JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering, Vol. 9, No. 2, June 2015, pp. 51-72

Scratchpad Memory Architectures and Allocation Algorithms for Hard Real-Time Multicore Processors

Yu Liu and Wei Zhang*

Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA

yuliu@siu.edu, wzhang4@vcu.edu

Abstract

Time predictability is crucial in hard real-time and safety-critical systems. Cache memories, while useful for improving the average-case memory performance, are not time predictable, especially when they are shared in multicore processors. To achieve time predictability while minimizing the impact on performance, this paper explores several time-predictable scratch-pad memory (SPM) based architectures for multicore processors. To support these architectures, we propose the dynamic memory objects allocation based partition, the static allocation based partition, and the static allocation based priority L2 SPM strategy to retain the characteristic of time predictability while attempting to maximize the performance and energy efficiency. The SPM based multicore architectural design and the related allocation methods thus form a comprehensive solution to hard real-time multicore based computing. Our experimental results indicate the strengths and weaknesses of each proposed architecture and the allocation method, which offers interesting on-chip memory design options to enable multicore platforms for hard real-time systems.

Category: Embedded computing

Keywords: Scratch-pad memory (SPM); Hard real-time system; Worst-case execution time (WCET); Multicore processors

I. INTRODUCTION

Time predictability is a crucial design consideration for hard real-time and safety critical systems such as aircraft and automotive control systems (e.g., x-by-wire), and medical systems (e.g., heart pacemakers, telesurgery). In these applications, missing deadlines of computing tasks may endanger human lives or lead to other catastrophic outcomes. To ensure safety and reliability for hard real-time and safety-critical systems, it is important to perform worst-case execution time (WCET) analysis to obtain a tight upper bound of the execution time, which can provide the basis for real-time schedulability analysis. The WCET, however, is not only determined by the application itself, but also heavily dependent on the target processor on which the application is running.

With the scaling of technology and the advancement of the design of computer architecture, modern microprocessors have become increasingly complex and unpredictable in timing behaviors. Historically, computer architecture research has concentrated on innovations to improve the average-case performance (and recently, energy efficiency), which are often harmful to the time predictability of computing. For example, modern features, such as superscalar pipelines, out-of-order execution, dynamic branch prediction, speculative execution, and caches, make it very difficult to accurately estimate the worst-case performance.

Open Access http://dx.doi.org/10.5626/JCSE.2015.9.2.51 http://jcse.kiise.org

Received 27 Feb 2015; **Revised** 27 Feb 2015; **Accepted** 31 Mar 2015

*Corresponding Author

In particular, cache memories have been widely used in modern processors to shorten the gap between the processor speed and memory access time. Unfortunately, cache memories are generally not time-predictable, as the execution time is highly dependent on the history of memory accesses, which can vary significantly for different inputs and cache states. This problem is aggravated by multicore architectures. In a multicore platform, the shared L2 cache architecture is widely used for multiple cooperative threads to efficiently share instructions, data, and the precious memory bandwidth for maximizing performance. In a multicore processor with a shared cache, different threads running on different cores can interfere in the shared cache, making it even more difficult and complex to predict the worst-case execution time.

Scratch-pad memories (SPMs) are memory arrays with decoding and column circuitry logic. An SPM is designed considering that the memory objects are mapped to the SPM in the last state of the compiler. The assumption here is that the SPMs occupy one distinct part of the memory address space with the remaining space occupied by main memory. Thus, unlike caches, we do not need to check the availability of the data/instruction in the SPM, which reduces the comparator and the signal miss/hit acknowledging circuitry compared to the caches. This contributes to the energy as well as area reduction [1]. Therefore, as an alternative technique to hardware controlled caches, SPMs offer the characteristics of time predictability and reasonable performance [2-5]. Industries have already employed SPMs in both single-core processors such as Freescale M-Core [6] and ARM7TDMI [7], and multicore processors such as IBM Cell [8], Tilera Tile64 [9], and NVIDIA Fermi GPU [10]. For example, NVIDIA's latest Fermi GPU has an SPM called shared memory, which can be partitioned into cache and SPM at configuration points 1:3 or 3:1, with SPM and L1 cache sitting on top of the L2 cache [11]. Similarly, the local store in IBM Cell broadband can be managed as a combination of direct buffers to store accesses with regular patterns and a software-controlled cache [12]. However, to the best of our knowledge, no prior work has performed a systematic design and evaluation of different SPM based architectures on multicore processors with the capability of full time-predictability.

In this paper, we study several different SPM based architectures on multicore processors with full time predictability. To understand the strengths and weaknesses of these proposed architectures, we evaluate them in terms of both execution time and energy consumption. It should be noted that the work of this paper has been published in the IEEE 30th International Conference on Computer Design (2012 ICCD) [13]. In this current journal version, we include the worst-case energy consumption [14] in the evaluation, which could provide a comprehensive result, as the energy consumption is also an important design consideration for real-time systems. Also, we add a new experiment to evaluate the impact of different bus band widths to the performance comparison between the SPM based and cache based architecture. Furthermore, since we have more space for this journal paper, we can provide more important details to better describe the proposed architectures and algorithms.

To efficiently exploit SPMs, it is crucial to determine the memory objects assignment to SPMs. Memory objects are chunks of data or instructions, such as basic blocks, global data, etc. Early studies on this topic mostly focus on compilation time algorithms to statically allocate hotspots of programs to SPMs [3, 4]. Later, researchers also studied dynamical allocation of memory objects, including software managed [15-18] and hardware assisted replacement methods [19-21]. Among these methods, both the static and software managed dynamic allocation methods can ensure full time predictability, since the SPM allocation is performed at the compiler stage. While [22] is the first paper to propose a set of optimal strategies to reduce the energy consumption of applications by sharing the SPM among multiple processes, it is based on a simple SPM architecture for energy saving. In this paper, we will extend both the static and dynamic allocation method for different SPM based architectures on multicore processors. Particularly, in order to support the shared L2 SPM, we propose a *dynamic allocation based partition L2 SPM strategy*, a *static allocation based partition L2 SPM strategy*, and a *static allocation based priority L2 SPM strategy*. In order to provide the 100% time predictability required by hard real-time systems, all these algorithms avoid any inter-core conflicts, which can greatly simplify the WCET analysis for multicore processors.

Our experimental results provide some interesting memory design options to enable real-time multicore computing. First, the performance of the two-level SPM architecture is superior to that of the one-level SPM architecture, while the latter is simpler in design and implementation. Second, the separated L1 instruction and data SPMs better fit the data-intensive real-time applications. Third, the dynamic allocation based partition method achieves the best performance on each core because of its flexibility for memory object allocation. However, the increased complexity and computation time may limit its applicability to large applications. Fourth, the WCET and worst-case energy consumption (WCEC) of the SPM-based multicore architectures are superior to the cache-based and hybrid architectures, while achieving time predictability.

II. RELATED WORK

In this section, we review related work on the WCET analysis of caches, time-predictable cache design, and SPM allocation for both single-core and multicore processors.

A. Related Work on Cache Timing Analysis

For real-time systems, especially hard real-time systems, it is crucial to obtain the WCET of each real-time task, which provides the basis for schedulability analysis. Missing deadlines in these systems may lead to serious consequences. In light of this, many research efforts have been made in the past two decades on WCET analysis focusing on cache based architectures [23]. Arnold et al. [24] proposed a static analysis method called the static cache simulation to bound worst-case instruction cache performance through classifying instruction cache accesses into first misses, always misses, first hits, and always hits. This technique was further summarized and extended in [11] and [12]. Li and Malik [25] and Li et al. [26] proposed to estimate worst case cache performance by using integer linear programming (ILP). Alt et al. [27] described an approach to predicting cache timing behavior based on abstract interpretation. Sebek and Gustafsson [28] proposed a static approach to determining the worst-case instruction cache miss ratio. Liu and Zhang [29, 30] utilized stack distance to predict cache misses, including instruction and data caches. Recently, several studies [31-34] have been carried out to examine the WCET estimation on shared L2 caches of multicore processors. These static analysis approaches, however, still have very high complexity and overestimation, which in general may not be scalable to a larger number of cores or larger benchmarks.

B. Related Work on Time-Predictable Microprocessor Architecture

Meanwhile, a number of researchers have attempted to exploit time-predictable caches to mitigate the complexity problem of static WCET analysis. Cache locking mechanism is a common solution that trades performance for predictability. Vera et al. [35] introduced a method that combines static cache analysis and cache locking in order to achieve both predictability and good performance. However, their work only focuses on data caches, which makes the method only a partial solution to real-time systems. Paolieri et al. [36] explored partitioned caches to improve cache time predictability, which however leads to performance degradation. Also, Plazar et al. [37] proposed a WCET-aware optimization technique for static I-cache locking which improves a program's performance and predictability.

Yan et al. [38] studied several time-predictable cache based multicore architectures, including prioritized and prioritized-partitioned caches to guarantee time predictability for real-time threads without significantly impacting the performance. Their work, however, only provided the solution and evaluation for dual-core systems. Moreover, its applicability was limited to the execution model where one core runs a real-time application while another core runs a non-real-time application, which may limit its use in real-time systems. Cullmann et al. [39] argue that some architectural features make timing analysis on multicore very difficult, but also show that smart configurations of existing complex architectures can alleviate this problem. Also, they point out that disjoint instruction and data caches, as well as private caches impair the precision of timing analysis and lead to a more complex analysis.

C. Related Work on SPM Allocation

An alternative to the cache is to use the SPM to achieve time predictability. Steinke et al. [3] first proposed an ILP based method in the compiler stage to statically allocate hot spots of programs into SPMs to save energy consumption maximally. Avissar et al. [4] also worked on the allocation of the static memory objects in the compiler stage, with a particular focus on the data objects, including global and stack data. Later, Suhendra et al. [40] exploited the ILP based method in the compiler stage, but their objective was to minimize a task's WCET by designing a heuristic search to achieve near-optimal allocations. These three works have a commonality whereby the memory objects allocated to SPMs are fixed during the run time. In addition, Marwedel et al. [41] described a comprehensive set of algorithms applied in the design time to maximally exploit SPMs.

Researchers have also attempted to dynamically allocate the memory objects to SPMs to improve the performance. Whitham and Audsley [21] designed a specific hardware named SPM management unit to provide OPEN and CLOSE operations for implementing dynamic allocation. Li et al. [18] split the live ranges of arrays to create potential data transfer statements between the SPM and off-chip memory, and finally adapted an existing graph-coloring algorithm for register allocation to achieve dynamic allocation. Furthermore, an inter-procedural heuristic for identifying hot instruction traces to insert in the SPMs was proposed by Ravindran et al. [17]. Verma and Marwedel [16] first proposed an ILP based method to solve the dynamic allocation problem in the compiler stage and Deverge and Puaut [15] later utilized the same ILP based method to improve the performance on the worst-case performance path. It should be noted that compiler stage based methods can offer full time predictability regardless of whether it is static or dynamic. Although these works contributed to the significant progress of SPM allocation to attain time predictability, the target SPM architectures are very simple and are far behind the requirements of multicore computing.

However, all of the studies mentioned above target SPMs for traditional single-core processors. These allocation algorithms [3, 4, 15-18, 21, 40, 41] focus on maximizing performance, energy efficiency, or WCET for a single thread. In a multicore environment, since multiple threads run concurrently, the allocation algorithms that concentrate on optimizing each single thread may not lead to a global optimal result. Moreover, a local SPM for each core is unlikely to be efficient for multicore processors; thus, the SPM architecture for multicore processors is likely to differ from that for single-core processors. For example, for multicore processors with shared SPM, a thread with many frequently used instructions or data can easily use most shared SPM spaces if not managed properly. This can cause other concurrent threads that are not able to use the SPM space, and hence degrade their performance or even miss their deadlines. Besides addressing the inter-core conflicting problem, this paper also studies a two-level SPM to replace the one-level SPM in order to boost the performance of SPM based architectures. For the two-level SPM, the allocator must be aware of the instructions/data that have already been allocated to the level-1 SPM, and the allocation must be aware of the different latencies to access different levels of SPMs, none of which are considered in traditional SPM allocation for single-core processors.

Kandemir et al. [42] studied a compiler strategy to optimize data accesses in regular array-intensive applications running on embedded multiprocessors with a virtually shared scratch-pad memory (VS-SPM). Their SPM allocation algorithm can increase the application-wide reuse of data that resides in SPMs of processors and thus reduce the extra off-chip memory accesses caused by inter-processor communication. This approach [42], however, does not consider the worst-case path, and thus cannot guarantee optimization of the WCET.

While SPMs based architectures can provide time predictability for real-time systems, this may come at the cost of performance if the SPM space cannot be used efficiently. Consequently, evaluation of the performance compared to the cache based system is important. Banakar et al. [43] carried out a comprehensive evaluation between SPM and cache memories. However, their work focused on a single level shared SPM and cache memory, which is not a representative architecture used in modern high-performance microprocessors. Liu and Zhang [5] studied the two-level SPM based architecture, which can result in better performance and energy results. Moreover, a two-level SPM, resembling a two-level (or multi-level in general) cache architecture that is typical for multicore processors, allows us to make a fair comparison between the SPMs and the caches based architectures. Therefore, in this paper, we focus on studying a two-level SPM cache architecture for multicore processors.

III. SPM ARCHITECTURES FOR MULTICORE PROCESSORS

A. Proposed SPM Based Architectures

Most modern high-performance multicore computer systems typically employ a two-level (or generally multi-level) cache based architecture. The level 1 cache is the fastest form of memory, and is built onto the chip but has a limited size. Also, two separated L1 caches are normally used to store instructions and data, respectively. A typical cache-based multicore architecture is shown in Fig. 1.

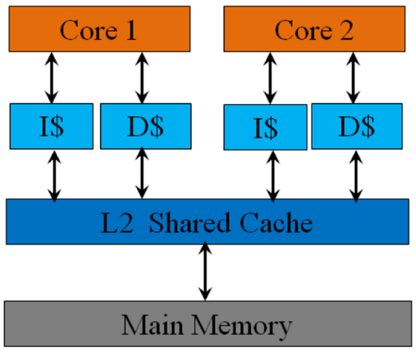


Fig. 1. Two-level cache based architecture for dual-core processors with separated L1 instruction and data cache of each core.

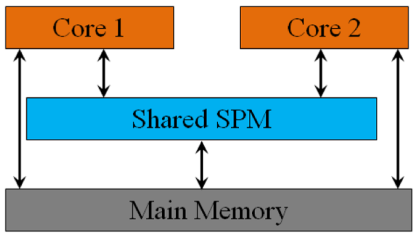


Fig. 2. One-level SPM based architecture (OLS-arch) for dual-core processors. SPM: scratch-pad memory.

Compared to caches, SPMs can achieve both time predictability and low energy consumption. Based on the characteristics of SPMs and the typical architectures of modern multicore processors, we design several SPM based memory architectures on multicore processors. First, a one-level SPM based architecture (called OLS-arch) is proposed, in which different cores share the same one-level SPM, as shown in Fig. 2. The obvious advantage of this architecture is its simplicity of design and implementation. Second, we propose a two-level SPM based architecture (called TLS-arch) with a unified L1 SPM on each core and a L2 SPM shared by all cores, as depicted in Fig. 3. The two-level architecture can trade-off between the speeds and sizes of different SPMs. Third, we propose another two-level SPM based architecture, of which the L1 SPMs are separated for each core as the L1 instruction and data SPMs (called TLSL1S-arch), as demonstrated in Fig. 4. In all these SPM based architectures, the L1 SPM is the fastest static random access memory (SRAM) with a small size, while the L2 SPM is a slower SRAM with a larger size.

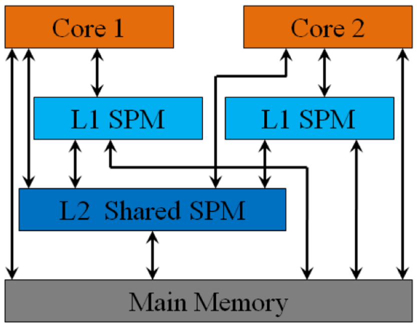


Fig. 3. Two-level SPM based architecture (TLS-arch) for dual-core processors with the unified L1 SPM of each core. SPM: scratch-pad memory.

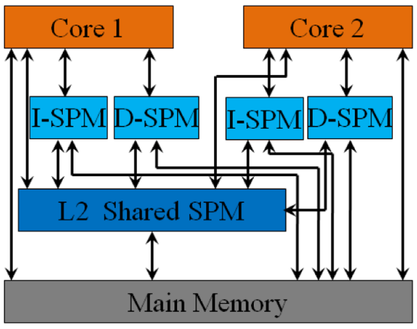


Fig. 4. Two-level SPM based architecture (TLSL1S-arch) for dual-core processors with separated inst. and data L1 SPM of each core. SPM: scratch-pad memory.

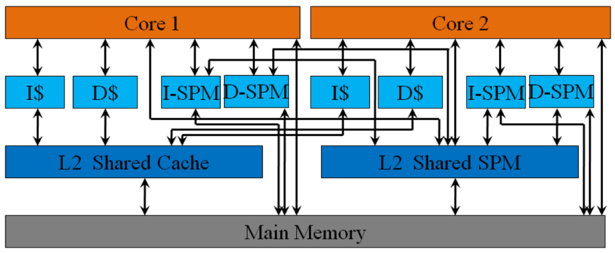


Fig. 5. Two-level hybrid architecture for dual-core processors. SPM: scratch-pad memory.

To quantitatively understand the strengths and weaknesses of these SPM-based architectures on multicore processors, we quantitatively evaluate these architectures in terms of both performance and energy efficiency through extensive simulation. We also compare the SPM-based multicore architecture with a similar cache based-architecture and a hybrid architecture. The hybrid architecture utilizes both SPMs and caches, as depicted in Fig. 5. In this architecture, the memory objects not allocated to any SPM will pass through the caches instead of being fetched from the main memory directly.

B. Timing Performance Models

Besides time predictability, all the proposed SPM-based architectures on multicore processors clearly share two advantages in terms of performance. First, the cache-based architecture needs to fetch the size of an L1 cache line from an L2 cache for an L1 cache miss (i.e., an L1 cache load), and the size of an L2 cache line from the main memory for an L2 cache miss (i.e., an L2 cache load). In contrast, the SPM-based architecture only needs to fetch the size of one instruction or data object (i.e., char, int, float, double, etc.) from any level of SPM or memory directly. Second, any fetch from the main memory requires the upper levels of memories to be updated in the cache-based architecture, while no such updating is needed in the SPM-based architecture due to the direct connection between the processor and the SPMs.

We assume that both the SPM and the corresponding cache memories have the same capacity and use the same types of hardware with the same device response time for unit data operations. This is why the same latency is selected for memories regardless of whether their type is cache or SPM. We use Eqs. (1), (2), and (3) to compute the latency of any memory operation in the L1 instruction, L1 data, L2 shared SPM, and the main memory in our architecture.

We assume that the smallest unit of memory operation allowed on the bus is one word. It should be noted that setting the memory bandwidth to be one word is the common assumption in the SPM related research. The work in [20, 21] was based on a 1 word memory bandwidth (defined as data size per unit time) to evaluate the SPM performance. Ravindran et al. [17] utilized 2 bytes memory bandwidth in their research. Li et al. [18] even used 1 byte memory bandwidth to study their SPM management algorithm. To understand the impact of higher bandwidth on the proposed SPM architectures and allocation algorithms, in Section VIII-C3, we also provide the evaluation results based on different bandwidths.

We use inclusive caches in our evaluation. While inclusive caches may waste some space and bandwidth in order to enforce the inclusion property compared to exclusive caches, inclusive caches are highly desirable for multicore processors because they facilitate memory controller and processor design by limiting the effects of cache coherency messages to higher levels in the memory hierarchy [44]. We plan to compare the SPM-based architectures with exclusive caches in our future work.

We adopt the performance model described in [43] and formalize the timing relation between the SPMs and the caches in the following equations. In Eq. (1), T_{s1} and T_{c1} are the latencies of the first-level SPM and cache, respectively. In Eq. (2), T_{s2} and T_{c2} are the latencies of the second-level SPM and cache, respectively, S_{c1} is the size of a cache line in an L1 cache, S_{inst} is the size of one instruction, and S_{data} is the size of any data type. In Eq. (3), T_{sm}

and T_{cm} are the latencies of the main memory in the SPM-based and the cache-based architectures respectively and S_{c2} is the size of a cache line of an L2 cache. We need to reduce double latencies of the L2 cache operation because any L2 cache miss needs to write data from the main memory into the L2 cache and then read data from the L2 cache in the cache-based architecture. We have the same consideration for the L1 cache miss.

$$T_{s1} = T_{c1}, \text{ for instruction/data objects} \tag{1}$$

$$T_{s2} = \begin{cases} (T_{c2} - 2 \times T_{c1})/S_{c1} \times S_{inst}, & \text{for inst.,} \\ (T_{c2} - 2 \times T_{c1})/S_{c1} \times S_{data} & \text{for data} \end{cases} \tag{2}$$

$$T_{sm} = \begin{cases} (T_{cm} - 2 \times T_{c2} - 2 \times T_{c1})/S_{c2} \times S_{inst}, & \text{for inst.,} \\ (T_{cm} - 2 \times T_{c2} - 2 \times T_{c1})/S_{c2} \times S_{data} & \text{for data} \end{cases} \tag{3}$$

Note that we do not include the shared bus and other factors in the timing models as our first multicore SPM related work. A major reason is that the very long instruction word (VLIW) compiler and simulator tool (i.e., Trimaran [45]) we used cannot support bus simulation and is difficult to extend, but Trimaran is currently the best tool for VLIW related research. Refer to Section VII for detailed information on the way we extend Trimaran for the evaluation work.

C. Energy Performance Models

The main components in a cache include the decoder, the tag memory array, the tag column multiplexers, the tag sense amplifiers, the tag comparators, the tag output drivers, the data memory array, the data column multiplexers, the data sense amplifiers, and the data output drivers, while the SPM only needs the decoding and the column circuitry logic. Thus, the SPM is essentially more energy efficient than the cache.

Based on the cache components, Kamble and Ghose [46] proposed an analytical energy dissipation model for the low power cache, which has been widely used in the research of cache energy estimation. In Eq. (4), the total amount of energy dissipated by a cache can be expressed as the sum of four components, including bit-line dissipations, word-line dissipations, dissipations in output lines, and dissipations in input lines [46]. First, the bit-line dissipations are due to precharging, readout, and writes of bit-line transitions. Second, word-line dissipations include the energy expended in driving the gate of the row driver. Third, output line dissipations are the energy dissipated when driving interconnects lines external to the cache towards the CPU side or the memory side. Last, the input line dissipations are energy expended in the input gates of the row decoder. The energy model of an SPM can largely reuse this equation but needs to remove the consideration of tag bits in the calculation of E_{bit} and E_{word}. Also, the SPM energy estimation does not need to consider the address output in the calculation of E_{output} due to the direct connection between the SPMs and the processor. In practice, we adapted the energy consumption evaluation tool EPIC-Explorer [47] to conduct the performance evaluation, and the development of EPIC-Explorer is based on CACTI [48].

$$E_{dissipation} = E_{bit} + E_{word} + E_{output} + E_{input} \tag{4}$$

IV. MEMORY OBJECTS ALLOCATION DESIGN CONSIDERATION

The memory objects assignment algorithms used to offer the full-time predictability should avoid both intra-core and inter-core conflicts. In the related work section, we mentioned that the ILP based method can avoid intra-core conflicts for either the static based method [43] or the dynamic based method [15, 16] (called the baseline methods). Consequently, it is attractive to extend the static and dynamic based ILP methods to support the proposed SPM based multicore architectures to maintain time predictability while maximizing performance/energy. Specifically, we extended the static based method in [43] and the dynamic based method in [16] to support the multi-level multicore SPM architectures proposed in this paper. The work in [15] is also based on [16], but its goal is to optimize the performance on the worst-case path, which differs from ours. Our design consideration and architectural extension are explained as follows.

The decision of memory allocation is made in the compiler stage, which is common to both the static and dynamic allocation methods. The difference is that the static allocation does not relocate the memory objects in the run-time. In other words, the memory objects are loaded to SPMs and the main memory prior to the run-time and remain there during the entire running time. Thus, the static allocation does not need any control flow graph (CFG) information. On the contrary, dynamic allocation permits the possible memory objects to copy (called Spill Actions) among different memories at the run-time, but the copying actions have been determined in each edge of the CFG during the compiler stage, not the running stage. The motivation to use the dynamic approach is that the memory objects only need to stay in SPMs during their live range. Thus, the dynamic approach requests a liveness analysis on the CFG, and the SPMs could then be reused for different memory objects during the run-time.

Besides avoiding the intra-core conflicts, we also need to avoid the inter-core instruction/data access conflicts that are specific to multicore processors. This is because

Table 1. Comparison between baseline methods and extended methods

Item	Baseline method	Extended method
Level	Only support one level SPM	Support multiple levels of SPMs
Type	Only support one mixed data and instruction SPM	Support separated the data and instruction SPMs
Core	Only support single core architecture	Support multicore architectures by different ways of partitioning the shared L2 SPM
Target	Optimize the WCET performance [15] or improve the performance [16]	Keep the full time predictability on different multicore platforms
Evaluation	Only provide limited quantitative analysis results	Provide performance impact of different architectures, complexity performance analysis, etc.

SPM: scratch-pad memory, WCET: worst-case execution time.

accurate prediction of the worst-case run-time inter-core conflicts is very complex, if not impossible, especially for large programs running on a large number of cores.

Naturally, partitioning is the best way to avoid any conflict. It can either be manually performed by the users or automatically performed by the system. In this paper, we have three methods, which will be proposed in the following sections. Dynamic partition and static partition depend on users to manually partition the shared L2 SPM. The other static method is called static priority, which involves the partition automatically based on the demand of applications on different cores. The memory objects allocated to the shared L2 SPM by different cores are interwoven for the static based priority method, but they are not overlapped. In other words, we do not have a separated space for each core. On the contrary, both the dynamic based and static based partition methods have separated space for each core in the L2 SPM. We will quantitatively compare and analyze these methods in order to understand their respective strengths and weaknesses.

Table 1 shows a summary of the extension of the ILP methods used in this paper (called the extended methods) compared to the baseline methods. Also, the detailed design of the dynamic memory object allocation is described in Section V, and the static allocation is described in Section VI.

V. MULTICORE DYNAMIC MEMORY OBJECT ALLOCATION

Generally, the dynamic allocation method needs to depend on the CFG information of each core. Unfortunately, one cannot simply merge the CFGs of programs running on different cores because the programs on different cores have totally independent codes, which are not guaranteed to execute synchronously. Therefore, the partitioned L2 SPM is a viable solution to utilize the dynamic allocation for the SPM based architectures on multicore processors, of which the L2 SPM is shared by

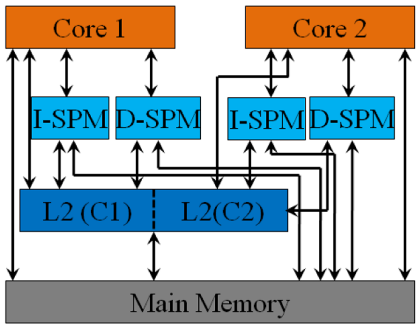


Fig. 6. Dynamic allocation based partition L2 SPM strategy. C1: core 1, C2: core 2, SPM: scratch-pad memory.

all cores. Accordingly, we call this dynamic allocation based L2 SPM sharing strategy the *Dynamic allocation based Partition L2 SPM strategy* (called DPaL2). Fig. 6 demonstrates this strategy.

It is worth clarifying the reason why we cannot use the priority based strategy for the dynamic allocation method. First, the priority based approach also needs to partition the L2 SPM for different cores. The difference between these approaches is that the priority based approach involves partitioning the L2 SPM based solution on the demand of each core through the ILP method, while in the partition based approach, the L2 SPM needs to be partitioned before setting up ILP. Second, as it is impossible to merge the CFGs of different cores, we cannot set unique ILP objects and constraints for the dynamic based method. This is because the programs running on different cores cannot be guaranteed to execute synchronously. In other words, one program may start earlier or later than other programs. The static allocation has no run-time copying actions so that it does not care about this asynchronous concern, which is why the priority method fits for the static allocation rather than the dynamic allocation. Therefore, to use the dynamic approach, we need to partition the L2 SPM in advance, and set up the ILP objects and constraints for each core separately.

A. Overview

Dynamic allocation permits the possible memory objects copying (called *Spill Actions*) among different memories at the run-time, which makes the decision of memory objects allocation in each edge of the CFG. It is motivated by the observation that if the liveness of two hotspots in a program does not overlap, we can reuse the space of SPMs to boost the performance. We make the optimal allocation of memory objects to SPMs on each edge of the global CFG, and the candidates of memory objects are only within those that are alive in the destination basic block of a specific edge. The global CFG is derived from merging the CFGs of all functions of the program running on a core into one unique CFG. In other words, we do not merge CFGs of programs running on different cores. The liveness analysis can be conducted for each memory object on the CFG to determine its liveness range. In addition, the ILP-based method can be used in our two-level SPM dynamic allocation to determine the optimal allocation solution. *Spill Actions* include *Spill Load* (i.e., moving memory objects to a specific SPM from the main memory or another SPM) and *Spill Store* (i.e., moving a memory object from a specific SPM to the main memory or another SPM).

It should be noted that when memory objects are copied into SPM, their addresses change. This can change the behavior of a program unless all pointers to the memory objects are updated. Usually, the SPM MMU is designed to avoid these problems, which implements address transparency. An object can be relocated to SPM without changing its logical address. Therefore, relocation does not invalidate any pointers and causes no issues related to pointer aliasing. Also, we assume that a SPM MMU can deal with the fragmentation perfectly. The SPM MMU may further partition the memory objects to the optimal size of the region to mitigate the effect of fragmentation, although this is beyond the scope of this paper.

B. Memory Objects and Liveness Analysis

1) Memory objects: The memory objects covered in our work include instruction objects and data objects. The instruction objects consist of basic blocks and functions, and our data objects consist of global scalars, non-scalar variables, and stack data.

In order to support the liveness analysis requested below, we need to analyze the *Static Attributes* of memory objects based on the basic concept of a DEF-USE chain. A reference to a memory object can be classified as a *DEF*, *MOD*, *USE* or *CONT* as the static attribute [16]. If a reference assigns a value to all the elements of a memory object, then it is classified as a *DEF*. If only some elements but not all are assigned, then the reference is assumed to be a *MOD*. Any reference reading a value of the elements of a memory object is assumed to be a *USE* [16]. In the liveness analysis, if a memory object is alive in a basic block but does not have any reference to the basic block, its attribute is regarded as *CONT*.

2) Liveness analysis: Liveness analysis relies on the global CFG and the static attributes for memory objects classified above. Specifically, we create the *DEF* attributes in the root node (the first basic block) of the global CFG for all the instruction memory objects. We also need to create the *DEF* attributes in the root node for some data objects, in which the elements are only partially set values in the whole program.

The memory assignment problem is formulated such that the memory objects are assigned to SPMs on the edges rather than at the nodes in the CFG [16]. Consequently, we need to derive the static attributes of memory objects on the edges from their attributes in the basic blocks. In this work, a *DEF*, *CONT*, *MOD* or *USE* attribute of a memory object is defined on every in-edge of a basic block with the *DEF*, *CONT*, *MOD* or *USE* attribute of that memory object reference. Note that the *in-edges* of a basic block are the edges following which the control flow enters this basic block. To ensure all basic blocks have *in-edges*, a virtual in-edge needs to be set up for the first basic block of the program to satisfy the requirements of this definition.

If more than one static attribute for a memory object can be assigned to an edge, the following priority is used to determine the appropriate attribute: *DEF* > *MOD* > *USE* > *CONT* [16]. The technique used to compute the live range of memory objects is changed to obtain the *CONT* attribute of these memory objects on each basic block of the global CFG. Finally, we use the DEF-USE chain based *live-in*/*live-out* sets computing algorithm to achieve this goal.

3) Consideration of stack data: Actually, the allocation of stack data to different memories (e.g., SPM, main memory) is used to address the research question of distributing stacks for heterogeneous memory units. A feasible solution is to combine the stack data into a single aggregated data object of each function. This is because if the stack data in the same function is allocated to different memory units, multiple stack registers instead of single stack registers are needed to track them, which violates the processor's internal architecture (normally there is only one stack register). The handling of the aggregated stack data object is the same as the normal data object, which is the benefit of this solution.

In practice, we can further improve this solution to the stack data by separating the aggregated stack data object into several smaller data objects. Otherwise, the size of the aggregated stack data object of each function may be too large to be allocated to any SPM. In compilers, the area on the stack devoted to the local variables, parameters, return address, and other temporaries for a function

Table 2. Notations used in the ILP variables for the dynamic allocation method

Notation	Explain
MO	The memory object
T	The time latency
j_1	The static attribute, which is in the set of static attributes (DEF, CONT, USE, MOD)
j_2	The spill action attributes, which is in the set of spill action attributes (LOAD, STORE)
i	i^{th} edge in the CFG
k_1	The instruction memory object, which is in the set of instruction memory objects
k_2	The data memory object, which is in the set of data memory objects
k	The memory objects, which is in the union of set k1 and k2
L_{1I}	The L1 instruction SPM
L_{1D}	The L1 data SPM
L_2	The L2 SPM
$L_1I\,M$	The spill action between the L1 instruction SPM and memory
$L_1D\,M$	The spill action between the L1 data SPM and memory
$L_1I\,L_2$	The spill action between the L1 instruction SPM and L2 SPM
$L_1D\,L_2$	The spill action between the L1 data SPM and L2 SPM
$L_2\,M$	The spill action between the L2 SPM and memory

ILP: integer linear programming, SPM: scratch-pad memory, CFG: control flow graph.

is called the function's activation record or stack frame. Specifically, in the Trimaran framework [45] used in our experiments, the stack frame is divided into four areas, including the Swap Area (aka, Register Spill Area), Local Variables Area, Outgoing Parameters Area, and Frame Make Area, and each area boundary can be tracked by the macro registers set in the Trimaran simulated VLIW architecture. Each area in this framework can be treated as a separated stack data object in our methods.

C. ILP-Based Algorithm

1) Binary variables: At the beginning, we define some binary variables used in the ILP formulas to attain the optimal SPM dynamic allocation results. These variables include $MO_{j_1k_1}^{iL_{1I}}$, $MO_{j_1k_2}^{iL_{1D}}$, $MO_{j_1k}^{iL_2}$, $MO_{j_2k_1}^{iL_{1I}M}$, $MO_{j_2k_2}^{iL_{1D}M}$, $MO_{j_2k}^{iL_2M}$, $MO_{j_2k_1}^{iL_{1I}L_2}$ and $MO_{j_2k_2}^{iL_{1D}L_2}$. The notations used in these variables are explained in Table 2.

Based on the definition of the above binary variables, we can conclude the following SPM allocation results. For example, if an instruction memory object is allocated to an L1 instruction SPM on the i^{th} edge, $MO_{j_1k_1}^{iL_{1I}} = 1$, and $MO_{j_1k}^{iL_2} = 0$. Also, setting the spill action related binary variable as 1 means we enable this spill action. For example, $MO_{LOADk_1}^{iL_{1I}M} = 1$ means copying this instruction memory object from the main memory to an L1 instruction SPM on the i^{th} edge of the global CFG, and vice versa.

2) Objective function and constraints: The objective function is to achieve the shortest execution time on the two-level SPM memory architecture based on the dynamic allocation method. Eq. (5) is the ILP objective function. In this equation, $T_{profit}^{s_1}$ is the profit if a memory object is allocated to the SPM s_1, where s_1 belongs to the set of L_{1I}, L_{1D}, and L_2. The profit is defined as the saved number of clock cycles if we allocate the memory object to a specific SPM instead of the main memory. $T_{cost}^{s_2}$ is the cost of transferring the load/store of the memory object to/from a specific SPM, where s_2 belongs to the set of $L_{1I}M$, $L_{1D}M$, L_2M, $L_{1I}L_2$, and $L_{1D}L_2$. The cost is defined as the sum of reading and writing cycles, and an extra overhead cycle to move a memory object between two different memories. Particularly, Eq. (6a) is the sum of the profits of spill actions. Eq. (6b) is the sum of the costs of spill load actions between SPMs and main memory. Eq. (6c) is the sum of costs of spill load actions between L1 and L2 SPMs. Eq. (6d) is the sum of the costs of spill store actions between SPMs and main memory. Eq. (6e) is the sum of costs of spill store actions between L1 and L2 SPMs.

$$max(6a) - (6b) - (6c) - (6d) - (6e) \tag{5}$$

$$\sum_i \sum_k \{T_{profit}^{L_{1I}} \times MO_{j_1k_1}^{iL_{1I}} + T_{profit}^{L_{1D}} \times MO_{j_1k_2}^{iL_{1D}} + T_{profit}^{L_2} \times MO_{j_1k}^{iL_2}\} \tag{6a}$$

$$\sum_i \sum_k \{T_{cost}^{L_{1I}M} \times MO_{LOADk_1}^{iL_{1I}M} + T_{cost}^{L_{1D}M} \times MO_{LOADk_2}^{iL_{1D}M} + T_{cost}^{L_2M} \times MO_{LOADk}^{iL_2M}\} \tag{6b}$$

$$\sum_i \sum_k \{ T_{cost}^{L_{1I}L_2} \times MO_{LOADk_1}^{iL_{1I}L_2} + T_{cost}^{L_{1D}L_2} \times MO_{LOADk_2}^{iL_{1D}L_2} \} \tag{6c}$$

$$\sum_i \sum_k \{ T_{cost}^{L_{1I}M} \times MO_{STOREk_1}^{iL_{1I}M} + T_{cost}^{L_{1D}M} \times MO_{STOREk_2}^{iL_{1D}M} + T_{cost}^{L_2M} \times MO_{STOREk}^{iL_2M} \} \tag{6d}$$

$$\sum_i \sum_k \{ T_{cost}^{L_{1I}L_2} \times MO_{STOREk_1}^{iL_{1I}L_2} + T_{cost}^{L_{1D}L_2} \times MO_{STOREk_2}^{iL_{1D}L_2} \} \tag{6e}$$

To ensure that the allocation results match the liveness analysis, the flow information of global CFG, the sizes of SPMs, and the consistency of the SPM spilling actions, the following ILP constraints formulas need to be modeled.

SPM Spill Load Constraint: This constraint ensures that if we allocate a memory object to a specific SPM on an edge (also called the current edge), this memory object should be either allocated in the same SPM on the incoming edge or spill loaded to this SPM on the current edge. Eqs. (7)–(11) represent such constraint formulas, and Fig. 7(a) is utilized to illustrate these formulas. In Eq. (7), if the memory object is allocated to the L1 instruction SPM on the *j*th edge (the current edge), it should be already allocated to the L1 instruction SPM on the *i*th edge (the incoming edge), or spill loaded from the memory/L2 SPM to the L1 instruction SPM on the *j*th edge (the current edge). Eqs. (8) and (9) can be explained in a similar way. Also, Eqs. (10) and (11) are used to guarantee that the memory object should be already allocated to an L2 SPM on the *i*th edge (the incoming edge), if it is loaded to one of the L1 SPMs on the *j*th edge (the current edge).

$$MO_{j_1k_1}^{e_jL_{1I}} - MO_{LOADk_1}^{e_jL_{1I}M} - MO_{LOADk_1}^{e_jL_{1I}L_2} - MO_{j_1k_1}^{e_iL_{1I}} = 0 \tag{7}$$

$$MO_{j_1k_2}^{e_jL_{1D}} - MO_{LOADk_2}^{e_jL_{1D}M} - MO_{LOADk_2}^{e_jL_{1D}L_2} - MO_{j_1k_2}^{e_iL_{1D}} = 0 \tag{8}$$

$$MO_{j_1k}^{e_jL_2} - MO_{LOADk}^{e_jL_2M} - MO_{j_1k}^{e_iL_2} = 0 \tag{9}$$

$$MO_{LOADk_1}^{e_jL_{1I}L_2} - MO_{j_1k_1}^{e_iL_2} \leq 0 \tag{10}$$

$$MO_{LOADk_2}^{e_jL_{1D}L_2} - MO_{j_1k_2}^{e_iL_2} \leq 0 \tag{11}$$

SPM Spill Store Constraint: This constraint ensures that if we allocate a memory object to a specific SPM on an edge (the current edge), this memory object should be either still allocated in the same SPM on the next edge or spill stored from a specific SPM on the current edge. Fig. 7(b) is utilized to illustrate the SPM Spill Store Constraint.

Although the Spill Store seems to give more flexibility to the dynamic allocation algorithm, it has a negative impact if we use the SPM Spill Store Constraint together with the SPM Spill Load Constraint. Fig. 7(c) illustrates the '*deadlock*' symptom between these two constraints, and we take the L2 SPM allocation as an example. Assume that $MO_k^{e_i}$ is not allocated to the L2 SPM on the *i*th edge; thus, it is 0. Then, both $STORE_k^{e_i}$ and $MO_k^{e_j}$ must be 0. If $MO_k^{e_j}$ is 0, it will result in $LOAD_k^{e_j}$ as 0, which means that the Spill Loading this memory object cannot occur. However, it is possible we can Spill Load this memory on the *j*th edge to improve the performance. In our opinion, there are two solutions to address this '*deadlock*' problem. The first solution involves using these two constraints on the same edge pairs concurrently. However, it is impossible for us to determine which edge pair uses the SPM Spill Load Constraint and which one implements the Spill Store Constraint to achieve the optimal allocation. The second solution involves using the SPM Spill Load Constraint, which means we will not Spill Store any memory object in its live range. Actually, this solution follows the general solution to register an allocation that we will not spill the data from a register to the memory during the live range of this data. In our work, we utilize the second solution.

Merge Constraint: This constraint ensures that if a memory object with *DEF*, *MOD*, *USE* or *CONT* static attribute is allocated to one specific SPM on an edge, it should be also allocated to the same SPM in other edges merging to the same basic block, which are presented in Eqs. (12), (13), and (14) for the L1 instruction, L1 data, and L2 SPM, respectively. Fig. 7(d) illustrates Merge Constraint.

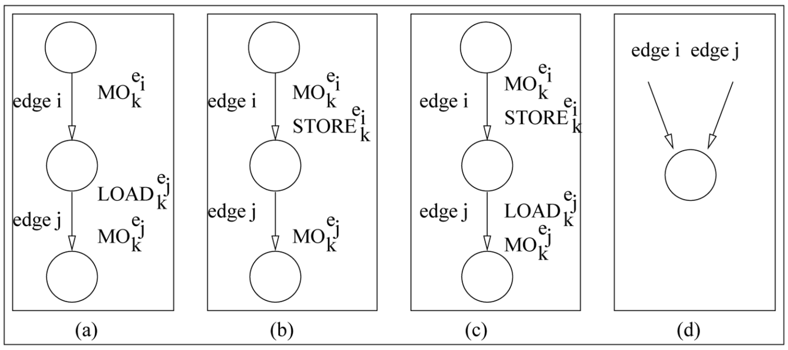


Fig. 7. SPM spill operations constraints of dynamic allocation (note that the simplified notations are used in this figure). (a) LOAD constraint, (b) STORE constraint, (c) 'deadlock', and (d) Merge constraint.

$$MO_{j_1k_1}^{e_iL_{1I}} = MO_{j_1k_1}^{e_jL_{1I}} = \ldots \tag{12}$$

$$MO_{j_1k_2}^{e_iL_{1D}} = MO_{j_1k_2}^{e_jL_{1D}} = \ldots \tag{13}$$

$$MO_{j_1k}^{e_iL_2} = MO_{j_1k}^{e_jL_2} = \ldots \tag{14}$$

Other Constraints: First, the SPM size constraint ensures the aggregate size of all memory objects allocated to the same SPM on an edge does not exceed the size of that SPM. Second, the SPM level constraint ensures that a memory object will not be allocated to the L1 SPM and L2 SPM at the same time. Third, the Instruction Object Parent Relation Constraint ensures that a basic block instruction object will not be allocated with its *parent function* instruction object to the SPMs at the same time. The parent function of a basic block is the function that includes this basic block.

D. An Example

In this section, we use a simple example, as shown in Fig. 8, to illustrate how to generate the ILP formulas. The example is extracted from a typical C code with an if-else statement inside a for loop. Due to the limited space, we use the memory object bb_2 on some edges to present the ILP formulas. Other memory objects can be used in the same way to generate the ILP formulas. Also, we only demonstrate the objective function, spill load, and merge constraint (other constraints are straightforward). The memory object bb_2 has *USE* attribute on the in-edges e_{1_2} and e_{5_2}, and *CONT* attribute on other in-edges according to the liveness analysis.

First, we use the memory object bb_2 on the in-edge e_{1_2} to show the objective function in Eq. (15). The profit of allocating a one word memory object to the L1 and the L2 SPM is 10 and 7 cycles, respectively. The cost of moving a one word memory object between the L1 SPM and the memory is 11 cycles, that between the L2 SPM and the memory is 14 cycles, and that between the L1 and the L2 SPMs is 4 cycles. The cost/profit per word is computed based on the equations in Section III-B and the definitions of profit and cost in Section V-C2. The edge weight values are based on profiling. Suppose the size of bb_2 is 5 words, and the weight of in-edge e_{1_2} is 1; the objective function for bb_2 can thus be computed by Eq. (15).

$$\begin{aligned}&(10\times5\times1)MO_{USEbb_2}^{e_{1\_2}L_{1I}}+(7\times5\times1)MO_{USEbb_2}^{e_{1\_2}L_2}\\&-(11\times5\times1)MO_{LOADbb_2}^{e_{1\_2}L_{1I}M}-(14\times5\times1)MO_{LOADbb_2}^{e_{1\_2}L_{2I}M}\\&-(4\times5\times1)MO_{USEbb_2}^{e_{1\_2}L_{1I}L_2}\end{aligned} \tag{15}$$

Second, we use the memory object bb_2 on the in-edge e_{1_2} and e_{2_3} to show the spill load constraint in Eqs. (16)–(18). Take the L1 instruction SPM as an example, if bb_2 is allocated in this SPM on the in-edge e_{2_3}, it should be either loaded from the memory or the L2 SPM on the in-edge e_{2_3} or already allocated in the L1 instruction SPM on the in-edge e_{1_2}.

$$MO_{CONTbb_2}^{e_{2\_3}L_{1I}}-MO_{LOADbb_2}^{e_{2\_3}L_{1I}M}-MO_{LOADbb_2}^{e_{2\_3}L_{1I}L_2}-MO_{USEbb_2}^{e_{1\_2}L_{1I}}=0 \tag{16}$$

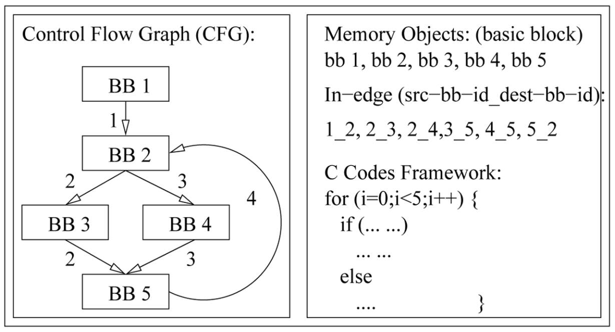
$$MO_{CONTbb_2}^{e_{2\_3}L_2}-MO_{LOADbb_2}^{e_{2\_3}L_2M}-MO_{USEbb_2}^{e_{1\_2}L_2}=0 \tag{17}$$

$$MO_{LOADbb_2}^{e_{2\_3}L_{1I}L_2}-MO_{USEbb_2}^{e_{1\_2}L_2}\le0 \tag{18}$$

Third, we use the memory object bb_2 on the in-edge e_{1_2} and e_{5_2} to show the merge constraints in Equations 19 and 20. In-edges e_{1_2} and e_{5_2} flow into the same basic block. If bb_2 is allocated to the L1 instruction/L2 SPM on the in-edge e_{1_2}, it should also be allocated to the same SPM on the in-edge e_{5_2}.

$$MO_{USEbb_2}^{e_{1\_2}L_{1I}}=MO_{USEbb_2}^{e_{5\_2}L_{1I}} \tag{19}$$

$$MO_{USEbb_2}^{e_{1\_2}L_2}=MO_{USEbb_2}^{e_{5\_2}L_2} \tag{20}$$



Memory Objects: (basic block)
bb 1, bb 2, bb 3, bb 4, bb 5
In−edge (src−bb−id_dest−bb−id):
1_2, 2_3, 2_4,3_5, 4_5, 5_2
C Codes Framework:

```
for (i=0;i<5;i++) {
  if (... ...)
    ... ...
  else
    ....              }
```

Fig. 8. Example illustrating the ILP formulas for dynamic allocation (the edge weight is marked beside the edge). ILP: integer linear programming.

VI. MULTICORE STATIC MEMORY OBJECTS ALLOCATION

In the static allocation method, the objective formula maximally saves the execution time, while the constraint formula ensures the sum of the size of the selected memory objects does not exceed the sizes of SPMs. With static allocation, un-allocated memory objects for the L1 SPMs of all cores can be considered as candidates for the shared L2 SPM. We can choose the best memory objects among all cores for the L2 SPM. In other words, we use the static allocation to improve the overall performance (i.e., the combined total execution time of all threads) of

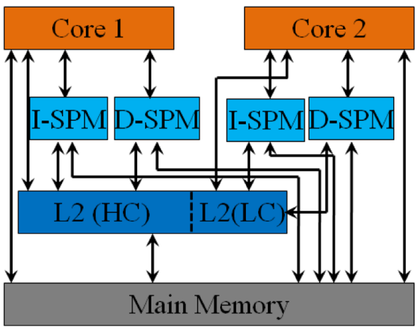


Fig. 9. Static allocation based priority L2 SPM strategy. HC: high priority core, LC: low priority core, SPM: scratch-pad memory.

all cores instead of considering each core separately, while the complexity of static allocation is considerably less than the dynamic allocation. Since the core with more significant hotspots will take more space in the shared L2 cache, this core has higher priority in nature. Actually, the priority strategy is to partition the L2 SPM according to the demand of the program running on each core. Consequently, we call this a static allocation based L2 SPM sharing strategy as the *Static allocation based Priority L2 SPM strategy* (called SPrL2). Fig. 9 demonstrates this strategy. Obviously, besides the priority strategy, the static allocation can also be used as the *Static allocation based Partition L2 SPM strategy* (called SPaL2). This strategy partitions the shared L2 SPM for each core, and uses the static allocation for each core under its L1 SPMs and partitioned L2 SPM.

The static allocation algorithms in our paper can better support programs running on different cores to share memory objects, since the memory objects of programs running on different cores are considered together. The dynamic allocation algorithms are also able to support programs sharing some memory objects, but it may have duplicated copies of the shared memory objects in the SPMs because the ILP equations of different programs are constructed independently for dynamic allocation algorithms.

An example is offered in Fig. 10 to illustrate the basic method of constructing ILP for the static based algorithms, and shows the difference between the SPrL2 and the SPaL2 strategies. We have two programs running on two cores separately. In this example, each program has three basic blocks and two data objects, and they share one data object. Notations used in the static allocation methods are shown in Table 3.

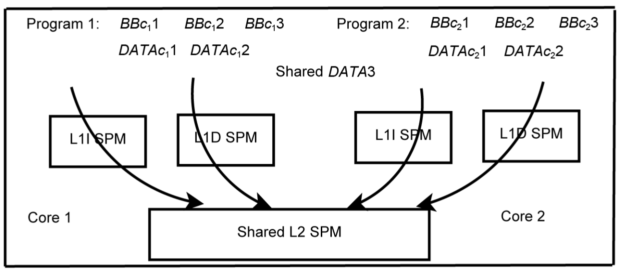


Fig. 10. Example illustrating the static based algorithms. SPM: scratch-pad memory.

Eq. (21) is used for both the SPrL2 and the SPaL2 strategies. For this example, the range of i is 1 to 2. k_1 is set as the $BBc_1 1$, $BBc_1 2$, and $BBc_1 3$ from the program running on core 1, and $BBc_2 1$, $BBc_2 2$, and $BBc_2 3$ from the program running on core 2. Also, k_2 is the set of $DATAc_1 1$ and $DATAc_1 2$ from the program running on core 1, and $DATAc_2 1$ and $DATAc_2 2$ from the program running on core 2. The shared $DATA3$ can only be added once in these equations. In other words, there is no difference for the L1 SPM between the SPrL2 and the SPaL2 strategies. After the ILP equations for the L1 SPM are solved, we obtain the memory objects selected for the L1 SPMs. In detail, Eq. (21a) is the objective function for core 1, and Eq. (21b) is the L1 SPM size constraint of core 1. Also, Eq. (21c) is the objective function for core 2 and Eq. (21d) is the L1 SPM size constraint of core 2.

$$max\left(\sum_{k_1}\{T_{profit}^{BBc_1 k_1}\times MO_{BBc_1 k_1}\}+\sum_{k_2}\{T_{profit}^{DATAc_1 k_2}\times MO_{DATAc_1 k_2}\}\right) \tag{21a}$$

$$\sum_{k_1}\{SIZE_{BBc_1 k_1}\}+\sum_{k_2}\{SIZE_{DATAc_1 k_2}\}<SIZE_{L_1 SPM\ of\ Core 1} \tag{21b}$$

Table 3. Notations used in the ILP variables for the static allocation method

Notation	Explain
MO	The memory object
k_1	Forall memory objects in the set of instruction memory objects
k_2	Forall memory objects in the set of data memory objects
i	Forall cores in the set of cores
j_1	Forall memory objects in the set of instruction memory objects but not selected for the L1 SPM
j_2	Forall memory objects in the set of data memory objects but not selected for the L1 SPM

ILP: integer linear programming, SPM: scratch-pad memory.

$$max\left(\sum_{k_1}\{T_{profit}^{BBc_2k_1}\times MO_{BBc_2k_1}\}+\sum_{k_2}\{T_{profit}^{DATAc_2k_2}\times MO_{DATAc_2k_2}\}\right) \tag{21c}$$

$$\sum_{k_1}\{SIZE_{BBc_2k_1}\}+\sum_{k_2}\{SIZE_{DATAc_2k_2}\}<SIZE_{L_1SPM\ of\ Core2} \tag{21d}$$

Assume BBc_11, BBc_12, BBc_21, and $DATA3$ are allocated to the L1 SPMs. The elements of the j_1 and j_2 set are those of the k_1 and k_2 set, minus these four memory objects selected for the L1 SPMs. The SPrL2 strategy considers all candidates of memory objects for the L2 SPM together, which is shown in Eq. (22). In detail, Eq. (22a) is the objective equation for all cores, and Eq. (22b) is the L2 SPM size constraint for all cores.

$$max\left(\sum_{i}\sum_{j_1}\{T_{profit}^{BBc_ij_1}\times MO_{BBc_ij_1}\}+\sum_{i}\sum_{j_2}\{T_{profit}^{DATAc_ij_2}\times MO_{DATAc_ij_2}\}\right) \tag{22a}$$

$$\sum_{i}\sum_{j_1}\{SIZE_{BBc_ij_1}\}+\sum_{i}\sum_{j_2}\{SIZE_{DATAc_ij_2}\}<SIZE_{L_2SPM} \tag{22b}$$

On the contrary, the SPaL2 strategy addresses the separated cache for each core under the separated size of L2 SPM for each core, which is illustrated in Eq. (23). In detail, Eq. (23a) is the objective equation for core 1, and Eq. (23b) is the separated L2 SPM size constraint for core 1. Also, Eq. (23c) is the objective equation for core 2, and Eq. (23d) is the separated L2 SPM size constraint for core 2.

$$max\left(\sum_{j_1}\{T_{profit}^{BBc_2j_1}\times MO_{BBc_2j_1}\}+\sum_{j_2}\{T_{profit}^{DATAc_2j_2}\times MO_{DATAc_2j_2}\}\right) \tag{23a}$$

$$\sum_{j_1}\{SIZE_{BBc_2j_1}\}+\sum_{j_2}\{SIZE_{DATAc_2j_2}\}<SIZE_{L_1SPM\ Seperated\ for\ Core1} \tag{23b}$$

$$max\left(\sum_{j_1}\{T_{profit}^{BBc_2j_1}\times MO_{BBc_2j_1}\}+\sum_{j_2}\{T_{profit}^{DATAc_2j_2}\times MO_{DATAc_2j_2}\}\right) \tag{23c}$$

$$\sum_{j_1}\{SIZE_{BBc_2j_1}\}+\sum_{j_2}\{SIZE_{DATAc_2j_2}\}<SIZE_{L_2SPM\ Seperated\ for\ Core2} \tag{23d}$$

VII. EVALUATION METHODOLOGY

We study the proposed SPM based architectures under the Trimaran compiler/simulator infrastructure [45], which is extended to evaluate the timing performance of the target VLIW based multicore processor. Meanwhile, the energy performance evaluation is based on our extended EPIC-Explorer, a parameterized VLIW-based platform framework for design space exploration [47].

To comparatively evaluate our proposed SPM based architectures on multicore processors and compare them to the cache based and hybrid architecture targeting real-time systems, we select four real-time benchmarks [49] as the small size benchmark group and four Powerstone benchmarks [50] as the relatively large size benchmark group, the salient characteristics of which are given in Table 4. These benchmarks are divided into four groups to perform the dual-core experiments and two groups to perform the four-core experiments. The detailed grouping information is also shown in Table 4. Both the real-time and Powerstone benchmarks are commonly used in the SPMs related publications [15, 19, 51, 52].

All benchmarks are compiled by the Trimaran compiler under a gcc setting of optimization 0 (i.e., set gcc -O option as 0), closing gcc inline feature, and using Critical Path Scheduling. We set gcc at optimization 0 because if we enable gcc optimization, the size of the basic blocks of the real-time benchmarks is very small. Most hot spots can easily fit the higher level SPM, which will give us a very optimistic evaluation result for the SPM based architecture compared to the cache architecture. Turning off the -O optimization could offer a fair size of memory objects used to perform this evaluation. Also, we attempted several different inputs to determine the worst-case inputs for these benchmarks with variable execution

Table 4. Salient characteristics of selected real-time (upper four benchmarks) and Powerstone benchmarks (latter four benchmarks)

Benchmark	Description	Compute cycles	Static inst. (#)	Global data	Local data	2-Core	4-Core
compress	A demonstration for data compression program	5551	484	1871	0	Group 1	Group 1
lms	An least mean square adaptive signal enhancement	540626	748	1240	0	Group 1	Group 1
ludcmp	Simultaneous linear equations by LU decomposition	3603	362	20816	800	Group 2	Group 1
minver	Matrix inversion for 3x3 floating point matrix	2246	562	448	2000	Group 2	Group 1
adpcm	Adaptive DPCM voice compression algorithm	103690	1929	3028	16	Group 3	Group 2
jpeg	Joint photographic experts group algorithm	1017892	1395	79749	264	Group 4	Group 2
pocsag	An asynchronous protocol used to transmit paper data	65780	1082	1442	0	Group 4	Group 2
pscomp	Power station demonstration of compression algorithm	75963	1723	33533	24	Group 3	Group 2

Table 5. Default configuration of memory size for dual-core simulation on different architectures (the left small value is for the small size benchmark group, while the right large value is for the large size benchmark group)

Architecture	Memory information
One-level SPM based	2048/8192 bytes shared SPM by two cores
Two-level SPM based	512/1024 bytes unified L1 SPM for each core
Two-level with unified L1 SPMs	1024/4096 bytes shared L2 SPM by two cores
Two-level SPM based	256/1024 bytes L1 inst. SPM and 256/1024 bytes data SPM for each core
Two-level with separated L1 SPMs	1024/4096 bytes shared L2 SPM by two cores
Cache based	256/1024 bytes L1 inst. cache and 256/1024 bytes data cache for each core
Cache with separated L1 caches	1024/4096 bytes shared L2 cache by two cores
Hybrid based	(1) 128/512 bytes L1 inst. SPM and 128/512 bytes data SPM for each core 512/1024 bytes shared L2 SPM by two cores (2) 128/512 bytes L1 inst. cache and 128/512 bytes data cache for each core 512/2048 bytes shared L2 cache by two cores

SPM: scratch-pad memory.

times, based on which we obtained the observed WCET. For the simulation based on the dual-core architectures, these eight benchmarks are classified into four groups, while they are categorized into two groups for the four-core simulation, which are shown in Table 4.

For a fair evaluation among the different architectures, by default, we fix the aggregated size of all memories to 2K bytes for the small size benchmark group, and 8K bytes for the large size benchmark group. In different architectures, this number is changed for different memories, as shown in Table 5. For the cache and hybrid based architecture, the L1 cache line size is 16 bytes, and 32 bytes for the L2 cache. Note that the evaluation of small size SPMs is useful since SPMs of such size are the real applications in the embedded systems. For example, the Maxim ultra-high speed flash controller includes the 8051 processor core and 1K SPM [53]. Also, note that both of our small and large size SPM settings are the focus of SPM related publications [2, 15, 19].

In addition, we evaluate caches with associativity varying from direct-mapped, 2-way associative to 4-way associative and fully associative. We choose the best performance cache among these four types in order to ensure a fair comparison to the SPMs. The replacement policy of set-associative caches is least recently used (LRU), and all caches are inclusive.

VIII. EXPERIMENTAL RESULTS

In our experiments, we first focus on the simulation of the dual-core processors to understand the strengths and weaknesses of our different proposed architectures and memory objects allocation methods from the point of view of both WCET and WCEC. Then, we extend some of our experiments on the four-core processors to examine the sensitivity of our designs on further cores. Our timing performance evaluation results consist of computation cycles, an instruction cache stall, and use stall cycles. Also, our energy consumption evaluation results consist of caches/SPMs energy consumption, the main memory energy consumption, and the processor energy consumption.

We notice that, normally, similar evaluation results are observed for WCET and WCEC in the following experimental results. In the WCET evaluation, the dominating difference is due to memory architecture latency. Also, as the WCET becomes longer, the lower level memories (SPMs/caches/main memory) are visited more frequently. The lower level SPM/cache visit consumes more energy than the higher level SPM/cache visit based on our energy consumption model. Thus, normally, as the WCET becomes longer, the WCEC becomes larger.

A. Evaluation among SPM based Architectures on Multicore Processors

Three different SPM based architectures on multicore processors are evaluated in this section. The dynamic based partition L2 SPM strategy is utilized for all of these architectures to guarantee a fair comparison among them. By default, the total L2 SPM space is equally partitioned between both cores.

One-level vs. Two-level: The WCET and WCEC comparison results between the OLS-arch (Fig. 2) and TLS-arch (Fig. 3) for dual-core processors are shown in Figs. 11 and 12, respectively.

We observe the obvious performance improvement on the two-level architecture in terms of both WCET and WCEC. However, we also notice that the performance of the benchmark *jpeg* drops slightly. This is because *jpeg* has a very large and heavily weighted basic block as its

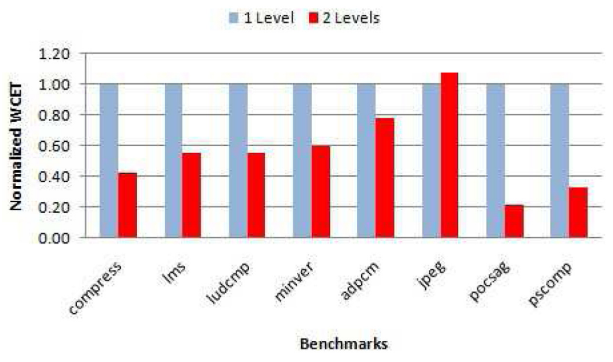


Fig. 11. WCET comparison between the OLS-arch and TLS-arch for dual-core processors (TLS-arch performance is normalized to that of OLS-arch, and the same normalization is performed for all subsequent performance evaluations). WCET: worst-case execution time.

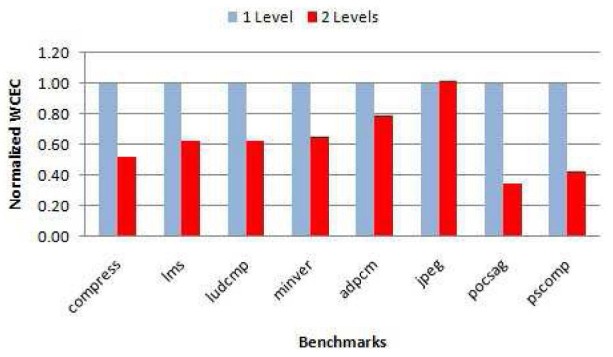


Fig. 12. WCEC comparison between the OLS-arch and TLS-arch for dual-core processors. WCEC: worst-case energy consumption.

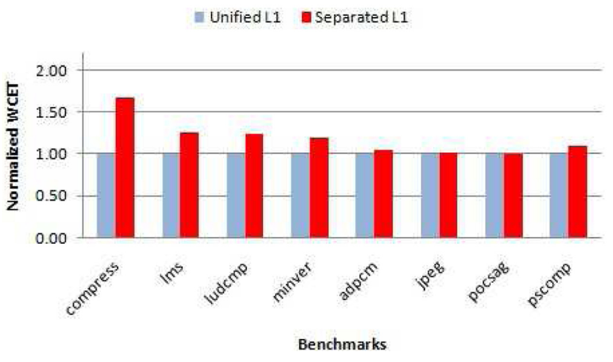


Fig. 13. WCET comparison between the two-level SPM based dual-core architecture with the unified L1 SPM for each core and that with the separated L1 SPMs for each core. WCET: worst-case execution time, SPM: scratch-pad memory.

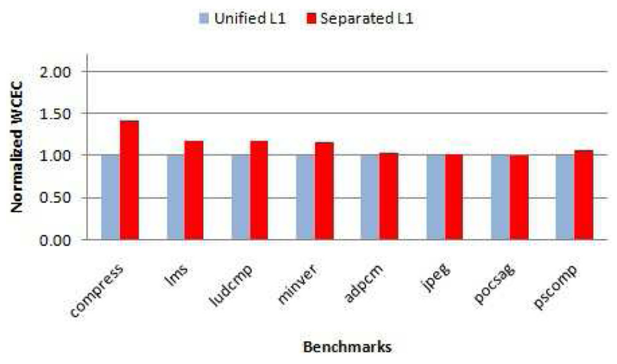


Fig. 14. WCEC comparison between the two-level SPM based dual-core architecture with the unified L1 SPM for each core and that with the separated L1 SPMs for each core. WCEC: worst-case energy consumption, SPM: scratch-pad memory.

hotspot. It cannot be allocated into any SPM in the two-level architecture as its size exceeds that of the SPMs. On the contrary, it can fit into the SPM in the one-level architecture. Accordingly, this type of situation limits the performance improvement for the TLS-arch. In sum, this scenario implies that the sequential focused program (like jpeg), rather than the condition focused program, best fits the one level SPM architecture.

Unified L1 vs. Separated L1: The WCET and WCEC comparison results between the TLS-arch (refer to Fig. 3) and TLSL1S-arch (refer to Fig. 4) are shown in Figs. 13 and 14, respectively.

Interestingly, we observe that the performance of the two-level architecture with the separated L1 SPMs for each core reduces compared to that of the unified L1 SPMs for each core in terms of both WCET and WCEC. Usually, a data object offers less performance boost compared to an instruction object of the same size, if it is allocated to the SPM. This is because the data object usually cannot ensure that all of its elements are visited, while the instruction object is fully visited. Also, the data stall latency is mitigated by the use-stall model so that the benefit of allocating data objects into SPMs may be less than that of instruction objects. However, most Powerstone benchmarks (such as *jpeg* and *pocsag*) can still achieve very close performance on the TLSL1S-arch compared to the performance on the TLS-arch, since they have some very heavily weighted data objects. Therefore, the two-level architecture with the separated L1 SPMs is still a good design choice for data-intensive benchmarks.

B. Evaluation among Different Allocation Strategies to Share the L2 SPM

In this section, we compare the DPaL2 (Section V), SPaL2, and SPrL2 strategies (Section VI) in terms of both WCET and WCEC. For a fair comparison, all of these strategies are utilized on the TLSL1S-arch (Fig. 4).

Dynamic Allocation Based Partition vs. Static Allocation Based Partition L2 SPM Strategy: This experiment evaluates the performance promotion of dynamic allocation. Figs. 15 and 16 present the WCET and WCEC

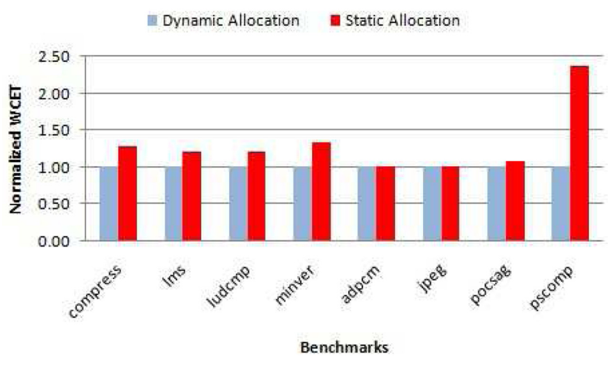


Fig. 15. WCET comparison of each benchmark between the DPaL2 and the SPaL2 strategies. WCET: worst-case execution time.

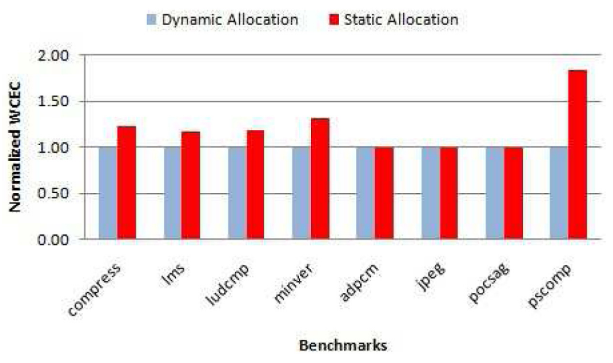


Fig. 16. WCEC comparison of each benchmark between the DPaL2 and the SPaL2 strategies. WCEC: worst-case energy consumption.

comparison results between these two allocation methods, respectively.

We can observe that the performance of dynamic allocation is superior to that of static allocation on all the benchmarks (some benchmarks have a slight performance boost, making them difficult to distinguish from the figures), and the average improvement rate is 31% on our selected benchmarks. As we discussed in Section IV, the dynamic allocation performs liveness analysis on the CFG and permits possible memory objects copying among different memories at the run-time. Consequently, more memory objects have the opportunity to be copied into SPMs through the dynamic allocation, and the better performance of dynamic allocation than the static allocation is then expected. In particular, the *pscomp* benchmark includes a large number of heavily weighted hotspots so that it best fits the dynamic allocation and then takes good performance promotion.

However, this performance improvement is not free. The complexity of the dynamic allocation algorithm is greatly increased, since it needs to consider each edge in the CFG of the program. Table 6 demonstrates the ILP solving time for both the dynamic and static allocation.

Table 6. ILP solving time for the dynamic and static allocation (unit: seconds)

Benchmark	Dynamic allocation	Static allocation
compress	60.71	0.01
lms	45.74	0.02
ludcmp	2.14	0.01
minver	3.63	0.01
adpcm	43.01	0.02
jpeg	3.92	0.01
pocsag	66.13	0.03
pscomp	204.42	0.03

ILP: integer linear programming

The average ILP solving time for the dynamic allocation is 53.71 seconds, while it is only 0.02 seconds for the static allocation. Also, we observe that the average ILP solving time of the large size benchmark group increases to 2.8 times that of the small size benchmark group for the dynamic allocation, while it only increases 1.8 times for the static allocation.

However, more constraints might reduce the size of the solution space for searching. We find that the ILP solver needs more solving time than the static allocation. The key is that the dynamic allocation method needs to obtain a solution for each edge in the CFG of the program, and there might be hundreds of edges. Meanwhile, the static allocation method only needs to find one solution for the whole program. Although one solution is more rapidly solved in the dynamic allocation, hundreds of solutions require significantly more work of the ILP solver.

In addition, Udayakumaran et al. [54] show a good comparison between the ILP and heuristic based methods. The ILP based method offers an optimistic solution, but a significantly large solving time is needed by the ILP solver, and the commercial ILP solver brings an extra cost. Meanwhile, the heuristic based method is more complex, but does not involve any extra solving time or cost. In the past decade, after Udayakumaran's publication, a large number of open source ILP solvers such as Coin_OR [55] and lp_solve [56] have been developed with good performance, and the computing power has greatly improved. Thus, the disadvantages of the ILP based method summarized in Udayakumaran's paper have been significantly alleviated.

Moreover, we can take the benchmark *pscomp* in Fig. 15 as an example to show the impact of the code size. The benchmark *pscomp* achieves the best performance on the dynamic algorithm compared to that of the static allocation, since it has the most spill actions among all benchmarks. Within one specific date set, there are 156 instruction objects spill actions and 36 data objects

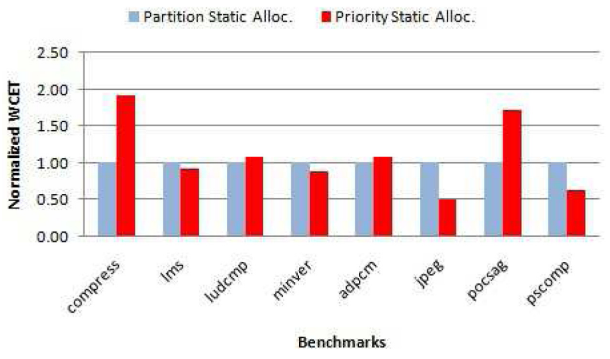


Fig. 17. WCET comparison of each benchmark between the SPaL2 and the SPrL2 strategies. WCET: worst-case execution time.

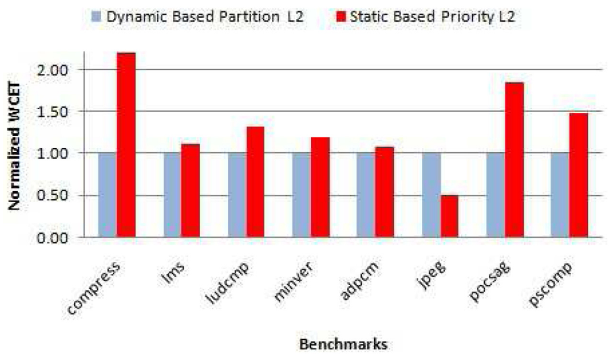


Fig. 19. WCET comparison of each benchmark between the DPaL2 and the SPrL2 strategies. WCET: worst-case execution time.

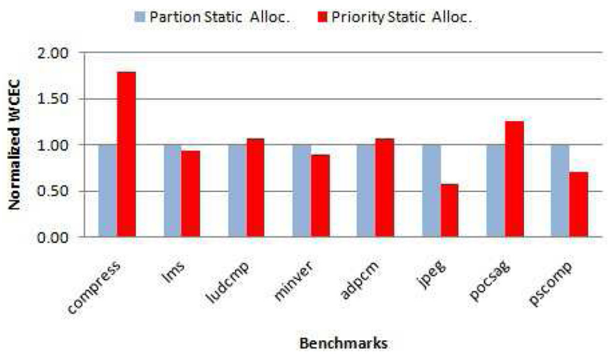


Fig. 18. WCEC comparison of each benchmark between the SPaL2 and the SPrL2 strategies. WCEC: worst-case energy consumption.

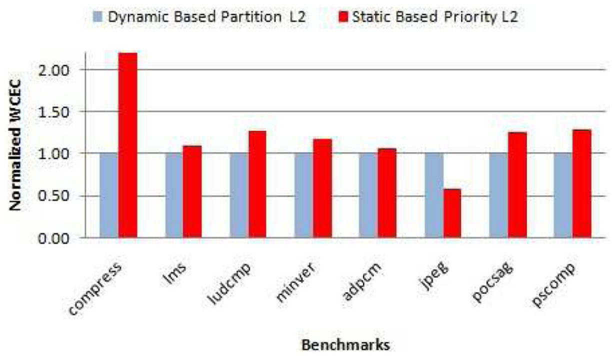


Fig. 20. WCEC comparison of each benchmark between the DPaL2 and the SPrL2 strategies. WCEC: worst-case energy consumption.

spill actions among the edges of CFG. We can assume one instruction is needed to support the spill action on a memory object. Thus, 192 extra instructions are needed. In Table 4, we can see that the benchmark *pscomp* has 1723 static instructions. Consequently, the dynamic algorithm could result in an 11% code size boot. However, if the addresses of some memory objects are consecutive, we do not need 192 extra instructions. Therefore, the actual code size increase rate is greater than 11%. Nevertheless, around a 10% code size change can be accepted based on the improved performance we achieve.

Static Allocation Based Partition vs. Static Allocation Based Priority L2 SPM Strategy: This experiment evaluates the performance change of the SPrL2 strategy. Figs. 17 and 18 present the WCET and WCEC comparison results between these two strategies, respectively.

As we expect, we can observe that one benchmark achieves performance improvement, while its peer benchmark suffers performance loss in each benchmark group. The benchmark with performance boost is the high priority benchmark, which occupies more L2 SPM space. On the contrary, the other benchmark with performance loss is named the low priority benchmark, which takes less L2 SPM space. The static based priority strategy will naturally allow the core with higher demand to take more space in the shared L2 SPM (Section VI).

Dynamic Allocation Based Partition vs. Static Allocation Based Priority L2 SPM Strategy: Figs. 19 and 20 compare the WCET and WCEC results. We can observe that the performance of most benchmarks under the DPaL2 strategy is still better than that under the SPrL2 strategy, except for the *jpeg* benchmark. Generally, the relative performance of the static based priority strategy compared to that of the dynamic based partition strategy is affected by two factors: the performance drop through using the static allocation instead of the dynamic allocation as the SPM space cannot be reused at run-time, and the performance enhancement by occupying more L2 SPM space due to the high priority core. If the second factor dominates the first factor, we can improve performance on the high priority core (such as that of the *jpeg* benchmark), although we use the static allocation. If the first factor dominates the second factor, the dynamic allocation still performs better (as for the other seven benchmarks).

Table 7. Default configuration of memory size for four-core simulation of different architectures

Architecture	Memory information
Two-Level SPM based	256/1024 bytes L1 inst. SPM and 256/1024 bytes data SPM for each core
Two-level with separated L1 SPMs	2048/8192 bytes shared L2 SPM by four cores
Cache based	256/1024 bytes L1 inst. cache and 256/1024 bytes data cache for each core
Cache with separated L1 caches	2048/8192 bytes shared L2 cache by four cores

SPM: scratch-pad memory.

C. Four-Core Experiments

Our four-core experiments focus on comparing the two proposed strategies (i.e., the DPaL2 and the SPrL2 strategies) on the shared L2 SPM, and comparing the performance between SPM based and cache based architecture in terms of both WCET and WCEC. Our purpose is to examine the sensitivity of the larger shared L2 SPM/cache with work load from four cores instead of two cores. The strengths and weaknesses of our different proposed SPM architectures have been fully evaluated in the dual-core experiments. Also, the hybrid architecture is proved to be obviously inferior to SPM and the cache based architecture, so that we do not need to perform repeated experiments on the four-core processor. The default size setting for the SPM and cache based architecture is shown in Table 7. The L1 cache line size is 16 bytes, while that for the L2 cache is 32 bytes. Also, the aggregated size of SPMs is 4K for the small size benchmark group, and 16K for the large size benchmark group.

1) Evaluation between the Dynamic Allocation Based Partition and Static Allocation Based Priority L2 SPM Strategy for Four-core Processors: The evaluation results of each benchmark are shown in Figs. 21 and 22. In benchmark group 1, the *lms* benchmark leads to a significant drop in the performance of all other benchmarks. This is because the hotspots of *lms* still dominate the shared L2 SPM, since the weight of hotspots of other benchmarks is too small compared to that in *lms*. In benchmark group 2, the *jpeg* and *adpcm* benchmarks obtain more space in the shared L2 SPM and achieve performance improvement compared to the SPaL2 strategy, while the other two benchmarks in this group suffer from some performance drop. Therefore, these results indicate the same strengths and weaknesses of these two strategies on dual-core processors (refer to the detailed summary in Section VIII-B).

2) Results on 4-Core Processors: Similar to the experiments on dual-core processors, here we use the DPaL2 strategy on the SPM based architecture. Also, the associativity of cache memories used in the cache based and hybrid architectures is set as direct-mapped, 2-way, 4-way, and fully-associative, respectively. The best performance results between these four associativity settings are chosen to compare with the SPM based architecture in order to be generous to cache memories. In addition, the metric *Weighed Performance* is utilized in the comparison. The comparison results are shown in Figs. 23 and 24. From these figures, we can observe that the weighted performance of cache based architecture compared to the SPM based architecture on four-core processors improves compared to performance on dual-core processors. On the four-core processors, only 62.5% of the benchmarks perform better on the SPM based architecture, while the percentage is 87.5% on dual-core pro-

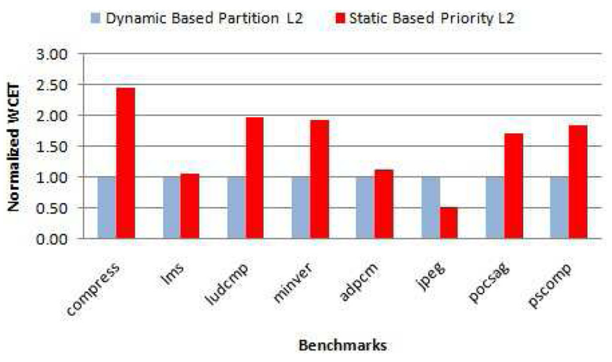


Fig. 21. WCET comparison of each benchmark between the DPaL2 and the SPrL2 strategies on four-core processors. WCET: worst-case execution time.

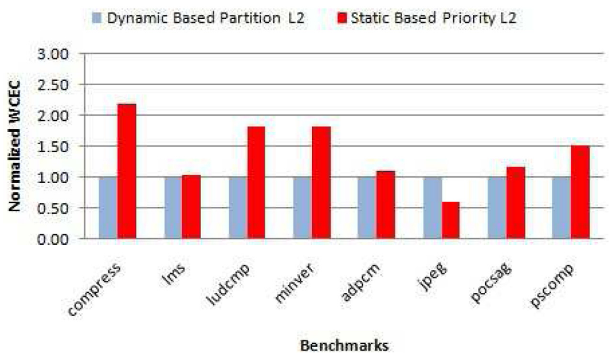


Fig. 22. WCEC comparison of each benchmark between the DPaL2 and the SPrL2 strategies on four-core processors. WCEC: worst-case energy consumption.

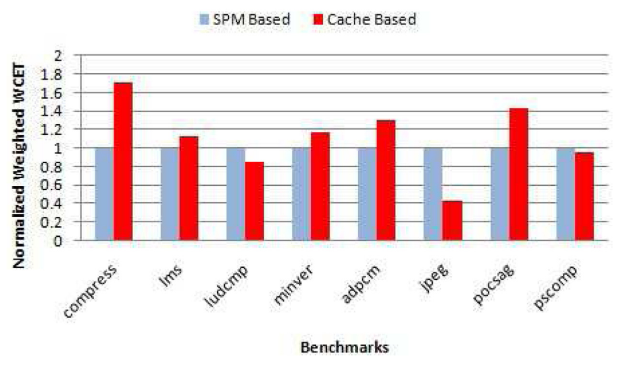


Fig. 23. Weighted WCET comparison between the SPM based and the cache based architectures for four-core processors. WCET: worst-case execution time, SPM: scratch-pad memory.

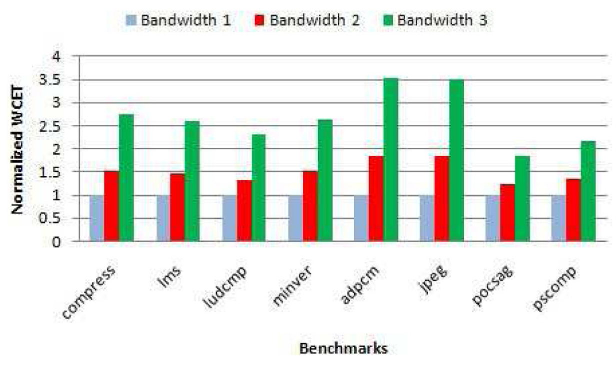


Fig. 25. WCET change of each benchmark under different memory bandwidth settings (bandwidth 1: 1 word, bandwidth 2: 2 words, and bandwidth 3: 4 words). WCET: worst-case execution time.

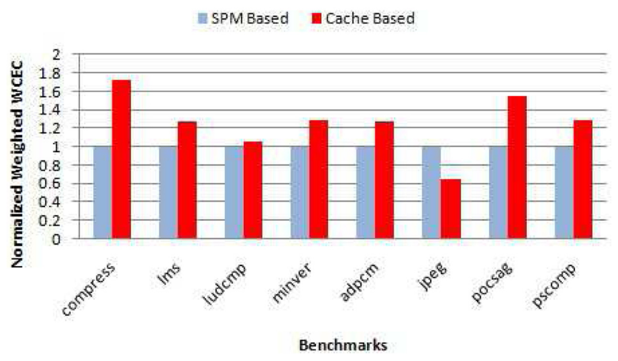


Fig. 24. Weighed WCEC comparison between the SPM based and the cache based architectures for four-core processors. WCEC: worst-case energy consumption, SPM: scratch-pad memory.

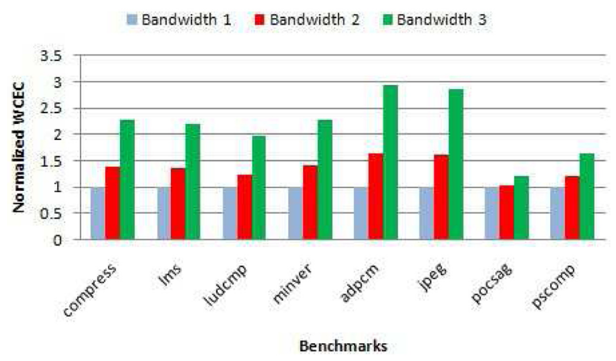


Fig. 26. WCEC change of each benchmark under different memory bandwidth settings (bandwidth 1: 1 word, bandwidth 2: 2 words, and bandwidth 3: 4 words). WCEC: worst-case energy consumption.

cessors. This is because we double the size of L2 memory, accomplishing double the number of cores. The larger L2 cache has more cache sets, which mitigates the interference among cores and then increases the cache performance.

3) Evaluation the Sensitivity to the Bus Bandwidth Change: As aforementioned in Section III-B, we design the experiment in this section to evaluate the performance influence of the memory bandwidth change. The increasing memory bandwidth should offer better performance for both caches and SPMs; however, the objective of this experiment is to evaluate the relative performance boost we can achieve on SPMs compared to caches under different bandwidths. Therefore, we fix the latency of caches under different bandwidths to calculate the relative latency of SPMs. We set the memory bandwidth as 1 word, 2 words, and 4 words, respectively, and still use Eqs. (1)–(3) to compute the latency of SPM operations based on the same cache latency setting. Consequently, the latency of accessing SPMs increases as the memory bandwidth increases. Figs. 25 and 26 show the WCET and WCEC change, respectively, under these three memory bandwidth settings, based on the TLSL1S-arch, which are normalized to the WCET and WCEC with 1 word memory bandwidth, respectively. Clearly, we observe that both the WCET and WCEC increase with the increase of the memory bandwidth. These results indicate that with larger memory bandwidth, the SPM's performance and energy advantages over caches are reduced.

IX. CONCLUSIONS

Traditional cache-based multicore processors can provide high performance, but are harmful to time predictability. In this paper, we studied three time-predictable SPM based architectures to provide time predictability with good performance for multicore processors. We explore multi-level SPM hierarchies and propose the DPaL2, SPaL2, and SPrL2 strategies for the L2 SPM shared by all cores. Our experimental results indicate the strengths and weaknesses of each architecture and the associated allocation strategy as follows.

First, the performance of the TLS-arch is superior to that of the OLS-arch, while the latter is simpler in design and implementation. The strength of the TLS-arch is that it provides better performance on the real-time applications without excessive data accessing, while the weakness is that the instruction and data cannot be accessed concurrently based on this architecture. On the contrary, the TLSL1S-arch better fits data-intensive real-time applications, which not only maintains good performance but also achieves a higher bandwidth by accessing both instruction and data SPMs at the same time.

Second, the DPaL2 strategy achieves the best performance on each core because of its flexibility of memory object allocation. However, the increased complexity and computation time may limit its applicability to large applications. In contrast, the strength of the static allocation based approaches is their low complexity, but the performance is usually less than that of the dynamic allocation.

In addition, our evaluation indicates that among the SPM based, cache based, and hybrid architectures, the SPM based architecture performs the best in terms of both WCET and WCEC, while attaining time predictability. We believe that the proposed SPM based architectures can provide interesting memory design options to enable real-time multicore computing.

In our future work, we plan to include more cache policies such as an exclusive cache to perform the evaluation. Also, we would like to extend our analysis and evaluation to include the timing effects of the shared bus. Moreover, we plan to extend our research framework to evaluate the scalability and effectiveness of the proposed SPM architectures and allocation strategies for larger benchmarks and larger SPMs, particularly targeting their use in smartphones and tablets, etc.

ACKNOWLEDGMENTS

This work is partly supported by the National Science Foundation, under grant CCF 1063645.

REFERENCES

1. R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, Estes Park, CO, 2002, pp. 73-78.
2. L. Wehmeyer and P. Marwedel, "Influence of onchip scratchpad memories on WCET prediction," in *Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Catania, Italy, 2004, pp. 1-4.
3. S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Paris, 2002, pp. 409-415.
4. O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratchpad-based embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, pp. 6-26, 2002.
5. Y. Liu and W. Zhang, "Exploiting time predictable two-level scratchpad memory for real-time systems," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, Tai-Chung, Taiwan, 2011, pp. 395-396.
6. Freescale Semiconductor Inc., MC2114 MC2113 MC2112 Advance Information, http://www.datasheetarchive.com/dl/Datasheet-02/DSA0023664.pdf.
7. ARM Inc., ARM7TDMI technical reference manual, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf.
8. IBM Systems and Technology Group, Cell architecture, http://moss.csc.ncsu.edu/~mueller/cluster/ps3/workshop/Day1_03_CourseCode_L1T1H1-10_CellArchitecture.pdf.
9. Tile64 Processor Product Brief, file:///C:/Users/KIM/Downloads/1187632077329ProBrief_Tile64_Web.pdf.
10. NVIDIA Fermi Compute Architecture, http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_ whitepaper.pdf.
11. J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching scheme for scratchpad memory," in *Proceedings of the 48th Design Automation Conference* (DAC), San Diego, CA, 2011, 960-965.
12. T. Chen, T. Zhang, Z. Sura, and M. Tallada, "Prefetching irregular references for software cache on cell," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (*CGO), Boston, MA, 2008, pp. 155-164.
13. Y. Liu and W. Zhang, "Exploiting multi-level scratchpad memories for time-predictable multicore computing," in *Proceedings of the IEEE 30th International Conference on Computer Design (ICCD)*, Montreal, Canada, 2012, pp. 61-66.
14. R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Proceedings of the 12th IEEE Real-Time Technology and Applications Symposium*, San Jose, CA, 2006, pp. 81-90.
15. J. F. Deverge and I. Puaut, "WCET-directed dynamic scratchpad memory allocation of data," in *Proceedings of 19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, Pisa, Italy, 2007, pp. 179-190.
16. M. Verma and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 802-815, 2006.
17. R. A. Ravindran, P. D. Nagarkar, G.S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown, "Compiler managed dynamic instruction placement in a low-power code cache," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, 2005, pp. 179-190.
18. L. Li, L. Gao, and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," in *Proceedings of 14th International Conference on Parallel Architec-*

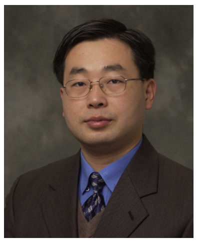
tures and Compilation Techniques (PACT'05), St. Louis, MO, 2005, pp. 329-338.

19. S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer, "Predictable dynamic instruction scratchpad for simultaneous multithreaded processors," in *Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA'08)*, Toronto, Canada, 2008, pp. 38-45.
20. H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic data scratchpad memory management for a memory subsystem with an MMU," in *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, CA, 2007, pp. 195-206.
21. J. Whitham and N. Audsley, "The scratchpad memory management unit for microblaze, implementation, testing and case study," University of York, Technical Report YCS-2009-439, 2009.
22. M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: a first approach," in *Proceedings of 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTMEDIA)*, New York, NY, 2005, pp. 115-120.
23. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al., "The worst-case execution-time problem: overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, article no. 36, 2008.
24. R. Arnold, F. Mueller, D. Whalley, and M. Harmon, "Bounding worst-case instruction cache performance," in *Proceedings of Real-Time Systems Symposium*, San Juan, Puerto Rico, 1994, pp. 172-181.
25. Y. T. S. Li, and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCTES)*, La Jolla, CA, 1995, pp. 88-98.
26. Y. T. S. Li, S. Malik, and A. Wolfe, "Cache modeling and path analysis for modern hardware architectures," in *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Los Alamitos, CA, 1996, pp. 254-263.
27. M. Alt, F. Christian, M. Florian, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," in *Proceedings of the Static Analysis Symposium (SAS'96)*, Aachen, Germany, 1996, pp. 52-66.
28. F. Sebek and J. Gustafsson, "Determining the worst-case instruction cache miss-ratio," in *Proceedings of Workshop on Embedded System Codesign (ESCODES'02)*, San Jose, CA, 2002, pp. 1-6.
29. Y. Liu and W. Zhang, "Stack distance based worst-case instruction cache performance analysis," in *Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC'11)*, Taichung, Taiwan, 2011, pp. 723-728.
30. Y. Liu and W. Zhang, "Bounding worst-case data cache performance by using stack distance," *Journal of Computing Science and Engineering*, vol. 3, no. 4, pp. 195-215, 2009.
31. J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, St. Louis, MO, 2008, pp. 80-89.
32. J. Yan and W. Zhang, "Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, Beijing, China, 2009, pp. 455-463.
33. Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multicores," in *Proceedings of 30th IEEE Real-time System Symposium (RTSS'09)*, Washington, DC, 2009, pp. 57-67.
34. M. Lv, W. Yi, N. Guan, and G. Yu,0" Combining abstract interpretation with model checking for timing analysis of multicore software," in *Proceedings of 31st IEEE International Real-time System Symposium (RTSS)*, San Diego, CA, 2010, pp. 339-349.
35. X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, 2003, pp. 272-282.
36. M. Paolieri, E. Quinones, and F. Cazorla, "Hardware support for WCET analysis of hard real-time multicore systems," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, 2009, pp. 57-68.
37. S. Plazar, J. Kleinsorge, P. Marwedel, and H. Falk, "WCET-aware static locking of instruction caches," in *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*, San Jose, CA, 2012, pp. 44-52.
38. J. Yan, W. Zhang, and Y. Liu, "Time-predictable and high-performance cache architectures for multi-core processors," in *Proceedings of the WiP Session of the 30th IEEE Real-Time Systems Symposium (RTSS'09*), Washington, DC, 2009, pp. 9-12.
39. C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, "Predictability considerations in the design of multi-core embedded systems," in *Proceedings of Embedded Real Time Software and Systems*, Toulouse, France, 2010.
40. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS'05)*, Miami, FL, 2005.
41. P. Marwedel, L. Wehmeyer, M. Verma, S. Steinke, and U. Helmig, "Fast, predictable and low energy memory references through architecture-aware compilation," in *Proceedings of Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2004, pp. 4-11.
42. M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu, "Compiler-directed scratch pad memory optimization for embedded multiprocessors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 3, pp. 281-287, 2004.
43. R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel, "Comparison of cache-and scratch pad based memory systems with respect to performance, area and

energy consumption," University of Dortmund, Technical Report No. 762, 2001.
44. M. Zahran, K. Albayraktaroglu, and M. Franklin, "Non-inclusion property in multi-level caches revisited," *International Journal of Computers and Their Applications*, vol. 14, no. 2, pp. 1-10, 2007.
45. Trimaran, http://www.Trimaran.org.
46. M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low power cache," in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED),* Monterey, CA, 1997, pp. 143-148.
47. G. Ascia, V. Catania, M. Palesi, and D. Patti, "EPIC-explorer: a parameterized VLIW-based platform framework for design space exploration," in *Proceedings of the 1st Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)*, Newport Beach, CA, 2003, pp. 65-72.
48. CACTI, http://www.cacti.net/.
49. Mälardalen Real-Time Research Center, "Real-time benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
50. Powerstone Benchmarks, http://www.cse.iitd.ac.in/~asahu/cs718/BenchMarks/others/PowerStone/pocsag.c.
51. O. Golubeva, M. Loghi, M. Poncino, and E. Macii, "Architectural leakage-aware management of partitioned scratchpad memories," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Nice Acropolis, France, 2007, pp. 1665-1670.
52. M. I. Aouad, R. Schott, and O. Zendra, "A tabu search heuristic for scratch-pad memory management," in *Proceedings of the International Conference on Software Engineering and Technology (ICSET'10),* Kandy, Sri Lanka, 2010, pp. 386-390).
53. Maxim Integrated Product Inc., Ultra-high-speed flash controller user guide, http://www.maximintegrated.com/en/app-notes/index.mvp/id/4833.
54. S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratchpad memory using compile time decisions," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, no. 2, pp. 472-511, 2006.
55. Coin_OR, https://projects.coin-or.org/Cbc.
56. lp_solve, http://lpsolve.sourceforge.net/5.5/.

Yu Liu

Yu Liu is currently a research scientist at the Canadian Nuclear Laboratories (formerly Atomic Energy of Canada Ltd.). He received his B.S. and M.S. degrees from Sichuan University, China in 2000 and 2003, respectively, and his Ph.D. degree from Southern Illinois University, Carbondale in 2011. He was employed at Motorola from 2003 through 2007, and IBM from 2011 through 2013. He also worked on two summer research projects at the Pacific Northwest National Lab, U.S. Department of Energy in 2009 and 2010, respectively. His research interests include real-time systems, wireless sensor networks, cyber-physical systems, and high performance computing.



Wei Zhang

Wei Zhang is a professor at the Department of Electrical and Computer Engineering at Virginia Commonwealth University, USA. Dr. Wei Zhang received his Ph.D. from Pennsylvania State University in 2003. From August 2003 to July 2010, Dr. Zhang worked as an assistant professor and then as an associate professor at Southern Illinois University Carbondale. His research interests are in embedded and real-time computing systems, computer architecture, and compiler and low-power systems. Dr. Zhang received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. Dr. Zhang has received 5 research grants from the National Science Foundation. In addition, his research and educational efforts have been supported by industry including leading IT companies such as IBM, Intel, Motorola, and Altera. Dr. Zhang has published more than 120 papers in refereed journals and conference proceedings. He is a senior member of the IEEE, and an associate editor of Journal of Computing Science and Engineering. He served as a member of organizing or program committees for several IEEE/ACM international conferences and workshops.