JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 8, No. 4, December 2014, pp. 215-227

Two-Level Scratchpad Memory Architectures to Achieve Time Predictability and High Performance

Yu Liu and Wei Zhang*
Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
yuliu@siu.edu, wzhang4@vcu.edu

Abstract
In modern computer architectures, caches are widely used to shorten the gap between processor speed and memory access time. However, caches are time-unpredictable, and thus can significantly increase the complexity of worst-case execution time (WCET) analysis, which is crucial for real-time systems. This paper proposes a time-predictable two-level scratchpad-based architecture and an ILP-based static memory objects assignment algorithm to support real-time computing. Moreover, to exploit the load/store latencies that are known statically in this architecture, we study a Scratchpad Sensitive Scheduling method to further improve the performance. Our experimental results indicate that the performance and energy consumption of the two-level scratchpad-based architecture are superior to the similar cache based architecture for most of the benchmarks we studied.

Category: Embedded computing

Keywords: Hard real-time systems; Scratch-pad memory; Time predictability

I. INTRODUCTION

Worst-case execution time (WCET) is crucial for hard real-time systems, such as aircraft and automobile control software. Missing deadlines in such systems may result in putting human lives in danger or other catastrophic outcomes. Many architectural features of modern microprocessors, such as caches, pipelines, and dynamic branch prediction, however, have been designed to boost the average-case performance. Unfortunately, these advanced architecture features are often harmful to time-predictability, because their timing is dependent on the program's historic dynamic behavior. In particular, a cache-based memory structure is used to shorten the gap between processor speed and memory access time. Although there has been some progress in timing analysis for caches, generally, it is extremely hard and complex to accurately obtain the WCET for processors with cache memories, especially for data caches, unified caches, multi-level caches, and shared caches in multi-core/multi-threaded processors.

Scratchpad memories (SPMs), as an alternative technique to hardware-controlled caches, offer the characteristic of time-predictability and reasonable performance [1-3]. SPMs are small, physically separate memories directly mapped into the address space of a memory system, which always use high-speed SRAM. Compared with caches of the same capacity, SPMs generally are more area- and energy-efficient because SPMs do not need to use tag arrays.

To efficiently exploit SPMs, however, it is crucial to determine the memory objects assignment to SPMs. Early

Open Access http://dx.doi.org/10.5626/JCSE.2014.8.4.215 http://jcse.kiise.org

Received 1 July 2014; **Revised** 13 September 2014; **Accepted** 20 November 2014
*Corresponding Author

works on this topic focused mostly on studying compilation time algorithms to statically allocate hot spots of programs and/or data into the SPMs, where the objective was either to save energy consumption or access time to the greatest extent possible [2, 3]. To further improve the performance, researchers have also studied how to dynamically allocate the memory objects, including software-managed [4] and hardware-assisted replacement methods [5]. However, these dynamic allocation approaches may harm the time-predictability, because of their dynamic replacement of memory objects. In contrast, static algorithms are very friendly to WCET estimations, because it is fully predictable which memory will be used for a certain memory access [1].

Banakar et al. [6] described a comprehensive evaluation between scratchpad and cache memories in their research. However, their work focused on a single-level shared scratchpad and cache memory, which is not a representative architecture used in modern high-performance microprocessors. To our knowledge, no two-level SPM based architecture with separated L1 instruction and data SPMs and a unified L2 SPM to support both instructions and data has been thoroughly studied and evaluated. Because two-level cache architectures have been used widely in modern processors to boost performance, it is worthwhile to exploit the SPM-based two-level architecture and compare it with the similar two-level caches to understand their pros and cons in terms of both performance and energy efficiency.

However, two-level SPMs or multi-level SPMs generally can bring new challenges to SPM allocation. First, because different levels of SPMs have various access latencies, it is important to make intelligent decisions for placing data/instructions into the appropriate level of SPM to achieve the best performance. Second, because the L2 SPM is shared by both data and instructions, just like a unified L2 cache, the SPM allocation algorithm must decide whether it is profitable to place instructions or data into the L2 SPM and how much SPM space should be used for data or instructions. Third, unlike multi-level caches that are typically inclusive, two-level SPMs are exclusive. This brings the question as to whether it is worth designing a two-level SPM architecture. For example, can we simply use a larger L1 SPM. This paper will explore the design space to answer these questions.

Additionally, to achieve time-predictability, this paper chooses a statically-scheduled processor based on very long instruction word (VLIW) processor architecture, although the two-level SPM architecture can also be applied to other architectural styles. VLIW microprocessors execute instructions in parallel, based on a fixed schedule determined by the compiler. Consequently, both the integer linear programming (ILP)-based memory objects allocation method and the SPM-sensitive scheduling discussed in this paper are implemented in the backend of the compiler.

Our contributions in this paper are thus two-fold. First, we propose a two-level architecture with small, separate L1 instruction and data SPMs and a larger shared (i.e., unified) L2 SPM. An ILP-based static memory objects assignment algorithm is used for the proposed architecture so as not to harm the characteristic of time predictability of SPMs. Also, both the performance and energy consumption of our two-level SPM based architecture are completely evaluated and compared with those of the traditional two-level cache based architecture. Second, we study a novel Scratchpad Sensitive Scheduling method to further improve the performance of the proposed architecture without compromising time predictability, which exploits the statically known load/store latencies due to the use of SPMs. Our experimental results indicate that both the observed WCET and energy consumption of the two-level SPM based architecture are superior to those of the cache based architecture with the same size.

We have undertaken comprehensive research work on SPM-based architecture performance evaluation and optimization. We have published several papers based on our work, and each has a specific focus. Specifically, we have a paper [7] focusing on the comparison of the static and dynamic memory object-allocation method on different SPM-based multi-core architectures. This paper focuses mainly on introducing the design of two-level SPM-based architectures and SPM Sensitive Scheduling, which use the advantages of SPM.

II. TWO-LEVEL SCRATCHPAD BASED ARCHITECTURE

The scratchpad is a memory array with decoding and column circuitry logic, and we need not check for the availability of data/instruction in the scratchpad [8]. Thus, SPMs have the essential characteristics of time-predictability and low energy consumption. Most modern high-performance computer systems offer two-level cache based architectures, as shown in Fig. 1. The level 1 cache is the fastest form of memory, which is built into the chip but is limited in size. Also, two separate L1 caches are normally embedded into the processor to store either instructions or data, which isolates possible interference between them in an otherwise unified cache. The level 2 cache is slower than the L1 cache but is larger in size, trading off between speed and size, which is often shared by both instructions and data. The level 2 cache is important to mitigate the L1 cache miss latency, which otherwise has to fetch from memory for every L1 miss.

To combine the advantages of both SPMs and two-level architectures for balancing access latency and capacity, we propose a two-level SPM-based architecture for real-time systems, as shown in Fig. 2. Similarly, the L1 SPM

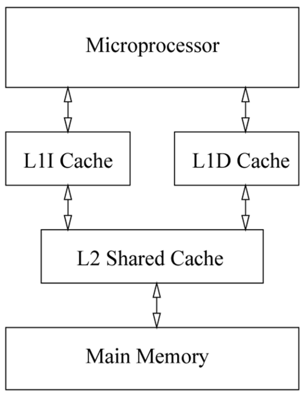


Fig. 1. The architecture of the two-level cache-based memory system.

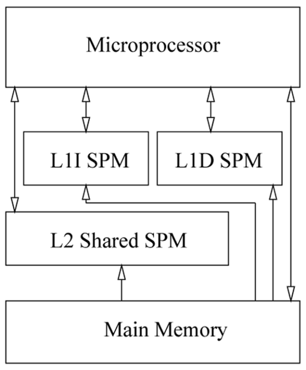


Fig. 2. The architecture of the two-level SPM-based memory system. SPM: scratchpad memory.

is the fastest SRAM with a small size, while the L2 SPM is a slower SRAM with a larger size. Because there is no replacement requirement in any higher level memory of our architecture, the L1 instruction, L1 data, L2 shared scratchpad, and the main memory are all connected directly to the microprocessor.

Such an SPM-based architecture is clearly time-predictable. Compared with a single-level SPM-based architecture, the added L2 SPM is still time-deterministic. Although the data access latency can vary, depending on whether it is in the L1 SPM, the L2 SPM, or the main memory, we plan to develop a static 2-level SPM allocation so that we can statically determine which data is stored at which level, thus making the execution time statically predictable.

In addition to time predictability, the proposed two-level SPM based architecture can also potentially attain better performance. First, because the SPM does not have a tag array, it can be faster to access the SPM than a cache. Moreover, because the access latency in the SPM-based memory architecture is known statically, this makes it possible to use the compiler to optimize the data access order so that the total execution time can be reduced in a time-predictable manner. Additionally, the use of the L2 SPM can dramatically reduce accesses to the main memory, which can further increase the performance. There is no hierarchical path between the L1 and L2 SPM in Figs. 1 and 2. The reason is that we focus on the static-based allocation method in this paper. Consequently, we do not need to transfer memory objects between the L1 and L2 SPM. That is why we do not present a path between the L1 and L2 SPM in the figures. In [7], we investigated the dynamic-based method, which requests memory object transferring between different level SPMs. Thus, the path between the L1 and L2 SPM is used in [7].

III. MEMORY OBJECT ASSIGNMENT

In this paper, we use a static algorithm to allocate memory objects to SPMs, because the static method can keep the characteristic of time-predictability, which is the most important design objective of a real-time system. An ILP-based method is used to allocate the hot spots to the three SPMs in our architecture, and the details are introduced in the following subsections. Additionally, as mentioned, the two-level SPM-based architecture is based on the VLIW architecture. VLIW microprocessors execute instructions in parallel, based on a fixed schedule determined by the compiler. So, both the memory object assignment and scheduling are implemented in the back-end of the compiler and executed in the compiling stage rather than the running stage. Thus, none of them has any impact on the execution time. Furthermore, the run-time of the memory allocation spent in the compiling stage mostly depends on the ILP solver rather than the memory allocation method itself.

A. Memory Objects

The memory objects covered in our work include instruction objects and data objects. First, the instruction objects consist of basic blocks, functions, and combinations of consecutive basic blocks. A basic block B_i is the basic unit of instruction objects in the control flow graph (CFG), and the smallest one used in our ILP-based algorithm. The function F_i is another instruction object considered in our work. Also, to minimize jump instructions, moving consecutive basic blocks is preferred [3]. Thus, the combination of consecutive basic blocks C_i is used in our algorithm. For example, the function F_0 has three basic blocks B_0, B_1, and B_2. So, we have two combinations of consecutive basic blocks, including C_0 (B_0 and B_1), and C_1 (B_1 and B_2). Second, our data objects consist of global scalars and non-scalar variables [3], which are represented as D_i. Note that the stack data is not considered in the memory objects, because generally the global scalars and non-scalar variables dominate the data in a real-time system.

However, the ILP-based static algorithm can be easily extended to handle the stack data following some recent research work on stack data allocation [2, 4, 9].

It should be noted that the granularity of memory objects is important to the performance. If the size of a memory object is larger than the size of SPMs, it will be allocated to the main memory rather than SPMs, and then the performance, like WCET, will surely be worse. To address this, a memory object can be broken into smaller parts, and they may fit the SPMs. Thus, we have multiple types of memory objects including the basic blocks, functions and combinations of consecutive basic blocks. For example, if a whole function can fit into SPMs, it is the best outcome. If we cannot make it because of the size, we can partially allocate some basic blocks of this function into SPMs so as not to jeopardize performance.

B. Assignment Algorithm

In our scratchpad-based architecture, we have three SPMs: the L1 instruction, L1 data, and L2 SPMs. Accordingly, we need to use the ILP-based static allocation algorithm for each of them.

- L1 Instruction SPM: All instruction objects mentioned above need to be considered as candidates for this SPM. The objective of setting ILP is to select the most significant instruction hotspots into this L1 SPM to maximally save execution time. Eq. (1) is the objective formula. In this equation, $S(x_i)$ is the size of instruction objects, and $W(x_i)$ is the weight of instruction objects, where x_i can be B_i, F_i, and C_i. That is, $W(x_i)$ means the number of references to any memory object. Also, B_i, F_i, and C_i can be either 1 or 0, where 1 means it is allocated to the L1 instruction SPM, and *vice versa*. T_{l1i} is the time saved by assigning a single byte of instruction to this scratchpad.
  We have two constraints in the ILP formulas. The first is Eq. (2), which ensures that the sum of the size of all selected objects does not exceed the size of this scratchpad S_{l1i}. The second one is Eq. (3), which prevents a basic block from being selected more than once, because one basic block can occur in B_i, F_i, and several C_i.

$$max\sum_{i=1}^{n}\{B_i\times W(B_i)\times S(B_i)\times T_{l1i}\} +\sum_{i=1}^{n}\{F_i\times W(F_i)\times S(F_i)\times T_{l1i}\} +\sum_{i=1}^{n}\{C_i\times W(C_i)\times S(C_i)\times T_{l1i}\} \tag{1}$$

$$\sum_{i=1}^{n}\{B_i\times S(B_i)\}+\sum_{i=1}^{n}\{F_i\times S(F_i)\} +\sum_{i=1}^{n}\{C_i\times S(C_i)\}\le S_{l1i} \tag{2}$$

$$B_i+F_i+\sum_{i=1}^{j}\{C_i\}\le 1 \tag{3}$$

- L1 Data SPM: All data objects mentioned above need to be considered as candidates for this scratchpad. The objective of setting ILP is to select the most significant data hotspots into this L1 data SPM to minimize execution time. Eq. (4) is the objective formula, where $S_t(D_i)$ is the size of data type (data type determines the size of each memory operation of data object, such as char, int, float, and double), $W(D_i)$ is the weight of memory operations of data objects, $S(D_i)$ is the size of data objects, and T_{l1d} is the time saving from assigning a single byte of data to this scratchpad. Also, D_i can be either 1 or 0, where 1 means it is allocated to the L1 data SPM, and *vice versa*. The constraint in the ILP formulas is Eq. (5), which ensures the sum of the size of all selected objects will not exceed the size of this scratchpad S_{l1d}.

$$max\sum_{i=1}^{n}\{D_i\times W(D_i)\times S_t(D_i)\times T_{l1d}\} \tag{4}$$

$$\sum_{i=1}^{n}\{D_i\times S(D_i)\}\le S_{l1d} \tag{5}$$

- L2 Shared SPM: All instruction and data objects not selected to be allocated in L1 scratchpad need to be considered as candidates for the L2 SPM. The objective of setting ILP is to select the most significant instruction/data hotspots in the remaining candidates into the L2 scratchpad to reduce execution time as much as possible. Eq. (6) is the objective formula. In this equation, $S(x_i)$ is the size of instruction/data objects; $W(x_i)$ is the weight of instruction/data objects, where x_i can be B_i, F_i, C_i, and D_i; and $S_t(D_i)$ is the size of data type. Also, B_i, F_i, C_i, and D_i can be 1 or 0, where 1 means it is allocated to the L2 shared SPM and 0 indicates it is not. T_{l2} is the time saved from assigning a single byte of instruction/data to this scratchpad.
  Similarly, we have two constraints in the ILP formulas. The first is Eq. (7), which ensures that the sum of the size of all selected objects will not exceed the size of this scratchpad S_{l2}. The second one is Eq. (8), which prevents a basic block from being selected more than once, because one basic block can occur in B_i, F_i, and several C_i.

$$max\sum_{i=1}^{m}\{B_i\times W(B_i)\times S(B_i)\times T_{l2}\} +\sum_{i=1}^{m}\{F_i\times W(F_i)\times S(F_i)\times T_{l2}\} +\sum_{i=1}^{m}\{C_i\times W(C_i)\times S(C_i)\times T_{l2}\} +\sum_{i=1}^{m}\{D_i\times W(D_i)\times S_t(D_i)\times T_{l2}\} \tag{6}$$

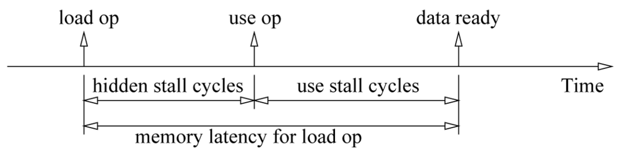


Fig. 3. The relationship between hidden stall cycles, use stall cycles and memory latency for load op in the stall-on-use model.

$$\sum_{i=1}^{m}\{B_i \times S(B_i)\}+\sum_{i=1}^{m}\{F_i \times S(F_i)\} + \sum_{i=1}^{m}\{C_i \times S(C_i)\}+\sum_{i=1}^{m}\{D_i \times S(D_i)\} \leq S_{l2} \tag{7}$$

$$B_i + F_i + \sum_{i=1}^{j}\{C_i\} \leq 1 \tag{8}$$

IV. SCRATCHPAD-SENSITIVE SCHEDULING

In this section, to exploit the load/store latencies that are known statically in our two-level SPM-based memory architecture, we studied a Scratchpad-Sensitive Scheduling method to improve the performance without compromising time predictability by optimizing the *Load-To-Use Distance* (see Section IV-A for details).

A. Background of Load Sensitive Scheduling

Dynamically-scheduled processors such as out-of-order superscalar processors are known to have a timing anomaly [10], which can significantly complicate WCET analysis and hurt time predictability. To achieve time-predictability, this paper chooses a statically scheduled processor based on VLIW processor architecture, although the two-level SPM architecture can also be applied to other architectural styles. VLIW microprocessors execute instructions in parallel based on a fixed schedule determined by the compiler. Consequently, the instruction scheduling in the compiler stage is very important to determine the execution time of a program on the VLIW microprocessor. When an instruction scheduler becomes more aware of the latencies of load instruction and uses such information during scheduling, then we say it is a *Load-Sensitive Scheduler* [11].

In a cache-based architecture, it is generally very hard and complex to determine statically the latency of each load instruction because the operation of the cache is essentially dynamic. To deal with the lack of perfect knowledge of load latencies, an *Optimistic Scheduler* assumes that a load instruction always hits in the cache, while a *Pessimistic Scheduler* assumes that a load instruction always misses in the cache. While the *Pessimistic Scheduler* is always safe to schedule instructions that depend on the loads, it may lead to poor performance because many loads may actually hit in the cache at run-time. In contrast, the *Optimistic Scheduler* may schedule instructions too aggressively, thus special hardware needs to be used to enforce that whenever a load instruction actually misses in the cache, the processor is stalled until the load returns to ensure the correctness of execution.

We call a load instruction as *Load Op*, which loads data from memories. Also, we call an instruction that uses the data loaded by a Load Op a *Use Op*. The number of execution cycles scheduled between the Load Op and Use Op is called the *Load-To-Use Distance* in this paper. Also, the *Stall-On-Use* model is commonly used to handle data cache misses in VLIW, in which the microprocessor will not be stalled until the Use Op is executed. Obviously, the *Stall-On-Use* model may reduce the *Use Stall Cycles* based on the *Load-To-Use Distance*, and the reduced stall cycles are called the *Hidden Stall Cycles*. The relationship in *Stall-On-Use* model is illustrated in Fig. 3.

Basically, the execution time consists of the computation and the memory stall time. The Optimistic Scheduling has the shorter *Load-To-Use Distance*, which may have longer use stall time if the data loaded is ready later. In contrast, the Pessimistic Scheduler has the longer *Load-To-Use Distance* but short *Use Stall Cycles*, which may have longer execution time because the instruction level parallelism is low.

B. Scratchpad Sensitive Scheduling

According to the analysis above, properly scheduling the *Load-To-Use Distance* can boost performance. For example, if we can schedule a Load Op having a large memory latency earlier and schedule its Use Op later, we can get the maximum *Hidden Stall Cycles* to shorten the Use Stall Cycles. Meanwhile, the microprocessor cycles within the *Load-To-Use Distance* can be used to schedule other instructions that do not depend on this load instruction [11], which can improve the degree of instruction level parallelism to shorten the execution time.

The key advantage of our scratchpad-based architecture is to preserve time predictability because the memory objects are allocated statically to SPMs in the compiler stage and no dynamic replacement is requested during the running time. Therefore, we can develop a scheduling algorithm named *Scratchpad Sensitive Scheduling* to be sensitive to the varying memory latencies of load instructions, as the data can be loaded from the L1 data SPM, the L2 SPM or the main memory.

Our *Scratchpad Sensitive Scheduling* is a basic block-based list scheduling algorithm. Its basic idea is to maintain a list of instructions ready to execute, to choose the instruction to schedule from the ready list, and then to update the ready list for the next microprocessor cycle. Generally, the list scheduling selects instructions from the ready list, based on the priorities of instructions. The priority is a function of what the algorithm designer views as an important criterion in selecting instructions [11].

In this paper, we extend the default *Critical Path Scheduling* algorithm in Trimaran infrastructure [12] to schedule the appropriate *Load-To-Use Distance* to improve the performance of programs running on our scratchpad-based architecture. Basically, the Critical Path Scheduling in Trimaran is a typical list scheduling algorithm, whose priorities for instructions are based on the height of instructions above the last exit instruction in basic blocks [12]. Eq. (9) is its priority function of each instruction *i* in basic blocks, where α is the weighted factor for the height of instructions. Because the sorting of instructions in the ready list is based on the ascending order, the instruction with larger height has higher priority. The α is set as -1 by default in Trimaran [12].

$$priority\ (i) = \alpha \times height\ (i) \tag{9}$$

In our Scratchpad Sensitive Scheduling, we consider two factors related to the *Load-To-Use Distance*, including the memory latency for a *Load Op* (*curLat*) and the related Load Op memory latency for a *Use Op* (*preLat*). The goal is to increase the priority of *Load Ops* with large memory access latencies, while decreasing the priority of *Use Ops* whose corresponding *Load Ops* have large access memory latencies. Then, we can enlarge the *Load-To-Use Distance* if there is flexibility in the scheduling. Eq. (10) is its priority function for each instruction *i* in a basic block, where α is the weighted factor for the height of instructions (set as -0.1 to ensure that instructions with larger height have higher priority), β is the weighted factor for the memory latency of Load Op (set as -0.45 to make the Load Op with higher priority), and γ is the weighted factor for the Load Op memory latency of a Use Op (set as 0.45 to let the Use Op with lower priority). The exact values of α, β, and γ are set by hand optimization, based on several experiments. However, these values can be set by multivariate analysis in future work. To be specific, for a different set of benchmarks, the optimized α, β, and γ might be different, because it depends on the patterns of load and use Op. For the set of benchmarks used in this paper, we simply manually adjusted the values of α, β, and γ for several rounds of testing, compared the results, and chose the set of α, β, and γ offering the best performance result as the hand-optimized values. We expected that the values of α, β, and γ can be optimized by the multivariate analysis or other optimization methods. However, this needs a significant research effort and is beyond the scope of this paper.

It is worth noting that if an instruction's successor instruction is a *Load Op*, it can inherit the *curLat* of its successor to raise the flexibility of Load Sensitive Scheduling. The reason is that the scheduling flexibility of a *Load Op* may be limited because its predecessor is scheduled prior to it. However, we can try to schedule the predecessor instruction earlier by giving it higher priority, and then the *Load Op* may be scheduled earlier to enlarge the *Load-To-Use Distance*. An algorithm to determine the priorities of instructions in the basic block is summarized in Fig. 4. The complexity of this algorithm is *O*(*N*), where *N* is the number of instructions in the basic block.

```
The Algorithm to Determine the Instruction Priority in Scratchpad Sensitive Scheduling
N: the set of instructions in the basic block; L: the set of Load Op;
S1: the set of data objects allocated in L1 SPM; U: the set of Use Op;
S2: the set of data objects allocated in L2 SPM;
1:  WHILE (N ≠ ∅) DO {
2:     pick an instruction i ∈ N;
3:     calculate Height(i);
4:     IF (i ∈ L) DO {
5:        IF (the data loaded by i ∈S1) DO curLatency(i) = L1 SPM Latency;
6:        ELSE IF (the data loaded by i ∈ S2) DO curLatency(i) = L2 SPM Latency;
7:        ELSE DO curLatency(i)= Memory Latency;
8:     }
9:     IF (i ∈ U) DO {
10:       get its Load OP j ∈ L;
11:       IF (the data loaded by j ∈ S1) DO preLatency(i) = L1 SPM Latency;
12:       ELSE IF (the data loaded by j∈ S2) DO preLatency(i) = L2 SPM Latency;
13:       ELSE predLatency(i) = Memory Latency;
14:    }
15:    IF (i ∉ L) DO {
16:       IF (its successor instruction j ∈ L) DO
17:          curLatency(i) = curLatency(j); //inherit the latency from successor Load Op;
18:    }
19:    priorty(i) = alpha * Height(i) + beta * curLatency(i) + gamma * predLatency(i);
20:    N = N - i;
21: }
```

Fig. 4. An algorithm to determine the priorities of instructions in the Scratchpad Sensitive Scheduling.

$$priority(i) = \alpha \times height(i) + \beta \times curLat(i) + \gamma \times predLat(i) \quad (10)$$

V. EVALUATION METHODOLOGY

We study the proposed two-level SPM based architecture and Scratchpad Sensitive Scheduling on a VLIW processor with four integer functional units, two floating point functional units, one load/store unit, and one branch unit. The VLIW processor has a global register file with 32 registers. The Trimaran compiler/simulator infrastructure [12] is used to evaluate the performance of the target VLIW processor. Trimaran consists of a front-end compiler IMPACT, a back-end compiler ELCOR, and a cycle-level VLIW processor simulator. Also, the energy consumption data of the SPM and cache are obtained using Cacti [13], which are incorporated into EPIC-Explorer, a parameterized VLIW-based platform framework for design space exploration [14] to report the total energy consumption. The ILP equations and inequalities were solved using a commercial ILP solver CPLEX [15]. The cache-based architecture uses the same processor with the same configuration, except for the cache memory part.

The default value of latencies in different levels of memories for the cache-based and the SPM-based architectures can be found in Table 1. The small values for the L2 shared scratchpad memory are used for fetching instructions/data of one or two words, as the SPMs do not have to fetch a whole cache block, like the cache. LRU is used as a cache replacement policy. Also, the write policy of data cache is chosen as Write-Back and Write-Allocate.

Table 1. Configuration parameters and their values in the base configuration of the simulated cache-based and SPM-based architecture

Configuration parameter	Value
Cache-based architecture	
L1 Instruction/data cache	LRU, 128 Bytes
	16 Bytes cache line
	1 Cycle latency
L2 Shared cache	LRU, 256 Bytes
	32 Bytes cache line
	10 Cycles latency
Memory	Unlimited size
	100 Cycles latency (8 Words fetch)
SPM-based architecture	
L1 Instruction/data SPM	128 Bytes
	1 Cycle latency
L2 Shared SPM	256 Bytes
	3 Cycles latency (1 Word Fetch)
	5 Cycles latency (2 Words Fetch)
Memory	Unlimited size
	100 Cycles latency (8 Words fetch)

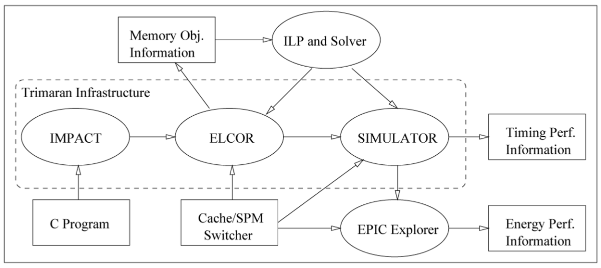


Fig. 5. The framework to evaluate our two-level SPM architecture and SPM-based scheduling. SPM: scratchpad memory, ILP: integer linear programming.

The framework of the extended Trimaran infrastructure for evaluation our work is demonstrated in Fig. 5. ELCOR outputs the memory objects information for setting ILP objective function and constraints to assign memory objects hotspots in scratchpad memories, which is solved by a commercial ILP solver CPLEX [15]. The results from the ILP solver were used by ELCOR again to assist Scratchpad-Sensitive Scheduling, if enabled, and to support the simulation by the Trimaran simulator. A cache/SPM switcher is defined to switch the function of our framework between the cache-based and scratchpad-based architecture. Simulation results are also used by EPIC-Explorer to generate an energy performance report. Also, the EPIC-Explorer and Trimaran simulator were extended to support both the cache-based and the SPM-based architectures.

To comparatively evaluate our SPM-based architecture and the cache-based architecture for real-time systems, we selected eight real-time benchmarks (i.e., kernels) from the MRTC real-time benchmark suite [16], the salient characteristics of which are given in Table 2. These benchmarks are either with the single fixed execution time, or with the variable execution time depending on different inputs.

VI. EXPERIMENTAL RESULTS

A. Observed WCET and Energy Results

- WCET Comparison: Our performance evaluation focuses on the observed WCET in terms of the number of clock cycles, including the computation cycles, the instruction cache stall cycles and the use stall cycles. Also, we focus on the WCET on both scratchpad and cache memories. The comparison results are shown in Fig. 6. All the WCET results of the cache-

Table 2. Salient characteristics of selected real-time benchmarks

Benchmark	Description	Path type	Compute cycles	Static instructions
compress	A demonstration for data compression program	Multiple paths	4438	429
crc	A demonstration for CRC operation	Multiple paths	25208	219
fdct	Forward Discrete Cosine Transform	Single path	2014	718
fft1	FFT using Cooly-Turkey algorithm	Multiple paths	2894	508
lms	An LMS adaptive signal enhancement	Multiple paths	457557	663
ludcmp	Simultaneous linear equations by LU decomposition	Multiple paths	3138	305
minver	Matrix inversion for 3x3 floating point matrix	Multiple paths	1915	490
qsort	Non-recursive version of quick sort algorithm	Multiple paths	1769	276

CRC: cyclic redundancy check, FFT: fast Fourier transform, LMS: least mean square.

based architecture are normalized to that of the SPM-based architecture. Thus, the performance of the SPM-based architecture is better than that of the cache-based architecture if the ratio is larger than 1, and *vice versa*. From Fig. 6, we can observe that the SPM based architecture performs better on all benchmarks except *lms* and *minver*, compared to either the best or the median WCET of the cache based architecture among all the four different associativity settings.

- Energy Consumption Comparison: Our energy consumption evaluation results consist of caches/SPMs energy consumption, the main memory energy consumption and the microprocessor energy consumption. The comparison results are shown in Fig. 7. All the energy dissipation results of the cache-based architecture are normalized to that of the SPM-based architecture. Thus, the energy consumption of SPM-based architecture is better than the cache-based architecture if the data is larger than 1, and *vice versa*. From Fig. 7, we can observe that the SPM-based architecture performs better on 75% benchmarks compared to either the best or the median energy consumption of the cache-based architecture among the four different set associativities.

In summary, these experimental results show that the proposed two-level SPM-based architecture has better observed WCET and energy consumption than the cache-based architecture in most cases. However, we also observed a few cases where the cache-based architecture performed better. Usually, if a benchmark does not have significant hotspots, its performance on the SPM-based architecture may not be as good as that of the cache-based architecture. The reason is that, unlike SPMs that are statically allocated, caches can dynamically reuse the space by replacing old instructions/data. For example, the benchmarks *lms* and *minver* consist of several loops with similar sizes, which are short of significant hotspots and thus lead to inferior performance. In detail, the memory allocation method in this paper is a static-based one. Only some of these loops can be fitted into the SPMs, because we cannot load other loops into the SPMs during the runtime. Additionally, the solution to this issue is actually the dynamic based allocation method, discussed in our previous paper [7].

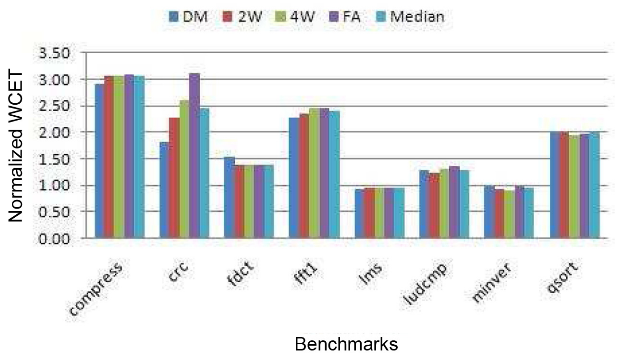


Fig. 6. The WCET comparison between the SPM-based and the cache-based architectures, which are normalized to the WCET of the SPM-based architecture (L1 size: 128 Bytes, L2 size: 256 Bytes). WCET: worst-case execution time, SPM: scratchpad memory.

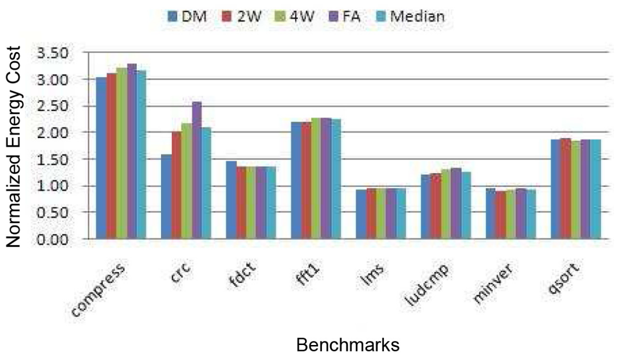


Fig. 7. The energy consumption comparison between the SPM-based and cache-based architectures, which are normalized to the energy consumption of the SPM-based architecture (L1 size: 128 Bytes, L2 size: 256 Bytes). SPM: scratchpad memory.

Table 3. Size settings of the cache-based and SPM-based architectures (unit: Bytes)

Level	Setting 1	Setting 2	Setting 3
L1 Inst.	128	256	512
L1 Data	128	256	512
L2 Shared	256	512	1024

B. Sensitivity Study

To evaluate the size sensitivity of our SPM-based architecture and the cache-based architecture, three pairs of size settings in L1 and L2 memories were used in our experiments, as shown in Table 3. The observed WCET results on different size settings are shown in Fig. 8 for the caches, and Fig. 10 for the SPMs. The energy consumption results on different size settings are shown in Fig. 9 for the caches, and Fig. 11 for the SPMs. Note that the results of the cache-based architecture are the best results among the four associativity settings. From these figures, it can be seen that both the observed WCET and energy consumption improved with the increase in memory size on both the SPM- and cache-based architectures.

Fig. 12 shows the difference between the observed WCET of cache- and SPM-based architectures. Also, Fig. 13 demonstrates the energy consumption difference between the cache- and SPM-based architectures. All results were normalized to the performance of the SPM-based architecture. However, because the improvement rate of these two architectures on the same benchmark with the increase of the cache/SPM size is not the same, their performance difference does not always follow the increase of the SPM/cache size. The exact performance of the SPM-based architecture is determined by whether there are outstanding hotspots in the program and the exact size of hotspots, because we used the static memory objects allocation algorithm to maintain the time predictability. Also, different benchmarks have different best SPM working sizes, which depend on the exact size of their significant hotspots. Basically, if the SPM size almost fits the size of hotspots, the benchmark has the best performance on this size of SPM. For example, the benchmark *compress* performs best on the size setting 2 from Figs. 12 and 13. If the size of the SPM is larger than

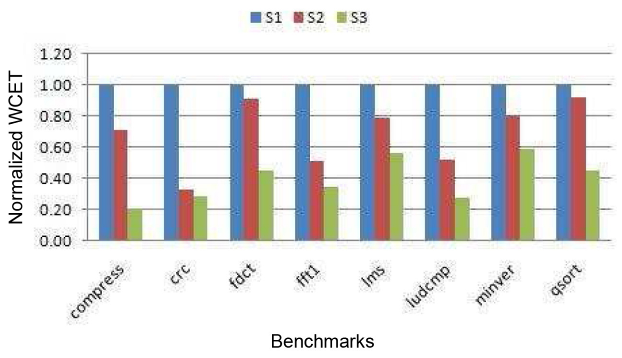


Fig. 8. The observed WCET comparison between the caches with different size settings (S1: size setting 1, S2: size setting 2, S3: size setting 3). WCET: worst-case execution time.

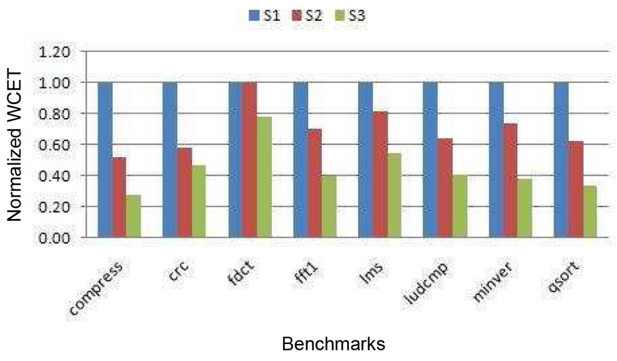


Fig. 10. The observed WCET comparison between the SPMs with different size settings (S1: size setting 1, S2: size setting 2, S3: size setting 3). WCET: worst-case execution time, SPM: scratchpad memory.

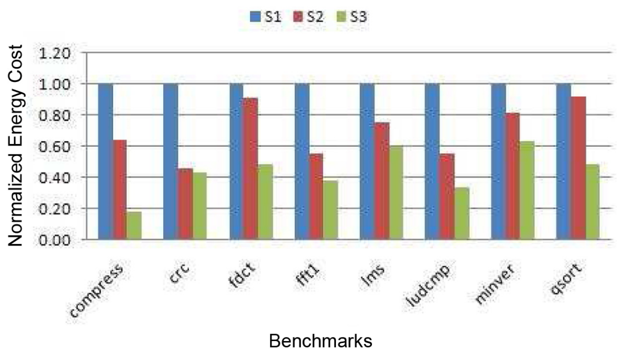


Fig. 9. The energy consumption comparison between the caches with different size settings (S1: size setting 1, S2: size setting 2, S3: size setting 3).

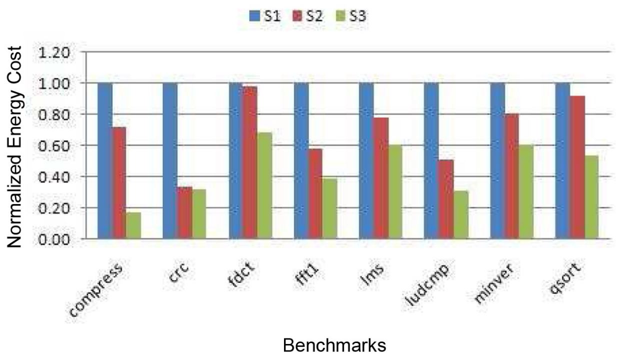


Fig. 11. The energy consumption comparison between the SPMs with different size settings (S1: size setting 1, S2: size setting 2, S3: size setting 3). SPM: scratchpad memory.

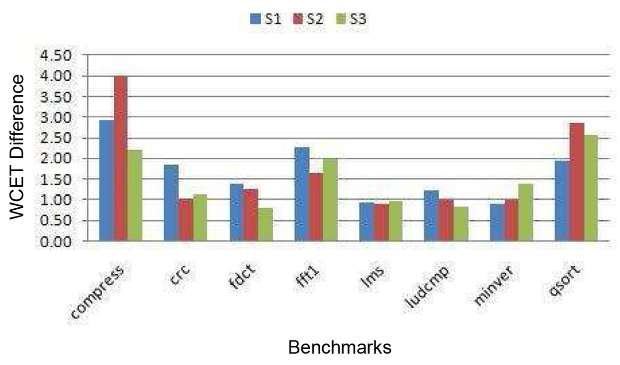


Fig. 12. The difference between the observe WCET of caches and SPMs under different size settings (S1: size setting 1, S2: size setting 2, S3: size setting 3). WCET: worst-case execution time, SPM: scratchpad memory.

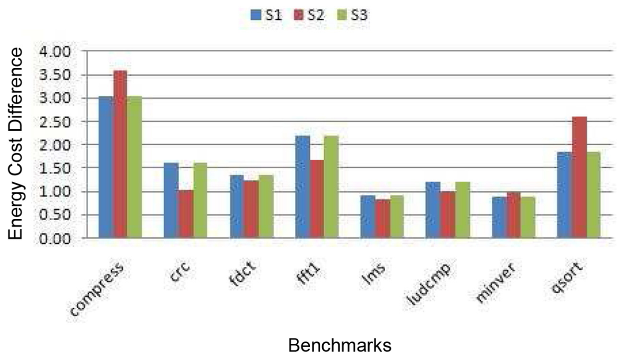


Fig. 13. The energy consumption difference between the caches and SPMs under different size setting (S1: size setting 1, S2: size setting 2, S3: size setting 3). SPM: scratchpad memory.

the size of hotspots, some non-hotspots will be allocated to the SPM. On the other hand, if the size of the SPM is smaller than the size of hotspots, some hotspots cannot be allocated to the SPM. Therefore, both cases will limit the performance of the SPM compared to that of the cache.

C. Why Two-Level SPMs?

We also evaluated the improvement of our two-level SPM-based architecture by comparing it to the performance and energy efficiency of the one-level SPM architecture, which has larger L1 instruction and data scratchpad memories but does not have the L2 scratchpad. In this one-level SPM-based architecture, the total size of its level-one instruction and data SPMs is equal to the sum of the sizes of SPMs (i.e., L1 instruction, L1 data and L2 SPMs) in a two-level SPM architecture. More specifically, the size of the L1 SPMs were set as 128 bytes for each of the instruction and data SPM, and the L2 scratchpad size was set as 256 bytes in the two-level architecture, and the size of L1 SPMs was set as 256 bytes for the instruction and data in the one-level SPM architecture,

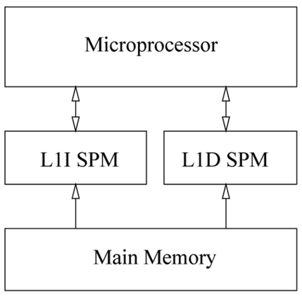


Fig. 14. One-level SPM-based architecture. SPM: scratchpad memory.

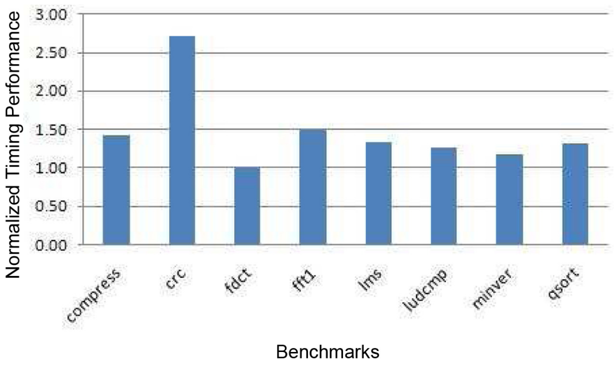


Fig. 15. The observed WCET comparison between the one-level and two-level SPM-based architecture. WCET: worst-case execution time, SPM: scratchpad memory.

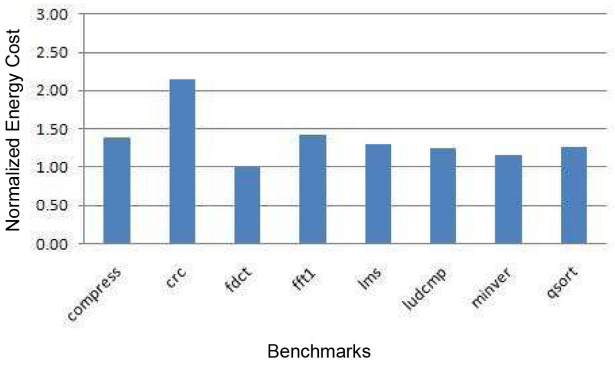


Fig. 16. The energy consumption comparison between the one-level and two-level SPM-based architecture.

respectively. Because the size of L1 SPMs in the one-level architecture is enlarged, they have larger latencies than the small SPMs used in the two-level architecture. In this experiment, the latency of large L1 SPMs in the one level architecture is set as the same as the latency of the L2 scratchpad in the two-level architecture, since their size is the same. It should be noted that in high-performance microprocessors, the L1 caches are typically kept small because a larger L1 caches often result in longer L1 cache access latency, which is also another reason that

Table 4. The comparison of performance improvement of Scratchpad Sensitive Scheduling (L1 size: 128 Bytes, L2 size: 256 Bytes)

Benchmark	Scratchpad (Default)		Scratchpad (SSS)		Difference	
256-128	Computation	Use Stall	Computation	Use Stall	Diff (C)	Diff (U)
compress	4438	294	4389	294	0.989	1.000
crc	25208	984	25208	984	1.000	1.000
lms	457557	121527	452632	113346	0.989	0.933
ludcmp	3138	2698	3017	2554	0.961	0.947

Table 5. The comparison of performance improvement of Scratchpad Sensitive Scheduling (L1 size: 256 Bytes, L2 size: 512 Bytes)

Benchmark	Scratchpad (Default)		Scratchpad (SSS)		Difference	
512-256	Computation	Use Stall	Computation	Use Stall	Diff (C)	Diff (U)
compress	4438	1176	4389	1176	0.989	1.000
crc	25208	984	25208	984	1.000	1.000
lms	457557	81802	456850	73621	0.998	0.900
ludcmp	3138	3223	3017	3169	0.961	0.983

Table 6. The comparison of performance improvement of Scratchpad Sensitive Scheduling (L1 size: 512 Bytes, L2 size: 1024 Bytes)

Benchmark	Scratchpad (Default)		Scratchpad (SSS)		Difference	
1024-512	Computation	Use Stall	Computation	Use Stall	Diff (C)	Diff (U)
compress	4438	1335	4389	1335	0.989	1.000
crc	25208	329	25208	329	1.000	1.000
lms	457557	136403	456853	129461	0.998	0.949
ludcmp	3138	3336	3017	3336	0.961	1.000

those processors often employ a larger L2 cache while keeping the L1 caches small.

The performance comparison results are presented in Fig. 15, and the energy consumption comparison results are demonstrated in Fig. 16. The data are normalized to the execution time and energy consumption of a two-level SPM-based architecture, respectively. We can observe that both the performance and energy consumption of a two-level architecture are better than those of a one-level SPM architecture, indicating the advantage of adding an L2 SPM instead of using large L1 SPMs. We also notice that the benchmark *fdct* showed only slight performance improvements. The reason is that the size of hotspots in this benchmark was larger than with the L1 SPM size. Thus, they can only be allocated to the L2 scratchpad in the two-level architecture, which limits the performance improvement.

D. Results of Scratchpad Sensitive Scheduling

Our experimental results of Scratchpad Sensitive Scheduling are shown in Tables 4–6 for different cache/SPM size settings, respectively. As in our analysis in Subsection IV-A, our Scratchpad Sensitive Scheduling may improve the computation and Use Stall Cycles if there is flexibility in scheduling. Among our eight real-time benchmarks, we observed four with obvious Use Stall Cycles, which have possibility of being improved by our scheduling algorithm. From Tables 4–6, we find that 75% of these four benchmarks *did* show performance improvement. The maximum improvement of computation cycles is about 3.9%, and the maximum improvement of use stall cycles is about 10%.

Also, we observe larger improvements in Use Stall Cycles on the benchmarks with originally large Use Stall Cycles (e.g., the benchmarks *lms* and *ludcmp*), while there is almost no improvement on the benchmarks with originally small Use Stall Cycles (e.g., the benchmarks *compress* and *crc*). The reason is that for benchmarks with more original *Use Stall Cycles*, the compiler is more likely to set the *Load-to-Use Distance* to reduce the *Use Stall Cycles*. In contrast, there is less room for the compiler to reduce the *Use Stall Cycles* for benchmarks with originally small *Use Stall Cycles*.

VII. CONCLUSIONS

In this paper, we propose a time-predictable two-level SPM-based architecture, and use an ILP-based static memory objects assignment algorithm to maintain the characteristic of time predictability. Both the performance and energy consumption of our SPM-based architecture are compared quantitatively with a cache-based architecture. Also, we studies the Scratchpad Sensitive Scheduling to further improve the performance of our proposed architecture. Our experimental results indicated that our proposed SPM-based architecture performed better than the cache-based architecture for the majority of benchmarks we assessed.

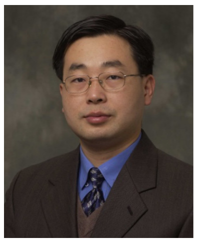
In future work, we would like to assess time-predictable dynamic SPM allocation schemes for multi-level SPMs. Also, we plan to extend the scope of the Scratchpad Sensitive Scheduling using trace-based scheduling or superblock/hyperblock scheduling while keeping time-predictability.

REFERENCES

1. 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*, Sicily, Italy, 2004, p. 1-4.
2. O. Avissar, R. Barua, and D. Stewart, “An optimal memory allocation scheme for scratch-pad-based embedded systems,” *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, pp. 6-26, 2002.
3. S. Steinke, L. Wehmeyer, B. S. Lee, and P. Marwedel, “Assigning program and data objects to scratchpad for energy reduction,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Paris, France, 2002, pp. 409-415.
4. J. F. Deverge and I. Puaut, “WCET-directed dynamic scratchpad memory allocation of data,” in *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS2007)*, Pisa, Italy, 2007, pp. 179-190.
5. 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 (MEDEA2008)*, Toronto, Canada, 2008, pp. 38-45.
6. 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,” *Technical Report 762*, University of Dortmund, Germany, 2001.
7. 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 (ICCD2012)*, Montreal, Canada, 2012, pp. 61-66.
8. 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 (CODES2002)*, Estes Park, CO, 2002, pp. 73-78.
9. J. Whitham and N. Audsley, “Implementing time-predictable load and store operations,” in *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT2009)*, Grenoble, France, 2009, pp. 265-274.
10. T. Lundqvist and P. Stenstrom, “Timing anomalies in dynamically scheduled microprocessors,” in *Proceedings of the 20th Real-Time Systems Symposium*, Phoenix, AZ, 1999, pp. 12-21.
11. C. R. Hardnett, K. V. Palem, R. M. Rabbah, and W. F. Wong, “Scheduling load operations on VLIW machines,” *Technical Report GITCC-01-015*, Georgia Institute of Technology, Atlanta, GA, 2001.
12. Trimaran, http://www.trimaran.org.
13. P. Shivakumar and N. P. Jouppi, “Cacti 3.0: an integrated cache timing, power, and area model,” *Technical Report 2001/2*, Compaq Computer Corporation, Harris County, TX, 2001.
14. 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.
15. CPLEX, http://www.ilog.com/products/cplex/.
16. Mälardalen Real-Time Research Center, “WCET project - benchmarks,” http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

Yu Liu

Yu Liu is currently a research scientist in Canadian Nuclear Laboratories (was Atomic Energy of Canada Ltd.). He received his B.S. and M.S. degrees from Sichuan University, China in 2000 and 2003 respectively, and Ph.D. degree from Southern Illinois University Carbondale in 2011. He worked in Motorola from 2003 through 2007, and IBM from 2011 through 2013. Also, he had two summer research works in Pacific Northwest National Lab, U.S. Department of Energy in 2009 and 2010, respectively. His research interest includes real-time system, wireless sensor network, cyber-physical system and high performance computing.



Wei Zhang

Wei Zhang is a professor in the Department of Electrical and Computer Engineering at Virginia Commonwealth University. Dr. Wei Zhang received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, Dr. Zhang worked as an assistant professor and then as an associate professor (tenured) at Southern Illinois University Carbondale. His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. Dr. Zhang has 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 the Journal of Computing Science and Engineering. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.