hacking and the selection of standard benchmarks.

4.1.1 Simulator

We use FaCSim [9], an ARM926EJ-S [10] processor simulator, to model the SPM on-chip memory setup. FaCSim supports a cycle-accurate simulation based on its functional frontend and accurate backend. Its memory subsystem is composed of instruction/data SPM (in the form of a tightly-coupled memory), instruction/data cache, unified TLB, MMU, write buffer, prefetch buffer, bus, and main memory. In our experiment, we optimize SPM management on data side alone. However, instruction-side optimization is theoretically similar because the proposed method does not distinguish between the instruction and data at runtime. The details of the experimental setup are summarized in Table 1.

Table 1 Parameters of the evaluation.

Core	*CPU type:* ARM926EJ-S *Frequency:* 200 MHz
Bus	*Bus type:* AHB *Core frequency*/*Bus frequency:* 3
Instruction cache	*Size:* 32 KB *Associativity:* 4 *Line size:* 32 B *Latency:* 1 cycle
Data cache	*Size:* 4 KB *Associativity:* 4 *Line size:* 32 B *Latency:* 1 cycle *Write back latency:* 1 cycle
Data side SPM	*Size:* 16 KB *Page size:* 1 KB *Latency:* 1 cycle *Replacement policy:* random
Memory	*Memory size:* 128 MB *Non-sequential read hit:* 8 cycle *Non-sequential read miss:* 11 cycle *Non-sequential write hit:* 3 cycle *Sequential read hit:* 1 cycle *Sequential write miss:* 4 cycle *Sequential write hit:* 1 cycle

4.1.2 Benchmarks

We select MiBench [8] and OOPACK [11] as the benchmark suits in our experiment. MiBench is a representative benchmark suit for embedded applications, and OOPACK is often adopted to simulate typical embedded programs with object-oriented features. For some constraints of the simulator, we avoid some multimedia benchmarks with a multitude of input data. However, we select applications of different locality degrees to verify the rationality of the proposed method on various memory access patterns. Table 2 shows the characteristics of the benchmarks.

4.2 Evaluation Metrics

For a fair comparison between SPM and the reference cache system, we should first determine the capacity of SPM and the cache in the experiment. Then an energy model is adopted to make an energy comparison.

4.2.1 Area and On-Chip Memory Capacity

We adopt a hybrid on-chip memory architecture in our experiment. According to Table 3, by following the principle that the total on-chip area of the hybrid design should not exceed the size of the cache-only architecture, we determine the SPM and cache capacity in the hybrid on-chip memory configuration for a fair comparison. Table 3 shows the on-chip area parameters of the cache and SPM calculated by the CACTI tool [12], from which we figure out that a hybrid on-chip memory configuration of 32 KB SPM plus 8 KB cache is approximately 78.7% of a 32 KB cache-only reference system. Unless stated otherwise, we use the 32 KB SPM plus 8 KB cache as the hybrid configurations in our following evaluation.

4.2.2 Energy Consumption Calculation

The energy consumption parameters of SPM and the cache are listed in Table 4, which are calculated by CACTI5.3. With tag memory and tag comparison undone, the CACTI tool [12] is used to estimate the energy values for the scratchpad. We reference the DRAM energy parameters from previous study [13]. The DRAM energy consumption is composed of a dynamic part, which is related to access types, and a static part, which remains stable in the whole

Table 2 Benchmarks description.

Benchmark	Category	Description
basicmath	Auto./Industrial	Simple mathematical calculations.
bitcount	Auto./Industrial	Bitcount algorithm tests the bit manipulation.
blowfish	Security	A symmetric block cipher with a variable length key.
fft	OOPACK	A fast fourier transform and its inverse transform.
dijkstra	Network	A well known algorithm for shortest path routing.
stringsearch	Office	A comparison algorithm for given words in phrases using.
MD5	OOPACK	A widely used cryptographic hash function with a 128-bit hash value.
matrix	OOPACK	Multiplies two matrices containing real numbers.

Table 3 Chip area parameters of cache and SPM (90 nm technology).

Size [KB]	Cache [mm^2]	SPM [mm^2]
4	0.4938	0.0779
8	0.7219	0.1456
16	1.3470	0.3006
32	1.6420	0.5706

Table 4 Parameters of energy consumption.

On-chip Memory				
SPM		Cache		
Size	Energy [nJ]	Size [KB]	Assoc.	Energy [nJ]
4 KB	0.009	4	4	0.087
8 KB	0.016	8	4	0.091
16 KB	0.018	16	4	0.126
32 KB	0.035	32	4	0.138
off-chip Memory				
Reference		Dynamic [nJ]		Static [mW]
Random read		11.75		7.6
Burst Read		26.98		
Random Write		10.40		
Burst Write		13.27		

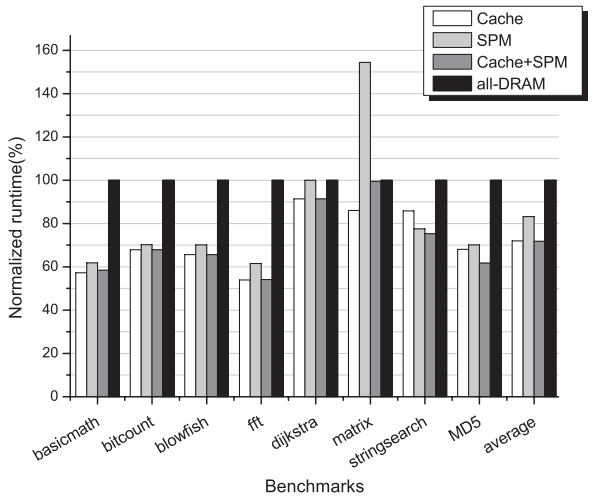

Fig. 6 The execution time comparison among 4 different on-chip memory configurations.

execution. We trace the memory references by the simulator and calculate the energy consumption of all memory levels by the energy model shown below. In our calculations, we assume that a cache line size is 32 bytes, and the associativity is 4 with a 90 nm technology. We reference the energy model in [14] to calculate the energy consumption. The components of the memory subsystem include the on-chip cache, SPM, and the main memory. The energy consumption of the memory subsystem is computed by

$$E_{total} = E_{SPM} + E_{cache} + E_{DRAM} \quad (3)$$

$$E_{SPM} = e_{SPM} * (read_{SPM} + write_{SPM}) \quad (4)$$

$$E_{cache} = e_{cache} * (hit + miss * linesize) \quad (5)$$

$$E_{DRAM} = e_{DRAMread} * read_{DRAM} + e_{DRAMwrite} * write_{DRAM} + T_{total} * P_{standby} \quad (6)$$

where e_{cache}, e_{SPM} $e_{DRAMread}$, and $e_{DRAMwrite}$ denote the access energy of the cache, SPM, and DRAM calculated by CACTI (in Table 4) for the memory types, respectively. $read_{SPM}$, $write_{SPM}$, $read_{DRAM}$, $write_{DRAM}$, hit, and $miss$ are the memory reference statistics, and T_{total} is the number of execution cycles collected by the simulator.

5. Results

In this section, we first compare the performance of the proposed method with a cache reference system. A comparison between a classical method and the one in this paper is then given. Our evaluations are concerned with the execution time and energy consumption to verify if the proposed SPM allocation method can indeed utilize the advantages of SPM in contrast with traditional cache systems. A complementary analysis on the SPM size impact and the execution time overhead of SPM follow in an attempt to gain a deeper understanding of the proposed method.

5.1 Comparison with Cache

In Fig. 6, we compare the execution time for the selected benchmarks under cache-only, cache+SPM, SPM, and DRAM memory configurations. Bars of different colors represent the 4 on-chip memory configurations. Particularly, the hybrid memory of SPM plus the cache is identified as the gray bar. The SPM+cache and cache-only configurations show the best performance among nearly all benchmarks, while the DRAM configuration performs the worst, against which the others are normalized. On the average, the SPM+cache and cache-only configurations reduce the execution time by 27.9% and 28.2% compared with DRAM, respectively. Particularly, the SPM-only design incurs a worse execution time than the DRAM setup in the *matrix* benchmark, which can be explained by the program locality features. Reviewing Fig. 2, we can observe that the memory references of *matrix* are more evenly distributed among several benchmarks, resulting in great challenges for a *CWS*-based runtime method in predicting the access pattern. This paper believes that a hybrid memory design along with cache plus SPM can enhance flexibility for applications with low data locality.

Figure 7 shows the normalized energy consumption between the SPM+cache and cache-only designs. The black bar represents the energy consumption of the cache, against which the SPM+cache configuration is normalized. We can observe that the energy consumption of the SPM+cache is much lower in nearly all benchmarks except for *matrix*. On the average, the hybrid on-chip memory managed by the proposed method can reduce the energy consumption by approximately 32.5% in contrast with the reference cache system. The reason for this is that the proposed method migrates a considerable amount of on-chip memory references from the cache to SPM, thus fully utilizing the inherent energy advantage of SPM.

Table 5 Comparison of the execution time between Egger's method and the proposed method.

Benchmarks	$movedblocks_{Egger}$	$movedblocks_{Proposed}$	$overhead_{Proposed}$ (%)
basicmath	7	16	102.0
bitcount	4	4	99.9
blowfish	1	3	98.9
fft	0	5	97.3
dijkstra	5	22	99.8
matrix	4	2432	100.3
stringsearch	1	6	87.6
MD5	1	4	90.6
average			97.1

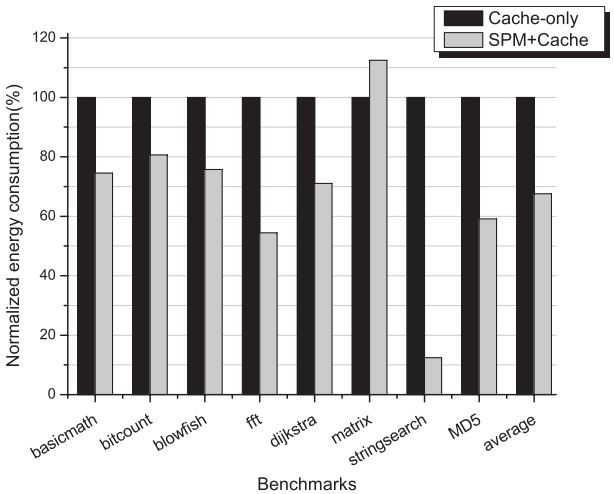

Fig. 7 The energy consumption comparison between cache and SPM+cache configurations.

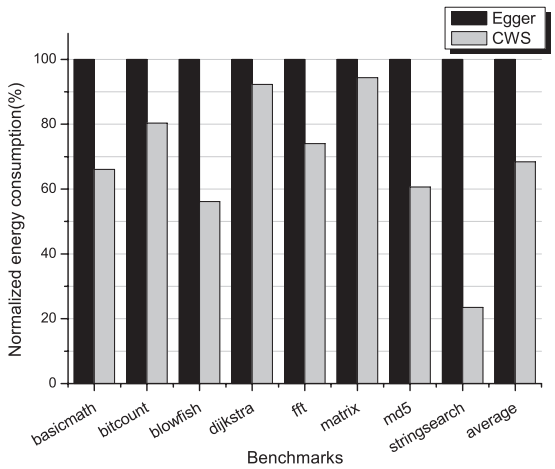

Fig. 8 The energy consumption comparison between the proposed method and Egger's method.

5.2 Comparison with Egger's Method

[21] proposed a profiling-driven dynamic SPM management method by Egger et al.. Egger's method divides the binary into pageable, cacheable, and uncacheable regions based on profiling information and a heuristic energy model. Only the pageable region is capable of being allocated into SPM. Their method and the one in this paper both use MMU to assist the address redirection after SPM allocation. However, the proposed method differs from Egger's method not by utilizing the profiling information, but by monitoring the runtime access pattern. For a fair comparison, we adopt a hybrid on-chip memory configuration of 16 KB data SPM plus 4 KB data mini-cache [21] in this experiment. The SPM is managed by the proposed method and Egger's method, respectively.

Table 5 shows an execution time comparison of the two methods. The second and third columns list the numbers of SPM allocation blocks by Egger's method and the proposed method, respectively, whereas the last column shows the normalized execution time of the proposed method with respect to Egger's method. It depicts that Egger's method outperforms the proposed method only in *basicmath* and *matrix*; however, the superiority is very limited. The result indicates that the proposed method based on runtime decision does not incur unacceptable huge runtime overhead compared with the profiling-driven method.

Figure 8 depicts the energy consumption comparison of the proposed method and Egger's method. The energy consumptions of the proposed method are normalized with respect to Egger's method. The result shows that the proposed method gains a lower energy consumption in all the benchmarks. On average, the CWS method outperforms the reference system by 31.6%. There are two possible causes of the disparity. First, the method in [21] adopts some empirical factors, such as the cache miss ratio and the average page miss number. The accuracy of these parameters is closely related to the application's characteristics and the profiling numbers. Second, Egger's method is more applicable to SPM management in instruction side. Some optimizations regarding the code region are ignored in our experiment, which may degrade the performance to some extent. However, the implementation of Egger's method still reflects the basic idea of a profiling-driven method.

Notably, compared with the previous methods, the proposed scheme operates the SPM allocation without profiling information and the compiler support, achieving more flexibility and less constraints. The new method is especially meaningful in some circumstances where the profiling-driven methods may lose their efficiency.

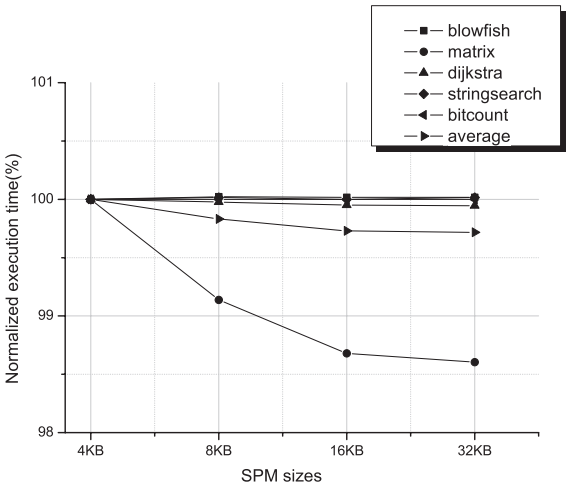

Fig. 9 Execution time in different SPM sizes.

5.3 SPM Size Impact

In this section, we examine the effect of varied SPM sizes on the execution time by configuring the SPM sizes from 4 KB to 8 KB, 16 KB, and 32 KB. Figure 9 shows the execution time changes according to different SPM sizes. We select *blowfish, matrix, dijkstra, stringsearch*, and *bitcount* as the benchmarks. Clearly, the increasing SPM size reduces the execution time by 0.16%, 0.27%, and 0.28% on average. However, the execution time reduction degree degrades when the SPM size is varied from 4 KB to 32 KB. The execution time of *blowfish* and *bitcount* in a larger SPM configuration is even higher than that in the smaller SPM cases. The reason for this is that only a small subset of program data is frequently used. Therefore, a relatively stable performance improvement is achieved when the SPM size increases continuously. This result is very similar that of the cache.

5.4 Runtime Overhead

The runtime overhead is composed of the latencies of data transfer, reference counting, and page table remapping. We use the following model

$$T_{overhead} = T_{transfer} + T_{referencecounting} + T_{pagetableremapping} \quad (7)$$

to calculate the overhead resulting from SPM management, in which $T_{transfer}$, $T_{referencecounting}$, and $T_{pagetableremapping}$ represent the three types of latencies mentioned above. Specifically, when the SPM allocation invokes a block replacement, $T_{transfer}$ is composed of both the overhead of block eviction and block filling. For example, from Table 6, 8 *CWS* blocks are identified in *stringsearch*. We compute the SPM management overhead by $T_{overhead} = (32 + 24) * 256 * 8 + 16 * (32 + 24) * 8 * 2 + 56 * 8 * 2 = 129920$ cycles, where the DRAM read and write latencies are 32 and 24, respectively. All of these results are calculated under the assumption that a memory block is identified as a *CWS* member by the predicate 16 *B*.

Table 6 *CWS* based SPM management overhead ratio.

Benchmarks	Total cycles	*CWS* blocks	Overhead
basicmath	832762752	**16**	0.03%
bitcount	119721978	14	0.18%
blowfish	7466869	14	2.85%
fft	531747	23	61.79%
dijkstra	97036984	53	0.81%
matrix	251297567	241	1.00%
stringsearch	**760486**	**8**	**16.58%**
MD5	133365	7	75.07%

We trace the selected benchmarks by simulating and analyzing the trace results through an analyzer program to determine the *CWS* for each application. Table 6 lists the block numbers of the benchmarks, from which we can approximately count the SPM management overhead and its percentage in the total execution time of the hybrid memory configuration. It indicates that the SPM management overhead in 5 benchmarks only occupies a small part (< 10%) of the total cycles. For benchmarks such as *fft* and *MD5*, the overhead percentage for SPM management is over 50%. The reason is that the program size is relatively small, which leads to a short execution time and a high SPM overhead. On the contrary, for benchmarks like *matrix*, the relatively high SPM management overhead is partially neutralized by the great program execution time. In these cases, the SPM overhead can be nearly ignored.

6. Related Work

In this section, we briefly review previous research on scratchpad memory management. In the literatures, SPM management is roughly divided into static methods [15]–[19], in which the SPM contents cannot be changed, and dynamic ones, in which the contents of SPM can be tuned after the compiling stage. Static approaches tend to model SPM allocation as a knapsack problem or use greedy strategies for efficiency. Meanwhile, the dynamic SPM management [14], [20]–[26] enables a change in SPM layout through compiler-inserted instructions or other runtime strategies. Compared with static methods, dynamic methods tune SPM layout at runtime, so they are more suitable to polytropical access patterns. SPM researchers show preference for dynamic methods over their static counterparts. Ramanujam et al. [20] was the first to proposed a dynamic SPM allocation scheme, which is a compiler-directed method to support loop and data transfers. Egger et al. [14], [21] proposed a horizontally partitioned on-chip memory architecture of mini-cache plus SPM, the purpose of the mini-cache is to cover the external memory access overhead by those memory reference that are not hit on SPM. The profiling information is mainly referenced for SPM allocation. This method handles address redirection by MMU, which is very similar to our method. [26] presented a scratchpad memory allocator for heap allocation in SPM, which uses a

variety of techniques to reduce its memory footprint while still remaining effective. The allocation algorithm supports both fixed-sized block allocation and a variable-sized region's allocations within these blocks. Similar to their design, our method also adopts a bitmap structure to record the SPM state at runtime.

SPM management is very similar to Stream Register File (SRF) allocation. In [27], a general-purpose compiler method called memory coloring is introduced, which adapts the array allocation problem to graph coloring for register allocation. The approach operates in three steps: (i) SPM partitioning to pseudo registers, (ii) live-range splitting to insert copy statements in an application code, and (iii) memory coloring to assign split array slices into the pseudo registers in SPM. This approach was further implemented in a real stream processor by Wang et al. [28] to address the problems of: (i) placing streams in SRF, (ii) exploiting stream use, and (iii) maximizing parallelism. However, these methods toward SRF allocation achieve better performance only for regular data accesses.

To avoid allocation overhead at runtime, existing dynamic SPM management is commonly based on profiling information and compiler knowledge, resulting in to statically decided dynamic methods. The most representative runtime method so far is software caching, which emulates the cache's automatic hardware management by software. In this method, a memory reference instruction is replaced by a series of instructions for tag comparison and address mapping, which incurs a huge overhead. Software caching is more widely adopted in certain applications with regular access pattern, such as the CELL BE stream processor, whose local memory is manually tuned by programmers. Previous studies [29]–[31] on software caching were expected to eliminate the problem of runtime overhead, but to dates there remains no still no successful solution for ordinary embedded systems.

Recently, more and more studies [3], [32], [33] have focused on real runtime SPM management because traditional SPM allocation methods have drawbacks for application portability. Further, they may be inadaptable in an environment with unpredictable access pattern at compile time. A compiler-independent SPM management method was introduced by Nguyen et al. [32] for java applications. The method first collects the most frequently accessed objects as the SPM allocation candidates. Next, the SPM size on particular devices is determined by making a call to the OS. Then, a run time decision is made to select the frequently used object for SPM allocation. Another representative runtime adaptive SPM management was proposed by Cho et al. [3] for multimedia applications. This method prepares several optional SPM layout schemes based on profiling and tracks memory reference pattern at runtime. A prepared SPM allocation scheme is selected when the runtime memory access record matches the pre-arranged profiling information. Both runtime methods reference the offline profiling message when doing a runtime SPM allocation. Deng et al. [33] discussed the hot data prediction problem not by profiling but by using a random sampling method completely during runtime. This method can detect *CWS* efficiently; however, the existing randomness cannot promise 100% accuracy in hot-spot prediction. We address this problem by using a counting-based method to predict *CWS* more accurately. Inevitably, most runtime SPM methods cost some hardware to track the memory reference pattern, and our method attempts to reduce such overhead by exploiting the potential of an existing architecture.

The notion of the working set was first formalized by Denning [4] to define those items that are accessed within a certain number of instructions. [36] introduced a software prefetch method for cache. Our method shares the same idea of prefetching hot data items before access. However, there are notable differences between the two methods. [36] is a compiler-based method that uses the static code analysis, whereas the method in this paper manages SPM by analyzing the dynamic access pattern. The compiler-based method inserts some instructions as hints to guide the prefetch operations during runtime, leading to a larger binary after the optimization. The definition of working set puts all items in a working set on equal footing, which is antithetical to many real computer workloads. Feitelson et al. [34] revealed the distinction by statistic analysis, and Etsion et al. [5] proposed a concept named *core working set*. In [35], a random sampling method was utilized for *CWS* prediction in a filter cache design to improve cache insertion efficiency. Moreover, based on *CWS*, [6] used a dual cache structure to give varied treatment to frequently used data and seldomly used items. All these improvements share the same idea that different treatment should be given to data items of different access pattern. This is the inspiration of our paper.

7. Conclusion

This paper associates a runtime SPM management with the *core working set* (*CWS*) by analyzing the traces of some typical embedded applications. In this method, a counting-based *CWS* predicate is used to identify the heavily referenced data blocks by monitoring the memory references at runtime. The novelty of the proposed method lies in its dependence on the program's dynamic access behavior as the main cue to guide the SPM allocation at runtime, thus offloading the SPM management from the compiler. We compare the proposed method with a cache reference system and a classical profiling-driven SPM management method, the results indicate that the method in this paper achieves considerable energy reduction without notable performance degradation. Moreover, the *CWS* based method manages the SPM in a more flexible and general-purpose manner.

References

[1] R.E. Matick and S.E. Schuster, "Logic-based edram: Origins and rationale for use," IBM J. Res. Dev., vol.49, no.1, pp.145–165, 2005.
[2] A.E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P.H. Oden, D.A. Prener, J.C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the cell

processor," PACT '05: Proc. 14th International Conference on Parallel Architectures and Compilation Techniques, pp.161–172, IEEE Computer Society, Washington, DC, USA, 2005.

[3] D. Cho, S. Pasricha, I. Issenin, N.D. Dutt, M. Ahn, and Y. Paek, "Adaptive scratch pad memory management for dynamic behavior of multimedia applications," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.28, no.4, pp.554–567, 2009.

[4] P.J. Denning, "The working set model for program behavior," Commun. ACM, vol.11, no.5, pp.323–333, 1968.

[5] Y. Etsion and D.G. Feitelson, "Probabilistic prediction of temporal locality," IEEE Comput. Archit. Lett., vol.6, no.1, pp.17–20, 2007.

[6] Y. Etsion and D.G. Feitelson, "Core working sets: Concept, identification, and use," The Hebrew University of Jerusalem, Tech. Rep., 2008.

[7] P.J. Denning, "The locality principle," Commun. ACM, vol.48, no.7, pp.19–24, 2005.

[8] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," 2001 IEEE International Workshop on Workload Characterization, WWC-4, pp.3–14, 2001.

[9] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han, "Facsim: A fast and cycle-accurate architecture simulator for embedded systems," LCTES '08: Proc. 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp.89–100, ACM, New York, NY, USA, 2008.

[10] ARM926EJ-S Technical Reference Manual, http://infocenter.arm.com, ARM Limited, April.

[11] E. Chatzigeorgiou and G. Stephanides, "Evaluating performance and power of object-oriented vs. procedural programming in embedded processors," LNCS 2361, pp.367–393, Springer, 2002.

[12] P. Shivakumar and P. Shivakumar, "Cacti 3.0: An integrated cache timing, power and area model," Tech. Rep., 2001, http://quid.hpl.hp.com:9081/cacti/

[13] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management in a multitasking environment," EMSOFT, pp.265–274, 2008.

[14] H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic data scratchpad memory management for a memory subsystem with an mmu," SIGPLAN Not., vol.42, no.7, pp.195–206, 2007.

[15] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," Trans. Embedded Computing Sys., vol.1, no.1, pp.6–26, 2002.

[16] O. Avissar, R. Barua, and D. Stewart, "Heterogeneous memory management for embedded systems," CASES '01: Proc. 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp.34–43, ACM, New York, NY, USA, 2001.

[17] P.R. Panda, N.D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems," ACM Trans. Des. Autom. Electron. Syst., vol.5, no.3, pp.682–704, 2000.

[18] S. Steinke, L. Wehmeyer, B. sik Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," Proc. Conference on Design, Automation and Test in Europe, pp.409–415, IEEE Computer Society, 2002.

[19] J. Sjödin and C. von Platen, "Storage allocation for embedded processors," CASES '01: Proc. 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp.15–23, ACM, New York, NY, USA, 2001.

[20] J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "Dynamic management of scratch-pad memory space," Design Automation Conference, pp.690–695, ACM Press, 2001.

[21] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management for portable systems with a memory management unit," EMSOFT '06: Proc. 6th ACM & IEEE International Conference on Embedded Software, pp.321–330, ACM, New York, NY, USA, 2006.

[22] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," CASES '03: Proc. 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp.276–286, ACM, New York, NY, USA, 2003.

[23] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," CODES+ISSS '04: Proc. 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp.104–109, ACM, New York, NY, USA, 2004.

[24] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," CASES '03: Proc. 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp.318–326, ACM, New York, NY, USA, 2003.

[25] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," J. Embedded Comput., vol.1, no.4, pp.521–540, 2005.

[26] R. McIlroy, P. Dickman, and J. Sventek, "Efficient dynamic heap allocation of scratch-pad memory," ISMM '08: Proc. 7th International Symposium on Memory Management, pp.31–40, ACM, New York, NY, USA, 2008.

[27] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," Proc. 14th International Conference on Parallel Architectures and Compilation Techniques, 2005, PACT 2005, pp.329–338, 2005.

[28] L. Wang, X. Yang, J. Xue, Y. Deng, X. Yan, T. Tang, and Q.H. Nguyen, "Optimizing scientific application loops on stream processors," LCTES '08: Proc. 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, pp.161–170, ACM, New York, NY, USA, 2008.

[29] E.G. Hallnor and S.K. Reinhardt, "A fully associative software-managed cache design," ISCA '00: Proc. 27th Annual International Symposium on Computer Architecture, pp.107–116, ACM, New York, NY, USA, 2000.

[30] J.E. Miller and A. Agarwal, "Software-based instruction caching for embedded processors," ASPLOS-XII: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.293–302, ACM, New York, NY, USA, 2006.

[31] C.A. Moritz, M.M. Frank, W. Lee, and S. Amarasinghe, Hot pages: Software caching for raw microprocessors, MIT/LCS Tech. Memo. LCS-TM-599, Aug. 1999.

[32] N. Nguyen, A. Dominguez, and R. Barua, "Scratch-pad memory allocation without compiler support for java applications," CASES '07: Proc. 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp.85–94, ACM, New York, NY, USA, 2007.

[33] N. Deng, W. Ji, and F. Shi, "A novel adaptive scratchpad memory management strategy," 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009.

[34] D.G. Feitelson, "Metrics for mass-count disparity," Proc. 14th International Symposium on Modeling, Analysis, and Simulation of Computer Systems, pp.61–68, 2006.

[35] Y. Etsion and D.G. Feitelson, "L1 cache filtering through random selection of memory references," PACT '07: Proc. 16th International Conference on Parallel Architecture and Compilation Techniques, pp.235–244, IEEE Computer Society, Washington, DC, USA, 2007.

[36] E.E. Johnson, "Working set prefetching for cache memories," SIGARCH Comput. Archit. News, vol.17, no.6, pp.137–141, 1989. ISSN 0163-5964.

[37] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," EuroSys '09: Proc. 4th ACM European Conference on Computer Systems, pp.89–102, ACM, New York, NY, USA, 2009. ISBN 978-1-60558-482-9.

Ning Deng received the B.E. degree in School of Computer from National University of Defense Technology, Chang Sha, in 2007. He is currently a Ph.D. Student at Beijing Institute of Technology. His primary research interest is on-chip memory management of microprocessors, with particular emphasis on scratchpad memory management for embedded systems currently. He is a student member of ACM.

Weixing Ji received his Ph.D. degree from Beijing Institute of Technology, Beijing, China, in 2008. He is currently a Assistant Professor with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests are in embedded systems, object-oriented programming, and computer architecture.

Jiaxin Li received the B.E. degree in computer science from Beijing Institute of Technology, Beijing, China, in 2004. She is currently working toward the Ph.D. degree in computer science at Beijing Institute of Technology, Beijing, China. She is the author of some papers in her areas of research interest, which include computing models of chip multi-processors and routing algorithms of network on chip.

Qi Zuo received the B.E. degree in computer science and engineering in 2005 from Beijing Institute of Technology. She is currently pursuing her Ph.D. degree in computer science at Beijing Institute of Technology. Her research focuses on parallel and distributed computing, computer architecture and memory management.

Feng Shi received his B.E. degree in physics in 1983 from Peaking University and received his Ph.D. degree from Beijing Institute of Technology, Beijing, China, in 1999. He is currently a Professor with the School of Computer Science and Technology, Beijing Institute of Technology. His research focuses on parallel computing and computer architecture.